Introduction of the Sirepo-Bluesky interface and its application to the optimization problems

Rakitin, Maksim, Giles, Abigail, Swartz, Kaleb, Lynch, Joshua, Moeller, Paul, et al.

Maksim S. Rakitin, Abigail Giles, Kaleb Swartz, Joshua Lynch, Paul Moeller, Robert Nagler, Daniel B. Allan, Thomas A. Caswell, Lutz Wiegart, Oleg Chubar, Yonghua Du, "Introduction of the Sirepo-Bluesky interface and its application to the optimization problems," Proc. SPIE 11493, Advances in Computational Methods for X-Ray Optics V, 1149311 (21 August 2020); doi: 10.1117/12.2569000

SPIE.

Event: SPIE Optical Engineering + Applications, 2020, Online Only

Introduction of the Sirepo-Bluesky interface and its application to the optimization problems

Maksim S. Rakitin[a,*], Abigail Giles[a,b,*], Kaleb Swartz[c], Joshua Lynch[a], Paul Moeller[d], Robert Nagler[d], Daniel B. Allan[a], Thomas A. Caswell[a], Lutz Wiegart[a], Oleg Chubar[a], and Yonghua Du[a]

[a]National Synchrotron Light Source II, Brookhaven National Laboratory, Upton, NY 11973, USA
[b]Mathematics/Computer Science Department, St. Joseph's College New York, Patchogue, NY 11772
[c]College of Engineering and Computing, Miami University, Oxford, OH 45056, USA
[d]RadiaSoft LLC, 3380 Mitchell Ln, Boulder, CO 80301

ABSTRACT

Simulation of beamlines at light sources is an essential part of their design and commissioning. Such simulations can be performed by the Synchrotron Radiation Workshop (SRW) code, which now has a user-friendly, browser-based interface, Sirepo. The simulations, utilizing a concept of a "virtual" beamline, can aim to optimize the specific aspects of a beamline, such as maximization of the flux, minimization of the beam size, *etc.* These tasks are also performed at the physical beamlines using various alignment procedures. At NSLS-II these procedures are executed by the Bluesky data collection framework. The Sirepo-Bluesky interface leverages both systems to allow for the multiparameter optimization of the X-ray source and beamline optics with the power of bluesky's plans used for the daily experiments at NSLS-II, and databroker's capabilities to retrieve the captured data and metadata to perform further analysis. Such a "collaboration" between the frameworks can be used to store the simulated results in the same database as for the experimental data, and seamlessly apply the same analysis pipelines, demonstrated in recent publications. In a simulation, multiple parameters can be changed at once and be stored as a snapshot of the "virtual" beamline in time along with the corresponding results of the simulations. A global optimization algorithm (*e.g.*, a genetic algorithm) can then utilize the data to find the best configuration for the desired outcome. Such an optimization procedure can be seamlessly applied to the real hardware by substituting the virtual motors and detectors by the real ones.

Keywords: Sirepo, SRW, Bluesky, Databroker, Ophyd, optimization, simulations

1. INTRODUCTION

Sirepo is a browser-based framework allowing for running simulations on a personal computer, a supercomputer or in the cloud.[1] Synchrotron Radiation Workshop (SRW)[2] is one of the simulation codes interfaced with Sirepo. SRW can be used to perform simulations of the Synchrotron Radiation sources as well as the wavefront propagation through optical systems of X-ray beamlines. For these purposes there are corresponding "Source" and "Beamline" pages in the Sirepo interface. Frequently, multiple parameters are required to be optimized to achieve the desired cost function (such as maximization of the flux or total intensity, or minimization of the beam size). Despite the convenience of the browser-based interactive interface of Sirepo, it can be time consuming to change multiple parameters by hand (especially in different pages/tabs) and keep track of the changes and corresponding results. The Bluesky framework,[3,4] designed to solve such problems, and routinely used at daily operations of NSLS-II beamlines, can be leveraged to facilitate multiple parameter change and to record the results of simulations as a function of these parameters. The *bluesky* library is responsible for changing the parameters and "triggering" a simulation, while the *databroker* library provides access to the recorded data and

* These authors contributed equally.
Corresponding authors: Maksim Rakitin: mrakitin@bnl.gov; Yonghua Du: ydu@bnl.gov

Advances in Computational Methods for X-Ray Optics V, edited by Oleg Chubar, Kawal Sawhney, Proc. of SPIE Vol. 11493, 1149311 · © 2020 SPIE
CCC code: 0277-786X/20/$21 · doi: 10.1117/12.2569000

metadata (*e.g.*, stored in a MongoDB database). The Sirepo-Bluesky library[5] was created to integrate the two frameworks — Sirepo and Bluesky. In the present work we cover only the single-electron simulations, but it is possible to reuse the infrastructure to perform partially-coherent (typically long-running) simulations in a scripting manner. Below we will explain each component in details and provide examples.

2. SOFTWARE LIBRARIES AND FRAMEWORKS

2.1 Bluesky Data Collection Framework

The Bluesky data collection framework consists of a few core libraries: *bluesky*, *ophyd*, *event-model*, and *databroker*. The relationships between the libraries are depicted in the Fig. 1.

2.1.1 Bluesky

The *bluesky* library handles experiment control and scientific data collection, allowing control of an instrument irrespective of its software/hardware configuration. It uses a concept of plans, which define the experimental procedures to run as Python functions. The plans allow implementation of complex procedures in a way convenient for a beamline scientist or a user. Users can either employ predefined plans (such as `count`, `scan`, `grid_scan`), or custom plans may be designed if the preexisting plans do not satisfy the experimental requirements. The `RunEngine` is a core part of the bluesky library, which passes instructions from the plans to the lower hardware levels (*e.g.*, to move motors and/or read detectors) and organizes metadata and measurements into documents emitted to user-configurable consumers.

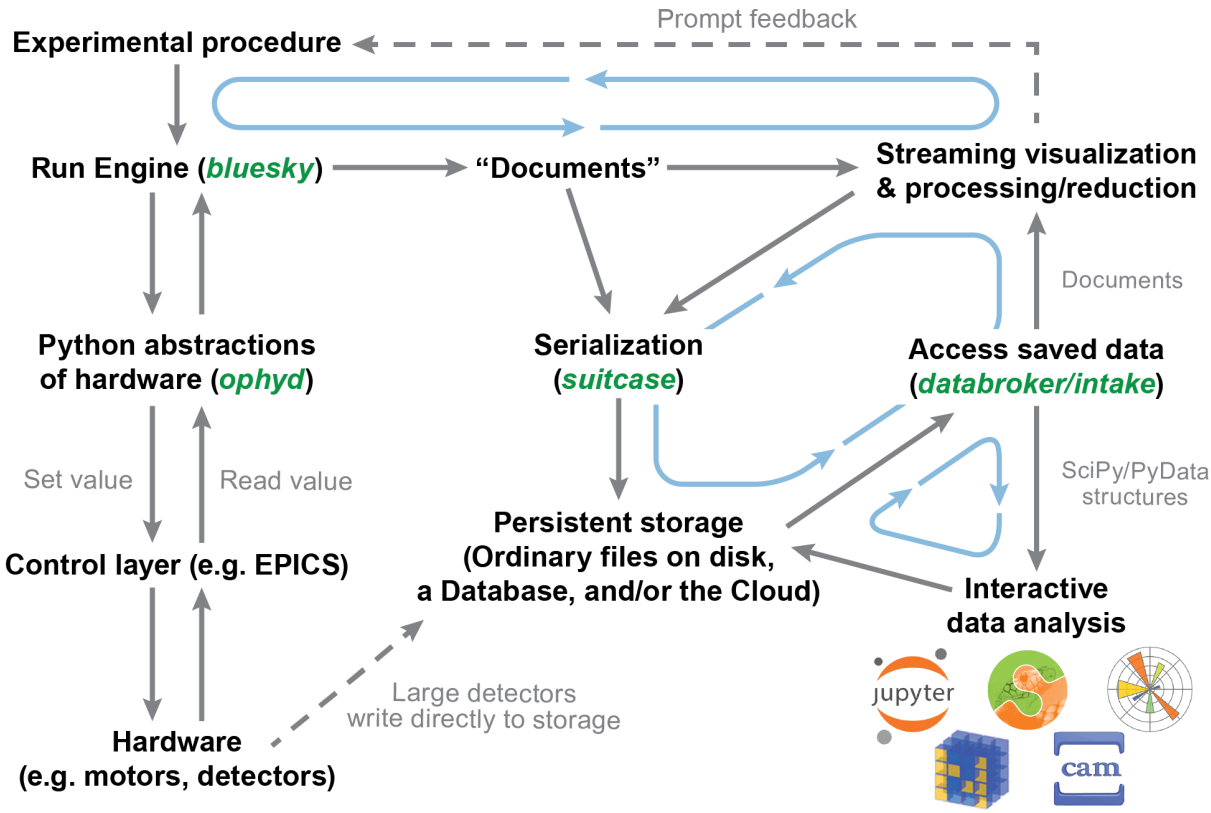

Figure 1. The relationships between the libraries in the Bluesky data collection framework.

