raytraverse Documentation

Release 1.4.3

Stephen Wasilewski

Apr 29, 2024

COMMAND LINE INTERFACE

raytraverse is a workflow for climate based daylight simulation for the evaluation of architectural spaces. Built around a wavelet guided adaptive sampling strategy, raytraverse can fully explore the daylight conditions throughout a space with efficient use of processing power and storage space. The code base has been split into three seperate packages: this one as well as two others that provide much of the backend functionality. craytraverse (https://pypi.org/project/craytraverse) is doccumented here and contains code to run radiance simulations from within python. raytools is documented here: https://raytools.readthedocs.io/en/latest/ and contains a lot of the library functions with working with hdr and ray based data. It is its' own repository because it includes stand alone tools for evaluating hdr images for glare metrics.

- Free software: Mozilla Public License 2.0 (MPL 2.0)
- Documentation: https://raytraverse.readthedocs.io/en/latest/.

CHAPTER

ONE

INSTALLATION

The easiest way to install raytraverse is with pip:

```
pip install --upgrade pip setuptools wheel
pip install raytraverse
```

or if you have cloned this repository:

```
cd path/to/this/file
pip install .
```

while raytraverse installs with the necessary essentials of radiance, it is recommended to also install radiance (see: https://github.com/LBNL-ETA/Radiance/releases and make sure you or the installer also sets the $RAYPATH variable) this is especially important if your material or light source definitions rely on .cal files distributed with radiance, such as perezlum.cal, window.cal, etc. Missing .cal files or other scene errors can cause raytraverse traverse to abort with cryptic eror messages (the number value is not meaningful and will be different every time):

```
python: : Unknown error -1624667552
```

If you encounter such an error, make sure your scene is valid in your current environment, using rvu, rpict, or rtrace.

1.1 Windows

Currently raytraverse is only compatible with macOS and linux operating systems. One way to use raytraverse on a Windows machine is with Docker. In adddition to the Docker installation, this process will require about 2.5 GB of disk space.

1. Install Docker from: https://www.docker.com/products/docker-desktop/ (click on "Windows") and then follow the installation instructions.
2. Open the newly installed Docker Desktop application (you do not need to sign in or create an account)
3. In an empty directory make a file called Dockerfile_first with the following contents:

```
# syntax=docker/dockerfile:1
# docker build -f Dockerfile_first . --tag raytraverse:latest
FROM python:3.9

WORKDIR /build
RUN apt-get update
RUN apt-get -y install man

SHELL ["/bin/bash", "-c"]
RUN pip3 install raytraverse
RUN curl -s https://api.github.com/repos/LBNL-ETA/Radiance/releases\?per_page\=1␣
```

(continues on next page)

(continued from previous page)

```
↪\
| grep "browser_download_url.*Linux.zip" | cut -d: -f2,3 | tr -d \" | wget -i -
RUN unzip Radiance_*_Linux.zip
RUN tar -xzf radiance-*-Linux.tar.gz
WORKDIR /radiance
RUN rm -rf bin lib man
RUN mv /build/radiance-*-Linux/usr/local/radiance/* ./
RUN rm -rf /build

ENV RAYPATH=.:/radiance/lib
ENV MANPATH=/radiance/man
ENV PATH=/radiance/bin:$PATH
RUN raytraverse --help
WORKDIR /working
```

4. in a command prompt navigate to this folder and run the following to create a docker image with raytraverse and radiance installed:

```
docker build - --tag raytraverse:latest < Dockerfile_first
```

5. To use raytraverse, navigate to a local folder that contains all necessary files (radiance scene files, sky data, etc.).

6. Now, in this folder (note that you may need to change the syntax of "$(pwd)" to be compatible with your shell, this works with the basic windows command prompt):

```
docker run -it --name rayt --mount type=bind,source="$(pwd)",target=/working␣
↪raytraverse /bin/bash
```

7. You now have a linux/bash command prompt in an environment with raytraverse, radiance, and python 3.9 installed. The currrent directory will be named "working" within the linux environment and is a shared resource with the host (changes on the host side are immediately seen in the container and vice versa). When you are finished, exit the linux shell ("exit"), then in the (now) windows command prompt:

```
docker rm rayt
```

8. for ease of use, you can put these to lines in a .bat file somewhere in your execution PATH, just make sure that docker desktop is running before calling:

```
docker run -it --name rayt --mount type=bind,source="$(pwd)",target=/working␣
↪raytraverse /bin/bash
docker rm rayt
```

9. to update raytraverse, the process is similar to step 4, but with a slightly different dockerfile:

```
# syntax=docker/dockerfile:1
# docker build -f Dockerfile_update . --tag raytraverse:latest
FROM raytraverse:latest

WORKDIR /build

SHELL ["/bin/bash", "-c"]
RUN pip3 install --upgrade --no-deps craytraverse
RUN pip3 install --upgrade --no-deps clasp
RUN pip3 install --upgrade --no-deps raytraverse
RUN curl -s https://api.github.com/repos/LBNL-ETA/Radiance/releases\?per_page\=1␣
↪\
| grep "browser_download_url.*Linux.zip" | cut -d: -f2,3 | tr -d \" | wget -i -
```

