hklpy Documentation

Brookhaven National Laboratory

Oct 24, 2020

CONTENTS

Diffractometer computation library with ophyd pseudopositioner support.

Based on the *hkl* library (https://repo.or.cz/hkl.git, also https://github.com/picca/hkl but that repository may not be synchronized with the latest version from repo.or.cz). Documentation for *hkl* is here: https://people.debian.org/~picca/hkl/hkl.html

Integrates with ophyd pseudopositioners.

Documentation about *ophyd* and the *bluesky* framework (https://blueskyproject.io/).

Always import the `gobject-introspection` package first and require `Hkl` version 5.0, as in:

```
import gi
gi.require_version('Hkl', '5.0')

from hkl.diffract import E4CV
```

Contents:

CALC

class hkl.calc.**CalcE4CH**(***kwargs*)

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hkl.calc.**CalcE4CV**(***kwargs*)

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hkl.calc.**CalcE6C**(***kwargs*)

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hkl.calc.**CalcK4CV**(***kwargs*)

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hkl.calc.**CalcK6C**(***kwargs*)

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hkl.calc.**CalcMed2p3**(***kwargs*)

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hkl.calc.**CalcPetra3_p09_eh2**(***kwargs*)

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hkl.calc.**CalcRecip**(*dtype*, *engine='hkl'*, *sample='main'*, *lattice=None*, *degrees=True*, *units='user'*, *lock_engine=False*, *inverted_axes=None*)
Reciprocal space calculations

Parameters

- **dtype** (*str*) – Diffractometer type (usually specified by a subclass)

- **engine** (*str, optional*) – 'hkl', for example
- **sample** (*str, optional*) – Default sample name (default: 'main')
- **lattice** (*Lattice, optional*) – Lattice to use with the default sample
- **degrees** (*bool, optional*) – Use degrees instead of radians (default: True)
- **units** (*{'user', }*) – The type of units to use internally
- **lock_engine** (*bool, optional*) – Don't allow the engine to be changed during the life of this object
- **inverted_axes** (*list, optional*) – Names of axes to invert the sign of

property Position
Dynamically-generated physical motor position class

__init__(*dtype*, *engine='hkl'*, *sample='main'*, *lattice=None*, *degrees=True*, *units='user'*, *lock_engine=False*, *inverted_axes=None*)
Initialize self. See help(type(self)) for accurate signature.

_get_axis_by_name(*name*)
Given an axis name, return the HklParameter

Parameters **name** (*str*) – If a name map is specified, this is the mapped name.

_invert_physical_positions(*pos*)
Invert the physical axis positions based on the settings

Parameters **pos** (*OrderedDict*) – NOTE: Modified in-place

add_sample(*sample*, *select=True*)
Add an HklSample

Parameters

- **sample** (*HklSample instance*) – The sample name, or optionally an already-created HklSample instance
- **select** (*bool, optional*) – Select the sample to focus calculations on

property energy
The energy associated with the geometry, in keV

property engine_locked
If set, do not allow the engine to be changed post-initialization

forward(*position*, *engine=None*)
Forward-calculate a position from pseudo to real space

forward_iter(*start*, *end*, *max_iters*, ***, *threshold=0.99*, *decision_fcn=None*)
Iteratively attempt to go from a pseudo start -> end position

For every solution failure, the position is moved back. For every success, the position is moved closer to the destination.

After up to max_iters, the position can be reached, the solutions will be returned. Otherwise, ValueError will be raised stating the last position that was reachable and the corresponding motor positions.

Parameters

- **start** (*Position*) –
- **end** (*Position*) –
- **max_iters** (*int*) – Maximum number of iterations

- **threshold** (*float, optional*) – Normalized proximity to *end* position to stop iterating
- **decision_fcn** (*callable, optional*) – Function to choose a solution from several. Defaults to picking the first solution. The signature of the function should be as follows:

  >> def decision(pseudo_position, solution_list): >> return solution_list[0]

Returns solutions

Return type list

Raises ***UnreachableError*** **(ValueError)** – Position cannot be reached The last valid HKL position and motor positions are accessible in this exception instance.

property inverted_axes
The physical axis names to invert

new_sample(*name*, *select=True*, ***kwargs*)
Convenience function to add a sample by name

Keyword arguments are passed to the new HklSample initializer.