2.1.2 Ophyd

The lower hardware levels are abstracted using the *ophyd* hardware abstraction library. At NSLS-II, it is abstracted from the EPICS control layer using the *pyepics* library, however non-EPICS devices can be integrated. An *ophyd* object, based on `Device`, can have multiple components (`Signal`s or other `Device`s). The latter is the core class of *ophyd*, which encodes the nodes of the hierarchical structure of the device and provides the core API. The `Signal`s are the "leaves" of the structure, and are responsible for communicating with the underlying control system. A readable *ophyd* `Device` provides a consistent API, for example, it can be "staged" (made ready to collect the data), "triggered" (instructed to acquire data), "read" (read the acquired data in the form of documents, see below), and "unstaged" (reverted to the original configuration), which allows them to be used interchangeably.

2.1.3 Document model

Bluesky makes data available to consumers in a streaming fashion during data acquisition. Data and metadata are organized into JSON-like documents, validated against a formal schema by the *event-model* library. These documents may be stored or used to feed online visualization or data analysis routines. Each scientifically-meaningful unit of data are bracketed by `start` and `stop` documents. The `start` document can contain the information known before the start of a *bluesky* (as a library) run (such as start time, user-provided metadata, *etc.*). The data to be collected during a run is described in terms of source, data type, dimensions, units, *etc.*, by one or more `descriptor` documents. The `stop` document has the information about the end of the run. The readings of devices during a run are stored in the `event` documents (such as positions of motors or readings from detectors at each point of a scan).

2.1.4 Databroker

The *databroker* library, a part of the Bluesky project, provides convenient methods to access the documents containing data and metadata from experimental measurements and operate on them in the form of standard scientific Python data structures (Pandas dataframes, NumPy arrays, *etc.*).

The Bluesky framework's libraries are installed at the NSLS-II beamlines, and are readily available to use and interface with other frameworks such as Sirepo.

2.2 Sirepo Simulation Framework

As mentioned above, Sirepo is an "umbrella" for simulation codes such as SRW, Shadow3,[6] and particle accelerator simulation codes. Its rich interactive capabilities make it an attractive environment to integrate other simulation codes. Sirepo is a distributed system: the backend server can run on a personal computer, or can execute on HPC resources to perform simulations, while the frontend, based on Javascript, is executed in a browser on a user's computer. For quick simulations, the backend and the frontend parts can run on the same computer, while for computationally intensive simulations it is recommended to use multi-core servers or supercomputers for running the backend server. The exchange between the server and the client sides is done using the JSON format. The readers are encouraged to check a detailed explanation of the framework in the corresponding paper.[1]

In this paper we will focus on the SRW interface in Sirepo. Sirepo provides a number of predefined examples, however users can define their own optical scheme with the predefined optical elements. Each element has a number of parameters which can be varied (in the graphical interface, or programmatically).

2.2.1 Explanation of the unique identifiers (UIDs) used in Sirepo and Bluesky

Every Sirepo simulation has a unique identifier (called `sim_id` throughout the text) in the URL (such as the `yuL5LMmD` part in the URL in the caption of the Fig. 7), by which one can access simulations programmatically. This UID is a random string consisting of 8 alpha-numerical symbols to reasonably guarantee different simulations to have different identifiers to avoid collisions of the corresponding simulation directories on a Sirepo server.

Each Bluesky-generated document has a unique, persistent identifier. We use a "universally unique identifier" (UUID) Version 4 (random), which is a 128-bit number used to identify information in computer systems. The canonical "8-4-4-4-12" format string (the numbers indicate the number of symbols in each dash-separated group)

is based on the record layout for the 16 bytes of the UUID.[7] In Python, such UUIDs can be generated by the *uuid* library:

```
> python
Python 3.6.4 |Anaconda, Inc.| (default, Mar 12 2018, 20:05:31)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import uuid
>>> print(uuid.uuid4())
4ae79507-0358-4d44-9554-880f7031946b
```

For users' convenience and better readability, this long UUID4 string can be shortened to the first 8 symbols (`4ae79507`), or sometimes even to the first 6 symbols (`4ae795`). As this part of the Bluesky ecosystem is user-facing (especially when one works with *databroker*), there is a more human-oriented, but transient identifier `scan_id`, which is a sequential number incrementing after each scan. Users can access their experiment data by either full UUID4, shortened 8-symbol UUID4, or the `scan_id` identifier, but accessing experiment data using UUID4 is the only reliable (and hence recommended) approach long term.

2.3 Sirepo-Bluesky library

The *sirepo-bluesky* library was developed to interface the two frameworks explained above. It is an open-source Python library with the source code available on GitHub.[5] A concept of a virtual detector is used in this library. The idea is to have an abstracted object which would conform to the interface of the *ophyd* `Device`, however triggering of such a detector would mean to start a simulation in Sirepo and return results in the same fashion as it would be done with the real, EPICS-based detectors. The *sirepo-bluesky* library requires a running Sirepo server to perform the simulations. The Sirepo server can be started via Vagrant as a VirtualBox Linux machine, or as a Docker Linux container (Sirepo is designed to work on Linux in production environments). The repository https://github.com/NSLS-II/sirepo-bluesky contains step-by-step installation and configuration instructions. The instructions also indicate how to prepare a Python environment (conda is recommended, but any approaches should work as long as one can install packages with `pip`). It can be the same physical machine where Sirepo runs, or it can be a different machine, and the communications with Sirepo will be done remotely over http(s). For convenience, all examples below are recommended to run in IPython.[8] All visualization in Python is done with Matplotlib.[9]

3. SIREPO-BLUESKY EXAMPLES

3.1 Performing a 2D Raster Scan on the "Beamline" Page of Sirepo

This example covers a 2D raster scan of the aperture opening in horizontal and vertical directions while taking the resulted intensity distribution images after X-ray propagation through the aperture on the "Beamline" page of Sirepo/SRW interface.