(continues on next page)

(continued from previous page)

```
RUN unzip Radiance_*_Linux.zip
RUN tar -xzf radiance-*-Linux.tar.gz
WORKDIR /radiance
RUN rm -rf bin lib man
RUN mv /build/radiance-*-Linux/usr/local/radiance/* ./
RUN rm -rf /build

ENV RAYPATH=.:/radiance/lib
ENV MANPATH=/radiance/man
ENV PATH=/radiance/bin:$PATH
RUN raytraverse --help
WORKDIR /working
```

and this command:

```
docker build - --tag raytraverse:latest < Dockerfile_update
```

10. see the Docker settings for information about resource allocation to the docker container.

CHAPTER

TWO

USAGE

raytraverse includes a complete command line interface with all commands nested under the *raytraverse* parent command enter:

```
raytraverse --help
```

raytraverse also exposes an object oriented API written primarily in python. calls to Radiance are made through Renderer objects that wrap the radiance c source code in c++ classes, which are made available in python with pybind11. see craytraverse (https://pypi.org/project/craytraverse/).

For complete documentation of the API and the command line interface either use the Documentation link included above or:

```
pip install -r docs/requirements.txt
make docs
```

to generate local documentation.

CHAPTER

THREE

COMMAND LINE INTERFACE

The raytraverse command provides command line access to executing common tasks. The best way to manage all of the options is with a .cfg file. First, generate a template:

```
raytraverse --template > options.cfg
```

and then edit the options for each file. for example, here is a simplified configuration for a low accuracy sample simulation, assuming a model scaled in meters where plane.rad is betwee 4m and 10m on each side:

```
[shared]
weather_file = weather.epw

[raytraverse_scene]
out = outdir
scene = room.rad

[raytraverse_area]
ptres = 2.0
zone = plane.rad

[raytraverse_suns]
epwloc = True
loc = ${shared:weather_file}

[raytraverse_skydata]
wea = ${shared:weather_file}
skyres = 10

[raytraverse_skyengine]
accuracy = 2.0
skyres = 10

[raytraverse_sunengine]
accuracy = 2.0
rayargs = -ab 0
nlev = 5

[raytraverse_skyrun]
accuracy = 2.0
edgemode = reflect
nlev = 2

[raytraverse_sunrun]
accuracy = 3.0
edgemode = reflect
nlev = 2
```

(continues on next page)

(continued from previous page)

```
srcaccuracy = 2.0
srcnlev = 2

[raytraverse_images]
basename = results
blursun = True
interpolate = highc
res = 800
resampleview = True
sdirs = None
sensors = None
skymask = 0:24

[raytraverse_evaluate]
basename = results
sdirs = None
sensors = None
skymask = None

[raytraverse_pull]
col = metric point
gridhdr = True
ofiles = results
skyfill = ${shared:weather_file}
viewfilter = 0
```

and then from the command line run:

```
raytraverse -c options.cfg skyrun directskyrun sunrun evaluate pull
```

3.1 raytraverse.scene

3.1.1 BaseScene

class raytraverse.scene.**BaseScene**(*outdir*, *scene=None*, *frozen=True*, *formatter=<class 'raytraverse.formatter.formatter.Formatter'>*, *reload=True*, *overwrite=False*, *log=True*, *loglevel=10*, *utc=False*)

Bases: `object`

container for scene description

Parameters

- **outdir** (*str*) – path to store scene info and output files
- **scene** (*str, optional (required if not reload)*) – space separated list of radiance scene files (no sky) or octree
- **frozen** (*bool, optional*) – create a frozen octree
- **formatter** (raytraverse.formatter.Formatter, *optional*) – intended renderer format
- **reload** (*bool, optional*) – if True attempts to load existing scene files in new instance overrides 'overwrite'
- **overwrite** (*bool, optional*) – if True and outdir exists, will overwrite, else raises a FileExistsError

- **log** (*bool, optional*) – log progress events to outdir/log.txt
- **loglevel** (*int, optional*) – maximum sampler level to log

property scene

render scene files (octree)

Getter
Returns this samplers's scene file path

Setter
Sets this samplers's scene file path and creates run files

Type
str

reflection_search_scene()

log(*instance*, *message*, *err=False*, *level=0*)

print a message to the log file or stderr

Parameters

- **instance** (*Any*) – the parent class for the progress bar
- **message** (*str, optional*) – the message contents
- **err** (*bool, optional*) – print to stderr instead of self._logf
- **level** (*int, optional*) – the nested level of the message

progress_bar(*instance*, *iterable=None*, *message=None*, *total=None*, *level=0*, *workers=False*)

generate a tqdm progress bar and concurrent.futures Executor class

Parameters

- **instance** (*Any*) – the parent class for the progress bar
- **iterable** (*Sequence, optional*) – passed to tqdm, the sequence to loop over
- **message** (*str, optional*) – the prefix message for the progress bar
- **total** (*int, optional*) – the number of expected iterations (when interable is none)
- **level** (*int, optional*) – the nested level of the progress bar
- **workers** (*Union[bool, str], optional*) – if "thread/threads/t" returns a ThreadPoolExecutor, else if True returns a ProcessPoolExecutor.

