Supporting Information - Closing the loop: Autonomous experiments enabled by machine-learning-based online data analysis in synchrotron beamline environments

Linus Pithan,[a]* Vladimir Starostin,[a] David Marecek,[b] Lukas Petersdorf,[c] Constantin Völter,[a] Valentin Munteanu,[a] Maciej Jankowski,[d] Oleg Konovalov,[d] Alexander Gerlach,[a] Alexander Hinderhofer,[a] Bridget Murphy,[c] Stefan Kowarik[b]* and Frank Schreiber [a]*

[a]Universität Tübingen, Inst. für Angewandte Physik, Auf der Morgenstelle 10, 72076 Tübingen, Germany, [b]Universität Graz, Physikalische und Theoretische Chemie, Heinrichstraße 28, 8010 Graz, Austria , [c]Universität Kiel, Inst. für Experimentelle und Angewandte Physik, Leibnizstraße 19, 24118 Kiel, Germany , and [d]ESRF, The European Synchrotron, 71 avenue des Martyrs, CS 40220, 38043 Grenoble Cedex 9, France . E-mail: linus.pithan@uni-tuebingen.de, stefan.kowarik@uni-graz.at, frank.schreiber]@uni-tuebingen.de

machine learning; reflectometry; xrr; nr; synchrotron; autonomous experiments; closed loop control; beamline control

At the time of the experiment BLISS 1.10 and pytango 9.3 were available on the ID10 beamline.

SI-1. Tango device servers wrapping around ML Models

A tango device server with specific API to wrap around the ML models developed based on pytorch and tensorflow used in the context of this study.

```
from typing import Iterable
import numpy as np
from pprint import pprint

from tango.server import Device
from tango.server import run
from tango.server import attribute, command, device_property
from tango import AttrWriteType

# python code wrapping around pytorch model
# available on request or in follow-up publication
from reflectorch.inference import InferenceModel


def energy_to_wavelength(energy: float):
    """Conversion from photon energy (eV) to photon wavelength (
    angstroms)"""
    return 1.2398 / energy * 1e4


class TorchEvaluator(Device):
    preprocess_attributes = {
        "beam_width": np.float64,
        "sample_length": np.float64,
        "energy_keV": np.float64,
    }

    input_attributes = {
        "tth": [np.float64],
        "intensity": [np.float64],
        "transm": [np.float64],
        "priors": [[np.float64]],
        "count_time": [np.float64],
    }
    prediction_attributes = {
        "q": [np.float64],
        "q_interp": [np.float64],
        "refl": [np.float64],
        "refl_interp": [np.float64],
        "refl_predicted": [np.float64],
        "refl_predicted_polished": [np.float64],
        "parameters": [np.float64],
        "parameters_polished": [np.float64],
        "parameter_names": [str],
```

```
44          "sld_profile": [np.float64],
45          "sld_x_axis": [np.float64],
46          "sld_profile_polished": [np.float64],
47      }
48
49      def __init__(self, *args, **kwargs):
50          Device.__init__(self, *args, **kwargs)
51
52          self._prefix_dict = dict()
53
54          self._input_data = self._initialize_dynamic_attributes(
55              "input", self.input_attributes
56          )
57          self._prediction_output = self._initialize_dynamic_attributes(
58              "prediction", self.prediction_attributes
59          )
60          self._preprocess_data = self._initialize_dynamic_attributes(
61              "preprocess", self.preprocess_attributes
62          )
63          self.inference_model = InferenceModel("l2q256_exp_3")
64
65      @property
66      def prediction_output(self):
67          return self._prediction_output
68
69      @command(dtype_out=int)
70      def predict(self) -> dict:
71          """
72          blocking command that runs pytorch model inference
73          """
74          intensity = self._input_data["intensity"]
75          scattering_angle = self._input_data["tth"]
76          attenuation = self._input_data["transm"]
77          priors = self._input_data["priors"]
78
79          preprocess_params = {
80              "wavelength": energy_to_wavelength(
81                  self._preprocess_data["energy_keV"] * 1000
82              ),
83              "beam_width": self._preprocess_data["beam_width"],
84              "sample_length": self._preprocess_data["sample_length"],
85          }
86
87          self.inference_model.set_preprocessing_parameters(**
    preprocess_params)
88
89          res_dict = self.inference_model.predict(
90              intensity, scattering_angle, attenuation, priors,
91          )
92
93          pprint(res_dict)
94
95          self.prediction_output["q"] = res_dict["q_values"]
96          self.prediction_output["q_interp"] = res_dict["q_interp"]
97          self.prediction_output["refl"] = res_dict["curve"]
98          self.prediction_output["refl_interp"] = res_dict["curve_interp"
```

```
    ]
99          self.prediction_output["refl_predicted"] = res_dict["
    curve_predicted"]
100         self.prediction_output["parameters"] = res_dict["params"]
101         self.prediction_output["sld_profile"] = res_dict["sld_profile"]
102         self.prediction_output["sld_x_axis"] = res_dict["sld_x_axis"]
103         self.prediction_output["parameter_names"] = res_dict["
    param_names"]
104
105         self.prediction_output["refl_predicted_polished"] = res_dict["
    curve_polished"]
106         self.prediction_output["parameters_polished"] = res_dict["
    params_polished"]
107         self.prediction_output["sld_profile_polished"] = res_dict[
108             "sld_profile_polished"
109         ]
110
111         return 0
112
113     def _initialize_dynamic_attributes(self, prefix, attributes):
114         """
115         boiler plate code to achieve generic handling of data arrays
    and
116         parameters that have to be passed into the tango server and
    back
117         to the client
118         """
119
120         if prefix in self._prefix_dict:
121             att_dict = self._prefix_dict[prefix]
122         else:
123             att_dict = dict()
124
125         for k, v in attributes.items():
126             att_type = None
127
128             if isinstance(v, tuple):
129                 att_type = v[0]
130                 att_dict[k] = v[1]
131             elif isinstance(v, Iterable):
132                 att_type = v
133                 att_dict[k] = []
134             else:
135                 att_type = v
136                 att_dict[k] = None
137
138             attr = attribute(
139                 name=prefix + "_" + k,
140                 dtype=att_type,
141                 access=AttrWriteType.READ_WRITE,
142                 fget=self._generic_read,
143                 fset=self._generic_write,
144                 max_dim_x=256,  # self.N_MAX_X,
145                 max_dim_y=256,  # self.N_MAX_Y,
146                 # fisallowed=self.generic_is_allowed,
147             )
```

```
            self.add_attribute(attr)

        self._prefix_dict[prefix] = att_dict
        return self._prefix_dict[prefix]

    def _generic_read(self, attr):
        prefix, name = attr.get_name().split("_", 1)
        value = self._prefix_dict[prefix][name]
        # unlike a normal static attribute read, one has to modify the value
        # inside this attr object, rather than just returning the value
        attr.set_value(value)

    def _generic_write(self, attr):
        prefix, name = attr.get_name().split("_", 1)
        self._prefix_dict[prefix][name] = attr.get_write_value()


if __name__ == "__main__":
    run((TorchEvaluator,))
```