The following IPython code snippet will prepare the environment for execution, assuming the `basic.zip` example (see Fig. 2) was uploaded to the running Sirepo server (its `sim_id` will be used later on).

```
%run -i examples/prepare_det_env.py
import sirepo_bluesky.sirepo_detector as sd
import bluesky.plans as bp
sirepo_det = sd.SirepoDetector(sim_id='<sim_id>', reg=db.reg)
sirepo_det.select_optic('Aperture')
param1 = sirepo_det.create_parameter('horizontalSize')
param2 = sirepo_det.create_parameter('verticalSize')
sirepo_det.read_attrs = ['image', 'mean', 'photon_energy']
sirepo_det.configuration_attrs = ['horizontal_extent', 'vertical_extent', 'shape']
```

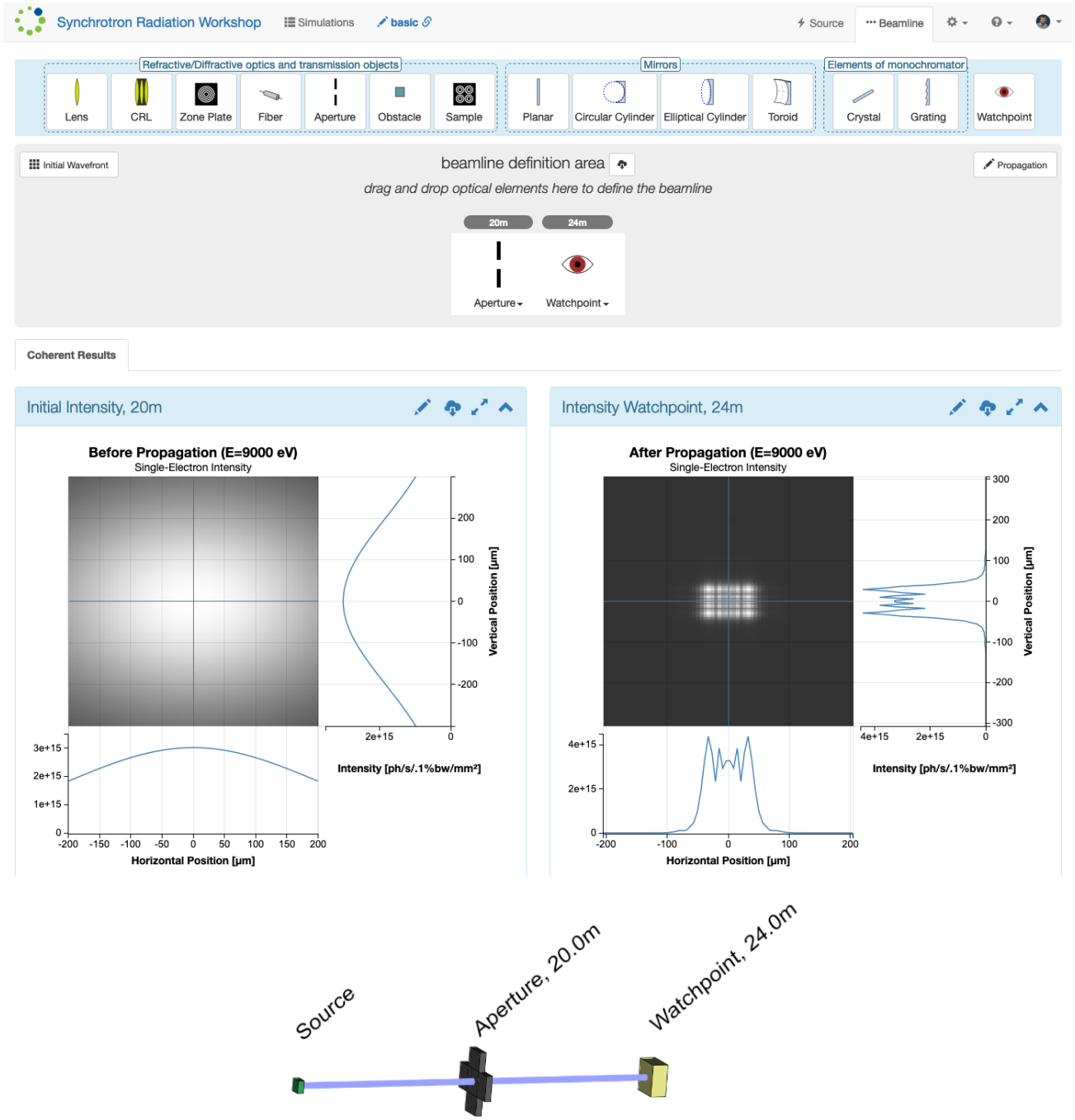

Figure 2. The top picture displays the Beamline page of Sirepo where a simple optical scheme is generated from the imported archive `basic.zip` (however one can interactively construct a "virtual" beamline with the available optical elements). The widgets below the beamline definition area display the initial intensity distribution before the first optical element (Aperture) at 20 m from the center of the source, and the resulting intensity distributions after wavefront propagation through the aperture and the subsequent "drift" space at the Watchpoint element located at 24 m from the source. The bottom picture displays the 3D representation of that optical layout produced by the experimental feature of Sirepo (not available on production servers as of time of this writing).

Line 1 in the example above uses the IPython "magic" command to run a Python module interactively in the current Python environment. That module has definitions of bluesky's `RunEngine`, databroker (`db`), and sets up all required callbacks. The next two lines import required parts of the *sirepo-bluesky* and *bluesky* libraries. Line 4 shows the code responsible for creation of the `SirepoDetector` *ophyd* object, which is instantiated using the `sim_id` string corresponding to the simulation in Sirepo (see Subsection 2.2.1 for details). Optionally, the server URL can be provided — that way one can point the object to a desired server given that it is configured to accept the external requests. The authentication with the Sirepo server is done with the default "secret" keyword, but it can be parametrized. On line 5, an optical element of interest (`Aperture`) is selected from the list of available elements in the "virtual" beamline. The next two lines (6-7) set the parameters to be scanned later. Line 8 defines the read attributes — the quantities to be read after each simulation (image reference to the intensity distribution on the Watchpoint at 24 m from the center of the source, averaged intensity from that image, and the corresponding photon energy). Line 9 defines the configuration attributes used for further processing/visualization.

The following command will start the grid scan with `RunEngine` using the defined detector and the horizontal and vertical `Aperture` sizes as "motors" over a mesh of 10×10 points (9×9 intervals) from 0 to 1 mm for each motor. The last argument `True` on line 4 means to perform a "snake" type of scanning trajectory, following snake-like, winding trajectory instead of a simple left-to-right trajectory. This command returns the UUID4 at the end of execution. A snapshot of the live visualization during the scan is depicted in Fig. 3 (left) showing the averaged intensity for each grid point. It is worth to mention that for this type of scan, the "slow" (vertical) axis in the visualization is for the first "motor" — `horizontalSize`, while the `verticalSize` "motor" corresponds to the "fast" (horizontal) axis.

```
1 RE(bp.grid_scan([sirepo_det],
2                 param1, 0, 1, 10,
3                 param2, 0, 1, 10,
4                 True))
```

To get one of the intensity distribution images, one would access *databroker*'s records such as in the example below. It uses *databroker*'s V1 API, where `db[-1]` on line 1 returns a "header" for the latest scan (which can equivalently be retrieved by the returned UUID from `RunEngine`), which can be used to get the image data (line 2). The `sirepo_det_image` string is the name of the image field, and `list(...)` is needed to convert the "lazy" generator Python object into a Python list. Line 6 is to show the image corresponding to the scan point number 22 (indexing in Python starts from 0) depicted on Fig. 3 (right)

```
1 hdr = db[-1]
2 imgs = list(hdr.data('sirepo_det_image'))
3 cfg = hdr.config_data('sirepo_det')['primary'][0]
4 hor_ext = cfg['{}_horizontal_extent'.format(sirepo_det.name)]
5 vert_ext = cfg['{}_vertical_extent'.format(sirepo_det.name)]
6 plt.imshow(imgs[21], aspect='equal', extent=(*hor_ext, *vert_ext))
```

3.2 Single-Electron Spectrum Calculation on the "Source" Page of Sirepo

This is another useful example of a calculation that can be performed on the "Source" page of Sirepo. In this example, we used a predefined "NSLS-II CHX beamline" simulation (see Fig. 4 (left)) corresponding to a different `sim_id` in Sirepo, which can be found in the URL to the simulation.

```
1 %run -i examples/prepare_det_env.py
2 import sirepo_bluesky.sirepo_detector as sd
3 import bluesky.plans as bp
4 sirepo_det = sd.SirepoDetector(sim_id='<sim_id>', reg=db.reg, source_simulation=True)
5 sirepo_det.read_attrs = ['image']
6 sirepo_det.configuration_attrs = ['photon_energy', 'shape']
```