Returns
a subclass of tqdm that decorates messages and has a pool property for multiprocessing.

Return type
TStqdm

Examples

with an iterable:

```
for i in self.scene.progress_bar(self, np.arange(10)):
    do stuff...
```

with workers=True:

with self.scene.progress_bar(self, total=len(jobs) workers=True) as pbar:
exc = pbar.pool do stuff... pbar.update(1)

3.1.2 Scene

class raytraverse.scene.**Scene**(*outdir*, *scene=None*, *frozen=True*, *formatter=<class 'raytraverse.formatter.radianceformatter.RadianceFormatter'>*, ***kwargs*)

Bases: *BaseScene*

container for radiance scene description

WARNING!! if scene parameter contains and instance primitive, sunsampler will throw a segmentation fault when it tries to change the source. As scene instanciation will make a frozen octree, it is better to feed complete scene description files, or an octree.

Parameters

- **outdir** (*str*) – path to store scene info and output files
- **formatter** (*raytraverse.formatter.RadianceFormatter, optional*) – intended renderer format

reflection_search_scene()

reflection_search(*vecs*, *res=5*)

source_scene(*srcfile*, *srcname*)

3.1.3 ImageScene

class raytraverse.scene.**ImageScene**(*outdir*, *scene=None*, *reload=True*, *log=False*)

Bases: *BaseScene*

scene for image sampling

Parameters

- **outdir** (*str*) – path to store scene info and output files
- **scene** (*str, optional*) – image file (hdr format -vta projection)

3.2 raytraverse.mapper

3.2.1 SkyMapper

class raytraverse.mapper.**SkyMapper**(*loc=None*, *skyro=0.0*, *sunres=9*, *name='suns'*, *jitterrate=0.5*)

Bases: AngularMixin, Mapper

translate between world direction vectors and normalized UV space for a given view angle. pixel projection yields equiangular projection

Parameters

- **loc** (*any, optional*) – can be a number of formats:
  1. either a numeric iterable of length 3 (lat, lon, mer) where lat is +west and mer is tz*15 (matching gendaylit).
  2. an array (or tsv file loadable with np.loadtxt) of shape (N,3), (N,4), or (N,5):
     a. 2 elements: alt, azm (angles in degrees)
     b. 3 elements: dx,dy,dz of sun positions
     c. 4 elements: alt, azm, dirnorm, diffhoriz (angles in degrees)

d. 5 elements: dx, dy, dz, dirnorm, diffhoriz.

3. path to an epw or wea formatted file
4. None (default) all possible sun positions are considered self.in_solarbounds always returns True

in the case of a geo location, sun positions are considered valid when in the solar transit for that location. for candidate options, sun positions are drawn from this set (with one randomly chosen from all candidates within bin.

- **skyro** (*float, optional*) – counterclockwise sky-rotation in degrees (equivalent to clockwise project north rotation)
- **sunres** (*float, optional*) – initial sampling resolution for suns
- **name** (*str, optional*)

property skyro

property loc

property solarbounds

property candidates

in_solarbounds(*xyz*, *level=0*, *include='any'*)

for checking if src direction is in solar transit

Parameters

- **xyz** (*np.array*) – source directions
- **level** (*int*) – for determining patch size, 2**level resolution from sunres
- **include** (*{'center', 'all', 'any'}, optional*) – boundary test condition. 'center' tests uv only, 'all' requires for corners of box centered at uv to be in, 'any' requires atleast one corner. 'any' is the least restrictive and 'all' is the most, but with increasing levels 'any' will exclude more positions while 'all' will exclude less (both approaching 'center' as level -> N)

Returns
result – Truth of ray.src within solar transit

Return type
np.array

shape(*level=0*)

solar_grid(*jitter=True*, *level=0*, *masked=True*)

generate a grid of solar positions

Parameters

- **jitter** (*bool, optional*) – if None, use the instance default, if True jitters point samples within stratified grid
- **level** (*int, optional*) – sets the resolution of the grid as a power of 2 from sunress
- **masked** (*bool, optional*) – apply in_solarbounds to suns before returning

Returns
shape (N, 3)

