Syris Documentation

Tomas Farago

May 18, 2022

Contents:

CHAPTER 1

Application Programming Interface

1.1 Syris Configuration

1.1.1 Initialization

Synchrotron Radiation Imaging Simulation (SYRIS) initialization.

`syris.`**`init`**(*platform_name=None*, *device_type=None*, *device_index=None*, *profiling=False*, *profiling_file='profile.dat'*, *loglevel=20*, *logfile=None*, *double_precision=False*)

Initialize syris with *device_index*.

1.1.2 Initialization routines and global variables

OpenCL, logging and precision configuration. This module also defines global variables which hold the configuration objects for precision and OpenCL. Furthermore, pmasf path is specified here and caching policy for data as well.

`class` `syris.config.`**`OpenCL`**

OpenCL runtime information holder.

`class` `syris.config.`**`Precision`**(*double=False*)

A precision object holds the precision settings of the floating point and complex numpy and OpenCL data types. If *double* is True, double precision is used.

`is_single`()

Return True if the precision is single.

`set_precision`(*double*)

If *double* is True set the double precision.

`syris.config.`**`init_logging`**(*level=10*, *logger_file=None*)

Initialize logging with output to *logger_file*.

1.2 Image Processing

Module for GPU-based image processing.

class `syris.imageprocessing.`**`Tiler`**(*shape*, *tiles_count*, *outlier=True*, *supersampling=1*, *cplx=False*)

Class for breaking images into smaller tiles.

`average`(*tile*, *out=None*)

Average `pyopencl.array.Array` *tile* based on supersampling and outlier specified for the tiler. If *out* is not None, it will be used for returning the sum.

`insert`(*tile*, *indices*)

Insert a non-supersampled, outlier-free *tile* into the overall image. *indices* (y, x) are tile indices in the overall image.

`result_tile_shape`

Result tile shape without outlier and supersampling.

`tile_indices`

Get the supersampled tile indices which are starting points of a given tile in (y, x) fashion.

`tile_shape`

Get the supersampled tile shape based on tile counts *tile_counts* as (y, x) and *shape* (y, x).

`syris.imageprocessing.`**`bin_image`**(*image*, *summed_shape*, *offset=(0, 0)*, *average=False*, *out=None*, *queue=None*, *block=False*)

Bin an *image*. The resulting buffer has shape *summed_shape* (y, x). *Offset* (y, x) is the offset to the original *image*. *summed_shape* has to be a divisor of the original shape minus the *offset*. If *average* is True, the summed pixel is normalized by the region area. *out* is the pyopencl Array instance, if not specified it will be created. *out* is also returned. If *block* is True, wait for the copy to finish.

`syris.imageprocessing.`**`blur_with_gaussian`**(*image*, *sigma*, *queue=None*, *block=False*)

Blur *image* with a gaussian kernel, where *sigma* is the standard deviation. Use command *queue*, if *block* is True, wait for the copy to finish.

`syris.imageprocessing.`**`crop`**(*image*, *region*, *out=None*, *queue=None*, *block=False*)

Crop a 2D *image*. *region* is the region to crop as (y_0, x_0, height, width), *out* is the pyopencl Array instance, if not specified it will be created. *out* is also returned. If *block* is True, wait for the copy to finish.

`syris.imageprocessing.`**`decimate`**(*image*, *shape*, *sigma=None*, *average=False*, *queue=None*, *block=False*)

Decimate *image* so that its dimensions match the final *shape*, which has to be a divisor of the original shape. Remove low frequencies by a Gaussian filter with *sigma* pixels. If *sigma* is None, use the FWHM of one low resolution pixel. Use command *queue*, if *block* is True, wait for the copy to finish.

`syris.imageprocessing.`**`fft_2`**(*data*, *queue=None*, *block=True*)

2D FFT executed on *data*. *block* specifies if the execution will wait until the scheduled FFT kernels finish. The transformation is done in-place if *data* is a pyopencl Array class and has complex data type, otherwise the data is converted first.

`syris.imageprocessing.`**`get_gauss_2d`**(*shape*, *sigma*, *pixel_size=1*, *fourier=False*, *queue=None*, *block=False*)

Get 2D Gaussian of *shape* with standard deviation *sigma* and *pixel_size*. If *fourier* is True the fourier transform of it is returned so it is faster for usage by convolution. Use command *queue* if specified. If *block* is True, wait for the kernel to finish.

`syris.imageprocessing.`**`get_num_tiles`**(*tiles*, *num_tiles=None*)

Determine number of tiles in the *tiles* list.

`syris.imageprocessing.`**`ifft_2`**(*data*, *queue=None*, *block=True*)

2D inverse FFT executed on *data*. *block* specifies if the execution will wait until the scheduled FFT kernels finish. The transformation is done in-place if *data* is a pyopencl Array class and has complex data type, otherwise the data is converted first.

`syris.imageprocessing.`**`make_tile_offsets`**(*shape*, *tile_shape*, *outlier=(0, 0)*)

Make tile offsets in pixels so that one tile has *tile_shape* and all tiles form an image of *shape*. *outlier* specifies the the overlap of the tiles, so if the outlier width is m, the tile overlaps with the previous and next tiles by m / 2. If the tile width is n, the tile must be cropped to (m / 2, n - n / 2) before it can be placed into the resulting image. This is convenient for convolution outlier treatment.

`syris.imageprocessing.`**`make_tiles`**(*func*, *shape*, *tile_shape*, *iterable=None*, *outlier=(0, 0)*, *queues=None*, *args=()*, *kwargs=None*)

Make tiles using *func* which can either have signature func(item, *args, **kwargs) or func(item, queue, *args, **kwargs), where queue is the OpenCL command queue. In the latter case, multiple command queues are mapped to different computation items. *shape* (y, x) is the final image shape, *tile_shape* (y, x) is the shape of one tile, *iterable* is the sequence to be mapped to *func*, if not specified, the offsets from `make_tile_offsets()` are used. *outlier* (y, x) is the amount of overlapping region between tiles, *queues* are the OpenCL command queues to use, *args* and *kwargs* are additional arguments passed to *func*. Returns a generator.

`syris.imageprocessing.`**`merge_tiles`**(*tiles*, *num_tiles=None*, *outlier=(0, 0)*)

Merge *tiles* which is a list to one large image. *num_tiles* is a tuple specifying the number of tiles as (y, x) or None, meaning there is equal number of tiles in both dimensions. The tiles must be stored in the row-major order.

`syris.imageprocessing.`**`pad`**(*image*, *region=None*, *out=None*, *value=0*, *queue=None*, *block=False*)

Pad a 2D *image*. *region* is the region to pad as (y_0, x_0, height, width). If not specified, the next power of two dimensions are used and the image is centered in the padded one. The final image dimensions are height x width and the filling starts at (y_0, x_0), *out* is the pyopencl Array instance, if not specified it will be created. *out* is also returned. *value* is the padded value. If *block* is True, wait for the copy to finish.

`syris.imageprocessing.`**`rescale`**(*image*, *shape*, *sampler=None*, *queue=None*, *out=None*, *block=False*)

Rescale *image* to *shape* and use *sampler* which is a `pyopencl.Sampler` instance. Use OpenCL *queue* and *out* pyopencl Array. If *block* is True, wait for the copy to finish.