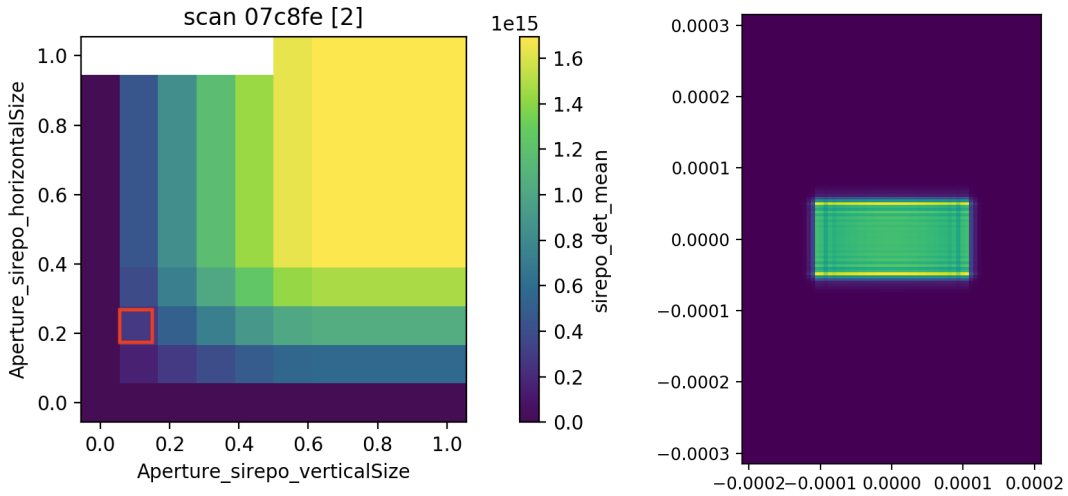

Figure 3. A snapshot of a live grid visualization captured during a running *bluesky* grid scan (left) and the intensity distribution image from the scan point #22 retrieved from the *databroker* after the scan (right). The red square on the left image indicates horizontal and vertical `Aperture` sizes for the image on the right, *i.e.* 0.22(2) mm (h) × 0.11(1) mm (v). The upper left corner of the left image with missing data indicates that the scan did not get to those data points yet. Please note, the units on the left plot match the ones in the Sirepo interface (mm), while the right plot is obtained from the raw SRW-generated file `res_int_pr_se.dat`, which uses SI units (m). The string "`scan 07c8fe [2]`" on top of the left plot comes from *bluesky*'s `LiveGrid` live visualization callback, where `07c8fe` is a shortened version of UUID4 and `[2]` is a transient sequential `scan_id` (in square brackets) corresponding to the running scan.

This example's setup is very similar to the previous one, but the `source_simulation=True` keyword argument instructs the detector object to perform a simulation on the "Source" page of Sirepo. The read and configuration attributes were updated to reflect a changed nature of the scan/simulation. The parameter variables are not used in this example as a simple `count` *bluesky* plan will be executed:

```
RE(bp.count([sirepo_det]))
```

And to get the resulting spectrum, a very similar code snippet is used (see Fig. 4 (right) for the result):

```
hdr = db[-1]
cfg = hdr.config_data('sirepo_det')['primary'][0]
energies = cfg['sirepo_det_photon_energy']
spectrum, = hdr.data('sirepo_det_image')
plt.plot(energies, spectrum)
```

3.3 Simulated Sirepo Flyer to Run Multiple Sirepo Simulations

This example is based on the "Young's Double Slit Experiment" example in Sirepo (see Fig. 5) that can be found in the "Wavefront Propagation" folder on the SRW simulations page, and uses a concept of flyers. Flyers are *ophyd* devices that have the required methods to meet the "flyable" interface: `kickoff`, `complete`, `collect`, and `describe_collect`. Flyer devices continuously collect data as motors move, so rapid data acquisition can occur. Flyers are instrument-specific, so each beamline or simulation engine would need to have a unique flyer implementation. See the *bluesky* documentation[10] for more details.

Start with preparing the environment:

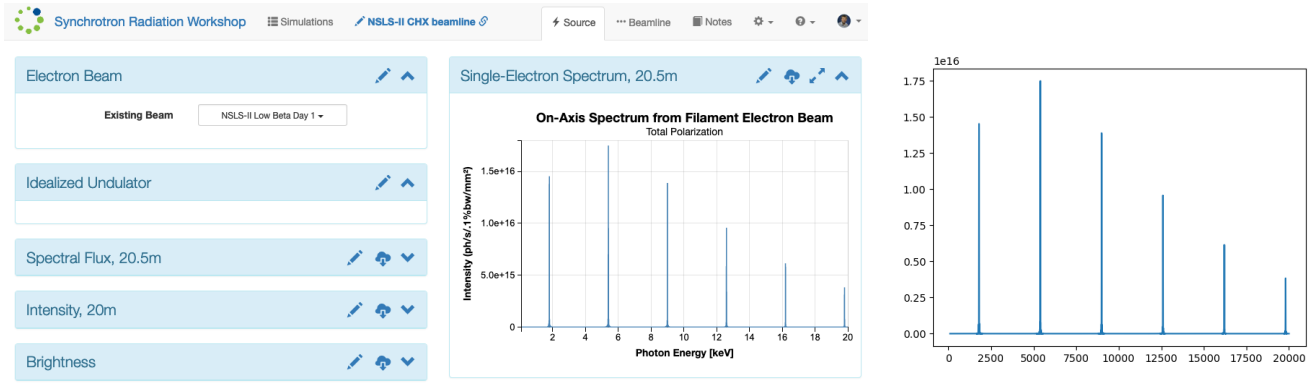

Figure 4. A single-electron spectrum report on the "Source" page of Sirepo (left) and the corresponding data retrieved from the *databroker* collected by `sirepo_det` (right).