Return type
np.array

3.2.2 PlanMapper

class raytraverse.mapper.**PlanMapper**(*area*, *ptres=1.0*, *rotation=0.0*, *zheight=None*, *name='plan'*, *jitterrate=0.5*, *autorotate=False*, *autogrid=None*)

Bases: Mapper

translate between world positions on a horizontal plane and normalized UV space for a given view angle. pixel projection yields a parallel plan projection

Parameters

- **area** (*str np.array, optional*) – radiance scene geometry defining a plane to sample, tsv file of points to generate bounding box, or np.array of points. if area is a radiance scene, or a 3d array of points, vertices are used to directly define the borders of the planmapper. if the array is 2d (or loaded from a tsv) points are used to generate an offset (see ptres) convexhull. points are stored as candidates for use as a static point sampler (jitter=false and level=0).
- **ptres** (*float, optional*) – resolution for considering points duplicates, border generation (1/2) and add_grid(). updateable
- **rotation** (*float, optional*) – positive Z rotation for point grid alignment
- **zheight** (*float, optional*) – override calculated zheight
- **name** (*str, optional*) – plan mapper name used for output file naming
- **jitterrate** (*float, optional*) – proportion of cell to jitter within
- **autorotate** (*bool, optional*) – if true set rotation based on long axis of area geometry
- **autogrid** (*int, optional*) – if given, autoset ptres based on this minimum number of points at level 0 along the minimum dimemsion (width or height)

ptres

point resolution for area look ups and grid

Type
float

property dxyz

(float, float, float) central view point

property rotation

ccw rotation (in degrees) for point grid on plane

Type
float

property bbox

boundary frame for translating between coordinates [[xmin ymin zmin] [xmax ymax zmax]]

Type
np.array

update_bbox(*plane*, *level=0*, *updatez=True*)

handle bounding box generation from plane or points

uv2xyz(*uv*, *stackorigin=False*)

transform from mapper UV space to world xyz

in_view_uv(*uv*, *indices=True*, ***kwargs*)

in_view(*vec*, *indices=True*)

check if point is in boundary path

Parameters

- **vec** (*np.array*) – xyz coordinates, shape (N, 3)
- **indices** (*bool, optional*) – return indices of True items rather than boolean array

Returns
mask – boolean array, shape (N,)

Return type
np.array

header(***kwargs*)

borders()

world coordinate vertices of planmapper boundaries

property boundary

bbox_vertices(*offset=0*, *close=False*)

shape(*level=0*)

point_grid(*jitter=True*, *level=0*, *masked=True*, *snap=None*)

generate a grid of points

Parameters

- **jitter** (*bool, optional*) – if None, use the instance default, if True jitters point samples within stratified grid
- **level** (*int, optional*) – sets the resolution of the grid as a power of 2 from ptres
- **masked** (*bool, optional*) – apply in_view to points before returning
- **snap** (*int, optional*) – level to snap samples to when jitter=False should be > level

Returns
shape (N, 3)

Return type
np.array

point_grid_uv(*jitter=True*, *level=0*, *masked=True*, *snap=None*)

add a grid of UV coordinates

Parameters

- **jitter** (*bool, optional*) – if None, use the instance default, if True jitters point samples within stratified grid
- **level** (*int, optional*) – sets the resolution of the grid as a power of 2 from ptres
- **masked** (*bool, optional*) – apply in_view to points before returning
- **snap** (*int, optional*) – level to snap samples to when jitter=False should be > level

Returns
shape (N, 2)

Return type
np.array

3.2.3 MaskedPlanMapper

class raytraverse.mapper.**MaskedPlanMapper**(*pm*, *valid*, *level*)

Bases: *PlanMapper*

translate between world positions on a horizontal plane and normalized UV space for a given view angle. pixel projection yields a parallel plan projection

Parameters

- **pm** (raytraverse.mapper.PlanMapper) – the source mapper to copy
- **valid** (*np.array*) – a list of valid points used to make a mask, grid cells not represented by one of valid will be masked
- **level** (*int, optional*) – the level at which to grid the valid candidates

update_mask(*valid*, *level*)

in_view_uv(*uv*, *indices=True*, *usemask=True*)

3.3 raytraverse.formatter

3.3.1 Formatter

class raytraverse.formatter.**Formatter**

Bases: object

scene formatter readies scene files for simulation, must be compatible with desired renderer.

comment = '#'

line comment character

scene_ext = ''

extension for renderer scene file

static make_scene(*scene_files*, *out*, *frozen=True*)

compile scene

static get_scene(*scene*)

static get_skydef(*color=None*, *ground=True*, *name='skyglow'*)

assemble sky definition

static get_sundef(*vec*, *color*, ***kwargs*)

assemble sun definition

3.3.2 RadianceFormatter

class raytraverse.formatter.**RadianceFormatter**

Bases: *Formatter*

scene formatter readies scene files for simulation, must be compatible with desired renderer.

