MFX Code Repository

User Cheat Sheet & Function Reference

LCLS MFX Beamline

Generated: March 05, 2026

Code Repository Summary

Last updated: 2026-03-05 20:46:47

Table of Contents

DOD

DropsDriver - `DropsDriver.py`

`def parse_arguments(obj)`

`def __init__(self, ip, port, supported_json = 'supported.json', reload = True, queue = None, **kwargs)`

`def __del__(self)`

`def middle_invocation_wrapper(func)`

`def inner(self, *args)`

`def connect(self, user: str)`

`def disconnect(self)`

`def get_status(self)`

`def move(self, position: str)`

`def get_position_names(self)`

`def get_task_names(self)`

`def get_current_positions(self)`

`def execute_task(self, value: str)`

`def auto_drop(self)`

`def move_to_interaction_point(self)`

`def move_x(self, value: float)`

`def get_pulse_names(self)`

`def get_nozzle_status(self)`

`def select_nozzle(self, channel: str)`

`def dispensing(self, state: str)`

`def setLED(self, duration: int, delay: int)`

`def move_y(self, value: float)`

`def move_z(self, value: float)`

`def take_probe(self, channel: int, probe_well: str, volume: float)`

`def get_task_details(self, task_name)`

`def get_drive_range(self)`

`def set_nozzle_parameters(self, active_nozzles: str, selected_nozzles: str, volts: int, pulse: str, frequency: int)`

`def stop_task(self)`

`def set_ip_offest(self)`

`def set_humidity(self, value)`

`def set_cooling_temp(self, temp)`

`def close_dialog(self, reference, selection)`

`def reset_error(self)`

`def send(self, endpoint)`

`def get_response(self)`

HTTPTransceiver - `HTTPTransceiver.py`

`def __init__(self, conn: HTTPConnection, queue: Queue, q_ready: Semaphore)`

`def send(self, endpoint: str)`

`def get_response(self)`

JsonFileHandler - `JsonFileHandler.py`

`def __init__(self, file_name: str)`

`def create_new_supported_file(self)`

`def reload_endpoints(self)`

`def get_endpoint_data(self, endpoint: str)`

`def add_endpoint(self, endpoint: str, payload: str, args = None, comment = None)`

ServerResponse - `ServerResponse.py`

`def __init__(self, httObj: HTTPResponse)`

`def __str__(self)`

SupporEndsHandler - `SupporEndsHandler.py`

`def __init__(self, file: str, conn: HTTPConnection)`

`def get_endpoints(self)`

`def reload_endpoint(self, endpoint: str)`

`def reload_all(self)`

TestJsonFileHandler - `TestJsonFileHandler.py`

`def test_file_load(self, capsys)`

`def test_file_add_endpoint(self, capsys)`

TestResponse - `TestResponse.py`

`def test_get_status(self, capsys)`

codi - `codi.py`

`def __init__(self, reload_presets = False)`

`def get_CoDI_predefined(self)`

`def update_CoDI_predefined(self)`

`def set_CoDI_predefined(self, name, base, left, right, z)`

`def get_CoDI_pos(self, precision_digits = 1)`

`def set_CoDI_pos(self, pos_name, wait = True)`

`def set_CoDI_current_pos(self, name)`

`def set_CoDI_current_z(self, verbose = True)`

`def remove_CoDI_pos(self, name)`

`def move_z_rel(self, z_rel)`

`def move_rot_left_rel(self, rot_rel)`

`def move_rot_right_rel(self, rot_rel)`

`def move_rot_base_rel(self, rot_rel)`

`def move_z_abs(self, z_abs)`

`def move_rot_left_abs(self, rot_abs)`

`def move_rot_right_abs(self, rot_abs)`

`def move_rot_base_abs(self, rot_abs)`

common - `common.py`

`def busy_wait(timeout: int)`

dod - `dod.py`

`def __init__(self, modules = 'None', ip = '172.21.72.187', port = 9999, supported_json = '/cds/group/pcds/pyps/apps/hutch-python/mfx/dod/supported.json')`

`def stop_task(self, verbose = True)`

`def clear_abort(self, verbose = True)`

`def get_status(self, verbose = False)`

`def busy_wait(self, timeout)`

`def get_task_details(self, task_name, verbose = False)`

`def get_task_names(self, verbose = False)`

`def get_current_position(self, verbose = False)`

`def get_nozzle_status(self, verbose = False)`

`def set_nozzle_dispensing(self, mode = 'Off', verbose = False)`

`def do_move(self, position, safety_test = False, verbose = False)`

`def move_x_abs(self, position_x, safety_test = False, verbose = False)`

`def move_y_abs(self, position_y, safety_test = False, verbose = False)`

`def move_z_abs(self, position_z, safety_test = False, verbose = False)`

`def do_task(self, task_name, safety_check = False, verbose = False)`

`def get_forbidden_region(self, rotation_state = 'both')`

`def set_forbidden_region(self, x_start, x_stop, y_start, y_stop, rotation_state = 'both')`

`def test_forbidden_region(self, x_test, y_test)`

`def set_timing_update(self)`

`def set_timing_zero_nozzle(self, nozzle, timing_rel)`

`def set_timing_rel_LED(self, timing_rel)`

`def set_timing_rel_reaction(self, timing_rel)`

`def set_timing_abs_Xray(self, timing_abs)`

`def set_timing_relative_nozzle(self, nozzle, timing_rel)`

`def logging_string(self)`

mfxDOD_old - `mfxDOD_old.py`

`def __init__(self)`

`def set_current_position(self, motor, value)`

`def led_scan(self, start, end, nsteps, duration)`

`def lxt_fast_set_absolute_zero(self)`

`def takeRun(self, nEvents = None, duration = None, record = True, use_l3t = False)`

`def pvascan(self, motor, start, end, nsteps, nEvents, record = None)`

`def pvdscan(self, motor, start, end, nsteps, nEvents, record = None)`

`def listscan(self, motor, posList, nEvents, record = True, use_l3t = False)`

`def cleanup_RE(self)`

`def scanExamples(self)`

`def perform_run(self, events, record = True, comment = '', post = True, **kwargs)`

`def dummy_daq_test(self, events = 360, sleep = 3, record = False)`

`def __init__(self)`

`def setOffset_nano(self, offset)`

`def setOffset_micro(self, offset)`

`def getOffset_nano(self)`

`def __call__(self, micro)`

mfx_dod_old - `mfx_dod_old.py`

`def set_current_position(motor, value)`

`def led_scan(start, end, nsteps, duration)`

`def lxt_fast_set_absolute_zero(self)`

`def takeRun(nEvents = None, duration = None, record = True, use_l3t = False)`

`def pvascan(motor, start, end, nsteps, nEvents, record = None)`

`def pvdscan(motor, start, end, nsteps, nEvents, record = None)`

`def listscan(self, motor, posList, nEvents, record = True, use_l3t = False)`

`def cleanup_RE(self)`

`def scanExamples(self)`

`def perform_run(events, record = True, comment = '', post = True, **kwargs)`

`def dummy_daq_test(events = 360, sleep = 3, record = False)`

`def __init__(self)`

`def setOffset_nano(self, offset)`

`def setOffset_micro(self, offset)`

`def getOffset_nano(self)`

`def __call__(self, micro)`

testServer - `testServer.py`

`def do_GET(self)`

testServerSpawner - `testServerSpawner.py`

`def __init__(self, hostName: str, serverPort: int, supportedJson: str)`

`def launch_web_server(self)`

`def server_launch()`

`def kill_web_server(self)`

MFX

attenuator_scan - `attenuator_scan.py` Attenuator transmission scan utilities for MFX beamline.