SI-2. Beamline / BLISS integration to run ML inference on remote tango device server and publish prediction results together with raw data

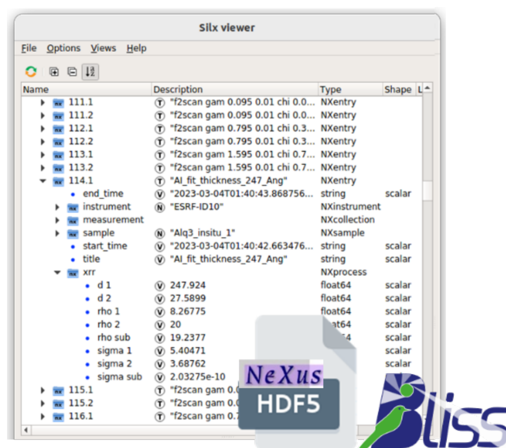

Fig. SI-1.: NeXus compliant data file generated by the beamline control system including results of the online data analysis.

Script that can be loaded as BLISS user macro into the beamline software environment without any additional external dependencies. The function predict_and_save takes a bliss scan object as input, runs ML inference and processes the results so that they are saved together with the raw data in a coherent fashion.

References:

https://bliss.gitlab-pages.esrf.fr/bliss/1.10.x/scan_group.html#sequences (accessed: 05.2023)

https://bliss.gitlab-pages.esrf.fr/bliss/1.10.x/data/data_scan_metadata.html (accessed: 05.2023)

```
import numpy as np
from dataclasses import dataclass

from bliss.common import tango
from bliss import setup_globals
from bliss.scanning.group import Sequence
from bliss.scanning.chain import AcquisitionChannel
from bliss.scanning.scan_meta import USER_SCAN_META

# Add XRR Metadata entry to BLISS
if "xrr" not in USER_SCAN_META.categories_names():
    USER_SCAN_META.add_categories({"xrr"})
    USER_SCAN_META.xrr.set("xrr", {"@NX_class": "NXprocess"})


@dataclass
class Priors:
    d_top: tuple = (0.0, 1000.0)
    d_sio2: tuple = (10.0, 30.0)

    sigma_top: tuple = (0.0, 50.0)
    sigma_sio2: tuple = (
        0.0,
        10.0,
    )
    sigma_si: tuple = (
        0.0,
        2.0,
    )

    rho_top: tuple = (7.0, 14.0)
    rho_sio2: tuple = (19, 23)
    rho_si: tuple = (19, 21)
    max_d_change: float = 5.0
    fit_growth: bool = True

    def to_arr(self):
        return np.array(
            [
                self.d_top,
                self.d_sio2,
                self.sigma_top,
                self.sigma_sio2,
                self.sigma_si,
```

```
45                 self.rho_top,
46                 self.rho_sio2,
47                 self.rho_si,
48             ]
49         )
50
51
52 @dataclass
53 class PreprocessParams:
54     energy_keV: float = 17.0
55     beam_width: float = 0.03
56     sample_length: float = 9.0
57
58
59 def predict_and_save(
60     scans, priors=None, preprocess: PreprocessParams = None,
61 ):
62     """
63     takes a list of scans that will be concatenated to a single XRR
    profile,
64     preforms basic background correction, pushes data via TANGO into
    the ML model,
65     triggers inference in the TANGO device, retrieves prediction
    results and
66     publishes results via BLISS to be saved together with original
    scans in
67     the NeXus hdf5 file produced by the BLISS NeXusWriter
68     """
69
70     # specify the tango uri of the device server embedding the ML-Model
71     tango_uri_or_proxy = ("//VISA-HOST.esrf.fr:PORT/PATH/TO/TANGO-ML-
    DEVICE",)
72
73     # specify counters that should be used to collect the data from the
     scan object
74     TTH = setup_globals.gam  # gam: angle of detector axis
75     CROI = "pilatus300k:roi_counters:roi2_sum"  # ROI with specular
    signal
76     BGROI1 = (
77         "pilatus300k:roi_counters:roi3_sum"  # ROI left of CROI (half
    the size of CROI)
78     )
79     BGROI2 = (
80         "pilatus300k:roi_counters:roi4_sum"  # ROI right of CROI (half
    the size of CROI)
81     )
82     TRANSM = "autof_eh1:transm"  # transmission of Autoabsorber
83
84     # extract detector angle from scans
85     tth_data = np.concatenate(np.array([x.get_data(TTH) for x in scans
    ]))
86
87     # extract transmission of attenuators from scans
88     transm_data = np.concatenate(np.array([x.get_data(TRANSM) for x in
    scans]))
89
```

```
90      # extract measured intensities from scans and perform basic
     background correction