`syris.imageprocessing.`**`varconvolve`**(*kernel_name*, *shape*, *kernel_args*, *local_size=None*, *program=None*, *queue=None*, *block=False*)

Variable convolution with OpenCL kernel function *kernel_name*, gloal size *shape* (y, x), kernel arguments *kernel_args*, work group size *local_size* (can be None, i.e. OpenCL will determine it automatically), OpenCL *program* (can be None in which case the default syris variable convolution program is used with all predefined kernels). *queue* is the command queue, if *block* is True wait for the kernel to finish. Return OpenCL event from the kernel execution.

$$(f * g)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y, \xi, \eta) g(x - \xi, y - \eta) d\xi d\eta$$

`syris.imageprocessing.`**`varconvolve_disk`**(*image*, *radii*, *normalized=True*, *smooth=True*, *sampler=None*, *queue=None*, *out=None*, *block=False*)

Variable convolution of input *image* with an elliptical disk with y and x radii. *radii* specify the convolution kernel disk y and x radius for every output point. They are specified as two 2D arrays and can be either a tuple of two 2D arrays or a pyopencl.array.Array instance with vfloat2 data type, meaning both 2D arrays are encoded in it. If *normalized* is True the convolution kernel sum is always 1. Use OpenCL *sampler*, command *queue*, *out* as output and wait for execution end if *block* is True. Convolution window is always odd-shaped and the middle pixel is set to 0. This means that if the *radii* are smaller numbers than 1, the convolution returns the original image. This has a consequence that it is not possible to create a disk with even number of pixels accross one of the principal axes, so the disk radius will be exact from the middle if you specify it in half pixels, e.g. if the radius is 1.5, then pixels [-1, 0, 1] will be selected, i.e. the disk diameter is 3 pixels.

`syris.imageprocessing.`**`varconvolve_gauss`**(*image*, *sigmas*, *normalized=True*, *sampler=None*, *queue=None*, *out=None*, *block=False*)

Variable convolution of input *image* with a Gaussian with y and x sigmas. *sigmas* specify the convolution kernel y and x sigmas for every output point. They are specified as two 2D arrays and can be either a tuple of two 2D arrays or a pyopencl.array.Array instance with vfloat2 data type, meaning both 2D arrays are encoded in it. If *normalized* is True the convolution kernel sum is always 1. Use OpenCL *sampler*, command *queue*, *out* as output and wait for execution end if *block* is True. Convolution window is always odd-shaped and the middle pixel is set to 0. This means that if the *sigmas* are smaller numbers than 1, the convolution returns the original image.

1.3 Geometry

Geometrical operations from primitive mathematical routines like rotation and translation to complex motion description by a spline-based *`Trajectory`* class. *`BoundingBox`* used to constraint physical bodies is defined here as well.

All the transformation operations are in the backward form, which means if the order of operations is: A = trans_1 B = trans_2 C = trams_3, then in the forward form of the resulting transformation matrix would be T = ABC yielding x' = ABCx = Tx. Backward form means that we calculate the matrix in the form T^{-1} = C^{-1}B^{-1}A^{-1} = (ABC)^{-1}. Thus, we can easily obtain x = T^{-1}x'.

`class` `syris.geometry.`**`BoundingBox`**(*points*)

Class representing a graphical object's bounding box.

`get_max`(*axis=0*)

Get maximum along the specified *axis*.

`get_min`(*axis=0*)

Get minimum along the specified *axis*.

`get_projected_points`(*axis*)

Get the points projection by releasing the specified *axis*.

`merge`(*other*)

Merge with *other* bounding box.

`overlaps`(*other*)

Determine if the bounding box XY-projection overlaps XY-projection of *other* bounding box.

`points`

The object border points.

`roi`

Return range of interest defined by the bounding box as (y_0, x_0, y_1, x_1).

`class` `syris.geometry.`**`Trajectory`**(*control_points*, *pixel_size=None*, *furthest_point=None*, *time_dist=None*, *velocity=None*, *num_points=None*)

Class representing object's trajectory.

Trajectory is a spline interpolated from a set of points.

`bind`(*pixel_size=None*, *furthest_point=None*)

Bind the trajectory to a *pixel_size* to make sure two positions are not more than *pixel_size* apart between two time points. *furthest_point* is the furthest point from a body center used to compute rotational displacement and it can be None.

`bound`

Return True if the trajectory is currently bound.

`control_points`
Control points used by the trajectory.

`furthest_point`
Furthest point for which the trajectory is interpolated.

`get_direction`(*abs_time*, *norm=True*)
Get direction of the trajectory at the time *abs_time*. It is the derivative of the trajectory at *abs_time*. If *norm* is True, the direction vector will be normalized.

`get_distances`(*u=None*, *u_0=None*)
Get the distances from the trajectory beginning to every consecutive point defined by the parameter.

`get_maximum_dt`(*distance=None*)
Get the maximum time difference which moves the object no more than *distance*. If distance is None the pixel size this trajectory is bound to is used. Consider both rotational and translational displacement.

`get_maximum_du`(*distance=None*)
Get the maximum parameter difference which moves the object no more than *distance*. If distance is None the pixel size this trajectory is bound to is used. Consider both rotational and translational displacement.

`get_next_time`(*t_0*)
Get time from *t_0* when the trajectory will have travelled more than pixel size.

`get_parameter`(*abs_time*)
Get the spline parameter from the time *abs_time*.

`get_point`(*abs_time*)
Get a point on the trajectory at the time *abs_time*.

`length`
Trajectory length.

`pixel_size`
Pixel size for which the trajectory is interpolated.

`points`
Return interpolated points.

`stationary`
Return True if the trajectory is stationary.

`time`
Total time needed to travel the whole trajectory.

`time_tck`
The tck tuple of `scipy.interpolate.splrep()` for time-distance spline.

`times`
Return the time points for which the distance is defined.

exception `syris.geometry.`**`TrajectoryError`**
Exceptions related to trajectory.

`syris.geometry.`**`angle`**(*vec_0*, *vec_1*)
Angle between vectors *vec_0* and *vec_1*. The vectors might be 2D with 0 dimension specifying (x, y, z) components.

`syris.geometry.`**`closest`**(*values*, *min_value*)
Get the minimum greater value *min_value* from *values*.

syris.geometry.**derivative_fit**(*tck*, *u*, *max_distance*)
Reinterpolate curve in a way that all the f' * du are smaller than *max_distance*. The original spline is given by *tck* and parameter *u*.

syris.geometry.**get_rotation_displacement**(*d_0*, *d_1*, *length*)
Return the displacement of a sphere with radius *length* caused by rotation around vectors *d_0* and *d_1*. The displacement is returned for every axis (x, y, z).

syris.geometry.**interpolate_1d**(*x_0*, *y_0*, *size*)
Interpolate function y = f(x) with *x_0*, *y_0* as control points and return the interpolated x_1 and y_1 arrays of *size*.

syris.geometry.**is_normalized**(*vector*)
Test whether a *vector* is normalized.

syris.geometry.**length**(*vector*)
Get length of a *vector*.

syris.geometry.**make_points**(*x_ends*, *y_ends*, *z_ends*)
Make 3D points out of minima and maxima given by *x_ends*, *y_ends* and *z_ends*.

syris.geometry.**maximum_derivative_parameter**(*tck*, *u*, *max_distance*)
Get the maximum possible du, for which holds that dx < *max_distance*.

syris.geometry.**normalize**(*vector*)
Normalize a *vector*.

syris.geometry.**overlap**(*interval_0*, *interval_1*)
Check if intervals *interval_0* and *interval_1* overlap.

syris.geometry.**reinterpolate**(*tck*, *u*, *n*)
Arc length reinterpolation of a spline given by *tck* and parameter *u* to have *n* data points.

syris.geometry.**rotate**(*phi*, *axis*, *shift=None*)
Rotate the object by *phi* around vector *axis*, where *shift* is the translation which takes place before the rotation and -*shift* takes place afterward, resulting in the transformation TRT^-1. Rotation around an arbitrary point in space can be modeled in this way. The angle is _always_ rescaled to radians.

syris.geometry.**scale**(*scale_vec*)
Scale the object by scaling coefficients (kx, ky, kz) given by *sc_vec*.

syris.geometry.**transform_vector**(*trans_matrix*, *vector*)
Transform *vector* by the transformation matrix *trans_matrix* with dimensions (4,3) width x height.

syris.geometry.**translate**(*vec*)
Translate the object by a vector *vec*. The vector is _always_ transformed into meters.