Parameters

- **name** (*str*) – The sample name
- **select** (*bool, optional*) – Select the sample to focus calculations on

property physical_axes
Physical (real) motor positions as an OrderedDict

property physical_positions
Physical (real) motor positions

property pseudo_axes
Ordered dictionary of axis name to position

property pseudo_axis_names
Pseudo axis names from the current engine

property pseudo_positions
Pseudo axis positions/values from the current engine

property sample_name
The name of the currently selected sample

property units
The units used for calculations

update()
Calculate the pseudo axis positions from the real axis positions

property wavelength
The wavelength associated with the geometry, in angstrom

class hkl.calc.**CalcSoleilMars**(***kwargs*)

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hkl.calc.**CalcSoleilSiriusKappa**(***kwargs*)

`__init__`(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class `hkl.calc.`**`CalcSoleilSiriusTurret`**(***kwargs*)

`__init__`(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class `hkl.calc.`**`CalcSoleilSixs`**(***kwargs*)

`__init__`(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class `hkl.calc.`**`CalcSoleilSixsMed1p2`**(***kwargs*)

`__init__`(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class `hkl.calc.`**`CalcSoleilSixsMed2p2`**(***kwargs*)

`__init__`(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class `hkl.calc.`**`CalcTwoC`**(***kwargs*)

`__init__`(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class `hkl.calc.`**`CalcZaxis`**(***kwargs*)

`__init__`(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

exception `hkl.calc.`**`UnreachableError`**(*msg*, *pseudo*, *physical*)
Position is unreachable.

`pseudo`
Last reachable pseudo position in the trajectory

Type sequence

`physical`
Corresponding physical motor positions

Type sequence

`__init__`(*msg*, *pseudo*, *physical*)
Initialize self. See help(type(self)) for accurate signature.

`hkl.calc.`**`_keep_physical_position`**(*func*)
decorator: stores/restores the physical motor position during calculations

`hkl.calc.`**`_locked`**(*func*)
a decorator for running a method with the instance's lock

`hkl.calc.`**`default_decision_function`**(*position*, *solutions*)
The default decision function - returns the first solution

CONTEXT

class hkl.context.**TemporaryGeometry**(*calc*)

Context manager that restores physical geometry after a block of code

__init__(*calc*)

Initialize self. See help(type(self)) for accurate signature.

class hkl.context.**UsingEngine**(*calc*, *engine*)

Context manager that uses a calculation engine temporarily

__init__(*calc*, *engine*)

Initialize self. See help(type(self)) for accurate signature.

CHAPTER THREE

DIFFRACT

A local subclass of `hkl.diffract.Diffractometer` for the desired diffractometer geometry must be created to define the reciprocal-space axes and customize the EPICS PVs used for the motor axes. Other capabilities are also customized in a local subclass.

Examples are provided after the source code documentation.

These are the diffractometer geometries provided by the **hkl-c++** library[1]:

name	description
`E4CH`	Eulerian 4-circle, vertical scattering plane
`E4CV`	Eulerian 4-circle, horizontal scattering plane
`E6C`	Eulerian 6-circle
`K4CV`	Kappa 4-circle, vertical scattering plane
`K6C`	Kappa 6-circle
`TwoC`	2-circle
`Zaxis`	Z-axis

These special-use geometries are also provided by the **hkl-c++** library[1]:

- `Med2p3`
- `Petra3_p09_eh2`
- `SoleilMars`
- `SoleilSiriusKappa`
- `SoleilSiriusTurret`
- `SoleilSixs`
- `SoleilSixsMed1p2`
- `SoleilSixsMed2p2`

In all cases, see the **hkl-c++** documentation for further information on these geometries.

[1] **hkl-c++** documentation: https://people.debian.org/~picca/hkl/hkl.html

3.1 Source code documentation

3.1.1 diffract

Support for diffractometer instances

BASE CLASS

Diffractometer(prefix[, calc_kw, . . .])	Diffractometer pseudopositioner

DIFFRACTOMETER GEOMETRIES

E4CH(prefix[, calc_kw, decision_fcn, . . .])
E4CV(prefix[, calc_kw, decision_fcn, . . .])
E6C(prefix[, calc_kw, decision_fcn, . . .])
K4CV(prefix[, calc_kw, decision_fcn, . . .])
K6C(prefix[, calc_kw, decision_fcn, . . .])
TwoC(prefix[, calc_kw, decision_fcn, . . .])
Zaxis(prefix[, calc_kw, decision_fcn, . . .])