`def attenuator_scan(sample: str = '?', tag: str = None, transmissions: list = None, inspire: bool = False, duration: float = 10.0, record: bool = False, use_daq: bool = True, runs: int = 5, daq_delay: int = 5, picker: str = None, daq_num: int = 2)`

`def _setup_daq_lcls2(record)`

`def _cleanup_daq_lcls2()`

`def quick_att_scan(sample, record = True)`

autorun - `autorun.py` Automated data acquisition routines for MFX beamline.

`def quote()`

`def post(sample, tag, run_number, post = True, inspire = False, daq_num = 2, spread = None, add_note = '')`

`def ioc_cam_recorder(cam_name, duration, tag)`

`def _autorun(sample: str = '?', tag: str = None, run_length: float = 60.0, inspire: bool = False, record: bool = False, runs: int = 5, daq_delay: int = 5, picker: str = None, close: bool = True, daq_num: int = 2, cam: str = None, run_type: str = 'data')`

`def _display_progress(run_length)`

`def _autorun_daq1(sample, tag, run_length, inspire, record, runs, daq_delay, cam, run_type)`

`def _autorun_daq2(sample, tag, run_length, inspire, record, runs, daq_delay, close, cam, run_type)`

`def autorun(**kwargs)`

`def geomrun(**kwargs)`

`def quick_run(sample, duration = 60, record = True)`

bash_utilities - `bash_utilities.py` Bash utility wrappers for MFX beamline operations.

`def xfel_gui(self)`

`def takepeds(self, daq_num: int = 2)`

`def makepeds(self, username: str, run_number: Optional[int] = None, onshift: bool = False, daq_num: int = 2, det: str = 'all')`

`def restartdaq(self, daq_num: int = 2)`

`def stopdaq(self, daq_num: int = 2)`

`def lecroy(self, res: str = '2560x1440', num: int = 1)`

`def mirrors(self)`

`def grabber(self)`

`def cleanup_shm(self)`

`def cameras(self, pro = False)`

`def camera_list_out(self)`

`def camera_list(self)`

`def focus_scan(self, camera, record = False, daq_num = 2)`

`def startami(self, ami_num: int = 1, daq_num: int = 1)`

`def coyote_gui(self, cfg: str = 'coyote', debug: bool = False)`

`def xfel_gui()`

`def takepeds(daq_num: int = 2)`

`def makepeds(username: str, run_number: Optional[int] = None, onshift: bool = False, daq_num: int = 2, det: str = 'all')`

`def restartdaq(daq_num: int = 2)`

`def stopdaq(daq_num: int = 2)`

`def lecroy(res: str = '2560x1440')`

`def grabber()`

`def cleanup_shm()`

`def cameras()`

`def camera_list()`

`def focus_scan(camera: str, record: bool = False, daq_num: int = 2)`

`def startami(ami_num: int = 1, daq_num: int = 1)`

beamline - `beamline.py`

`def mfx_reload(module_name)`

cctbx - `cctbx.py` CCTBX (Computational Crystallography Toolbox) integration for MFX beamline.

`def __init__(self, experiment: Optional[str] = None)`

`def sshproxy(self, user: str)`

`def image_viewer(self, user: str, facility: str = 'S3DF', image_type: str = 'avg', exp: Optional[str] = None, run: Optional[int] = None, group: Optional[str] = None, debug: bool = False)`

`def indexing(self, user: str, facility: str = 'S3DF', exp: Optional[str] = None, run: Optional[int] = None, group: str = '001_rg001', debug: bool = False)`

`def merge(self, user: str, facility: str = 'S3DF', exp: Optional[str] = None, group: str = '001_rg001', debug: bool = False)`

`def geom_refine(self, user: str, facility: str = 'S3DF', group: str = '001_rg001', level: Optional[int] = 0, exp: Optional[str] = None)`

`def xfel_gui(self, user: str, facility: str = 'NERSC', exp: str = '', debug: bool = False)`

`def notch_check(self, user, runs = [])`

dccm - `dccm.py` Double Crystal Channel-cut Monochromator (DCCM) control for MFX beamline.

`def forward(self, pseudo_pos: namedtuple)`

`def inverse(self, real_pos: namedtuple)`

`def energyToSi111BraggAngle(self, energy: float)`

`def thetaToSi111energy(self, theta)`

`def __init__(self, prefix: str, hutch = 'MFX', **kwargs)`

`def forward(self, pseudo_pos: namedtuple)`

`def inverse(self, real_pos: namedtuple)`

`def __init__(self, prefix: str, hutch: typing.Optional[str] = None, acr_status_suffix = 'AO805', pv_index = 2, **kwargs)`

`def __init__(self, prefix: str = 'SP1L0:DCCM', **kwargs)`

`def insert(self)`

`def remove(self)`

`def calculate_bragg_angle(energy_kev, d_spacing = default_dspacing)`

`def calculate_energy(angle_deg, d_spacing = default_dspacing)`

debug - `debug.py` Debugging and diagnostic utilities for MFX beamline infrastructure.

`def __init__(self)`

`def awr(self, hutch: str = 'mfx')`

`def motor_check(self)`

`def check_server(self, server: str)`

`def cycle_server(self, server)`

`def check_all_servers(self, server_type)`

`def server_list(self, server_type: str = 'all')`

delay_scan - `delay_scan.py` Laser-X-ray delay scanning for pump-probe experiments at MFX beamline.

`def delay_scan(daq, time_motor, time_points, sweep_time, duration = None, record = None, use_l3t = False, controls = None)`

`def inner_scan()`

`def forward(self, pseudo_pos)`

`def inverse(self, real_pos)`

`def quick_delay_scan(start_ps, end_ps, sweep_time = 2.0, duration = 60, record = True)`

`def calculate_sweep_velocity(start_mm, end_mm, sweep_time)`

`def time_to_position(delay_s)`

`def position_to_time(position_mm)`

devices - `devices.py` Custom device definitions for MFX beamline.

`def tweak(self, distance: int)`

`def voltage_check(self)`

`def _do_move(self, state)`

energy_control - `energy_control.py` Energy control and calibration utilities for MFX beamline.

`def __init__(self)`

`def all(self)`

`def vernier_ref(self)`

`def k_ref(self)`

`def vernier(self)`

`def k_energy(self)`

`def mono(self)`

`def __init__(self)`

`def all(self, energy, crystal_angle_offset = 0.0)`

`def vernier_ref(self, energy)`

`def k_ref(self, energy)`

`def vernier(self, energy)`

`def k_energy(self, energy)`

`def mono(self, energy)`

`def xrt(self, energy, crystal_angle_offset = 0.0)`

`def get_energy(pv_list: list = ['vr', 'kr', 'v', 'k', 'd'])`

`def set_energy(energy: float, pv_list: list = [], crystal_angle_offset: float = 0.0)`

exafs - `exafs.py` EXAFS (Extended X-ray Absorption Fine Structure) control for MFX beamline.