comment = '#'

line comment character

scene_ext = '.oct'

extension for renderer scene file

static make_scene(*scene_files*, *out*, *frozen=True*)

compile scene

static get_scene(*scene*)

recover scene file paths from compiled octree

Parameters
scene (*octree file*)

Returns

- **files** (*string to use in new octree generation. -i prepended before*)
- *each actree*
- **frozen** (*if result will be a frozen octree*)

static get_skydef(*color=(0.96, 1.004, 1.118)*, *ground=True*, *name='skyglow'*, *mod='void'*, *groundname=None*, *groundcolor=(1, 1, 1)*)

assemble sky definition

static get_sundef(*vec*, *color*, *size=0.5333*, *mat_name='solar'*, *mat_id='sun'*)

assemble sun definition

3.4 raytraverse.renderer

3.4.1 ImageRenderer

class raytraverse.renderer.**ImageRenderer**(*scene*, *viewmapper=None*, *method='linear'*, *color=False*, *uv=False*)

Bases: object

interface to treat image data as the source for ray tracing results

not implemented as a singleton, so multiple instances can exist in parallel.

Parameters

- **scene** (*str*) – path to hdr image file with projection matching ViewMapper
- **viewmapper** (*raytraverse.mapper.ViewMapper, optional*) – if None, assumes 180 degree angular fisheye (vta)
- **method** (*str, optional*) – passed to scipy.interpolate.RegularGridInterpolator

run(**args*, ***kwargs*)

alias for call, for consistency with SamplerPt classes for nested dimensions of evaluation

3.4.2 SpRenderer

class raytraverse.renderer.**SpRenderer**(*rayargs=None*, *scene=None*, *nproc=None*, *default_args=True*)

Bases: object

sub-process renderer for calling external executables

args = None

scene = None

name = 'rtrace'

defaultargs = ''

nproc = None

run(**args*, ***kwargs*)
alias for call, for consistency with SamplerPt classes for nested dimensions of evaluation

classmethod get_default_args()

classmethod reset()
reset engine instance and unset associated attributees

classmethod set_args(*args*, *nproc=None*)
prepare arguments to call engine instance initialization

Parameters

- **args** (*str*) – rendering options
- **nproc** (*int, optional*) – cpu limit

classmethod load_scene(*scene*)
load octree file to engine instance

Parameters
scene (*str*) – path to octree file

Raises
ValueError: – can only be called after set_args, otherwise engine instance will abort.

3.5 raytraverse.sky

3.5.1 skycalc

functions for loading sky data and computing sun position

raytraverse.sky.skycalc.**read_epw**(*epw*)
read daylight sky data from epw or wea file

Returns
out – (month, day, hour, dirnorn, difhoriz)

Return type
np.array

raytraverse.sky.skycalc.**read_epw_full**(*epw*, *columns=None*)

Parameters

- **epw**
- **columns** (*list, optional*) – integer indices or keys of columns to return

Return type
requested columns from epw as np.array shape (8760, N)

raytraverse.sky.skycalc.**get_loc_epw**(*epw*, *name=False*)
get location from epw or wea header

raytraverse.sky.skycalc.**sunpos_utc**(*timesteps*, *lat*, *lon*, *builtin=True*)
Calculate sun position with local time

Calculate sun position (altitude, azimuth) for a particular location (longitude, latitude) for a specific date and time (time is in UTC)

Parameters

- **timesteps** (*np.array(datetime.datetime)*)
- **lon** (*float*) – longitude in decimals. West is +ve
- **lat** (*float*) – latitude in decimals. North is +ve
- **builtin** (*bool*) – use skyfield builtin timescale

Returns

- *(skyfield.units.Angle, skyfield.units.Angle)*
- *altitude and azimuth in degrees*

raytraverse.sky.skycalc.**row_2_datetime64**(*ts*, *year=2020*)

raytraverse.sky.skycalc.**datetime64_2_datetime**(*timesteps*, *mer=0.0*)

convert datetime representation and offset for timezone

Parameters

- **timesteps** (*np.array(np.datetime64)*)
- **mer** (*float*) – Meridian of the time zone. West is +ve

Return type
np.array(datetime.datetime)

raytraverse.sky.skycalc.**sunpos_degrees**(*timesteps*, *lat*, *lon*, *mer*, *builtin=True*, *ro=0.0*)

Calculate sun position with local time

Calculate sun position (altitude, azimuth) for a particular location (longitude, latitude) for a specific date and time (time is in local time)

Parameters

- **timesteps** (*np.array(np.datetime64)*)
- **lon** (*float*) – longitude in decimals. West is +ve
- **lat** (*float*) – latitude in decimals. North is +ve
- **mer** (*float*) – Meridian of the time zone. West is +ve
- **builtin** (*bool, optional*) – use skyfield builtin timescale
- **ro** (*float, optional*) – ccw rotation (project to true north) in degrees