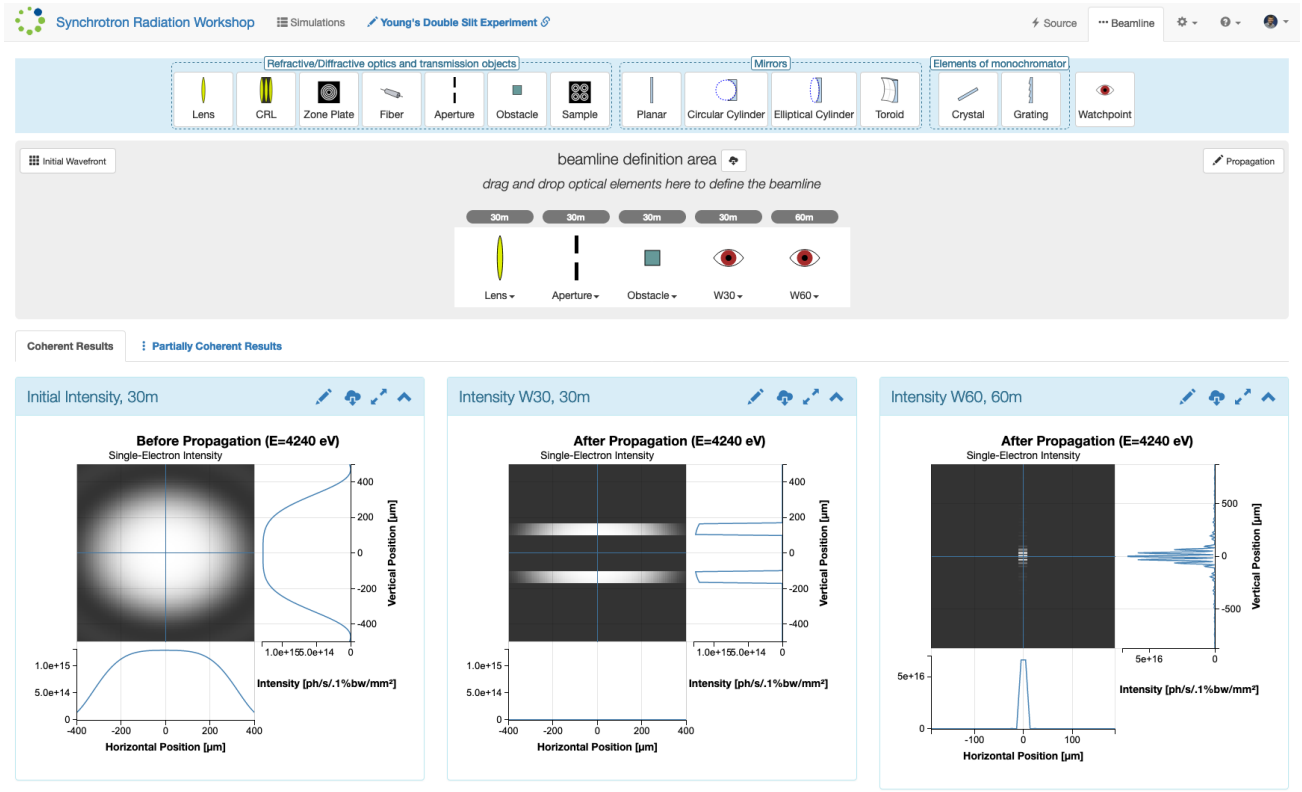

Figure 5. The "Beamline" page of the "Young's Double Slit Experiment" example in Sirepo, showing the optical layout consisting of a Lens, an Aperture, an Obstacle (all located at 30 meters after the source), and 2 watchpoints — one right after the Obstacle (`W30`) and another one (`W60`) at the distance of 60 meters after the source. The bottom part of the picture displays the intensity distribution widgets before propagation and after propagation corresponding to each watchpoint.

```
%run -i examples/prepare_flyer_env.py
import bluesky.plans as bp
import sirepo_bluesky.sirepo_flyer as sf
```

The following code will prepare input for the flyer which will perform 5 different simulations that change 4 parameters (`Aperture`'s `horizontalSize` and `verticalSize`, `Lens`' `horizontalFocalLength`, and `Obstacle`'s `horizontalSize`,) at a time:

```
params_to_change = []
for i in range(1, 6):
    key1 = 'Aperture'
    parameters_update1 = {'horizontalSize': i * .1, 'verticalSize': (16 - i) * .1}
    key2 = 'Lens'
    parameters_update2 = {'horizontalFocalLength': i + 7}
    key3 = 'Obstacle'
    parameters_update3 = {'horizontalSize': 6 - i}
    params_to_change.append({key1: parameters_update1,
                             key2: parameters_update2,
                             key3: parameters_update3})
```

All parameters are coupled as a function of `i`, but one can specify the desired values for each parameter to make them completely independent.

The following code will create the flyer and run a `fly` scan (where `sim_id` is the Sirepo identifier of this simulation), changing the specified parameters and getting the resulting intensity distribution images on the watchpoint `W60`:

```
sirepo_flyer = sf.SirepoFlyer(sim_id='<sim_id>', server_name='<URL>',
                              root_dir='<some_dir>', params_to_change=params_to_change,
                              watch_name='W60')

RE(bp.fly([sirepo_flyer]))
```

The flyer creates 5 copies of the simulation on the Sirepo server and runs them all in parallel — one simulation per a group of updated parameters. This feature allows for the N-dimensional optimization.

Data access is done as follows, utilizing the stream `sirepo_flyer` (generated by the flyer), resulting in the table with the following data/metadata:

```
In[13]: hdr = db[-1]
   ...: hdr.table(stream_name='sirepo_flyer')

Out[13]:
                              time                    sirepo_flyer_image  \
seq_num
1       2020-08-10 07:54:01.426501  ae51b7d7-1a0f-4613-9118-1626b4f89bf0
2       2020-08-10 07:54:01.426501  14183b1a-03f1-4333-a4a2-b9e16ccdbf29
3       2020-08-10 07:54:01.426501  2e372fb4-7fe3-47ce-acf8-9af3e2d1acad
4       2020-08-10 07:54:01.426501  7bea7ace-0be3-4b97-a936-f2cec48cb370
5       2020-08-10 07:54:01.426501  7e22377b-985c-49d9-aaf4-26c967b1bd22

        sirepo_flyer_shape  sirepo_flyer_mean  sirepo_flyer_photon_energy  \
seq_num
```

```
1                [250, 896]       3.677965e+13                      4240.0
2                [250, 546]       9.944933e+13                      4240.0
3                [250, 440]       1.492891e+14                      4240.0
4                [252, 308]       2.234285e+14                      4240.0
5                [252, 176]       3.885947e+14                      4240.0

                           sirepo_flyer_horizontal_extent  \
seq_num
1        [-0.0013627376425855513, 0.0013596958174904943]
2         [-0.001015813953488372, 0.0010120930232558139]
3        [-0.0009701657458563539, 0.0009701657458563542]
4        [-0.0008026143790849673, 0.0008026143790849673]
5        [-0.0005374045801526716, 0.0005312977099236639]

                             sirepo_flyer_vertical_extent  \
seq_num
1         [-0.00024950099800399 2, 0.00024750499001996017]
2         [-0.00024950099800399 2, 0.00024750499001996017]
3        [-0.00024650698602794426, 0.0002504990019960079]
4        [-0.00024850299401197 62, 0.00025249500998003984]
5        [-0.00025149700598802393, 0.0002495009980039921]

                  sirepo_flyer_hash_value sirepo_flyer_status  \
seq_num
1        d5d6628d50bd65a329717e8ffb942224           completed
2        d6f8b77048fe6ad48e007cfb776528ad           completed
3        e5f914471d873f156c31815ab705575f           completed
4        bf507c942bb67c7191d16968de6ddd5b           completed
5        1775724d932efa3e0233781465a5a67b           completed

         sirepo_flyer_Aperture_horizontalSize  \
seq_num
1                                         0.1
2                                         0.2
3                                         0.3
4                                         0.4
5                                         0.5

         sirepo_flyer_Aperture_verticalSize  \
seq_num
1                                       1.5
2                                       1.4
3                                       1.3
4                                       1.2
5                                       1.1

         sirepo_flyer_Lens_horizontalFocalLength  \
seq_num
1                                              8
2                                              9
3                                             10
4                                             11
```

```
5                                                    12

          sirepo_flyer_Obstacle_horizontalSize
seq_num
1                                                     5
2                                                     4
3                                                     3
4                                                     2
5                                                     1
```

The table contains 5 rows of data collected during the scan and the following columns:

- `seq_num`: the sequence number starting from 1;
- `time`: the time when the specific point's event was recorded;
- `sirepo_flyer_image`: the UUID4 strings each containing a reference to the raw SRW data file corresponding to the intensity distribution at the watchpoint `W60` for each set of parameters;
- `sirepo_flyer_shape`: the shape of each image referred in the `sirepo_flyer_image` field;
- `sirepo_flyer_mean`: the averaged intensities of each image referred in the `sirepo_flyer_image` field (in the units used in SRW, [ph/s/.1%bw/mm^2]);
- `sirepo_flyer_photon_energy`: the corresponding photon energy in each simulation (in the units used in SRW, [eV]);
- `sirepo_flyer_horizontal_extent` and `sirepo_flyer_vertical_extent`: horizontal and vertical extents of each image referred in the `sirepo_flyer_image` field (in the units used in SRW, [m]);
- `sirepo_flyer_hash_value`: the `MD5` hash sum corresponding to the raw SRW data files referred in the `sirepo_flyer_image` field, which is used to confirm the uniqueness and integrity of each file;
- `sirepo_flyer_status`: the status of each simulation in Sirepo;
- `sirepo_flyer_Aperture_horizontalSize` and `sirepo_flyer_Aperture_verticalSize` (input parameters): the horizontal and vertical sizes of the `Aperture` optical element in the corresponding Sirepo simulation (in the units used in Sirepo, [mm]);
- `sirepo_flyer_Lens_horizontalFocalLength` (input parameter): the horizontal focal length of the `Lens` optical element in the corresponding Sirepo simulation (in the units used in Sirepo, [m]);
- `sirepo_flyer_Obstacle_horizontalSize` (input parameter): the horizontal size of the `Obstacle` optical element in the corresponding Sirepo simulation (in the units used in Sirepo, [mm]).

The shape and the extents are changed for each simulation due to the automatic sampling method used in the initial wavefront input parameters in the predefined Sirepo example.