`def __init__(self)`

`def _move_dccm_energy_with_vernier(self, energy_keV)`

`def _move_k_energy(self, k_energy)`

`def _move_vernier_energy(self, vernier_energy)`

`def _current_k_energy(self)`

`def _delta_eV_to_k_energy(self, energy_keV, abs_value = False, rounding = 1)`

`def _next_k_energy(self, k_stepsize, k_offset, reverse, min_k_keV)`

`def _build_energy_and_wait_time(self, energies_list, wait_time_list, start_eV, end_eV, min_k, max_k, element, min_time_EXAFS, max_time_EXAFS, debug)`

`def _initialize_energies_and_move(self, energies, wait_time, reverse, k_offset, k_stepsize, track_feespec, crystal_angle_offset = 0.0)`

`def _init_tfs(self, energies, margin_mm, ref_focal_length_um, ref_z_stage_mm, avoid_forbidden, enable_prefocus, map_focus_track, lens_beam_energy_offset, target = 400.37)`

`def _init_tchk(self, map_tchk_track)`

`def _setup_daq_and_start_recording(self, sample, picker, inspire, record, run_index)`

`def check_beam_status(self, flux_threshold)`

`def align_vernier_to_dccm(self, energy_center_offset_eV = 0.0, energy_range_eV = 5.0, energy_steps = 21, events_per_step = 60, flux_threshold = None, diagnostic = 'dg2')`

`def _measure_vernier_offset(self, energy, track_tchk_data, diagnostic = 'dg2')`

`def _retrieve_vernier_offset(self, energy, track_tchk_data)`

`def _align_vernier_to_dccm(self, energy_eV, track_tchk_data, map_tchk_track, diagnostic = 'dg2')`

```
def _request_vernier_offset_measurement(self, energy, track_tchk_data, map_tchk_track)
```

```
def _move_tfs_to_energy(self, energy_eV, track_focus_data, attenuation = None, tfs_offset = 0.0)
```

```
def _request_k_energy_update(self, energy_keV, k_stepsize, k_offset, reverse, min_k_keV)
```

```
def _move_k_if_necessary(self, energy_keV, k_stepsize, k_offset, reverse, min_k_keV, track_feespec)
```

```
def _align_undulator(self, on_diagnostic, using_device, with_method, grid_bins)
```

```
def _get_track_data(self, json_file_name)
```

```
def _save_track_data(self, track_record, json_file_name)
```

```
def _save_track_tchk_data(self, track_record, display = True)
```

```
def _plot_track_tchk_data(self, json_file_name)
```

```
def _measure_lens_beam_offset(self, energy, track_lens_offset_data)
```

```
def _return_to_start(self, energy_start, k_energy_start)
```

```
def _handle_keyboard_interrupt(self, sample, tag, run_number, record, inspire, energy_start, k_energy_start, crystal_angle_offset = 0.0)
```

```
def _finalize_scan(self, energy_start, k_energy_start, crystal_angle_offset = 0.0)
```

```
def _wait(self, wait_time)
```

```
def _post(self, sample = '?', tag = None, run_number = None, post = False, inspire = False, add_note = '')
```

```
def long_escan(self, simulate = False, start_eV = 0.0, end_eV = None, min_k = 2.0, max_k = 12.0, energies_list = [], wait_time_list = [], element = 'Fe', sample = '?', tag = None, picker = None, inspire = False, daq_delay = 5, record = False, runs = 1, k_stepsize = 120, reverse = False, min_k_keV = 7.035, k_offset = 0, flux_threshold = None, attenuation = None, min_time_EXAFS = 0.5, max_time_EXAFS = 10.0, tchk = False, diagnostic = 'dg2', map_focus_track = False, map_tchk_track = False, map_lens_beam_energy_offset = False, lens_beam_energy_offset = 0.0, track_focus = False, crystal_angle_offset = 0.0, tfs_margin_mm = 5.0, ref_focal_length_um = None, ref_z_stage_mm = None, tfs_target = 400.37, avoid_forbidden_combo = True, enable_prefocus = True, track_feespec = False, track_feespec_cam = False, undulator_point = False, undulator_on_diagnostic = 'dg1', undulator_using_device = 'yag', undulator_with_method = 'calib', undulator_grid_bins = 5, debug = False, tfs_offset = 0.0)
```

`def __init__(self)`

`def K_to_eV(self, K_value)`

`def eV_to_K(self, energy_eV)`

`def build_energy_range(self, min_before_pre_edge = 7055.0, max_before_pre_edge = 7110.2, preedge_end = 7116.2, preedge_eV_increment = 0.5, min_K_value = 2.0, max_K_value = 12.0, before_edge_eV_increment = 5.0, edge_eV_increment = 1.0, K_spacing = 0.1, time_before_edge = 2, time_in_edge = 1, time_in_preedge = 2, min_time_EXAFS = 0.5, max_time_EXAFS = 10, debug = False)`

`def map_time_to_K_weighting(self, min_time, max_time, K_values)`

`def plot_scan_profile(self, energy_range, time_range, energy_K_range, K_values)`

`def output_scan_profile(self, elist_name = 'elist', tlist_name = 'tlist')`

find - `find.py` Device and signal discovery utilities for MFX beamline.

`def __init__(self)`

`def get_motor_by_pvname(self, pvname: str)`

`def get_signal_by_pvname(self, pvname: str)`

`def get_signal_motor_by_pvname(self, pvname: str)`

`def get_motor(pvname: str)`

`def get_signal(pvname: str)`

`def get_positioner(pvname: str)`

junk - `junk.py` junk_plot.py

`def get_image(PV)`

`def require_zero(PV, suffix, expected)`

`def check_color_mode(PV)`

`def check_data_stream(PV)`

`def check_camviewer_config(PV)`

`def gaussian2D(coords, amplitude, xo, yo, sigma_x, sigma_y, offset)`

`def live_view(PV, n_frames = 200)`

`def main()`

macros - `macros.py` Utility macros and helper functions for MFX beamline operations.

`def determine_dccm_bragg(energy: float)`

`def laser_in(wait: bool = False, timeout: float = 10)`

`def laser_out(wait: bool = False, timeout: float = 10)`

`def set_slits(dg1: Optional[float] = None, dg2_us: Optional[float] = None, dg2_ms: Optional[float] = None, dg2_ds: Optional[float] = None)`

`def get_exp(hutch = 'mfx', station: int = 0)`

`def get_run(hutch = 'mfx', station: int = 0)`

`def __init__(self, detname: str = 'Rayonix')`

`def _energy_keV_to_wavelength_A(self, energy_keV)`

`def _pixel_index_to_radius_mm(self, pixel_index)`

`def _pixel_radius_mm_to_theta_radian(self, pixel_radius_mm, det_dist_mm)`

`def _pixel_theta_radian_to_q_invA(self, pixel_theta_radian, wavelength_A)`

`def _pixel_q_invA_to_resol_A(self, pixel_q_invA)`

`def _pixel_radius_mm_to_q_invA(self, radius_mm, det_dist_mm, energy_keV)`

`def resolution_coverage(self, energy_keV = None, det_dist_mm = None)`

mfx_timing - `mfx_timing.py` MFX timing and sequencer control for synchronized experiments.