1.4 Materials

Sample material represented by a complex refractive index.

class syris.materials.**Material**(*name*, *refractive_indices*, *energies*, *f_1=None*, *f_2=None*)
A material represented by its *name* and *refractive_indices* calculated for *energies*.

energies
energies for which the complex refractive index was calculated.

get_attenuation_coefficient(*energy*)
Get the linear attenuation coefficient at *energy*.

get_refractive_index(*energy*)
Interpolate refractive indices to obtain the one at *energy*.

name
Material *name*.

refractive_indices
Get complex refractive indices (delta [phase], ibeta [absorption]) for all energies used to create the material.

save(*filename=None*)
Save this instance to a *filename*.

exception syris.materials.**MaterialError**
Material errors

syris.materials.**make_fromfile**(*filename*)
Load saved material from *filename*.

syris.materials.**make_henke**(*name*, *energies*, *formula=None*, *density=None*)
Use the https://henke.lbl.gov database to lookup a material *name* for *energies*, use the specified chemical *formula* and *density*.

syris.materials.**make_pmasf**(*name*, *energies*)
Make a material based on the PMASF program.

- *name* - compund name defined in "compound.dat"
- ***energies* - list of energies which will be taken** into account [keV]
- *steps* - number of intervals between the energies

Return a list of refractive indices.

syris.materials.**make_stepanov**(*name*, *energies*, *density=None*, *formula=None*, *crystal=None*)
Use the https://x-server.gmca.aps.anl.gov database to lookup a material *name* for *energies*, use the specified chemical *formula* and *density*.

1.5 Optical Elements

Optical Elements are entities capable of producing wavefields as a function of time.

class syris.opticalelements.**OpticalElement**
An optical element capable of producing a wavefield as a function of time.

get_next_time(*t_0*, *distance*)
Get next time at which the object will have traveled *distance*, the starting time is *t_0*.

transfer(*shape*, *pixel_size*, *energy*, *exponent=False*, *offset=None*, *t=None*, *queue=None*, *out=None*, *check=True*, *block=False*)
Transfer function of the element in real space on an image plane of size *shape*, use *pixel_size*, *energy*, *offset* is the physical spatial offset of the element as (y, x), transfer at time *t*. If *exponent* is true, compute the exponent of the transfer function without applying the wavenumber. Use *queue* for OpenCL computations and *out* pyopencl array. If *block* is True, wait for the kernel to finish. If *check* is True, the function is checked for aliasing artefacts.

transfer_fourier(*shape*, *pixel_size*, *energy*, *t=None*, *queue=None*, *out=None*, *block=False*)
Transfer function of the element in Fourier space of size *shape*, use *pixel_size*, *energy* and comput the function at time *t*. Use *queue* for OpenCL computations and *out* pyopencl array. If *block* is True, wait for the kernel to finish.

1.6 Math

Math helper functions.

`syris.math.`**`closest`**(*values*, *min_value*)
: Get the minimum greater value greater than *min_value* from *values*.

`syris.math.`**`difference_root`**(*x_0*, *tck*, *y_d*)
: Given a function $f(x) = y$, find x_1 for which holds $|f(x_1) - f(x_0)| = y_d$. *x_0* is the starting x_0 and $f(x)$ is defined by spline coefficients *tck*.

`syris.math.`**`fftfreq`**(*n*, *pixel_size*)
: Compute spatial frequencies for a 2D grid (*n*, *n*) with spacing *pixel_size*. Returns spatial frequencies f as (f_y, f_x).

`syris.math.`**`fwnm_to_sigma`**(*fwnm*, *n=2*)
: Return Gaussian sigma from full width at *n*-th maximum *fwnm*.

`syris.math.`**`get_surrounding_points`**(*points*, *threshold*)
: Get the closest points around a *threshold* from both sides, left and right. If one of the sides is empty than None is returned on its place.

`syris.math.`**`infimum`**(*x_0*, *data*)
: Return the greatest point from *data* which is less than *x_0*.

`syris.math.`**`match_range`**(*x_points*, *y_points*, *x_target*)
: Match the curve $f(x) = y$ to *x_target* points by interpolation of *x_points* and *y_points*.

`syris.math.`**`sigma_to_fwnm`**(*sigma*, *n=2*)
: Return Gaussian full width at n-th maximum given by *sigma* and *n*.

`syris.math.`**`supremum`**(*x_0*, *data*)
: Return the smallest point from *data* which is greater than *x_0*.

1.7 Physics

Physics on the light path.

`syris.physics.`**`compute_aliasing_limit`**(*n*, *wavelength*, *pixel_size*, *propagation_distance*, *fov=None*, *fourier=True*)
: Get the non-aliased fraction of data points when propagating a wavefield to a region $n \times pixel\_size$ to *propagation_distance* using *wavelength*, *pixel_size* and field of view *fov* (if not specified computed as *n* * *pixel_size*). If *fourier* is True then the limit is computed for the Fourier space.

`syris.physics.`**`compute_collection`**(*num_aperture*, *opt_ref_index*)
: Get the collection efficiency of the scintillator combined with a lens. The efficiency is given by $\eta = \frac{1}{2}\left(\frac{NA}{n}\right)^2$, where NA is the numerical aperture *num_aperture* of the lens, n is the optical refractive index *opt_ref_index* given by the *`Scintillator`*.

`syris.physics.`**`compute_diffraction_angle`**(*diameter*, *propagation_distance*)
: Compute the diffraction angle for a region where a wavefield within the *diameter* can interfere on a *propagation_distance*.

`syris.physics.`**`compute_propagation_distance_limit`**(*n*, *wavelength*, *pixel_size*)
: Compute the propagation distance which just fits the sampling theorem for *n* pixels, *wavelength* and *pixel_size*.

`syris.physics.`**`compute_propagation_sampling`**(*wavelength*, *distance*, *fov*, *fresnel=True*)
: Compute the required number of pixels and pixel size in order to satisfy the sampling theorem when propagating a wavefield with *wavelength* to *distance* and we want to propagate field of view *fov*. If *fresnel* is true, the same

distance computation approximation is done as when computing a Fresnel propagator (2nd order Taylor series expansion for the square root).

`syris.physics.`**`compute_propagator`**(*size*, *distance*, *lam*, *pixel_size*, *fresnel=True*, *region=None*, *apply_phase_factor=False*, *mollified=True*, *queue=None*, *block=False*)

Create a propagator with (*size*, *size*) dimensions for propagation *distance*, wavelength *lam* and *pixel_size*. If *fresnel* is True, use the Fresnel approximation, if it is False, use the full propagator (don't approximate the square root). *region* is the diameter of the the wavefront area which is capable of interference. If *apply_phase_factor* is True, apply the phase factor defined by Fresnel approximation. If *mollified* is True the aliased frequencies are suppressed. If command *queue* is specified, execute the kernel on it. If *block* is True, wait for the kernel to finish.

`syris.physics.`**`energy_to_wavelength`**(*energy*)

Convert *energy* [eV-like] to wavelength [m].