SPECIAL-USE DIFFRACTOMETER GEOMETRIES

Med2p3(prefix[, calc_kw, decision_fcn, . . .])
Petra3_p09_eh2(prefix[, calc_kw, . . .])
SoleilMars(prefix[, calc_kw, decision_fcn, . . .])
SoleilSiriusKappa(prefix[, calc_kw, . . .])
SoleilSiriusTurret(prefix[, calc_kw, . . .])
SoleilSixs(prefix[, calc_kw, decision_fcn, . . .])
SoleilSixsMed1p2(prefix[, calc_kw, . . .])
SoleilSixsMed2p2(prefix[, calc_kw, . . .])

class hkl.diffract.**Diffractometer**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

Diffractometer pseudopositioner

calc	The calculation instance
engine	The calculation engine associated with the diffractometer
forward(pseudo)	Calculate a RealPosition from a given PseudoPosition
inverse(real)	Calculate a PseudoPosition from a given RealPosition
_energy_changed([value])	Callback indicating that the energy signal was updated
_update_calc_energy([value])	writes self.calc.energy from value or self.energy

This has a corresponding calculation engine from **hklpy** that does forward and inverse calculations.

If instantiating a specific diffractometer class such as *E4C*, *E6C*, neither the *calc_inst* or the *calc_kw* parameters are required.

However, there is the option to either pass in a calculation instance (with *calc_inst*) or keywords for the default calculation class (using *calc_kw*) to instantiate a new one.

Parameters

- **`prefix`** (`str`) – PV prefix for all components
- **`calc_kw`** (`dict, optional`) – Initializer arguments for the calc class
- **`decision_fcn`** (`callable, optional`) – The decision function to use when multiple solutions exist for a given forward calculation. Defaults to arbitrarily picking the first solution.
- **`read_attrs`** (`list, optional`) – Read attributes default to all pseudo and real positioners
- **`configuration_attrs`** (`list, optional`) – Defaults to the UB matrix and energy
- **`parent`** (`Device, optional`) – Parent device
- **`name`** (`str, optional`) – Device name

`calc_class`

Reciprocal calculation class used with this diffractometer. If None (as in *hkl.diffract.Diffractometer*, *calc_inst* must be specified upon initialization.

Type sub-class of CalcRecip

See also:

`hkl.diffract.E4CH`, `hkl.diffract.E4CV`, `hkl.diffract.E6C`, `hkl.diffract.K4CV`, `hkl.diffract.K6C`, `hkl.diffract.Med2p3`, `hkl.diffract.Petra3_p09_eh2`, `hkl.diffract.SoleilMars`, `hkl.diffract.SoleilSiriusKappa`, `hkl.diffract.SoleilSiriusTurret`, `hkl.diffract.SoleilSixs`, `hkl.diffract.SoleilSixsMed1p2`, `hkl.diffract.SoleilSixsMed2p2`, `hkl.diffract.TwoC`, `hkl.diffract.Zaxis`

`__init__`(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

`property _calc_energy_update_permitted`

return boolean *True* if permitted

`_energy_changed`(*value=None*, ***kwargs*)

Callback indicating that the energy signal was updated

Note: The *energy* signal is subscribed to this method in the `Diffractometer.__init__()` method.

`_update_calc_energy`(*value=None*, ***kwargs*)

writes self.calc.energy from value or self.energy

`property calc`

The calculation instance

`check_value`(*pos*)

Raise exception if pos is not within limits.

In a scan, a subset of the pseudo axes may be directed, which are given in a dict from a set message from the bluesky RunEngine.

It is not permitted to scan both pseudo and real positioners.

property engine
The calculation engine associated with the diffractometer

forward(*pseudo*)
Calculate a RealPosition from a given PseudoPosition

Must be defined on the subclass.

Parameters **pseudo_pos** (`PseudoPosition`) – The pseudo position input

Returns **real_position** – The real position output

Return type RealPosition

inverse(*real*)
Calculate a PseudoPosition from a given RealPosition

Must be defined on the subclass.