91      intens_data = np.concatenate(
92          np.array(
93              [x.get_data(CROI) - x.get_data(BGROI1) - x.get_data(BGROI2)
      for x in scans]
94          )
95      )
96
97      # establish communication with tango device that is linked to ML
     model
98      ml_device = tango.DeviceProxy(tango_uri_or_proxy)
99
100     # deal with priors and preprocessing parameters
101     priors = priors or Priors()
102     preprocess = preprocess or PreprocessParams()
103
104     # push data to tango device
105     ml_device.input_tth = tth_data
106     ml_device.input_transm = transm_data
107     ml_device.input_intensity = np.clip(intens_data, 0, np.infty)
108     ml_device.input_count_time = [0]
109     ml_device.input_priors = priors.to_arr()
110     ml_device.postprocess_max_d_change = priors.max_d_change
111     ml_device.postprocess_fit_growth = priors.fit_growth
112     ml_device.preprocess_beam_width = preprocess.beam_width
113     ml_device.preprocess_sample_length = preprocess.sample_length
114     ml_device.preprocess_energy_keV = preprocess.energy_keV
115
116     # run inference (here built to run synchronous in blocking fashion)
117     ml_device.set_timeout_millis(5000)
118     ml_device.predict()
119
120     # retrieve predicted parameters
121     # polished parameters refers to results of subsequent LMS fit after
      inference
122     polished_params = ml_device.prediction_parameters_polished
123     param_names = ml_device.prediction_parameter_names
124
125     # create summary metadata structure that is to be published via
     BLISS
126     info = {"xrr": dict(zip(param_names, polished_params))}
127
128     # creates a new entry in the hdf5 file generated by BLISS and redis
      DB for online access
129     seq = Sequence(
130         title=f"AI_fit_thickness_{int(polished_params[0])}_Ang",
     scan_info=info
131     )
132
133     # specify additional data entries that should appear under "
     measurement" in the h5 file
134     seq.add_custom_channel(AcquisitionChannel("raw_tth", float, ()))
135     seq.add_custom_channel(AcquisitionChannel("corrected_intensity",
     float, ()))
136     seq.add_custom_channel(AcquisitionChannel("raw_q", float, ()))
```

```
137     seq.add_custom_channel(AcquisitionChannel("predicted_intensity",
    float, ()))
138     seq.add_custom_channel(AcquisitionChannel("predicted_q", float, ())
    )
139     seq.add_custom_channel(AcquisitionChannel("params", float, ()))
140     seq.add_custom_channel(AcquisitionChannel("params_names", str, ()))
141     seq.add_custom_channel(AcquisitionChannel("polished_intensity",
    float, ()))
142     seq.add_custom_channel(AcquisitionChannel("polished_params", float,
     ()))
143
144     # add references to the raw data scans as well as process data and
    analysis results
145     with seq.sequence_context() as scan_seq:
146         for scan in scans:
147             scan_seq.add(scan)
148
149         seq.custom_channels["raw_tth"].emit(tth_data)
150         seq.custom_channels["corrected_intensity"].emit(ml_device.
    prediction_refl)
151         seq.custom_channels["raw_q"].emit(ml_device.prediction_q)
152         seq.custom_channels["predicted_intensity"].emit(
153             ml_device.prediction_refl_predicted
154         )
155         seq.custom_channels["params"].emit(ml_device.
    prediction_parameters)
156         seq.custom_channels["params_names"].emit(ml_device.
    prediction_parameter_names)
157         seq.custom_channels["polished_intensity"].emit(
158             ml_device.prediction_refl_predicted_polished
159         )
160         seq.custom_channels["polished_params"].emit(polished_params)
161
162     # return polished_params e.g. for closed loop feedback
163     return polished_params
```