`syris.physics.`**`is_wavefield_sampling_ok`**(*wavefield_exponent*, *queue=None*, *out=None*)

Check the sampling of the *wavefield_exponent*. Use OpenCL *queue* and *out* array. Return True if the sampling is OK, False otherwise.

`syris.physics.`**`propagate`**(*samples*, *shape*, *energies*, *distance*, *pixel_size*, *region=None*, *apply_phase_factor=False*, *mollified=True*, *detector=None*, *offset=None*, *queue=None*, *out=None*, *t=None*, *check=True*, *block=False*)

Propagate *samples* with *shape* as (y, x) which are `syris.opticalelements.OpticalElement` instances at *energies* to *distance*. Use *pixel_size*, limit coherence to *region*, *apply_phase_factor* is as by the Fresnel approximation phase factor, *offset* is the sample offset. *queue* an OpenCL command queue, *out* a PyOpenCL Array. If *block* is True, wait for the kernels to finish. If *check* is True, check the transmission function sampling.

`syris.physics.`**`ref_index_to_attenuation_coeff`**(*ref_index*, *lam*)

Convert refractive index to the linear attenuation coefficient given by $\mu = \frac{4\pi\beta}{\lambda}$ based on given *ref_index* and wavelength *lam*.

`syris.physics.`**`transfer`**(*thickness*, *refractive_index*, *wavelength*, *exponent=False*, *queue=None*, *out=None*, *check=True*, *block=False*)

Transfer *thickness* (can be either a numpy or pyopencl array) with *refractive_index* and given *wavelength*. If *exponent* is True, compute the exponent of the function without applying the wavenumber. Use command *queue* for computation and *out* pyopencl array. If *block* is True, wait for the kernel to finish. If *check* is True, the function is checked for aliasing artefacts. Returned *out* array is different from the input one because of the pyopencl.clmath behavior.

`syris.physics.`**`transfer_many`**(*objects*, *shape*, *pixel_size*, *energy*, *exponent=False*, *offset=None*, *queue=None*, *out=None*, *t=None*, *check=True*, *block=False*)

Compute transmission from more *objects*. If *exponent* is True, compute only the exponent, if it is False, evaluate the exponent. Use *shape* (y, x), *pixel_size*, *energy*, *offset* as (y, x), OpenCL command *queue*, *out* array, time *t*, check the sampling if *check* is True and wait for OpenCL kernels if *block* is True. Returned *out* array is different from the input one because of the pyopencl.clmath behavior.

`syris.physics.`**`wavelength_to_energy`**(*wavelength*)

Convert wavelength [m-like] to energy [eV].

1.8 Experiments

Synchrotron radiation imaging experiments base module.

`class` `syris.experiments.`**`Experiment`**(*samples*, *source*, *detector*, *propagation_distance*, *energies*)

A virtual synchrotron experiment base class.

compute_intensity(*t_0*, *t_1*, *shape*, *pixel_size*, *queue=None*, *block=False*)
Compute intensity between times *t_0* and *t_1*.

get_next_time(*t*, *pixel_size*)
Get next time from *t* for all the samples.

make_sequence(*t_start*, *t_end*, *shape=None*, *shot_noise=True*, *amplifier_noise=True*, *source_blur=True*, *queue=None*)
Make images between times *t_start* and *t_end*.

make_source_blur(*shape*, *pixel_size*, *queue=None*, *block=False*)
Make geometrical source blurring kernel with *shape* (y, x) size and *pixel_size*. Use OpenCL command *queue* and *block* if True.

time
Total time of all samples.

1.9 Bodies

Bodies are used to model physical objects like samples, optical elements like gratings, etc. A `MovableBody` can be moved by using an instance of `syris.geometry.Trajectory`. It is also possible to move it by manipulating its transformation matrix directly. `CompositeBody` can contain multiple bodies in order to model complex motion patterns, e.g. a robotic arm.

1.9.1 Base

A base module for pysical bodies, which are optical elements having spatial extent.

class syris.bodies.base.**Body**(*material=None*)
An abstract body class with a *material*, which is a `syris.materials.Material` instance.

project(*shape*, *pixel_size*, *offset=None*, *t=None*, *queue=None*, *out=None*, *block=False*)
Project thickness at time *t* to the image plane of size *shape* which is either 1D and is extended to (n, n) or is 2D as HxW. *pixel_size* is the point size, also either 1D or 2D. *offset* is the physical spatial body offset as (y, x). *queue* is an OpenCL command queue, *out* is the pyopencl array used for result. If *block* is True, wait for the kernel to finish.

class syris.bodies.base.**CompositeBody**(*trajectory*, *orientation=array([0., 1., 0.]) * dimensionless*, *bodies=None*)
Class representing a body consisting of more sub-bodies. A composite body can be thought of as a tree structure with children representing another bodies.

add(*body*)
Add a body *body*.

all_bodies
All bodies inside this body recursively.

bind_trajectory(*pixel_size*)
Bind trajectory for *pixel_size*.

bodies
All bodies which are inside this composite body.

bounding_box
Get bounding box around all the bodies inside.

clear_transformation()
Clear all transformations.

direct_primitive_bodies
Return primitive bodies on the level immediately after this body's level.

furthest_point
Furthest point is 0 for composite object.

get_distance(*t_0*, *t_1*)
Return the translational and rotational travelled distance in time interval *t_0*, *t_1*.

get_maximum_dt(*pixel_size*)
Get the maximum delta time for which the body will not move more than *pixel_size* divided by the number of bodies because their movement can sum up constructively.

get_next_time(*t_0*, *pixel_size*, *xtol=1e-12*)
Get next time at which the body will have traveled *pixel_size*, the starting time is *t_0*. *xtol* is the absolute tolerance for bisection passed to `scipy.optimize.bisect()`.

move(*abs_time*, *clear=True*)
Move to a position of the body in time *abs_time*. If *clear* is true clear the transformation matrix first.

moved(*t_0*, *t_1*, *pixel_size*)
Return True if the body moves more than *pixel_size* in time interval *t_0*, *t_1*.

remove(*body*)
Remove body *body*.

remove_all()
Remove all sub-bodies.

restore_transformation_matrices()
Restore transformation matrices of all bodies.

rotate(*angle*, *vec*, *shift=None*)
Rotate the body by *angle* around vector *vec*, where *shift* is the translation which takes place before the rotation and -*shift* takes place afterward, resulting in the transformation TRT^-1. Sub-bodies are rotated with respect to their relative position to the composite body.

save_transformation_matrices()
Save transformation matrices of all bodies and return them in a dictionary {body: transform_matrix}.

time
The total trajectory time of the body and all its subbodies.

translate(*vec*)
Translate all sub-bodies by a vector *vec*.

class `syris.bodies.base.`**MovableBody**(*trajectory*, *material=None*, *orientation=array([0., 1., 0.]) * dimensionless*, *cache_projection=True*)
Class representing a movable body.

apply_transformation(*trans_matrix*)
Apply transformation given by the transformation matrix *trans_matrix* on the current transformation matrix.

bind_trajectory(*pixel_size*)
Bind trajectory for *pixel_size*.

bounding_box
Bounding box defining the extent of the body.

cache_projection
Whether or not projection cache is being used.

center
Center.

clear_transformation()
Clear all transformations.

furthest_point
The furthest point from body's center with respect to the scaling factor of the body.

get_distance(*t_0*, *t_1*)
Return the maximum principal axes translational and rotational travelled distance in time interval *t_0*, *t_1*.

get_maximum_dt(*pixel_size*)
Get the maximum delta time for which the body will not move more than *pixel_size* between any two time points.