Parameters **real_position** (`RealPosition`) – The real position input

Returns **pseudo_pos** – The pseudo position output

Return type PseudoPosition

class hkl.diffract.**E4CH**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**E4CV**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**E6C**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**K4CV**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**K6C**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**Med2p3**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**Petra3_p09_eh2**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**SoleilMars**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**SoleilSiriusKappa**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**SoleilSiriusTurret**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**SoleilSixs**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**SoleilSixsMed1p2**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**SoleilSixsMed2p2**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**TwoC**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

class hkl.diffract.**Zaxis**(*prefix*, *calc_kw=None*, *decision_fcn=None*, *calc_inst=None*, ***, *configuration_attrs=None*, *read_attrs=None*, ***kwargs*)

3.2 Examples

Demonstrate the setup of diffractometers using the *hkl* package.

- `sim6c` : simulated 6-circle
- `k4cv` : kappa 4-circle with EPICS PVs for motors
- `k4cve` : `k4cv` with energy from local control system

The *sample Examples* section describes how to setup a crystal sample with an orientation matrix.

3.2.1 `sim6c`: 6-circle with simulated motors

It is useful, sometimes, to create a simulated diffractometer where the motor axes are provided by software simulations, rather than using the actual motors provided by the diffractometer.

The `ophyd.SoftPositioner`[2] is such a software simulation.

Create the custom 6-circle subclass:

```
import gi
gi.require_version('Hkl', '5.0')
# MUST come before `import hkl`
import hkl.diffract
from ophyd import Component, PseudoSingle, SoftPositioner

class SimulatedE6C(hkl.diffract.E6C):
    """E6C: Simulated (soft) 6-circle diffractometer"""

    h = Component(PseudoSingle, '')
    k = Component(PseudoSingle, '')
    l = Component(PseudoSingle, '')

    mu = Component(SoftPositioner)
    omega = Component(SoftPositioner)
    chi = Component(SoftPositioner)
    phi = Component(SoftPositioner)
    gamma = Component(SoftPositioner)
    delta = Component(SoftPositioner)

    def __init__(self, *args, **kwargs):
        """
        start the SoftPositioner objects with initial values
        """
        super().__init__(*args, **kwargs)
        for axis in self.real_positioners:
            axis.move(0)
```

[2] `ophyd.SoftPositioner`: https://blueskyproject.io/ophyd/positioners.html#ophyd.positioner.SoftPositioner

Create an instance of this diffractometer with:

```
sim6c = SimulatedE6C('', name='sim6c')
```

3.2.2 `k4cv` : kappa 4-circle with EPICS motor PVs

To control a kappa diffractometer (in 4-circle geometry with vertical scattering plane) where the motor axes are provided by EPICS PVs, use *ophyd.EpicsMotor*.[3]

In this example, we know from our local control system that the kappa motors have these PVs:

kappa axis	EPICS PVs
komega	sky:m1
kappa	sky:m2
kphi	sky:m3
tth	sky:m4

Create the custom kappa 4-circle subclass:

```
import gi
gi.require_version('Hkl', '5.0')
# MUST come before `import hkl`
import hkl.diffract
from ophyd import Component, PseudoSingle, EpicsMotor

class KappaK4CV(hkl.diffract.K4CV):
    """K4CV: kappa diffractometer in 4-circle geometry"""

    h = Component(PseudoSingle, '')
    k = Component(PseudoSingle, '')
    l = Component(PseudoSingle, '')

    komega = Component(EpicsMotor, "sky:m1")
    kappa = Component(EpicsMotor, "sky:m2")
    kphi = Component(EpicsMotor, "sky:m3")
    tth = Component(EpicsMotor, "sky:m4")
```

Create an instance of this diffractometer with:

```
k4cv = KappaK4CV('', name='k4cv')
```

3.2.3 `k4cve` : `k4cv` with energy from local control system

Extend the `k4cv` example above to use the energy as provided by the local control system. In this example, assume these are the PVs to be used:

signal	EPICS PV
energy	optics:energy
energy units	optics:energy.EGU
energy locked?	optics:energy_locked
energy offset	no PV, same units as *energy* (above)

[3] `ophyd.EpicsMotor`: https://blueskyproject.io/ophyd/builtin-devices.html?highlight=epicsmotor#ophyd.epics_motor.EpicsMotor

The *energy locked?* signal is a flag controlled by the user that controls whether (or not) the *energy* signal will update the wavelength of the diffractometer's *calc* engine. We expect this to be either 1 (update the calc engine) or 0 (do NOT update the calc engine).