`def __init__(self, sequencer: Optional[object] = None)`

`def _seq_step(self, evt_code_name: Optional[str] = None, delta_beam: int = 0)`

`def _seq_init(self, sync_mark: float = 30)`

`def _seq_put(self, steps: List[Tuple[str, int]])`

`def _seq_120hz(self)`

`def _seq_120hz_truncated(self)`

`def _seq_120hz_yano(self)`

`def _seq_90hz_yano(self)`

`def _seq_60hz(self)`

`def _seq_60hz_truncated(self)`

`def _seq_60hz_yano(self)`

`def _seq_30hz(self)`

`def _seq_20hz(self)`

`def _seq_10hz(self)`

`def set_seq(self, rep: Optional[str] = None, sequencer: Optional[str] = None, laser: Optional[List[str]] = None)`

`def set_timing(rep: str, sequencer: Optional[str] = None, laser: Optional[List[str]] = None)`

notch_scan - `notch_scan.py` DCCM notch filter energy scanning for MFX beamline.

`def __init__(self)`

`def set_energy(self, energy_eV: float, vernier: bool = False, wait: bool = True)`

`def series(self, energy_scan_start_eV: float, energy_scan_end_eV: float, energy_scan_steps: int, run_length: int = 30, tag: str = 'dccm', picker: Optional[str] = None, inspire: bool = False, daq_delay: int = 5, record: bool = False, vernier: bool = False, daq_num: int = 2, exp: Optional[str] = None)`

`def output(self, user: str, facility: str = 'S3DF', exp: str = None, run: str = None, energy: float = None, step: float = None, num: int = None, daq_num: int = 2)`

`def notch_scan(energy_scan_start_eV: float, energy_scan_end_eV: float, energy_scan_steps: int, run_length: int = 30, tag: str = 'dccm', picker: str = None, inspire: bool = False, daq_delay: int = 5, record: bool = False, daq_num: int = 2, exp: str = None)`

om - `om.py` OnDA (Online Data Analysis) monitor control for MFX beamline.

`def __init__(self, experiment: Optional[str] = None)`

`def check(self)`

`def fix_yaml(self, yaml: str, mask: Optional[str] = None, geom: Optional[str] = None)`

`def fix_run_om(self, detector: str)`

`def run(self, node: int, det: str = 'epix')`

`def reset(self, node: int)`

`def start_onda(node: int, detector: str = 'epix')`

`def reset_onda(node: int)`

optimize/beam - `beam.py`

`def _predict_centroid(self, undp_xy: tuple[float, float], calib: dict)`

`def _undp_solve(self, goal: tuple[float, float], calib: dict)`

`def _move_und_in_chunks(self, target_xy: tuple[float, float], max_step: float = 50.0, start_with_x: bool = True)`

`def _save_calibration_plots(self, res, reg_x, reg_y, out_dir, on_diagnostic, ts)`

`def _load_calibration(self, on_diagnostic: Diagnostics)`

`def check_calibration(self, on_diagnostic: Diagnostics = 'dg1', xopt_obj: Optional[Xopt] = None, threshold_sigma: float = 2.0)`

`def calibrate(self, xopt_obj: Xopt, on_diagnostic: Diagnostics = 'dg1', grid_bins: int = 5)`

`def _calib(self, xopt, goal, on_diagnostic, using_device, mover, grid_bins = 5, retrain = False)`

`def _turbo(self, xopt, path_plot, xopt_rand_evaluate, xopt_steps, mover, xopt_turbo_option)`

`def align(self, with_goal: Optional[float] = None, on_diagnostic: Diagnostics = 'dg1', with_package: Packages = 'xopt', with_method: Methods = 'turbo', using_device: Devices = 'yag', mover: Movers = 'mirr', xopt_turbo_option: Turbo = 'optimize', xopt_rand_evaluate: int = 3, xopt_steps: int = 10, xopt_max_iter: int = 2000, blop_qr_n: int = 16, blop_qei_n: int = 16, blop_qei_iterations: int = 5, use_2d_markers: bool = False, with_goal_2d: Optional[tuple[float, float]] = None, xopt_obj: Optional[Xopt] = None, save_run: bool = True, num_frames: int = 1, grid_bins: int = 5, retrain: bool = False)`

`def scan(self, on_diagnostic: Diagnostics = 'dg1', using_device: Devices = 'yag', mirror_pitch_start = None, mirror_pitch_end = None, num_steps: int = 51, sequencer_fps: int = 120)`

`def __init__(self, name: str = 'c1', dof: str = 'x')`

`def scan(self, scan_start = None, scan_end = None, num_steps: int = 51)`

optimize/beam_status - `beam_status.py`

`def __init__(self, prefix = '', name = 'beam_status', **kwargs)`

`def gdet_ave(self, threshold = 0.1)`

optimize/beamline_hw - `beamline_hw.py` Initialize hardware for blop_scans and xopt_scans

```
def init_devices(force: bool = False)
```

```
def sim_devices()
```

```
def get_offsets()
```

```
def get_fake_dg1_wave8_x()
```

```
def get_fake_dg2_wave8_x()
```

```
def get_fake_dg1_wave8_y()
```

```
def get_fake_dg2_wave8_y()
```

```
def update_fake_dg1_yag(cam: FakeLCLSImagePlugin)
```

```
def update_fake_dg2_yag(cam: FakeLCLSImagePlugin)
```

```
def update_fake_xcs_yag1(cam: FakeLCLSImagePlugin)
```

```
def update_fake_ip1_yag(cam: FakeLCLSImagePlugin)
```

```
def __init__(self, value)
```

```
def __init__(self, name, value, pos_tracker)
```

```
def move(self, position, **kwargs)
```

```
def set(self, value, **kwargs)
```

```
def put(self, value, **kwargs)
```

```
def position(self)
```

```
def get(self)
```

```
def get_fake_dccm_energy()
```

```
def get_fake_intensity()
```

```
def wrapper_dccm()
```

```
def wrapper_intensity()
```

```
def get_vernier_pos_tracker()
```

```
def get_dccm_tracker()
```

```
def get_alignment_scan_mode()
```

```
def get_calibration_scan_mode()
```

```
def read_dccm_energy()
```

optimize/blop_scans - `blop_scans.py` To run for real, import get_blop_agent, generate agents, and try to agent.learn().

`def init_bluesky_objs(force: bool = False)`

`def init_re()`

`def clean_re(re: RunEngine, bec: BestEffortCallback)`

`def get_blop_agent(wave8: Diagnostics = 'dg1', mirror_nominal: float = MIRROR_NOMINAL, search_delta: float = 5, wave8_xpos: Optional[float] = None, wave8_max_value: float = 10, setup_re: bool = True)`

`def digestion(df: DataFrame)`

`def setup_sim_test()`

`def run_sim_test()`

optimize/constraints - `constraints.py` Define per-device optimization constraints.

(No public functions)

optimize/devices - `devices.py` Ophyd devices created for these optimize routines.

`def get_coordinate(self)`

`def specific_marker_target(self, marker_num: int)`

`def average_marker_target(self, marker_nums: tuple[int, ...] | list[int])`

`def standard_two_corners_target(self)`

`def standard_box_target(self)`

`def __init__(self, *args, **kwargs)`

`def image(self)`

`def get_centroid(self)`

`def sim_set_image(self, size: tuple[int, int] | None = None, centroid: tuple[int, int] | None = None, fwhm: int | None = None, peak: int | None = None)`

`def sim_install_updater(self, updater: Callable[[FakeLCLSImagePlugin], None])`

`def fake_yag_image(size: tuple[int, int], centroid: tuple[int, int], fwhm: int, peak: int)`

optimize/errors - `errors.py` Custom exception classes.

(No public functions)

optimize/interactive - `interactive.py` Load devices and functions for interactive use.