get_next_time(*t_0*, *pixel_size*)
Get time from *t_0* when the body will have travelled more than *pixel_size*.

get_rescaled_transform_matrix(*units*, *coeff=1*)
The last column of the transformation matrix holds displacement information has SI units, convert those to the *units* specified, apply coefficient *coeff* and return a copy of the matrix.

last_position
Last position.

move(*abs_time*, *clear=True*)
Move to a position of the body in time *abs_time*. If *clear* is true clear the transformation matrix first.

moved(*t_0*, *t_1*, *pixel_size*, *bind=True*)
Return True if the body moves more than *pixel_size* in time interval *t_0*, *t_1*. If *bind* is True bind the trajectory to the specified *pixel_size*, otherwise use the trajectory as-is to compute an estimate.

position
Current position.

project(*shape*, *pixel_size*, *offset=None*, *t=None*, *queue=None*, *out=None*, *block=False*)
Project thickness at time *t* (if it is None no transformation is applied) to the image plane of size *shape* which is either 1D and is extended to (n, n) or is 2D as HxW. *pixel_size* is the point size, also either 1D or 2D. *offset* is the physical spatial body offset as (y, x). *queue* is an OpenCL command queue, *out* is the pyopencl array used for result. If *block* is True, wait for the kernel to finish.

rotate(*angle*, *axis*, *shift=None*)
Rotate the body by *angle* around vector *vec*, where *shift* is the translation which takes place before the rotation and -*shift* takes place afterward, resulting in the transformation TRT^-1.

translate(*vec*)
Translate the body by a vector *vec*.

update_projection_cache(*t=None*, *shape=None*, *pixel_size=None*, *offset=None*, *projection=None*)
Update projection cache with time *t*, *shape*, *pixel_size*, *offset* and *projection*.

1.9.2 Simple Bodies

A static body.

class `syris.bodies.simple.`**`StaticBody`**(*thickness*, *pixel_size*, *material=None*, *queue=None*)

A static body is defined by its projected *thickness*, which is a quantity and it is always converted to meters, thus the `project()` method always returns the projection in meters. *pixel_size* is the pixel size of the *thickness* and *material* is a `syris.materials.Material` instance. Use OpenCL command *queue*.

`get_next_time`(*t_0*, *distance*)

A simple body doesn't move, this function returns infinity.

`syris.bodies.simple.`**`make_grid`**(*n*, *period*, *width=array(1.) * m*, *thickness=array(1.) * m*, *pixel_size=array(1.) * m*, *material=None*, *queue=None*)

Make a rectangluar grid with shape (*n*, *n*), the bars are spaced *period* and are *width* in diameter. *thickness* is the projected thickness and *pixel_size*, *material* and *queue*, which is an OpenCL command queue, are used to create `StaticBody`.

`syris.bodies.simple.`**`make_sphere`**(*n*, *radius*, *pixel_size=array(1.) * m*, *material=None*, *queue=None*)

Make a sphere with image shape (*n*, *n*), *radius* and *pixel_size*. Sphere center is in n / 2 + 0.5, which means between two adjacent pixels. *pixel_size*, *material* and *queue*, which is an OpenCL command queue, are used to create `StaticBody`.

1.9.3 Isosurfaces

Bodies based on isosurfaces.

class `syris.bodies.isosurfaces.`**`MetaBall`**(*trajectory*, *radius*, *material=None*, *orientation=array([0., 1., 0.]) * dimensionless*)

"Metaball bodies are smooth blobs formed by summing density functions representing particular bodies.

`bounding_box`

Bounding box of the metaball.

`furthest_point`

Furthest point is twice the radius because of the influence region of the metaball.

`get_transform_const`()

Precompute the transformation constant which does not change for x,y position.

`pack`()

Pack the body into a structure suitable for OpenCL kernels. Packed units are in meters.

class `syris.bodies.isosurfaces.`**`MetaBalls`**(*trajectory*, *metaballs*, *orientation=array([0., 1., 0.]) * dimensionless*)

Composite body composed of metaballs.

`syris.bodies.isosurfaces.`**`get_format_string`**(*string*)

Get string in single or double precision floating point number format.

`syris.bodies.isosurfaces.`**`get_moved_groups`**(*bodies*, *t_0*, *t_1*, *distance*)

Filter only *bodies* which truly move in the time interval *t_0*, *t_1* more than *distance*. Return a set of moved groups, where a group is defined by the last composite body which holds only primitive bodies. If a primitive body is in the *bodies* it is included without further testing because if it didn't move it wouldn't be in the list.

`syris.bodies.isosurfaces.`**`project_metaballs`**(*metaballs*, *shape*, *pixel_size*, *offset=None*, *queue=None*, *out=None*, *block=False*)

Project a list of `MetaBall` on an image plane with *shape*, *pixel_size*. *offset* is the physical spatial body offset as (y, x). Use OpenCL *queue* and *out* pyopencl Array instance for returning the result. If *block* is True, wait for the kernel to finish.

`syris.bodies.isosurfaces.`**`project_metaballs_naive`**(*metaballs*, *shape*, *pixel_size*, *offset=None*, *z_step=None*, *queue=None*, *out=None*, *block=False*)

Project a list of `MetaBall` on an image plane with *shape*, *pixel_size*. *z_step* is the physical step in the z-dimension, if not specified it is the same as *pixel_size*. *offset* is the physical spatial body offset as (y, x). Use OpenCL *queue* and *out* pyopencl Array instance for returning the result. If *block* is True, wait for the kernel to finish.

1.9.4 Meshes

Bodies made from mesh.

`class` `syris.bodies.mesh.`**`Mesh`**(*triangles*, *trajectory*, *material=None*, *orientation=array([0., 1., 0.]) * dimensionless*, *iterations=1*, *center='bbox'*)

Rigid Body based on *triangles* which form a polygon mesh. The triangles are a 2D array with shape (3, N), where N / 3 is the number of triangles. One polygon is formed by three consecutive triangles, e.g. when:

```
triangles = [[Ax, Bx, Cx]
             [Ay, By, Cy]
             [Az, Bz, Cz]]
```

then A, B, C are one triangle's points. *iterations* are the number of iterations within one pixel which try to find an intersection. *center* determines the center of the local coordinates, it can be one of None, 'bbox', 'gravity' or a (x, y, z) tuple specifying an arbitrary point.

`areas`
Triangle areas.

`bounding_box`
Bounding box implementation.

`center_of_bbox`
The bounding box center.

`center_of_gravity`
Get body's center of gravity as (x, y, z).

`compute_slices`(*shape*, *pixel_size*, *queue=None*, *out=None*, *offset=None*)
Compute slices with *shape* as (z, y, x), *pixel_size*. Use *queue* and *out* for outuput. Offset is the starting point offset as (x, y, z).

`diff`
Smallest and greatest difference between all mesh points in all three dimensions. Returns ((min(dx), max(dx)), (min(dy), max(dy)), (min(dz), max(dz))).

`extrema`
Mesh extrema as ((x_min, x_max), (y_min, y_max), (z_min, z_max)).

`furthest_point`
Furthest point from the center.

`get_degenerate_triangles`(*eps=array(0.001) * deg*)
Get triangles which are close to be parallel with the ray in z-direction based on the current transformation matrix. *eps* is the tolerance for the angle between a triangle and the ray to be still considered parallel.