We'll also create a (non-EPICS) signal to provide for an energy offset (in the same units as the control system energy). This offset will be *added* to the control system energy (in `_update_calc_energy()`), before conversion of the units to *keV* and then setting the diffractometer's *calc* engine energy:

calc engine *energy* (keV) = control system *energy* + *offset*

which then sets the wavelength:

calc engine *wavelength* (angstrom) = $h\nu$ / calc engine *energy*

($h\nu = 12.39842$ angstrom · keV) and account for the units of the control system *energy*. To combine all this, we define a new python class starting similar to *KappaK4CV* above, and adding the energy signals. Create the custom kappa 4-circle subclass with energy:

```python
import gi
gi.require_version('Hkl', '5.0')
# MUST come before `import hkl`
import hkl.diffract
from ophyd import Component
from ophyd import PseudoSingle
from ophyd import EpicsSignal, EpicsMotor, Signal
import pint

class KappaK4CV_Energy(hkl.diffract.K4CV):
    """
    K4CV: kappa diffractometer in 4-circle geometry with energy
    """

    h = Component(PseudoSingle, '')
    k = Component(PseudoSingle, '')
    l = Component(PseudoSingle, '')

    komega = Component(EpicsMotor, "sky:m1")
    kappa = Component(EpicsMotor, "sky:m2")
    kphi = Component(EpicsMotor, "sky:m3")
    tth = Component(EpicsMotor, "sky:m4")

    energy = Component(EpicsSignal, "optics:energy")
    energy_units = Component(EpicsSignal, "optics:energy.EGU")
    energy_offset = Component(Signal, value=0)
    energy_update_calc_flag = Component(
        EpicsSignal,
        "optics:energy_locked")
```

Create an instance of this diffractometer with:

```python
k4cve = KappaK4CV_Energy('', name='k4cve')
```

Note: If you get a log message such as this on the console:

```
W Fri-09:12:16 - k4cve not fully connected, k4cve.calc.energy not updated
```

this informs you the update cannot happen until all EPICS PVs are connected. The following code, will create the object, wait for all PVs to connect, then update the *calc* engine:

```
k4cve = KappaK4CV_Energy('', name='k4cve')
k4cve.wait_for_connection()
k4cve._energy_changed(k4cve.energy.get())
```

To set the energy offset from the command line:

```
%mov k4cve.energy_offset 50
```

which means the diffractometer (assuming the control system uses "eV" units) will use an energy that is 50 eV *higher* than the control system reports. The diffractometer's *calc* engine will **only** be updated when the energy signal is next updated. To force an update to the calc engine, call `_energy_changed()` directly with the energy value as the argument:

```
k4cve._energy_changed()
```

But this only works when the `optics:energy_locked` PV is 1 (permitted to update the calc engine energy). To update the diffractometer's *calc* engine energy and bypass the `k4cve.energy_update_calc_flag` signal, call this command:

```
k4cve._update_calc_energy()
```

Finally, to set the energy of the diffractometer's *calc* engine, use one of these two methods:

`energy_update_calc_flag`	command
obey signal	`k4cve._energy_changed()`
ignore signal	`k4cve._update_calc_energy()`

Each of these two methods will accept an optional `value` argument which, if provided, will be used in place of the `energy` signal from the control system.

CHAPTER

FOUR

ENGINE

class hkl.engine.**CalcParameter**(*param*, *geometry*, **args*, ***kwargs*)

Like calc parameter but needs reference to a geometry object. Updates to the parameter should be propagated back to the geometry.

Parameters

- **param** (*HklParameter*) –
- **geometry** (*Geomery object*) –

__init__(*param*, *geometry*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

property fit

True if the parameter can be fit or not

class hkl.engine.**Engine**(*calc*, *engine*, *engine_list*)

HKL calculation engine

__init__(*calc*, *engine*, *engine_list*)

Initialize self. See help(type(self)) for accurate signature.

property _units

The (internal) units used for calculations

property engine

The calculation engine

property mode

HKL calculation mode (see also *HklCalc.modes*)

property units

The units used for calculations

update()

Calculate the pseudo axis positions from the real axis positions

class hkl.engine.**Parameter**(*param*, *units='user'*, *name=None*, *inverted=False*)

HKL library parameter object

Example:

```
Parameter(
        name='None (internally: ux)',
        limits=(min=-180.0, max=180.0),
        value=0.0,
        fit=True,
```