4. OPTIMIZATION APPROACH AND RESULTS

The 2020 spring internship SULI project at the NSLS-II TES beamline extended the capabilities of the framework (including a concept of *bluesky* flyers) and developed a flexible optimizer for beam intensity based on Differential Evolution (DE). This optimizer operated on three backends: the Tender Energy X-ray Absorption Spectroscopy (TES) beamline at the NSLS-II,[11] a set of simulated EPICS motors provided via a Docker container,[12] and the Synchrotron Radiation Workshop simulations via the Sirepo framework (see Fig. 7 for the TES virtual beamline representation in Sirepo).

4.1 Optimization Methods

4.1.1 Differential Evolution

Differential Evolution (DE) is a population-based type of Evolutionary Algorithm (EA).[13,14] Some aspects of DE include genes, individuals, and the population. Genes are possible values of a single variable that are used in the objective function to calculate or measure fitness. Individuals are the set of genes that represent a possible solution for each variable to be optimized. The population is the set of individuals that make up the set of possible solutions to check. The objective function is the function or value to be optimized, and the evaluation of the objective function for a single individual can be called the fitness of that individual.[13,15] DE implements four genetic operators to create and choose the next generation's population: *Evaluate*, *Mutate*, *Crossover*, and *Select*.

Evaluate determines the fitness of each individual in the objective function.

Mutate creates a set, which is as large as the population, of mutant trial individuals. There are a number of ways to create the trial individuals, but currently only two mutation strategies are implemented (see Subsection 4.2.4).

Crossover goes through pairs of individuals, one from the original population and the other from the trial population, and exchanges genes based on a specified probability. The probability of crossover, P_c, is a user-defined value passed into the algorithm. Another random value is generated for each gene, and if the generated random value is less than P_c, the mutated gene is used. Otherwise, the original gene is used.

Select compares the mutated crossover individuals to the original individuals and chooses the individuals with the best fitness for the new population.[14]

4.1.2 OMEA

To improve the efficiency of the DE algorithm when optimizing a beamline, Observer Mode for Evolutionary Algorithm (OMEA) is implemented. The idea of OMEA is that as motors are moving to their next set of positions, the value of the objective function, which is beam intensity in this case, is monitored. If a better fitness is discovered, the set of positions resulting in the higher fitness value replaces the next individual in the population.[15] This method allows better positions to be found and utilized where they may not have been discovered without OMEA.

4.2 Optimization Functions

The beamline DE optimization utilizes Bluesky flyers. The `optimize` function is a *bluesky* plan that performs the beamline optimization. The DE optimization procedure is depicted in Fig. 6, and details about each block are provided in the following subsections.

4.2.1 Initialize Population

The first step of the optimization is to create the initial population. The first individual is made up of all the positions of the motors to be optimized. The rest of the initial population is randomized based on the bounds of each motor that will be moved.

4.2.2 Evaluate Population

This step is where the OMEA method is utilized using two functions. The first, `run_hardware_fly`, moves the motors while reading the resulting beam intensity, and the second, `omea_evaluation`, accesses and analyzes the collected data.

The `run_hardware_fly` function handles creating flyers for each individual in the population and running the *bluesky* `fly` plan on each flyer, one at a time. Flyers are used instead of a regular motor scan because flyers can collect data about the positions and the resulting beam intensity at a higher frequency, and it is important to have as much data as possible when using the OMEA method.

The `omea_evaluation` function gets the data collected from the `fly` scans and does some evaluation and replacement of individuals, if necessary. A single flyer moves from one individual to another, so each record

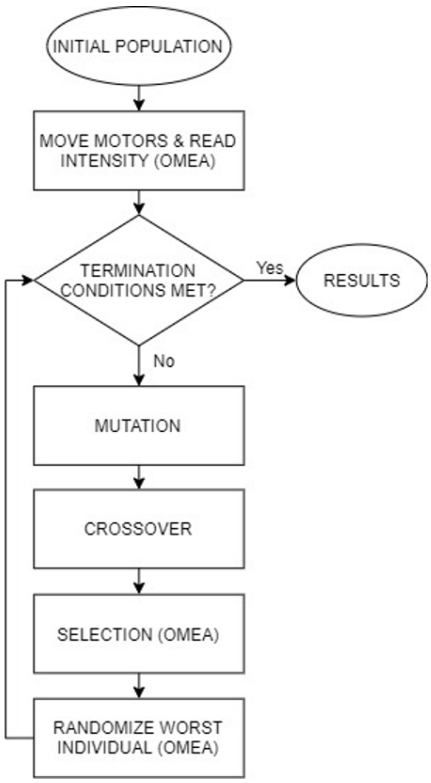

Figure 6. Flow chart of beamline optimization using DE.

table from the `fly` scans contains values measured as motors moved and the target individual to move to. The maximum fitness of the non-population positions in each record is found and compared to the target individual. If the target individual's fitness is less than the maximum fitness of the intermediate values, the target individual is replaced by the set of positions that resulted in the maximum fitness of the intermediate values. This is repeated for every `fly` scan record for the population, and the new population and fitness values based on the best individuals are returned to the primary optimize function.

4.2.3 Termination conditions

After the evaluation of the population, termination conditions are checked. Optimization will stop if the best fitness value is above a certain threshold and the best fitness remains the same for five generations in a row, or if the maximum number of iterations is reached. If the conditions are met, the best individual and fitness are output with a plot of convergence to the current solution. If not, the optimization continues.

4.2.4 Mutation

The current population is mutated in this step. There are two mutation strategies that are currently implemented: *DE/rand/1/bin* and *DE/best/1/bin*.

DE/rand/1/bin: this creates a trial individual by selecting three random individuals separate from the current individual from the population, taking the difference of two of the individuals, multiplying the difference by a mutation factor, and adding the final individual to the product.[13,16]

DE/best/1/bin: this follows the same idea of *DE/rand/1/bin* with a couple of differences. Two random individuals are selected to create the difference vector, and the difference vector is multiplied by the mutation factor. Then, the product is added to the individual with the best fitness to create the trial individual, instead of another randomly selected individual.[13]

4.2.5 Crossover

The mutated population is next combined with the original population by crossover, where each individual from the population and the corresponding trial individual exchange genes with a certain probability. A random number is generated, and if the random number is less than the probability of crossover, P_c, the trial gene is used. If not, the original population gene is used. This creates a new population of mutated crossover individuals.

4.2.6 Selection using OMEA

The mutated crossover individuals are compared with the original population, and the best individuals are chosen to form the next generation. This function follows the same OMEA evaluation method as described previously to evaluate the entire trial population. Afterwards, the original population is compared to the trial individuals, and the individual with the better fitness is used in the new generation.

4.2.7 Worst individual randomization using OMEA

After the new population is formed, the worst individual of the population is randomized and evaluated using OMEA. Once the individual has been created and its fitness measured, the optimization plan checks termination conditions and repeats from there.

4.3 Examples of different backends for the optimization code

The DE optimization code was applied to three different backends:

- the Sirepo/Synchrotron Radiation Workshop simulations;
- the Tender Energy X-ray Absorption Spectroscopy (TES) beamline of the NSLS-II;
- a set of simulated motors provided via a Docker container.

The code itself was generalized as much as possible to be able to use the same plan for all three backends. The code was added to the set of IPython startup scripts for each backend. The scripts set up custom plans and either the hardware to be used, or the environment for the Sirepo simulations.

In addition to required parameters for each backend optimization, the DE parameters can also be modified. The population size, crossover probability, mutation factor, mutation strategy, threshold value, and maximum number of iterations can be specified or left as defaults.

4.3.1 Optimization of the NSLS-II TES "virtual" beamline in Sirepo

An example of the "Beamline" page for the NSLS-II TES virtual beamline representation in Sirepo is depicted in Fig. 7. The link in the caption allows readers to access a copy of that simulation in Sirepo after authentication.