Returns
Sun position as (altitude, azimuth) in degrees

Return type
np.array([float, float])

raytraverse.sky.skycalc.**sunpos_radians**(*timesteps*, *lat*, *lon*, *mer*, *builtin=True*, *ro=0.0*)

Calculate sun position with local time

Calculate sun position (altitude, azimuth) for a particular location (longitude, latitude) for a specific date and time (time is in local time)

Parameters

- **timesteps** (*np.array(np.datetime64)*)
- **lon** (*float*) – longitude in decimals. West is +ve
- **lat** (*float*) – latitude in decimals. North is +ve
- **mer** (*float*) – Meridian of the time zone. West is +ve
- **builtin** (*bool*) – use skyfield builtin timescale

- **ro** (*float, optional*) – ccw rotation (project to true north) in radians

Returns
Sun position as (altitude, azimuth) in radians

Return type
np.array([float, float])

raytraverse.sky.skycalc.**sunpos_xyz**(*timesteps*, *lat*, *lon*, *mer*, *builtin=True*, *ro=0.0*)

Calculate sun position with local time

Calculate sun position (altitude, azimuth) for a particular location (longitude, latitude) for a specific date and time (time is in local time)

Parameters

- **timesteps** (*np.array(np.datetime64)*)
- **lon** (*float*) – longitude in decimals. West is +ve
- **lat** (*float*) – latitude in decimals. North is +ve
- **mer** (*float*) – Meridian of the time zone. West is +ve
- **builtin** (*bool*) – use skyfield builtin timescale
- **ro** (*float, optional*) – ccw rotation (project to true north) in degrees

Returns
Sun position as (x, y, z)

Return type
np.array

raytraverse.sky.skycalc.**generate_wea**(*ts*, *wea*, *interp='linear'*)

raytraverse.sky.skycalc.**coeff_lum_perez**(*sunz*, *epsilon*, *delta*, *catn*)

matches coeff_lum_perez in gendaylit.c

raytraverse.sky.skycalc.**perez_apply_coef**(*coefs*, *cgamma*, *dz*)

raytraverse.sky.skycalc.**perez_lum_raw**(*tp*, *dz*, *sunz*, *coefs*)

matches calc_rel_lum_perez in gendaylit.c

raytraverse.sky.skycalc.**perez_lum**(*xyz*, *coefs*, *intersky=True*)

matches perezlum.cal

raytraverse.sky.skycalc.**scale_efficacy**(*dirdif*, *sunz*, *csunz*, *skybright*, *catn*, *td=10.9735311509*)

raytraverse.sky.skycalc.**perez**(*sxyz*, *dirdif*, *md=None*, *ground_fac=0.2*, *td=10.9735311509*)

compute perez coefficients

Notes

to match the results of gendaylit, for a given sun angle without associated date, the assumed eccentricity is 1.035020

Parameters

- **sxyz** (*np.array*) – (N, 3) dx, dy, dz sun position
- **dirdif** (*np.array*) – (N, 2) direct normal, diffuse horizontal W/m^2
- **md** (*np.array, optional*) – (N, 2) month day of sky calcs (for more precise eccentricity calc)
- **ground_fac** (*float*) – scaling factor (reflectance) for ground brightness
- **td** (*np.array float, optional*) – (N,) dew point temperature in C

Returns
perez – (N, 10) diffuse normalization, ground brightness, perez coefs, x, y, z

Return type
np.array

raytraverse.sky.skycalc.**sky_mtx**(*sxyz*, *dirdif*, *side*, *jn=4*, *intersky=True*, *color=False*, ***kwargs*)

generate sky, ground and sun values from sun position and sky values

Parameters

- **sxyz** (*np.array*) – sun directions (N, 3)
- **dirdif** (*np.array*) – direct normal and diffuse horizontal radiation (W/m^2) (N, 2)
- **side** (*int*) – sky subdivision
- **jn** (*int, optional*) – sky patch subdivision n = jn^2
- **intersky** (*bool, optional*) – include interreflection between ground and sky (mimics perezlum.cal, not present in gendaymtx
- **kwargs** (*dict, optional*) – passed to perez()

Returns

- **skymtx** (*np.array*) – (N, side*side)
- **grndval** (*np.array*) – (N,)
- **sunval** (*np.array*) – (N, 4) - sun direction and radiance

raytraverse.sky.skycalc.**radiance_skydef**(*sunpos*, *dirdif*, *loc=None*, *md=None*, *ground_fac=0.2*, *td=10.9735311509*, *ro=0.0*)

similar to gendaylit, returns strings

Parameters

- **sunpos** (*Sequence*) – dx, dy, dz sun position or m,d,h (if loc is not None)
- **dirdif** (*Sequence*) – direct normal, diffuse horizontal W/m^2
- **loc** (*tuple, optional*) – location data given as lat, lon, mer with + west of prime meridian triggers sunpos treated as timestep
- **md** (*tuple, optional*) – month day of sky calcs (for more precise eccentricity calc with xyz sunpos)
- **ground_fac** (*float*) – scaling factor (reflectance) for ground brightness
- **td** (*np.array float, optional*) – (N,) dew point temperature in C
- **ro** (*float, optional*) – ignored if sunpos is xyz, else angle in degrees counterclockwise to rotate sky (to correct model north, equivalent to clockwise rotation of scene)