SI-3. Asynchronous BLISS integration to receive events via the BLISS data streaming API

The code snippet below represents basically a consumer of events emitted via the event based data streaming api of BLISS and publishes produced results using a beacon channel.

References: `https://bliss.gitlab-pages.esrf.fr/bliss/1.10.x/data/data_redis_api_low.html` (accessed: 05.2023)

`https://bliss.gitlab-pages.esrf.fr/bliss/1.10.x/beacon_channels.html` (accessed: 05.2023)

```
from bliss.data.node import get_session_node
from bliss.config.channels import Channel

# create BLISS Channel with unique name in the scope of the beamline
# to publish predictions results via beacon / redis
last_film_thickness = Channel("last_film_thickness", default_value=None
    )


def listen_scans_of_session(session):
    """
    script that can run in any bliss process
    (independent oft the process producing the data & controlling the
    hardware)
    takes as input argument the name of the bliss session publishing
    the data
    to the beamline beacon server / redis
    """

    session_node = get_session_node(session)
    print(f"Listening to {session}")

    received_scans = []
    for event_type, node, event_data in session_node.walk_on_new_events
    (
        include_filter=["scan"]
    ):
        # this loop deals with the event based data streaming api of
    BLISS
        #
        if event_type == event_type.NEW_NODE:
            print(f"new scan", node.db_name)

        elif event_type == event_type.END_SCAN:
            print(node.db_name)

            params = node.info.get("fastfit")
            if params.get("xrr_scan_group_first", False):
                received_scans = [node]
                print("add scan")
            if params.get("xrr_scan_group_last", False):
                received_scans.append(node)
                print("add scan and process")
                process_scans(received_scans)


def process_scans(scans):
    """
    Run ML inference
    """

    # missing here: code similar to synchronous version in example
    above
    # interacting with tango or triton server to retrieve prediction

    # ...
```

```
51
52      thickness = prediction["parameters_polished"][0]
53
54      # publish e.g. film thickness via bliss channel & beacon
55      # to make it available to all bliss processes running on the
        beamline
56      last_film_thickness.value = thickness
57
58      print("######### predicted thickness", thickness)
```

SI-4. Postprocessing gradient-descent fit with optional growth rate parameter.