`max_triangle_x_diff`
Get the greatest x-distance between triangle vertices.

normals
Triangle normals.

num_triangles
Number of triangles in the mesh.

sort()
Sort triangles based on the greatest x-coordinate in an ascending order. Also sort vertices inside the triangles so that the greatest one is the last one, however, the position of the two remaining ones is not sorted.

transform()
Apply transformation *matrix* and return the resulting triangles.

triangles
Return current triangle mesh.

vectors
The triangles as B - A and C - A vectors where A, B, C are the triangle vertices. The result is transposed, i.e. axis 1 are x, y, z coordinates.

syris.bodies.mesh.**make_cube**()
Create a cube triangle mesh from -1 to 1 m in all dimensions.

syris.bodies.mesh.**read_blender_obj**(*filename*, *objects=None*)
Read blender wavefront *filename*, extract only *objects* which are object indices.

1.10 Devices

Lenses used in experiments.

class syris.devices.lenses.**Lens**(*magnification*, *na=None*, *f_number=None*, *focal_length=None*, *transmission_eff=1*, *sigma=None*)
Class holding lenses.

numerical_aperture
Lens numerical aperture.

Cameras used by experiments.

class syris.devices.cameras.**Camera**(*pixel_size*, *gain*, *dark_current*, *amplifier_sigma*, *bits_per_pixel*, *shape*, *quantum_efficiencies=None*, *wavelengths=None*, *exp_time=array(1.) * s*, *fps=array(1.) * 1/s*, *dtype=<class 'numpy.uint16'>*)
Base class representing a camera.

get_image(*photons*, *shot_noise=True*, *amplifier_noise=True*, *psf=True*, *queue=None*)
Get digital counts image from incoming *photons*. The resulting image is based on the incoming photons and dark current. We apply noise based on EMVA 1288 standard according to which the variance $\sigma_y^2 = K^2(\sigma_e^2 + \sigma_d^2) + \sigma_q^2$, where K is the system gain, σ_e^2 is the poisson- distributed shot noise variance, σ_d^2 is the normal distributed electronics noise variance and σ_q^2 is the quantization noise variance. If *shot_noise* is False don't apply it. If *amplifier_noise* is False don't apply it as well. If *psf* is False don't apply the point spread function.

get_quantum_efficiency(*wavelength*)
Get quantum efficiency [dimensionless] at *wavelength*.

syris.devices.cameras.**is_fps_feasible**(*fps*, *exp_time*)
Determine whether frame rate given by *fps* can be accomplished with the exposure time *exp_time*. It is only possible to set frame rates for which $exposure\ time <= 1/fps$.

`syris.devices.cameras.`**`make_pco_dimax`**`()`
: Make a pco.dimax camera.

Module for beam filters which cause light attenuation. Filters are assumed to be homogeneous, thus no phase change effects are introduced when a wavefield passes through them.

`class` `syris.devices.filters.`**`Filter`**
: Beam frequency filter.

  `get_next_time``(`*t_0*, *distance*`)`
  : A filter doesn't move, this function returns infinity.

`class` `syris.devices.filters.`**`GaussianFilter`**`(`*energies*, *center*, *sigma*, *peak_transmission=1*`)`
: Gaussian beam filter.

  `get_next_time``(`*t_0*, *distance*`)`
  : A filter doesn't move, this function returns infinity.

`class` `syris.devices.filters.`**`MaterialFilter`**`(`*thickness*, *material*`)`
: Beam frequency filter.

  `get_attenuation``(`*energy*`)`
  : Get attenuation at *energy*.

  `get_next_time``(`*t_0*, *distance*`)`
  : A filter doesn't move, this function returns infinity.

`class` `syris.devices.filters.`**`Scintillator`**`(`*thickness*, *material*, *light_yields*, *energies*, *luminescence*, *wavelengths*, *optical_ref_index*`)`
: Scintillator emits visible light when it is irradiated by X-rays.

  `d_wavelength`
  : Wavelength spacing.

  `get_conversion_factor``(`*energy*`)`
  : Get the conversion factor to convert X-ray photons to visible light photons [dimensionless].

  `get_light_yield``(`*energy*`)`
  : Get light yield at *energy* [1 / keV].

  `get_luminescence``(`*wavelength*`)`
  : Get luminescence at *wavelength* [1 / nm].

  `wavelengths`
  : Wavelengths for which the emission is defined.

Detector composed of a scintillator, a lens and a camera.

`class` `syris.devices.detectors.`**`Detector`**`(`*scintillator*, *lens*, *camera*`)`
: A detector consisting of a camera and an objective lens.

  `convert``(`*photons*, *energy*, *wavelengths=None*`)`
  : Convert X-ray *photons* at *energy* to visible light photons with *wavelengths*.

  `get_visible_attenuation``(`*wavelengths=None*`)`
  : Get the attenuation coefficient for visible light *wavelengths* [dimensionless].

X-ray sources at synchrotrons. They provide X-ray photons used for imaging. Synchrotron radiation sources provide high photon flux of photons with different energies, which form a spectrum characteristic for a given source type.

class syris.devices.sources.**BendingMagnet**(*electron_energy*, *el_current*, *magnetic_field*, *sample_distance*, *dE*, *size*, *pixel_size*, *trajectory*, *profile_approx=True*, *phase_profile='plane'*)

Bending magnet X-ray source.

critical_energy

Critical energy of the source is defined as .. math:

```
\epsilon_c [keV] = 0.665 E^2 [GeV] B[T]
```

gama

$\frac{E}{m_e c^2}$

get_flux(*photon_energy*, *vertical_angle*, *pixel_size*)

Get the photon flux coming from the source consisting of photons with *photon_energy* and get it at the vertical observation angle *vertical_angle*.

get_next_time(*t_0*, *distance*)

Get the next time when the source will have moved more than *distance*.

class syris.devices.sources.**FixedSpectrumSource**(*energies*, *flux*, *sample_distance*, *size*, *trajectory*, *pixel_size=None*, *phase_profile='plane'*)

get_flux(*photon_energy*, *vertical_angle*, *pixel_size*)

Get linearly interpolated flux at *photon_energy*.

class syris.devices.sources.**Wiggler**(*electron_energy*, *el_current*, *magnetic_field*, *sample_distance*, *dE*, *size*, *pixel_size*, *trajectory*, *num_periods*, *profile_approx=True*, *phase_profile='plane'*)

Wiggler source.

get_flux(*photon_energy*, *vertical_angle*, *pixel_size*)

Get the photon flux coming from the source consisting of photons with *photon_energy* and get it at the vertical observation angle *vertical_angle*.

class syris.devices.sources.**XRaySource**(*sample_distance*, *size*, *trajectory*, *phase_profile='plane'*)

apply_blur(*intensity*, *distance*, *pixel_size*, *queue=None*, *block=False*)

Apply source blur based on van Cittert-Zernike theorem at *distance*.

get_next_time(*t_0*, *distance*)

Get the next time when the source will have moved more than *distance*.

exception syris.devices.sources.**XRaySourceError**

X-ray source related exceptions.

syris.devices.sources.**make_topotomo**(*dE=None*, *trajectory=None*, *pixel_size=None*, *ring_current=array(200.) * mA*)

Make the TopoTomo bending magnet source located at ANKA, KIT. Use *dE* for energy spacing (1 keV if not specified), *trajectory* for simulating beam fluctuations. If it is None a (1024, 1024) window is used with the beam center in the middle and no fluctuations. *pixel_size* specifies the pixel spacing between the window points, if not specified 1 um is used. *ring_current* is the storage ring electric current.

1.11 OpenCL GPU Utilities

Utility functions concerning GPU programming.

`syris.gpu.util.`**`are_images_supported`**`()`
Is the INTENSITY|FLOAT image format supported?

`syris.gpu.util.`**`cache`**`(`*mem*, *shape*, *dtype*, *cache_type=1*`)`
Cache a device memory object *mem* with *shape* and numpy data type *dtype* on host or device based on *cache_type*.