Returns

- **desc** (*str*) – comments with sky info
- **sund** (*str*) – solar material and sun object ("" if no sun)
- **skyd** (*str*) – perezlum brightfunc definition and sky/ground objects

3.5.2 SkyData

class raytraverse.sky.**SkyData**(*wea*, *loc=None*, *skyro=0.0*, *ground_fac=0.2*, *intersky=True*, *skyres=15*, *minalt=2.0*, *mindiff=5.0*, *mindir=0.0*, *ground=True*, *srcname='sky'*, *color=False*)

Bases: `object`

class to generate sky conditions

This class provides an interface to generate sky data using the perez sky model

Parameters

- **wea** (*str np.array*) – path to epw, wea, .npy file or np.array, or .npz file, if loc not set attempts to extract location data (if needed).
- **loc** (*tuple, optional*) – location data given as lat, lon, mer with + west of prime meridian overrides location data in wea (but not in sunfield)
- **skyro** (*float, optional*) – angle in degrees counter-clockwise to rotate sky (to correct model north, equivalent to clockwise rotation of scene)
- **ground_fac** (*float, optional*) – ground reflectance
- **intersky** (*bool, optional*) – include interreflection between ground and sky (mimics perezlum.cal, not present in gendaymtx)
- **skyres** (*int, optional*) – resolution of sky patches (sqrt(patches / hemisphere))
- **minalt** (*float, optional*) – minimum solar altitude for daylight masking
- **mindiff** (*float, optional*) – minumum diffuse horizontal irradiance for daylight masking
- **mindir** (*float, optional*) – minimum direct normal for daylight masking
- **ground** (*bool, optional*) – include ground component in matrix
- **srcname** (*str, optional*) – name
- **color** (*bool, optional*) – store sky data with 3 color channels (not implemented)

skyres

sky patch resolution

property skyro

sky rotation (in degrees, ccw)

property loc

lot, lon, mer (in degrees, west is positive)

skydata_dew()

property rowlabel

m,d,h (if known)

property skydata

sun position and dirnorm diffhoriz

write(*name='skydata'*, *scene=None*, *compressed=True*)

format_skydata(*dat*)

process dat argument as skydata

see sky.setter for details on argument

Returns

dx, dy, dz, dir, diff

Return type
np.array

property daysteps

property daymask

shape (len(skydata),) boolean array masking timesteps when sun is below horizon

property fullmask

property maskindices

property mask

an additional mask for smtx data

property smtx

shape (np.sum(daymask), skyres**2 + 1) (+(3,) if self.color) coefficients for each sky patch each row is a timestep, coefficients exclude sun

property sun

shape (np.sum(daymask), 5) sun position (index 0,1,2) and coefficients for sun at each timestep assuming the true solid angle of the sun (index 3) and the weighted value for the sky patch (index 4).

property sunproxy

corresponding sky bin for each sun position in daymask

smtx_patch_sun(*includesky=True*)

generate smtx with solar energy applied to proxy patch for directly applying to skysampler data (without direct sun components) can also be used in a partial mode (with sun view / without sun reflection.)

header()

generate image header string

fill_data(*x*, *fill_value=0.0*, *rowlabels=False*)

Parameters

- **x** (*np.array*) – first axis size = len(self.daymask[self.mask])
- **fill_value** (*Union[int, float], optional*) – value in padded array
- **rowlabels** (*bool, optional*) – include rowlabels

Returns
data in x padded with fill value to original shape of skydata

Return type
np.array

label(*x*)

masked_idx(*i*)

radiance_sky_matrix(*outf*, *fmt='float'*, *sun=True*, *sky=True*, *ncomps=3*)

sky_description(*i*, *prefix='skydata'*, *grid=False*, *sun=True*, *ground=True*, *sunpatch=False*)

generate radiance scene files to directly render sky data at index i

Parameters

- **i** (*int*) – index of sky vector to generate (indexed from skydata, not daymask)
- **prefix** (*str, optional*) – name/path for output files
- **grid** (*bool, optional*) – render sky patches with grid lines
- **sun** (*bool, optional*) – include sun source in rad file

- **ground** (*bool, optional*) – include ground source
- **sunpatch** (*bool, optional*) – include sun energy in sun_patch (sun should be false)

Returns
basename of 3 files written: prefix_i (.rad, .cal, and .dat) .cal and .dat must be located in RAYPATH (which can include .) or else edit the .rad file to explicitly point to their locations. note that if grid is True, the sky will not be accurate, so only use this for illustrative purposes.