`def get_parser()`

`def get_objects(sim: bool = False)`

`def misc_setup(autoreload: bool = False)`

optimize/mirror_pointing - `mirror_pointing.py`

`def get_yag_centroid(prefix, name, num_frames = 10, timeout = 10)`

`def optimize_mirror_pointing(instrument: str, diagnostic_pvname: str, window_size: float = 5.0, num_frames: int = 10, num_points: int = 10)`

`def main()`

optimize/plots - `plots.py` Matplotlib plotting utilities for tracking the optimizer's decisions.

`def refresh_mpl_plots()`

`def centroid_path_plot(image: np.ndarray, goal: Union[float, tuple[float, float]], markers: list[tuple[float, float]], centroids: list[tuple[float, float]], roi_center: Optional[tuple[int, int]] = None, roi_radius: Optional[int] = None, figure: Optional[Figure] = None)`

`def __init__(self, imager: YagCamera, goal: tuple[float, float], constraints: Optional[YagConstraints] = None)`

`def get_markers(self)`

`def add_point(self, centroid: tuple[float, float])`

`def add_points(self, centroids: list[tuple[float, float]])`

`def refresh(self)`

`def __init__(self, xopt: Xopt)`

`def refresh(self)`

optimize/type_checking - `type_checking.py` Type checking utilities for use in other submodules.

`def validate_w_lowercase_args(func)`

`def wrapper(*args, **kwargs)`

optimize/undpoint - `undpoint.py` Utilities for undulator pointing.

`def _get_2d_delta(mds: MultiDerivedSignal, items: SignalToValue)`

`def _put_2d_delta(mds: MultiDerivedSignal, value: tuple[float, float])`

```
def coerce_input_to_tuple(position: Union[tuple[float, float], float], ypos: Optional[float])
```

```
def __init__(self, prefix: str, done_pvname: str = DEFAULT_DONE_MOVE_PV, name: str, **kwargs)
```

```
def move(self, position: Union[tuple[float, float], float], y_delta: Optional[float] = None, wait: bool = True, timeout: Optional[float] = None, moved_cb: Optional[Callable] = None)
```

```
def _setup_move(self, position: tuple[float, float])
```

```
def __init__(self, prefix: str, name: str, done_pvname: str = DEFAULT_DONE_MOVE_PV, x_abs_pvname: str = 'BPMS:UNDH:4690:XOFF.D', y_abs_pvname: str = 'BPMS:UNDH:4690:YOFF.D', **kwargs)
```

```
def move(self, position: Union[tuple[float, float], float], y_abs: Optional[float] = None, wait: bool = True, timeout: Optional[float] = None, moved_cb: Optional[Callable] = None)
```

```
def get_delta_from_abs(self, position: tuple[float, float])
```

```
def _new_xpos(self, value: float, **kwargs)
```

```
def _new_ypos(self, value: float, **kwargs)
```

```
def _update_pos(self, **kwargs)
```

```
def position(self)
```

```
def __init__(self, prefix: str = 'MFX:USER:MCC:UND', name: str = 'mfx_undp', **kwargs)
```

```
def __init__(self, prefix: str, name: str, max_step: Optional[float] = 50.0, sleep_between: float = 2.0, **kwargs)
```

```
def move(self, position: Union[tuple[float, float], float], y_abs: Optional[float] = None, wait: bool = True, max_step: Optional[float] = None, sleep_between: Optional[float] = None, timeout: Optional[float] = None, moved_cb: Optional[Callable] = None)
```

```
def __init__(self, prefix: str = 'MFX:USER:MCC:UND', name: str = 'mfx_undp_safe', max_step: Optional[float] = 50.0, sleep_between: float = 2.0, **kwargs)
```

```
def _sim_new_move(self, value: int, **_)
```

```
def __init__(self, prefix = '', name = 'sim_2d_abs', **kwargs)
```

```
def _new_raw_x(self, value: float, **_)
```

```
def _new_raw_y(self, value: float, **_)
```

optimize/user_select - `user_select.py` Standard selectors for going from user inputs to various scan resources and settings.

```
def select_diagnostic(device_type: Devices, location: Diagnostics)
```

```
def select_goal(device_type: Devices, location: Diagnostics, goal: Optional[float] = None, goal_2d: Optional[tuple[float, float]] = None, use_2d_markers: bool = False)
```

optimize/utils - `utils.py` Utility functions for MFX optimization.

```
def _to_numpy(x)
```

```
def set_yag_constraints(location: str, roi_center: tuple[int, int] | None = None, roi_radius: int | None = None)
```

```
def set_und_constraints(xy_delta: float | None = None, max_travel_distance: float | None = None)
```

```
def plot_yag_optimization_setup(roi_center: tuple[int, int] = None, roi_radius: int = None, goal_2d: tuple[int, int] = None, yag_location: str = 'dg1', show_plot: bool = True)
```

```
def quick_yag_plot()
```

```
def plot_gp_landscape(opt, resolution: int = 50, figsize: tuple[int, int] = (12, 5), output_name: str = 'objective')
```

```
def snake_order(df, x = 'undp_x', y = 'undp_y', start = 'asc')
```

optimize/vernier_calibration - `vernier_calibration.py`

```
def __init__(self)
```

```
def configure(self, *args, **kwargs)
```

```
def stage(self, *args, **kwargs)
```

```
def unstage(self, *args, **kwargs)
```

```
def describe(self, *args, **kwargs)
```

```
def read(self, *args, **kwargs)
```

```
def collect(self, *args, **kwargs)
```

```
def __init__(self)
```

```
def _predict_vernier_offset(self, target_energy_eV: float, calib: dict)
```

```
def _vernier_solve(self, target_energy_eV: float, calib: dict)
```

```
def _save_calibration_plots(self, res, reg_offset, out_dir, ts)
```

`def _load_calibration(self)`

`def check_calibration(self, threshold_sigma: float = 2.0)`

`def calibrate(self, energy_start_eV: float, energy_end_eV: float, energy_steps: int = 10, events_per_step: int = 120, simulate: Optional[bool] = None)`

`def on_event(name, doc)`

`def measure_offset_at_energy(self, events: int = 120)`

`def fit(self, data_points: list[dict])`

`def align_to_dccm(self, energy_range_eV: float = 5.0, energy_steps: int = 21, events_per_step: int = 60, flux_threshold: float = None, simulate: Optional[bool] = None)`

`def move_to_energy_with_calibration(self, simulate: Optional[bool] = None)`

`def create_vernier_calibration()`

optimize/xopt_scans - `xopt_scans.py` To run for real, import get_xopt_obj and try to random_evaluate() and step() the Xopt object.