`syris.gpu.util.`**`execute_profiled`**`(`*function*`)`
Execute a *function* which can be an OpenCL kernel or other OpenCL related function and profile it.

`syris.gpu.util.`**`get_all_varconvolutions`**`()`
Create all variable convolutions.

`syris.gpu.util.`**`get_array`**`(`*data*, *queue=None*`)`
Get pyopencl.array.Array from *data* which can be a numpy array, a pyopencl.array.Array or a pyopencl.Image. *queue* is an OpenCL command queue.

`syris.gpu.util.`**`get_cache`**`(`*buf*`)`
Get a device memory object from cache *buf*, which can reside either on host or on device.

`syris.gpu.util.`**`get_command_queues`**`(`*context*, *devices=None*, *queue_kwargs=None*`)`
Create command queues for each of the *devices* within a specified *context*. If *devices* is None, they are obtained from *context*. *queue_kwargs* are passed to the CommandQueue constructor.

`syris.gpu.util.`**`get_cpu_platform`**`()`
Get any platform with CPUs.

`syris.gpu.util.`**`get_cuda_platform`**`()`
Get the NVIDIA CUDA platform if any.

`syris.gpu.util.`**`get_event_duration`**`(`*event*, *start=4738*, *stop=4739*`)`
Get OpenCL event duration. *start* and *stop* define the OpenCL timer start and stop.

`syris.gpu.util.`**`get_gpu_platform`**`()`
Get any platform with GPUs.

`syris.gpu.util.`**`get_host`**`(`*data*, *queue=None*`)`
Get *data* as numpy.ndarray.

`syris.gpu.util.`**`get_image`**`(`*data*, *access=4*, *queue=None*`)`
Get pyopencl.Image from *data* which can be a numpy array, a pyopencl.array.Array or a pyopencl.Image. The image channel order is pyopencl.channel_order.INTENSITY and channel_type is pyopencl.channel_type.FLOAT. *access* is either pyopencl.mem_flags.READ_ONLY or pyopencl.mem_flags.WRITE_ONLY. *queue* is an OpenCL command queue.

`syris.gpu.util.`**`get_intel_platform`**`()`
Get the Intel platform if any.

`syris.gpu.util.`**`get_metaobjects_source`**`()`
Get source string for metaobjects creation.

`syris.gpu.util.`**`get_platform`**`(`*name*`)`
Get the first OpenCL platform which contains *name* as its substring.

`syris.gpu.util.`**`get_platform_by_device_type`**`(`*device_type*`)`
Get platform with specific device type (CPU, GPU, …).

`syris.gpu.util.`**`get_precision_header`**`()`
Return single or double precision vfloat definitions header.

`syris.gpu.util.`**`get_program`**`(`*src*`)`
Create and build an OpenCL program from source string *src*.

`syris.gpu.util.`**`get_source`**(*file_names*, *precision_sensitive=True*)

Get source by concatenating files from *file_names* list and apply single or double precision parametrization if *precision_sensitive* is True.

`syris.gpu.util.`**`get_varconvolution_source`**(*name*, *header=”*, *inputs=”*, *init=”*, *compute_outer=”*, *compute_inner=’weight = 1.0;’*, *after=”*, *cplx=False*, *only_kernel=False*)

Create a shift dependent convolution kernel function with *name*. *header* is an OpenCL code which is placed in the front of the source before the kernel function. *inputs* are additional kernel inputs (see opencl/varconvolution.in for the fixed ones), *init* is the kernel initialization code, *compute_outer* is called at every iteration of the outer (y) loop, *compute_inner* is called in the inner (x) loop. *after* is the code after both loops. If *cplx* is True, the complex version of the kernel is used. Pseudo-code of the OpenCL source for the noncomplex version will look like this:

```
*header*

kernel void *name* (read_only image2d_t input,
                    global vfloat *output,
                    const sampler_t sampler,
                    int2 window, *inputs*)
{
    int idx = get_global_id (0);
    int idy = get_global_id (1);
    int width = get_global_size (0);
    int i, j, tx, ty, imx, imy;
    vfloat value, weight, result = 0.0;

    *init*

    for (j = 0; j < window.y; j++) {
        ty = window.y - j - 1;
        imy = idy + j - window.y / 2;
        *compute_outer*
        for (i = 0; i < window.x; i++) {
            imx = idx + i - window.x / 2;
            value = read_imagef (input, sampler, (int2)(imx, imy)).x;
            tx = window.x - i - 1;
            *compute_inner*
            result += value * weight;
        }
    }

    *after*

    output[idy * width + idx] = result;
}
```

The complex version uses two inputs, *input_real* and *input_imag* which are also image2d_t instances. *compute_inner* must set the *weight* variable in order to apply the convolution kernel weight.

`syris.gpu.util.`**`get_varconvolve_disk`**(*normalized=True*, *smooth=True*, *only_kernel=False*)

Create variable circular kernel convolution, kernel sum is 1 if *normalized* is True, if *smooth* is True smooth out sharp edges of the disk. If *only_kernel* is True only the kernel is returned.

`syris.gpu.util.`**`get_varconvolve_gauss`**(*normalized=True*, *window_fwnm=1000*, *only_kernel=False*)

Create variable Gaussian convolution. The kernel sum is 1 if *normalized* is True, window is computed automatically for every x, y position in the original image based on the sigma at x, y and *window_fwnm* as 2 * sqrt(2 * log(*window_fwnm*)) * sigma. If *only_kernel* is True only the kernel is returned.

syris.gpu.util.**get_varconvolve_propagator**(*only_kernel=False*)
Create the variable propagator convolution. If *only_kernel* is True only the kernel is returned.

syris.gpu.util.**init_programs**()
Initialize all OpenCL kernels needed by syris.

syris.gpu.util.**make_opencl_defaults**(*platform_name=None*, *device_type=None*, *device_index=None*, *profiling=True*)
Create default OpenCL context from *platform_name* and a command queue based on *device_index* to the devices list. If None, all devices are used in the context. If *platform_name* is not specified and *device_type* is, get a platform which has devices of that type. If *profiling* is True enable it.

syris.gpu.util.**make_vcomplex**(*value*)
Make complex value for OpenCL based on the set floating point precision.

syris.gpu.util.**qmap**(*func*, *items*, *queues=None*, *args=()*, *kwargs=None*)
Apply *func* to *items* on multiple command queues. The function *func* should block until the execution on a device is finished, otherwise the command queue which is assigned to it might return to the pool of usable resources too soon and stall execution. Consider using another mechanism if *func* is a kernel, i.e. the multi gpu execution might be realized without threading, which is used here. *func* is a callable with signature func(item, queue, *args, **kwargs) where item is an item to be processed and queue is the OpenCL command queue to be used. *queues* are the command queues to be used for computation, if not specified, all the default ones are used. *args* is a list and *kwargs* a dictionary, both passed to *func*.