In this version of the optimization code, found in the GitHub repository `https://github.com/NSLS-II-TES/profile_sirepo`, instead of moving motors and reading detectors, simulation parameters are changed and then the simulations are run to get an averaged intensity value that acts as beam intensity. Since there is no hardware to move or read, the flyer for the Sirepo simulations acts a bit differently from the hardware flyer. A `fly` scan can consist of a single flyer that performs multiple simulations at once or a group of flyers that each perform one simulation. The `fly` scan can utilize multiprocessing in either case to speed up execution.

Since Sirepo simulations take time to run, the multiprocessing was necessary to complete optimization in a more reasonable time frame. Parallel execution of the simulations decreases the amount of time needed to complete the optimization. Using a powerful server to run the simulations also speeds up the optimization by allowing a large number of simulations to be run simultaneously. For example, a standalone server at NSLS-II equipped with Intel Xeon CPU E5-2699 v4 processors (operating at the 2.2 GHz frequency) contains 44 physical cores (88 logical cores after hyper-threading), allowing to run simulations with up to 88 parallel processes with a reasonable efficiency. Running a simulation can take anywhere from a few seconds for simulations with fewer optical elements to a few minutes for larger single-electron simulations.

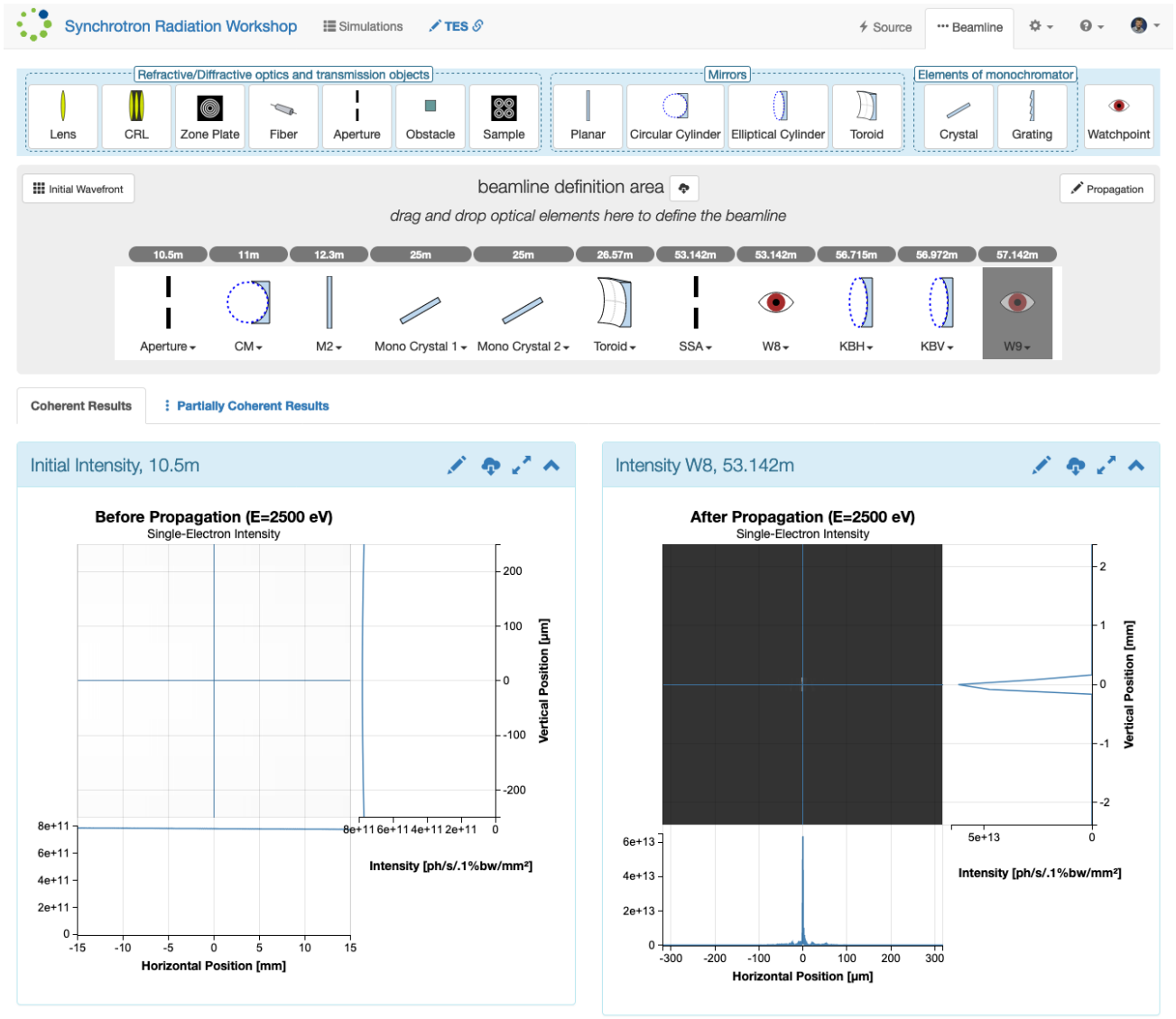

Figure 7. NSLS-II TES virtual beamline representation in Sirepo (URL: https://www.sirepo.com/srw#/beamline/yuL5LMmD).

A Sirepo model of the TES beamline was created and optimized using the enhanced Differential Evolution algorithm. The toroidal mirror's tangential radius and grazing angle and the circular cylinder mirror's bending radius were selected as the parameters to optimize. The watchpoint `W8` served as a detector. The grazing angle was optimized between 5 and 10 mrad (the default value of 7 mrad in the `yuL5LMmD` Sirepo simulation), and the tangential and bending radii were both optimized between 1,000 and 5,000 m (the default values in Sirepo are 1,000 m and 2,700 m respectively). The population size was five individuals, and one individual was checked between each individual in the population to simulate the OMEA method of looking for better individuals in between the original population.

After the optimization, the grazing angle reached an optimal position of 7 mrad, which matched the default value, while the tangential and bending radii were found to change from their default values to the optimal values of 3,188 m and 3,535 m respectively, resulting in a few orders of magnitude increase of the averaged intensity. The final fitness was $\sim 2\times10^{13}$ ph/s/.1%bw/mm^2 and was achieved in 21 generations over approximately one