`def get_variables(mover: Movers, narrow: bool = False, diagnostic: Optional[Diagnostics] = None)`

`def get_constraints(diagnostic: Diagnostics, device: Devices)`

`def get_vocs(mover: Movers, diagnostic: Diagnostics, device: Devices)`

`def evaluator_move(mover: Movers, input: dict)`

`def get_evaluator_wave8(wave8: str = 'dg1', wave8_xpos: Optional[float] = None, mover: Movers = 'mirr')`

`def evaluate(input: dict[str, float])`

`def evaluate_yag_processing(diagnostic: Diagnostics, fit: ImageProjectionFit, num_frames: int = 1, save_dir: Optional[str] = None)`

`def distance2d(pt1: tuple[float, float], pt2: tuple[float, float])`

`def evaluate_yag_results(diagnostic: Diagnostics, fit_result: ImageProjectionFitResult)`

`def get_evaluator_yag(yag: str = 'dg1', goal: Optional[float] = None, mover: Movers = 'mirr', num_frames: int = 1, images_dir: Optional[str] = None)`

`def evaluate(input: dict[str, float])`

`def get_evaluator_yag_2d(yag: Diagnostics, goal: tuple[float, float], mover: Movers, num_frames: int = 1, images_dir: Optional[str] = None)`

`def evaluate(input: dict[str, float])`

`def get_evaluator_wave8_2d(wave8: Diagnostics, goal: tuple[float, float], mover: Movers)`

`def evaluate(input: dict[str, float])`

`def get_xopt_obj(device_type: Devices, location: Diagnostics, mover: Movers, goal: Optional[float] = None, xopt_generator_turbo_controller: Optional[Turbo] = None, use_2d_markers: bool = False, goal_2d: Optional[tuple[float, float]] = None, max_iter: Optional[int] = None, dump_file: Optional[str] = None, num_frames: int = 1)`

`def test_write_permissions()`

plans/serp_seq_scan - `serp_seq_scan.py`

`def serp_seq_scan(shift_motor, shift_pts, fly_motor, fly_pts, seq)`

`def per_step(detectors, step, pos_cache)`

`def test_serp_scan()`

`def trigger(self)`

scan - `scan.py` Generic motor scanning utilities for MFX beamline.

`def __init__(self)`

`def scan(self, scan_start: float, scan_end: float, scan_steps: int, events_per_step: int = 120, sample: str = '?', tag: str = None, picker: str = None, runs: int = 1, inspire: bool = False, record: bool = False, pv: str = 'lxt_fast', close: bool = True, daq_num: int = 2, exp: str = None)`

`def series(self, pv_values: List[float] = None, daq_delay: int = 5, scan_start: float = None, scan_end: float = None, scan_steps: int = None, events_per_step: int = 120, sample: str = '?', tag: str = None, picker: str = None, runs: int = 1, inspire: bool = False, record: bool = False, pv: str = 'lxt_fast', close: bool = True, daq_num: int = 2, exp: str = None)`

`def _get_pv_value(self, pv: str)`

`def _set_pv_value(self, pv: str, value: float)`

`def quick_scan(start: float, end: float, steps: int, pv: str = 'lxt_fast', sample: str = 'quick_scan', record: bool = False, **kwargs)`

```
def delay_scan(start: float, end: float, steps: int, sample: str = 'delay_scan', record: bool = False, **kwargs)
```

```
def series_scan(pv_values: List[float], start: float, end: float, steps: int, pv: str = 'lxt_fast', sample: str = 'series', record: bool = False, **kwargs)
```

```
def alignment_scan(motor: str, center: float = 0.0, range: float = 2.0, steps: int = 21, sample: str = 'alignment', record: bool = False, **kwargs)
```

timetool - `timetool.py` Time tool drift correction and monitoring utilities for MFX beamline.

```
def write_log(message: str, logfile: str = '')
```

```
def is_good_measurement(tt_data: np.ndarray, amplitude_thresh: float, ipm_thresh: float, fwhm_threshs: Tuple[float, float])
```

```
def correct_timing_drift(amplitude_thresh: float = 0.02, ipm_thresh: float = 500.0, drift_adjustment_thresh: float = 0.05, fwhm_threshs: Tuple[float, float] = (30, 130), num_events: int = 61, will_log: bool = True)
```

```
def monitor_timing(duration: float = 60.0, logfile: str = '')
```

```
def start_drift_correction(sensitivity: str = 'normal', logfile: Optional[str] = None)
```

```
def quick_drift_check(duration: float = 60.0)
```

timing - `timing.py` Vernier energy control and calibration utilities for MFX beamline.

```
def __init__(self)
```

```
def check(self)
```

```
def clustered_points(self, y, z, n, power = 2.0, center = None, plot = False)
```

```
def scan(self, start: float, end: float, steps: int, events_per_step: int = 240, sample: str = '?', tag: str = 'timing', picker: str = None, inspire: bool = False, record: bool = True, daq_num: int = 2, pv: str = None, laser: int = None, analysis: bool = True, randomize: bool = False, cluster: bool = False, center: float = None, delay: bool = False, duration: float = 300.0, sweep_time: float = 5.0)
```

```
def output(self, user: str, facility: str = 'S3DF', exp: str = None, run: str = None, daq_num: int = 2)
```

vernier - `vernier.py` Vernier energy control and calibration utilities for MFX beamline.

```
def __init__(self)
```

```
def scan(self, energy_scan_start_eV: float, energy_scan_end_eV: float, energy_scan_steps: int, events_per_step: int = 120, sample: str = '?', tag: str = None,
```

`picker: str = None, inspire: bool = False, record: bool = False, daq_num: int = 2, mcc: str = None)`

`def series(self, energy_scan_start_eV: float, energy_scan_end_eV: float, energy_scan_steps: int, run_length: int = 10, tag: str = None, picker: str = None, inspire: bool = False, daq_delay: int = 5, record: bool = False, daq_num: int = 2)`

`def __init__(self)`

`def fee_spec_list(self, user, facility, exp = None, run_list = None)`

`def series(self, user: str, facility: str = 'S3DF', run_type: str = 'series', exp: str = None, run: str = None, energy: float = None, step: float = None, num: int = None)`

`def scan(self, user: str, facility: str = 'S3DF', run_type: str = 'scan', exp: str = None, run: str = None)`

vonhamos - `vonhamos.py` Von Hamos spectrometer control for MFX beamline.

`def __init__(self, *args, **kwargs)`

`def go(self, target: float, epsilon: float = 0.001, n_iterations_max: int = 10, smart: bool = False, stuck_threshold: float = 0.001, overshoot_factor: float = 1.2, wait: bool = True)`

`def optimize_crystal(crystal: DeterministicCrystal, x_target: Optional[float] = None, rot_target: Optional[float] = None, tilt_target: Optional[float] = None, epsilon_x: float = 0.001, epsilon_rot: float = 0.01, epsilon_tilt: float = 0.01, smart: bool = True)`

`def set_all_crystals(spectrometer: DeterministicVonHamos6Crystal, x: Optional[float] = None, rot: Optional[float] = None, tilt: Optional[float] = None, epsilon_x: float = 0.001, epsilon_rot: float = 0.01, epsilon_tilt: float = 0.01, smart: bool = True)`

`def read_crystal_positions(spectrometer: DeterministicVonHamos6Crystal)`

`def print_crystal_positions(spectrometer: DeterministicVonHamos6Crystal)`

wire - `wire.py` Wire scanner control and scanning utilities for MFX beamline.

`def __init__(self)`

`def scan(self, start: float, end: float, num_steps: int, num_events: int = 120, sample: str = 'wire', tag: str = None, picker: str = None, inspire: bool = False, record: bool = False, daq_num: int = 2, mcc: str = None)`

`def output(self, user: str, facility: str = 'S3DF', run_type: str = 'scan', exp: str = None, run: int = None)`

`def __init__(self)`

`def x()`

`def y()`

`def __init__(self)`

`def x(value: float)`

`def y(value: float)`

`def wire_scan(start: float, end: float, num_steps: int, mcc: str, num_events: int = 120, record: bool = False, **kwargs)`

`def get_wire_position()`

`def set_wire_position(x: Optional[float] = None, y: Optional[float] = None)`

`def analyze_wire_scan(user: str, facility: str = 'S3DF', exp: str = None, run: int = None)`

`def home_wire()`

`def park_wire(x_park: float = -10.0, y_park: float = -10.0)`

xas - `xas.py` X-ray Absorption Spectroscopy (XAS) utilities for MFX beamline.