The functions below are used to perform a postprocessing polishing operation by fitting the measured curve via gradient descent initialized based on AI result. Optionally, it allows fitting the linear change of the thickness of the top layer as a function of q for curves measured *in situ* during growth. Further details may be found in (Munteanu *et al.*, 2023).

References:

Munteanu, V., Starostin, V., Greco, A., Pithan, L., Gerlach, A., Hinderhofer, A., Kowarik, S. Schreiber, F. (2023). Neural network analysis of neutron and X-ray reflectivity data: incorporating prior knowledge for tackling the phase problem.

```
1  import numpy as np
2  from scipy.optimize import minimize, curve_fit
3
4  def abeles_np(
5          q: np.ndarray,
6          thickness: np.ndarray,
7          roughness: np.ndarray,
8          sld: np.ndarray):
9
10     c_dtype = np.complex128 if q.dtype is np.float64 else np.complex64
11
12     if q.ndim == thickness.ndim == roughness.ndim == sld.ndim == 1:
13         zero_batch = True
14     else:
15         zero_batch = False
16
```

```
17     thickness = np.atleast_2d(thickness)
18     roughness = np.atleast_2d(roughness)
19     sld = np.atleast_2d(sld)
20     batch_size, num_layers = thickness.shape
21     sld = np.concatenate([np.zeros((batch_size, 1)).astype(sld.dtype),
      sld], -1)[:, None]
22     thickness = np.concatenate([np.zeros((batch_size, 1)).astype(
      thickness.dtype), thickness], -1)[:, None]
23     roughness = roughness[:, None] ** 2
24 
25     sld = sld * 1e-6 + 1e-30j
26 
27     k_z0 = (q / 2).astype(c_dtype)
28     if len(k_z0.shape) == 1:
29         k_z0 = k_z0[None]
30     if len(k_z0.shape) == 2:
31         k_z0 = k_z0[..., None]
32     k_n = np.sqrt(k_z0 ** 2 - 4 * np.pi * sld)
33     k_n, k_np1 = k_n[..., :-1], k_n[..., 1:]
34 
35     beta = 1j * thickness * k_n
36 
37     exp_beta = np.exp(beta)
38     exp_m_beta = np.exp(-beta)
39     rn = (k_n - k_np1) / (k_n + k_np1) * np.exp(- 2 * k_n * k_np1 *
      roughness)
40 
41     c_matrices = np.stack([
42         np.stack([exp_beta, rn * exp_m_beta], -1),
43         np.stack([rn * exp_beta, exp_m_beta], -1),
44     ], -1)
45     c_matrices = np.moveaxis(c_matrices, -3, 0)
46     m, c_matrices = c_matrices[0], c_matrices[1:]
47     for c in c_matrices:
48         m = m @ c
49 
50     r = np.abs(m[..., 1, 0] / m[..., 0, 0]) ** 2
51     r = np.clip(r, None, 1.)
52 
53     if zero_batch:
54         r = r[0]
55     return r
56 
57 
58 def standard_restore_params(fitted_params) -> dict:
59     num_layers = (fitted_params.size - 2) // 3
60     return dict(
61         thickness=fitted_params[:num_layers],
62         roughness=fitted_params[num_layers:2 * num_layers + 1],
63         sld=fitted_params[2 * num_layers + 1:],
64     )
65 
66 
67 def mse_loss(curve1, curve2):
68     return np.sum((curve1 - curve2) ** 2)
69 
```

```
70
71 def standard_refl_fit(
72         q: np.ndarray, curve: np.ndarray,
73         init_params: np.ndarray,
74         bounds: np.ndarray = None,
75         refl_generator=abeles_np,
76         restore_params_func=standard_restore_params,
77         scale_curve_func=np.log10,
78         **kwargs):
79
80     if bounds is not None:
81         kwargs['bounds'] = bounds
82         init_params = np.clip(init_params, *bounds)
83
84     res = curve_fit(
85         get_scaled_curve_func(
86             refl_generator=refl_generator,
87             restore_params_func=restore_params_func,
88             scale_curve_func=scale_curve_func,
89         ),
90         q, scale_curve_func(curve),
91         p0=init_params, **kwargs
92     )
93
94     curve = refl_generator(q, **restore_params_func(res[0]))
95     return res[0], curve
96
97
98 def get_fit_with_growth(
99         q: np.ndarray, curve: np.ndarray,
100        init_params: np.ndarray,
101        bounds: np.ndarray = None,
102        init_d_change: float = 0.,
103        max_d_change: float = 30.,
104        scale_curve_func=np.log10,
105        **kwargs):
106
107    init_params = np.array(list(init_params) + [init_d_change])