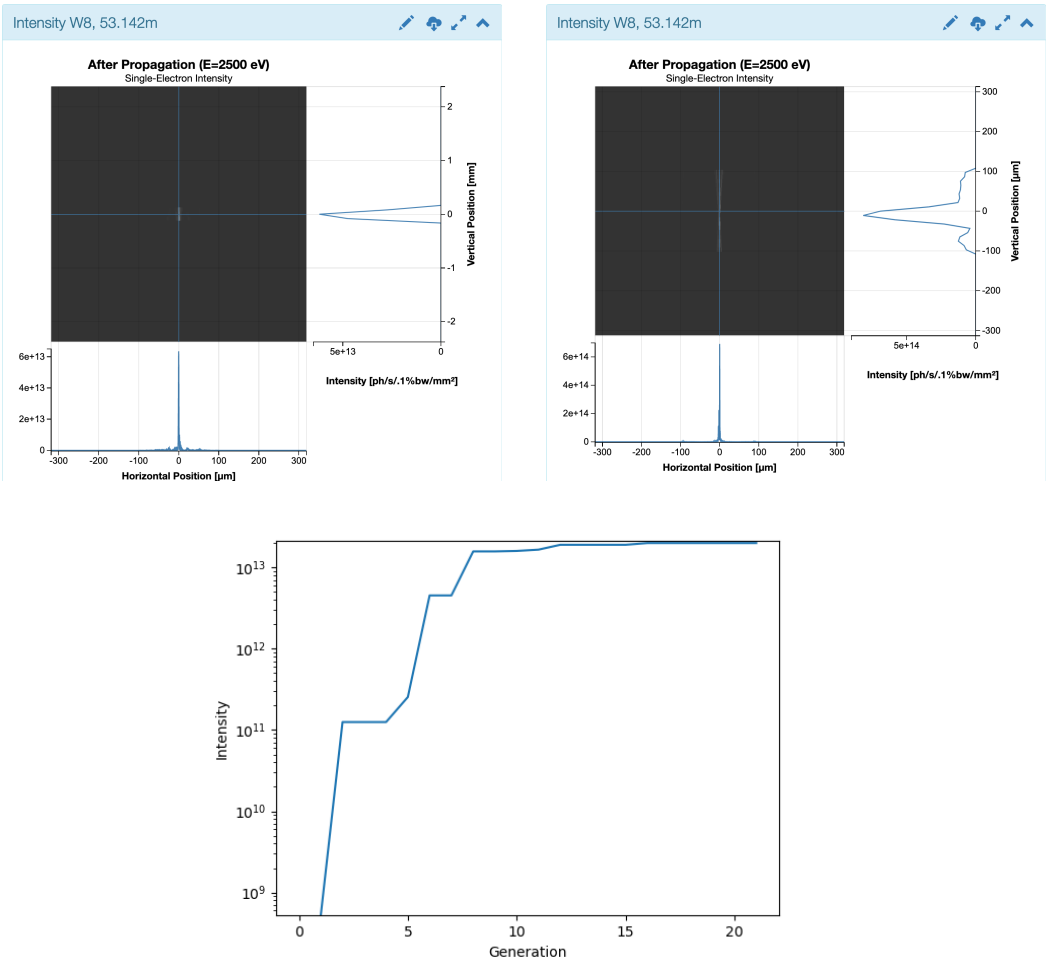

Figure 8. The original (top left) and the optimized (top right) intensity distribution plots from Sirepo, and the convergence plot of the 3-dimensional TES Sirepo optimization (bottom)

and a half hours. Figure 8 displays the original *vs.* optimized intensity distribution plots from Sirepo and the fitness convergence plot.

4.3.2 Optimization of the TES beamline

To optimize the TES beamline, the optimization plan requires information about the motors to be optimized and their bounds, the detector to use, and the type of `fly` plan to use. The motors and detectors are taken from the standard "collection" IPython startup files from the GitHub repository `https://github.com/NSLS-II-TES/profile_collection`, which is used for daily operations of the beamline. The motors and the bounds are two separate Python dictionaries, where the motor dictionary maps the names of the motors to the respective motor objects, and the bound dictionary describes the bounds of the search space for each motor. The detector is a predefined *ophyd* object in the startup scripts, and the `fly` plan is a plan designed for either hardware flyers or for Sirepo flyers.

The optimization plan for the hardware flyers was similar to the one explained in the Subsection 4.3.1, but with different *ophyd* objects used in the flyers. The optimization aimed to find the highest intensity produced by a small sample in the 3D-space, surveyed by the `x`, `y`, and `z` axis motors moving the sample stage at the TES beamline. The Xspress3 detector was used as a point detector producing a scalar signal value at each (x, y, z) point in space.

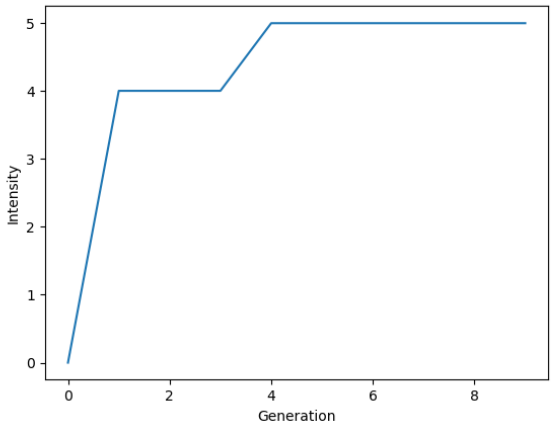

Figure 9. Convergence of the 3-dimensional simulated motor optimization.

The most recent version of the optimization code has been tested on the beamline to ensure that the motors moved correctly and that the detector was being started, read, and stopped when expected. Testing of the code with the X-ray beam is planned in the near future, and the results will be reported in a separate publication.

4.3.3 Optimization of a set of simulated motors using Docker

Simulated motors were used to test code when TES beam time was unavailable. The corresponding code is located in the GitHub repository `https://github.com/NSLS-II-TES/profile_simulated_hardware`. DE optimization using the Docker-run simulated motors works in a very similar fashion to optimization of the TES beamline. Both the real motors and the simulated motors are operated using the EPICS control system protocol, so the motors act very similarly and can be easily swapped. The only difference between the simulated motor optimization and the TES optimization cases is the detector. The simulated hardware case did not have a detector to read, so instead we used a computed signal satisfying the Gaussian distribution based on motor positions to simulate detector intensity.

Three simulated motors were optimized in a search space of ± 5 units from the initial motor positions. A set of optimal positions was discovered in 4 generations. Figure 9 shows the convergence of the fitness.

5. CONCLUSIONS

We demonstrated the flexibility of the combination of two frameworks — Bluesky and Sirepo. The Sirepo backend can be used in the same fashion as the real hardware on beamlines, which was confirmed by the three provided examples utilizing the same optimization code. The same approach will be used for the future developments of multi-dimensional optimization based on Artificial Intelligence and Machine Learning methods (AI/ML) as it can provide large training datasets for such methods.

ACKNOWLEDGMENTS

This project was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internships Program (SULI). This research used the Tender Energy X-ray Absorption Spectroscopy (TES) beamline (8-BM) of the National Synchrotron Light Source II, a U.S. Department of Energy (DOE) Office of Science User Facility operated for the DOE Office of Science by Brookhaven National Laboratory under Contract No. DE-SC0012704.

REFERENCES

[1] Rakitin, M. S., Moeller, P., Nagler, R., Nash, B., Bruhwiler, D. L., Smalyuk, D., Zhernenkov, M., and Chubar, O., "Sirepo: an open-source cloud-based software interface for X-ray source and optics simulations," *Journal of Synchrotron Radiation* **25**, 1877–1892 (Nov 2018).

[2] Chubar, O. and Elleaume, P., "Accurate and efficient computation of synchrotron radiation in the near field region," *Conf. Proc.* , 1177–1179 (1998).

[3] Allan, D., Caswell, T., Campbell, S., and Rakitin, M., "Bluesky's Ahead: A Multi-Facility Collaboration for an a la Carte Software Project for Data Acquisition and Management," *Synchrotron Radiation News* **32**(3), 19–22 (2019).

[4] "Bluesky Project, https://blueskyproject.io."

[5] "Sirepo-Bluesky repository, https://github.com/NSLS-II/sirepo-bluesky."

[6] Sanchez del Rio, M., Canestrari, N., Jiang, F., and Cerrina, F., "*SHADOW3*: a new version of the synchrotron X-ray optics modelling package," *Journal of Synchrotron Radiation* **18**, 708–716 (Sep 2011).

[7] "Wikipedia: Universally unique identifier, https://en.wikipedia.org/wiki/Universally_unique_identifier."

[8] Pérez, F. and Granger, B. E., "IPython: a system for interactive scientific computing," *Computing in Science and Engineering* **9**, 21–29 (May 2007).

[9] Hunter, J. D., "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering* **9**(3), 90–95 (2007).

[10] "Bluesky Documentation: Asynchronous Acquisition, https://blueskyproject.io/bluesky/async.html."

[11] Northrup, P., "The TES beamline (8-BM) at NSLS-II: tender-energy spatially resolved X-ray absorption spectroscopy and X-ray fluorescence imaging," *Journal of Synchrotron Radiation* **26**, 2064–2074 (Nov 2019).

[12] "EuropeanSpallationSource, MCAG_setupMotionDemo as a Docker image, https://github.com/EuropeanSpallationSource/MCAG_setupMotionDemo."

[13] Feoktistov, V., [*Differential Evolution*], Springer US (2006).

[14] Price, K., Storn, R. M., and Lampinen, J. A., [*Differential Evolution*], Springer-Verlag (2005).

[15] Xi, S., Borgna, L. S., and Du, Y., "General method for automatic on-line beamline optimization based on genetic algorithm," *Journal of Synchrotron Radiation* **22**, 661–665 (May 2015).

[16] Storn, R. and Price, K., "Differential Evolution — A Simple and Efficient Heuristic for global Optimization over Continuous Spaces," *Journal of Global Optimization* **11**(4), 341–359 (1997).