`def continuous_dccmscan(energies: List[float], pointTime: float = 1.0, move_vernier: bool = False, bidirectional: bool = False)`

`def run_dccmscan(energies: List[float], record: bool = True, pointTime: float = 1.0, move_vernier: bool = True, bidirectional: bool = False, **kwargs)`

`def build_xas_energy_list(pre_edge_start: float = 7.05, pre_edge_end: float = 7.095, edge_start: float = 7.1, edge_end: float = 7.14, post_edge_end: float = 7.2, pre_edge_spacing: float = 0.005, edge_spacing: float = 0.001, post_edge_spacing: float = 0.01)`

`def build_exafs_energy_list(edge_energy: float = 7.112, k_min: float = 2.0, k_max: float = 12.0, k_spacing: float = 0.05, include_pre_edge: bool = True)`

`def energy_to_k(energy: float, edge_energy: float = 7.112)`

`def k_to_energy(k: float, edge_energy: float = 7.112)`

`def estimate_edge_position(energies: np.ndarray, intensities: np.ndarray, method: str = 'derivative')`

`def normalize_xas(energies: np.ndarray, intensities: np.ndarray, pre_edge_range: Optional[Tuple[float, float]] = None, post_edge_range: Optional[Tuple[float, float]] = None)`

`def quick_xas_scan(element: str = 'Fe', record: bool = False, pointTime: float = 1.0)`

xlj_fast - `xlj_fast.py` XLJ (X-ray Liquid Jet) fast motor control with interactive keyboard interface.

`def __init__(self, motors: List, orientation: str = 'horizontal', scale: float = 0.1, mode: str = 'translation')`

`def _setup_translation_keys(self)`

`def _setup_rotation_keys(self)`

`def _setup_6axis_keys(self)`

`def scale_keys(self)`

`def print_help(self)`

`def print_status(self)`

`def update_scale(self, factor: float)`

`def execute_move(self, motor, direction: int)`

`def run(self)`

`def xlj_fast(orientation: str = 'horizontal', scale: float = 0.1)`

`def xlj_fast_rot(orientation: str = 'horizontal', scale: float = 0.1)`

`def xlj_6axis(orientation: str = 'horizontal', scale: float = 0.1)`

xrt_spec - `xrt_spec.py` Yano laser control and automated data acquisition for MFX beamline.

`def __init__(self)`

`def get_feespec_positions(self, energy_keV, crystal_angle_offset = 0.0, debug = False)`

`def move_feespec_energy(self, energy_keV, crystal_angle_offset = 0.0)`

`def check_feespec_crystal_angle(self, energy_keV, crystal_angle_offset = 0.0)`

`def track_feespec_camera(self, energy_keV, crystal_angle_offset = 0.0)`

`def scan_feespec_camera_angle(self, start_energy_keV, end_energy_keV, num_points, run_length: int = 30, tag: str = 'xrt_cam_scan', picker: Optional[str] = None, inspire: bool = False, daq_delay: int = 5, record: bool = False, daq_num: int = 2, check_xrt = True, exp: Optional[str] = None)`

`def output(self, user: str, facility: str = 'S3DF', exp: str = None, run: str = None, energy: float = None, step: float = None, num: int = None, daq_num: int = 2)`

yano - `yano.py` Yano laser control and automated data acquisition for MFX beamline.

`def __init__(self)`

`def shutter_status(self)`

`def configure_shutters(self, fiber1 = False, fiber2 = False, fiber3 = False, free_space = None)`

`def fiber_0(self)`

`def fiber_1(self)`

`def fiber_2(self)`

`def fiber_3(self)`

`def _delaystr(self, delay)`

`def _wrap_delay(self, delay, base_rate = 120)`

`def set_delay(self, delay, rep = 30)`

`def get_delay(self)`

`def post(self, sample = '?', tag = None, run_number = None, post = False, inspire = False, daq_num = 2, spread = None, add_note = '')`

`def track_focus(self, energy)`

`def _begin(self, events = None, duration = 300, record = False, use_l3t = None, controls = None, wait = False, end_run = False)`

`def plot_scan_profile(self, energy_seq, step_time, title = 'Energy Scan Sequence', highlight_cycles = True, figsize = (14, 7))`

`def generate_energy_seq(self, energy_scan_start_eV, energy_scan_end_eV, energy_scan_steps, run_length, step_time, brewster = 0, bs = None, spread_type = 'vernier', debug = False)`

`def run(self, sample = '?', tag = None, run_length = 300, record = True, runs = 5, inspire = False, daq_delay = 5, picker = None, fiber = 0, free_space = None, laser_delay = None, track_focus = False, rep = 30, daq_num = 2, spread = [], spread_type = None, step_time = None, brewster = 0, bs = None, debug = False)`

Scripts

analyze_timing - `analyze_timing.py` analyze_timing

`def proxy_jump(facility: str = 'S3DF', exp: str = None, run: str = None)`

`def get_scan_motor(run)`

`def custom_erf(x, a, sigma, mu, b)`

`def fit_irfs1(x_data, y_data, run_number, t_stage)`

`def output(facility: str = 'S3DF', exp: str = None, run: str = None)`

`def parse_args(args)`

`def main(args)`

`def run()`

auto_idle/idler - `idler.py`

(No public functions)

cctbx/average - `average.py` cctbx_start

`def average(exp, run, facility, debug)`

`def parse_args(args)`

`def main(args)`

`def run()`

cctbx/cctbx_start - `cctbx_start.py` cctbx_start

`def check_settings(exp, facility, cctbx_dir)`

`def parse_args(args)`

`def main(args)`

`def run()`

cctbx/energy_calib_output - `energy_calib_output.py` fee_spec

`def output(facility: str = 'S3DF', run_type: str = None, exp: str = None, run: str = None, energy: float = None, step: float = None, num: int = None)`

`def parse_args(args)`

`def main(args)`

`def run()`

cctbx/fee_calib - `fee_calib.py`

`def process_run(exp, run_num, detector_name, max_events)`

`def gaussian(x, amplitude, mean, sigma)`

`def find_peak_position(spectrum, fit_window, sg_window, poly_order)`

`def calibrate_energy_scale(peak_positions, energies)`

`def run(args)`

cctbx/fee_spec - `fee_spec.py` fee_spec

`def output(exp: str = None, runs: str = None, facility: str = 'S3DF')`

`def parse_args(args)`

`def main(args)`

`def run()`

cctbx/fee_summed - `fee_summed.py`

(No public functions)

cctbx/geom_refine - `geom_refine.py` cctbx_start

`def geom_refine(exp, facility, level, group)`

`def parse_args(args)`

`def main(args)`

`def run()`

cctbx/image_viewer - `image_viewer.py` cctbx_start

`def image_viewer(exp, run, facility, image_type, group, debug)`

`def parse_args(args)`

`def main(args)`

`def run()`

cctbx/mask - `mask.py` cctbx_start

`def mask(exp, run, facility, group)`

`def parse_args(args)`

`def main(args)`

`def run()`

conf_edit - `conf_edit.py` cctbx_start

`def is_valid_yaml(file_path)`

`def update_yaml(exp)`

`def parse_args(args)`

`def main(args)`

`def run()`

detector_image - `detector_image.py`

(No public functions)