(continues on next page)

(continued from previous page)

```
        inverted=False,
        units='Degree')
```

__init__(*param*, *units='user'*, *name=None*, *inverted=False*)
Initialize self. See help(type(self)) for accurate signature.

property default_units
A string representing the default unit type

property fit
True if the parameter can be fit or not

property hkl_parameter
The HKL library parameter object

property inverted
Is the value inverted internally?

property user_units
A string representing the user unit type

class hkl.engine.**Solution**(*engine*, *list_item*, *class_*)
solution of the conversion from (hkl) to axes

__init__(*engine*, *list_item*, *class_*)
Initialize self. See help(type(self)) for accurate signature.

CHAPTER FIVE

SAMPLE

class hkl.sample.**HklSample**(*calc*, *sample=None*, *units='user'*, ***kwargs*)

Represents a sample in diffractometer calculations

Parameters

- **calc** (*instance of CalcRecip*) – Reciprocal space calculation class
- **name** (*str*) – A user-defined name used to refer to the sample
- **sample** (*Hkl.Sample, optional*) – A Sample instance from the wrapped Hkl library. Created automatically if not specified.
- **units** (*{'user', 'default'}*) – Units to use
- **lattice** (*np.ndarray, optional*) – The lattice
- **U** (*np.ndarray, optional*) – The crystal orientation matrix, U
- **UB** (*np.ndarray, optional*) – The UB matrix, where U is the crystal orientation matrix and B is the transition matrix of a non-orthonormal (the reciprocal of the crystal) in an orthonormal system
- **ux** (*np.ndarray, optional*) – ux part of the U matrix
- **uy** (*np.ndarray, optional*) – uy part of the U matrix
- **uz** (*np.ndarray, optional*) – uz part of the U matrix
- **reflections** – All reflections for the current sample in the form:

  ```
  [(h, k, l), ...]
  ```

  This assumes the hkl engine is used; generally, the ordered set of positions for the engine in-use should be specified.

property U

The crystal orientation matrix, U

property UB

The UB matrix, where U is the crystal orientation matrix and B is the transition matrix of a non-orthonormal (the reciprocal of the crystal) in an orthonormal system

If written to, the B matrix will be kept constant: U * B = UB -> U = UB * B^-1

__init__(*calc*, *sample=None*, *units='user'*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

_create_reflection(*h*, *k*, *l*, *detector=None*)

Create a new reflection with the current geometry/detector

_refl_matrix(*fcn*)
Get a reflection angle matrix

add_reflection(*h*, *k*, *l*, *position=None*, *detector=None*, *compute_ub=False*)
Add a reflection, optionally specifying the detector to use

Parameters

- **h** (`float`) – Reflection h
- **k** (`float`) – Reflection k
- **l** (`float`) – Reflection l
- **detector** (`Hkl.Detector, optional`) – The detector
- **position** (`tuple or namedtuple, optional`) – The physical motor position that this reflection corresponds to If not specified, the current geometry of the calculation engine is assumed.
- **compute_ub** (`bool, optional`) – Calculate the UB matrix with the last two reflections

affine()
Make the sample transform affine

clear_reflections()
Clear all reflections for the current sample

compute_UB(*r1*, *r2*)
Compute the UB matrix with two reflections

Using the Busing and Levy method, compute the UB matrix for two sample reflections, r1 and r2.

Note that this modifies the internal state of the sample and does not return a UB matrix. To access it after computation, see *Sample.UB*.

Parameters

- **r1** (`HklReflection`) – Reflection 1
- **r2** (`HklReflection`) – Reflection 2

property hkl_calc
The HklCalc instance associated with the sample

property hkl_sample
The HKL library sample object

property lattice
The lattice (a, b, c, alpha, beta, gamma)

a, b, c [nm] alpha, beta, gamma [deg]

property name
The name of the currently selected sample

property reciprocal
The reciprocal lattice

property reflections
All reflections for the current sample in the form: [(h, k, l), . . .]

remove_reflection(*refl*)
Remove a specific reflection

property ux
ux part of the U matrix

property uy
uy part of the U matrix

property uz
uz part of the U matrix

hkl.sample.**check_lattice**(*lattice*)
Check an Hkl.Lattice for validity

Raises `ValueError` –

5.1 Examples

We limit our examples here to just a few brief examples will show how to define a sample, its crystal lattice, and orient the crystal to position it using reciprocal space coordinates. We'll leave it to others (for now) to show more comprehensive examples that show additional capabilities, such as limiting the ranges of the motors for acceptable solutions of the *forward* calculation from (*hkl*) to motor positions.