108    if bounds is not None:
109        bounds = np.concatenate([bounds, np.array([0, max_d_change])
    [..., None]], -1)
110
111    params, curve = standard_refl_fit(
112        q, curve, init_params, bounds, refl_generator=
    refl_with_changing_params,
113        restore_params_func=get_restore_params_with_growth_func(q_size=
    q.size, d_idx=0),
114        scale_curve_func=scale_curve_func, **kwargs
115    )
116    params[0] += params[-1] / 2
117    return params, curve
118
119
120 def fit_refl_curve(q: np.ndarray, curve: np.ndarray,
121                    init_params: np.ndarray,
122                    bounds: np.ndarray = None,
```

```
123                     refl_generator=abeles_np,
124                     restore_params_func=standard_restore_params,
125                     scale_curve_func=np.log10,
126                     **kwargs
127                     ) -> np.ndarray
128
129     fitting_func = get_fitting_func(
130         q=q, curve=curve,
131         refl_generator=refl_generator,
132         restore_params_func=restore_params_func,
133         scale_curve_func=scale_curve_func,
134     )
135
136     res = minimize(fitting_func, init_params, bounds=bounds, **kwargs)
137
138     if not res.success:
139         import warnings
140         warnings.warn(f"Minimization did not converge.")
141     return res.x
142
143
144 def get_scaled_curve_func(
145         refl_generator=abeles_np,
146         restore_params_func=standard_restore_params,
147         scale_curve_func=np.log10):
148
149     def scaled_curve_func(q, *fitted_params):
150         fitted_params = restore_params_func(np.asarray(fitted_params))
151         fitted_curve = refl_generator(q, **fitted_params)
152         scaled_curve = scale_curve_func(fitted_curve)
153         return scaled_curve
154
155     return scaled_curve_func
156
157
158 def get_fitting_func(
159         q: np.ndarray,
160         curve: np.ndarray,
161         refl_generator=abeles_np,
162         restore_params_func=standard_restore_params,
163         scale_curve_func=np.log10,
164         loss_func=mse_loss):
165
166     scaled_curve = scale_curve_func(curve)
167
168     def fitting_func(fitted_params):
169         fitted_params = restore_params_func(fitted_params)
170         fitted_curve = refl_generator(q, **fitted_params)
171         loss = loss_func(scale_curve_func(fitted_curve), scaled_curve)
172         return loss
173
174     return fitting_func
175
176
177 def restore_masked_params(fixed_params, fixed_mask):
178     def restore_params(fitted_params) -> dict:
```

```
179         params = np.empty_like(fixed_mask).astype(fitted_params.dtype)
180         params[fixed_mask] = fixed_params
181         params[~fixed_mask] = fitted_params
182         return standard_restore_params(params)
183 
184     return restore_params
185 
186 
187 def base_params2growth(base_params: dict, d_shift: np.ndarray, d_idx:
        int = 0) -> dict:
188     d_init = base_params['thickness'][None]
189     q_size = d_shift.size
190     d = d_init.repeat(q_size, 0)
191     d[:, d_idx] = d[:, d_idx] + d_shift
192 
193     roughness = np.broadcast_to(base_params['roughness'][None], (q_size
        , base_params['roughness'].size))
194     sld = np.broadcast_to(base_params['sld'][None], (q_size,
        base_params['sld'].size))
195 
196     return {
197         'thickness': d,
198         'roughness': roughness,
199         'sld': sld,
200     }
201 
202 
203 def get_restore_params_with_growth_func(q_size: int, d_idx: int = 0):
204     def restore_params_with_growth(fitted_params) -> dict:
205         fitted_params, delta_d = fitted_params[:-1], fitted_params[-1]
206         base_params = standard_restore_params(fitted_params)
207         d_shift = np.linspace(0, delta_d, q_size)
208         return base_params2growth(base_params, d_shift, d_idx)
209 
210     return restore_params_with_growth
211 
212 
213 def refl_with_changing_params(q: np.ndarray, **kwargs):
214     return abeles_np(q[..., None], **kwargs).flatten()
```