CHAPTER 2

Usage

2.1 Profiling GPU Code

Module for profiling OpenCL GPU code execution.

class syris.profiling.**DummyProfiler**
A profiler which does nothing for saving time.

add(*event*, *func_name=''*)
Does nothing with input arguments.

class syris.profiling.**ProfileReconstructor**(*file_name*, *str_units*)
Profile reconstructor which handles the profiling file created by `Profiler`.

get_data(*attr*)
Get data in a dictionary aggregated to a multidictionary. *attr* is a record attribute which will serve as a key to the top level of result dictionary. Return a dictionary in form {attr: {event_id: Event}}.

class syris.profiling.**Profiler**(*queues*, *file_name*)
An OpenCL GPU code PROFILER.

add(*event*, *func_name=''*)
Add an OpenCL *event* and function with name *func_name* into the PROFILER's queue.

run()
Run in a separate thread and serve the incoming events.

shutdown()
Wait for all events to finish and then stop the PROFILER loop.

syris.profiling.**plot**(*data*, *attribute*, *states*, *file_units*, *out_units*, *start_from=0*, *stop_at=1.7976931348623157e+308*, *delta=0.0*, *only_averages=False*)
Plot the profiling information, where

- *data* - a dictionary in the form {id: {event_id: values}}
- *attribute* - (event_id, device_id, queue_id)

- *states* - OpenCL Event states to use as beginning and end
- *file_units* - units used in the profiling file
- *out_units* - units used for output
- *start_from* - plot events started after start_from
- *stop_at* - plot events started before stop_at
- *delta* - plot only events with duration >= delta
- *only_averages* - outputs only the average timings

2.2 Examples

In this section we describe the examples which come directly with *syris* in order to demonstrate its usage.

2.2.1 Composite body

composite_body.py

Composite body example

Here we show how to use a composite body in order to move groups of objects around. This is possible in two ways, either manually by translating and rotating the composite body, or automatically by using a trajectory.

Manual

This example shows manual rotation of a grid of spheres with different radii around one of the spheres. CompositeObject is used to simplify the transformations workflow.

Trajectory

This example has the same result as the previous one but achieved by trajectories.

Subtrajectories

This example shows a circular global motion followed by the whole composite body and its sub-bodies, which are cuboids following their own local linear trajectories. The sub-bodies further move along their own trajectories.

2.2.2 Edge enhancement

edge_enhancement.py

Edge enhancement caused by free-space propagation. Control the accuracy by the –supersampling option. If this value is too low (1), the propagators are not resolved correctly and the resulting images contain artefacts. Increase this value to e.g. 4 and you will see how the propagators and the results change.

2.2.3 Energy Filter

energy_filter.py

Energy filter based on Gaussian profile.

2.2.4 Experiment

experiment.py

Experiment example.

`examples.experiment.`**`main`**`()`

Parse command line arguments and execute one of the experiments.

`examples.experiment.`**`make_devices`**`(`*n*, *energies*, *camera=None*, *highspeed=True*, *scintillator=None*`)`

Create devices with image shape (*n*, *n*), X-ray *energies*, *camera* and use the high speed setup if *highspeed* is True.

2.2.5 Fresnel Propagation

propagator.py

Show different propagators.

`examples.propagator.`**`main`**`()`

2.2.6 Fresnel Propagation Accuracy

fresnel.py

Comparison of analytical and numerical Fresnel diffraction pattern. The object is a square aperture from Introduction to Fourier Optics by J. W. Goodmann, 2nd edition.

`examples.fresnel.`**`crop_to_aperture`**`(`*image*, *w*, *ps*`)`

Crop *image* to 2x aperture width.

`examples.fresnel.`**`main`**`()`

Main function.

`examples.fresnel.`**`parse_args`**`()`

Parse command line arguments.

`examples.fresnel.`**`propagate_analytically`**`(`*n*, *w*, *ps*, *d*, *lam*`)`

Propagate square aperture analytically.

`examples.fresnel.`**`propagate_numerically`**`(`*n*, *w*, *ps*, *d*, *lam*`)`

Propagate square aperture numerically.

2.2.7 Mesh Bodies

mesh.py

Mesh projection and slice.

`examples.mesh.`**`main`**`()`

Main function.

`examples.mesh.`**`parse_args`**`()`
Parse command line arguments.

2.2.8 Laminography of Samples Defined by Meshes

mesh_scan.py

Laminography data set generation with mesh geometry.

`examples.mesh_scan.`**`log_attributes`**`(`*obj*`)`
Log object *obj* attributes.

`examples.mesh_scan.`**`make_ground_truth`**`(`*args*, *shape*, *mesh*`)`
Shape is (y, x), so the total number of slices is y.

`examples.mesh_scan.`**`scan`**`(`*shape*, *ps*, *axis*, *mesh*, *angles*, *prefix*, *lamino_angle=array(45.) * deg*, *index=0*, *num_devices=1*, *shift_coeff=10000.0*, *ss=1*`)`
Make a scan of tomographic angles. *shift_coeff* is the coefficient multiplied by pixel size which shifts the triangles to get rid of faulty pixels.

2.2.9 Metaballs

metaballs.py

2.2.10 Paganin Phase Retrieval

paganin.py

Show forward phase contrast simulation and backward phase retrieval using the Paganin method[1].

[1] Paganin, David, et al. "Simultaneous phase and amplitude extraction from a single defocused image of a homogeneous object." Journal of microscopy 206.1 (2002): 33-40.

2.2.11 Platform Benchmark

speed.py

Code speed on a specific platform.

2.2.12 Multi GPU Speedup

multigpu.py

Show multi-device speedup on a problem of size n x m^k, where n is the number of pixels to compute, m is the base number of operations per pixel powered to k.

2.2.13 X-ray Source

source.py

An X-ray source example.

2.2.14 X-ray Source Blur

source_blur.py

Source blur example.

2.2.15 Simple

simple.py

Simple propagation example.

2.2.16 Tomographic Rotation

tomographic_rotation.py

Example of a trajectory which simulates tomographic rotation.

2.2.17 4D Tomography

tomography_4D.py

Simple 4D tomography example. Two cubes rotate around the tomographic rotation axis and at the same time move along y-axis. The total vertical displacement between rotation start and end is the cube edge. This leads to an "incomplete" data set with increasingly more missing data in the sinogram space from top to bottom. There is exactly one complete sinogram, the middle one.

2.2.18 Trajectory

trajectory.py

Trajectory and motion example.

`examples.trajectory.`**`create_sample`**`(`*n*, *ps*, *radius=None*, *velocity=None*, *x_ends=None*, *y_ends=None*`)`
    Crete a metaball with a sine trajectory.

`examples.trajectory.`**`main`**`()`

`examples.trajectory.`**`make_circle`**`(`*n=128*, *axis='z'*, *overall_angle=None*, *phase_shift=None*`)`
    Axis specifies the axis of rotation, which can be 'x', 'y' or 'z'.

`examples.trajectory.`**`parse_args`**`()`
    Parse command line arguments.

2.2.19 Transformation Order

transformation.py

Demonstrates the order of transformations.

`examples.transformation.`**`main`**`()`
    Script execution.

`examples.transformation.`**`print_rounded`**`(`*vector*, *decimals=2*`)`
    Print a roundded version of *vector*.

`examples.transformation.`**`transform`**(*point=array([1., 0., 0.]) * m*, *x_rot=array(90.) * deg*, *y_rot=array(90.) * deg*, *z_rot=array(0.) * deg*)

Transform *point* by a series of rotations, *x_rot* around x axis and so on for *y_rot* and *z_rot*.

2.2.20 Transmission Function Sampling

transmission_function_sampling.py

Aliasing of the transmission function of a wedge which projection is computed as f(x, y) = x. Delta is chosen in such a way that it causes phase shift between two adjacent pixels 2Pi in case of no supersampling. Thus, The transmission function of the wedge along x has phase 0, 2Pi, 4Pi, . . . , which is lost due to insufficient pixel spacing in case of no supersampling.

The used material is a pure phase material, i.e. beta = 0. The results are the real part of the T(x, y), which is cos(-2 Pi / lambda x delta).

2.2.21 Linear Shift Dependent Convolution

varconvolution.py

Example showing variable convolution.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index