These examples use a diffractometer object called `fourc` which we'll take to be an instance of a 4-circle diffractometer in vertical scattering geometry (*E4CV*) with simulated motors. (This is similar to the 6-circle example *sim6c: 6-circle with simulated motors*.)

```
import gi
gi.require_version('Hkl', '5.0')
# MUST come before `import hkl`
import hkl.diffract
from ophyd import Component, PseudoSingle, SoftPositioner

class SimulatedE4CV(hkl.diffract.E4CV):

    h = Component(PseudoSingle, '')
    k = Component(PseudoSingle, '')
    l = Component(PseudoSingle, '')

    omega = Component(SoftPositioner)
    chi = Component(SoftPositioner)
    phi = Component(SoftPositioner)
    tth = Component(SoftPositioner)

    def __init__(self, *args, **kwargs):
        """
        start the SoftPositioner objects with initial values
        """
        super().__init__(*args, **kwargs)
        for axis in self.real_positioners:
            axis.move(0)
```

Then, create the `fourc` object for these examples:

```
fourc = SimulatedE4CV('', name='fourc')
```

5.1.1 define a sample with a lattice

Define a sample with the name *EuPtIn4_eh1_ver* and define its crystal lattice. Use angstroms as units for the unit cell edges and degrees for the angles.

```
from hkl.util import Lattice
fourc.calc.new_sample('EuPtIn4_eh1_ver',
    lattice=Lattice(
        a=4.542, b=16.955, c=7.389,
        alpha=90.0, beta=90.0, gamma=90.0))
```

5.1.2 define an orientation matrix

This is a brief example to define two reflections and then apply the method of Busing & Levy[1] to calculate an orientation matrix.

For this example, the reflection positions were found at a wavelength of 1.62751693358 angstroms.

```
fourc.calc.wavelength = 1.62751693358
```

We found the (*080*) reflection at these motor positions:

```
p = fourc.calc.Position(
    omega=22.31594, chi=89.1377, phi=0, tth=45.15857)
```

Associate *p* (the motor positions) with the (*080*) reflection:

```
r1 = fourc.calc.sample.add_reflection(0, 8, 0, p)
```

Do the same for the (*0 12 1*) reflection:

```
p = fourc.calc.Position(
    omega=34.96232, chi=78.3139, phi=0, tth=71.8007)
r2 = fourc.calc.sample.add_reflection(0, 12, 1, p)
```

Compute the **UB** matrix:

```
fourc.calc.sample.compute_UB(r1, r2)
```

See the *hkl-c++* documentation[2] for more details.

5.1.3 compute the motor positions for a reflection

Compute the motor positions for the (*100*) reflection:

```
fourc.forward(1,0,0)
```

This can be assigned to a python object:

```
p = fourc.forward(0,1,1)
```

[1] Acta Cryst 22 (1967) 457-464

[2] *hkl-c++* documentation: https://people.debian.org/~picca/hkl/hkl.html

Then, the motor positions can be accessed: `p.omega`, `p.chi`, `p.phi`, `p.tth`.

These are the motor angles computed as the first solution:

(hkl)	omega	chi	phi	tth
(1, 0, 0)	10.32101	0.20845	86.38310	20.64202

CHAPTER SIX

UTIL

`hkl.util.`**`Lattice`**
alias of `hkl.util.LatticeTuple`

`hkl.util.`**`get_position_tuple`**(*axis_names*, *class_name='Position'*)

`hkl.util.`**`hkl_euler_matrix`**(*euler_x*, *euler_y*, *euler_z*)

`hkl.util.`**`new_detector`**(*dtype=0*)
Create a new HKL-library detector

`hkl.util.`**`to_hkl`**(*arr*)
Convert a numpy ndarray to an hkl `Matrix`

Parameters **`arr`** (*`ndarray`*) –

Returns

Return type Hkl.Matrix

`hkl.util.`**`to_numpy`**(*mat*)
Convert an hkl `Matrix` to a numpy ndarray

Parameters **`mat`** (*`Hkl.Matrix`*) –

Returns

Return type ndarray

PYTHON MODULE INDEX

INDEX