door_watcher/door_watcher - `door_watcher.py`

`def __init__(self, door_state, voltage_read, voltage_write)`

`def start(self)`

`def callback(self, old_value = None, value = None, **kwargs)`

`def stop(self)`

`def main()`

focus_scan - `focus_scan.py`

`def get_markers(PV)`

`def roi_size(marker1X, marker1Y, marker2X, marker2Y)`

`def check_color_mode(PV)`

`def check_data_stream(PV)`

`def check_camviewer_config(PV)`

`def get_image(PV)`

`def get_roi(image, marker1X, marker1Y, marker2X, marker2Y)`

`def get_projections(image)`

`def gaussian(x, amplitude, mean, stddev)`

`def FWHM(x, y)`

`def get_lens_z()`

`def fwhm_calc(PV, marker1X, marker1Y, marker2X, marker2Y, axis_x_x, axis_x_y)`

`def park_lens()`

`def plot_data(scan_data)`

generate_code_summary - `generate_code_summary.py`

`def __init__(self, file_path: Path)`

`def parse(self)`

`def get_module_docstring(self)`

`def extract_all_functions(self)`

`def _get_full_signature(self, node: ast.FunctionDef)`

`def _get_parent_class(self, node: ast.FunctionDef)`

`def generate_summary()`

`def generate_pdf(markdown_file: Path)`

generate_mkdocs - `generate_mkdocs.py` Script to generate MFX documentation.

`def parse_gitignore(repo_path: Path)`

`def is_ignored(path: Path, repo_path: Path, gitignore_patterns: Set[str], exclude_dirs: Optional[Set[str]] = None)`

`def clean_docs_folder(docs_path: Path, valid_md_files: Set[Path], preserve_files: Optional[Set[str]] = None, backup: bool = False)`

`def create_md_file(md_path: Path, module_path: str)`

`def organize_by_module_structure(python_files: List[Path], repo_path: Path)`

`def build_nav_from_structure(structure: Dict, max_depth: int = 10, current_depth: int = 0)`

`def get_module_path(py_file: Path, repo_path: Path)`

`def create_mkdocs_config(nav_structure: List, repo_path: Path)`

`def create_extra_css(docs_path: Path)`

`def find_python_files(repo_path: Path, gitignore_patterns: Set[str], exclude_dirs: Optional[Set[str]] = None)`

`def generate_docs(repo_path: str = '.', docs_path: str = 'docs', backup: bool = False, show_ignored: bool = False, exclude_dirs: Optional[Set[str]] = None)`

get_info - `get_info.py`

`def get_info(argv)`

jungfrau/take_pedestal - `take_pedestal.py` Take a pedestal for the Jungfrau

`def __init__(self, hutch, src, *aliases)`

`def gainName(self, gain = 0)`

`def commit(self)`

`def takeJungfrauPedestals(record = True, nEvts = 1000)`

TFS

lens - `lens.py` Basic Lens object handling

`def mv_retry(self, position, retries = 3, tolerance = 0.01, settle_time = 0.2, timeout = None)`

`def _parse_lens_number(prefix)`

`def __init__(self, *args, **kwargs)`

`def focus(self, energy)`

`def image_from_obj(self, z_obj, energy)`

`def __init__(self, prefix, **kwargs)`

`def radius(self)`

`def z(self)`

`def sig_focus(self)`

`def _do_move(self, state)`

`def __init__(self, *args)`

`def effective_radius(self)`

`def tfs_radius(self)`

`def image(self, z_obj, energy)`

`def nlens(self)`

`def _info(self)`

`def show_info(self)`

`def connect(cls, array1, array2)`

offline_calculator - `offline_calculator.py`

`def __init__(self, tfs_lenses, prefocus_lenses = None, exclusions = None)`

`def combinations(self)`

`def _update_combos(self)`

`def get_pre_focus_lens(self, energy)`

`def get_combo_image(combo, z_obj = 0.0)`

`def check_forbidden(self, pre_focus_lens_radius, energy, radius)`

`def find_solution(self, target, energy, n = 4, z_obj = 0.0, avoid_forbidden = True, enable_prefocus = True)`

sim_transfocator - `sim_transfocator.py`

`def _do_move(self, state)`

`def attach(self, tfs_dev)`

`def mv(self, value)`

`def make_tfs_sim(tfs, prefix = 'SIM:', name = 'tfs_sim')`

`def _translation(self)`

tfs_plots - `tfs_plots.py`

`def plot_sweeps()`

`def plot_sweep_energy(xrt_lens, dbi, ax = None)`

`def plot_spreadsheet_data(xrt_lens, ax, df)`

`def plot_lens(xrt_lens)`

`def plot_lens_all()`

transfocator - `transfocator.py`

`def __init__(self, prefix, *args, **kwargs)`

`def __init__(self, prefix, nominal_sample = 400.37, **kwargs)`

`def lenses(self)`

`def xrt_lenses(self)`

`def tfs_lenses(self)`

`def current_focus(self, energy_eV = None)`

`def remove_all(self)`

`def find_best_combo(self, target = None, energy_eV = None, n = 4, z_obj = 0, show = True, exclusions = [], avoid_forbidden = False, enable_prefocus = True, **kwargs)`

`def try_combo(self, target = 400.37, energy = None, show = True, prefocus = None, tfs = [], **kwargs)`

`def set(self, value, **kwargs)`

`def focus_at(self, value = None, wait = False, timeout = None, **kwargs)`

`def plan_energy_schedule(self, low_eV, high_eV, step_eV = 10.0, target = None, n = 4, z_obj = 0.0, show = False)`

`def plot_focus_track(self, json_file_path)`

`def get_stage_limits(self, margin_mm)`

`def mv_stage_to_pos(self, z_mm)`

`def set_reference_combo(self, energy_eV, show = False, **kwargs)`

`def get_z_stage_target(self, energy_eV, combo, ref_focal_length_um, ref_z_stage_mm)`

`def mv_stage_to_target_pos(self, energy_eV, combo, target_z_mm, track_record)`

`def track_focus(self, energies, margin_mm = 10.0, show = False, ref_focal_length_um = None, ref_z_stage_mm = None, display = True, shrinking_rate = 4, enable_prefocus = True, lens_beam_energy_offset = 0.0, **kwargs)`

`def constant_energy(func)`

`def with_constant_energy(transfocator_obj, energy_type, tolerance, *args, **kwargs)`

transfocator_scan - `transfocator_scan.py`

`def __init__(self, camera_pv)`

`def check_color_mode(self)`

`def get_image(self)`

`def get_image_old(self)`

`def get_image_roi(self)`

`def get_markers(self)`

`def get_projections(self)`

`def scan_transfocator(self, transfocator_motor, positions, image_sec)`

`def twoD_gaussian_fixed_angle(xy, amplitude, xo, yo, sigma_x, sigma_y, offset)`

`def fit_2d_gaussian_fixed_angle(image)`

`def scan_transfocator_jb(self, transfocator_motor, positions, image_sec)`

`def fit_scan(self, x, y)`

`def gaussian(x, amplitude, mean, stddev)`

utils - `utils.py`

`def focal_length(radius, energy, N = 1)`

`def focal_length_old(radius, energy, N = 1)`

`def estimate_beam_fwhm(radius, energy, fwhm_unfocused = 0.0003, distance = 4.474)`

Repository Statistics

- **Total Modules:** 84
- **Total Functions:** 732