Return type
str

Raises
IndexError – if i is not in masked indices

3.5.3 SkyDataMask

class raytraverse.sky.**SkyDataMask**(*hours*)

Bases: *SkyData*

spoofed skydata class for use with light results

Parameters
hours (*np.array*) – hours of year given as (m, d, h) where hour is H.5 (assumes 8760) to use as daymask.

property skydata

sun position and dirnorm diffhoriz

3.6 raytraverse.sampler

3.6.1 draw

wavelet and associated probability functions.

raytraverse.sampler.draw.**get_detail**(*data*, **args*, *mode='reflect'*, *cval=0.0*)

convolve a set of kernels with data. computes the sum of the absolute values of each convolution.

Parameters

- **data** (*np.array*) – source data (atleast 2D), detail calculated over last 2D
- **args** (*np.array*) – filters
- **mode** (*str*) – signal extension mode (passed to scipy.ndimage.convolve)
- **cval** (*float*) – constant value (passed to scipy.ndimage.convolve, used when mode='constant')

Returns
detail_array – 1d array of detail coefficients (row major order) matching size of data

Return type
np.array

raytraverse.sampler.draw.**from_pdf**(*pdf*, *threshold*, *lb=0.5*, *ub=4*, *minsamp=0*)

generate choices from a numeric probability distribution

Parameters

- **pdf** (*np.array*) – 1-d array of weights

- **threshold** (*float*) – the threshold used to determine the number of choices to draw given by pdf > threshold
- **lb** (*float, optional*) – values below threshold * lb will be excluded from candidates (lb must be in (0,1)
- **ub** (*float, optional*) – the maximum weight is set to ub*threshold, meaning all values in pdf >= to ub*threshold have an equal chance of being selected. in cases where extreme values are much higher than moderate values, but 100% sampling of extreme areas should be avoided, this value should be lower, such as when a region is sampled at a very high resolution (as is the case with directional sampling). On the other hand, set this value higher for sampling schemes with a low final resolution (like area sampling). If ub <= 1, then a deterministic choice is made, returning the idx of all values in pdf > threshold.

Returns
idx – an index array of choices, size varies.

Return type
np.array

3.6.2 BaseSampler

class raytraverse.sampler.**BaseSampler**(*scene*, *engine*, *accuracy=1.0*, *stype='generic'*, *samplerlevel=0*, *featurefunc=<function max>*, *features=1*, *weightfunc=<function max>*, *t0=0.00390625*, *t1=0.0625*)

Bases: object

wavelet based sampling class this is a virutal class that holds the shared sampling methods across directional, area, and sunposition samplers. subclasses are named as: {Source}Sampler{SamplingRange}, for instance:

- **SamplerPt: virtual base class for sampling directions from a point**
  - SkySamplerPt: sampling directions from a point with a sky patch source.
  - SunSamplerPt: sampling directions from a point with a single sun source
  - SunSamplerPtView: sampling the view from a point of the sun
  - ImageSampler: (re)sampling a fisheye image, useful for testing
- SamplerArea: sampling points on a horizontal planar area with any source type
- SamplerSuns: sampling sun positions (with nested area sampler)

Parameters

- **scene** (raytraverse.scene.Scene) – scene class containing geometry and formatter compatible with engine
- **engine** – has a run() method
- **accuracy** (*float, optional*) – parameter to set threshold at sampling level relative to final level threshold (smaller number will increase sampling, default is 1.0)
- **stype** (*str, optional*) – sampler type (prefixes output files)
- **featurefunc** (*func, optional*) – takes detail array as an argument, shape: (features,N, M) and an axis=0 keyword argument, returns shape (N, M). could be np.max, np.sum np.average or us custom function following the same pattern.
- **features** (*int, optional*) – number of values evaluated for detail

lb = 0.25

lower bound for drawing from pdf passed to raytraverse.sampler.draw.from_pdf()

ub = 8

upper bound for drawing from pdf passed to raytraverse.sampler.draw.from_pdf()

scene

scene information

Type
raytraverse.scene.Scene

t0

initial sampling threshold coefficient this value times the accuracy parameter is passed to raytraverse.sampler.draw.from_pdf() at level 0 (usually not used)

t1

final sampling threshold coefficient this value times the accuracy parameter is passed to raytraverse.sampler.draw.from_pdf() at final level, intermediate sampling levels are thresholded by a linearly interpolated between t0 and t1

accuracy

accuracy parameter some subclassed samplers may apply a scale factor to normalize threshold values depending on source brightness (see for instance ImageSampler and SunSamplerPt)

Type
float

stype

sampler type

Type
str

weights

holds weights for self.draw

Type
np.array

featurefunc

func takes weights and axis=0 argument to reduce detail

weightfunc

func takes weights and axis=1 argument to reduce output from engine when engine produces more features than sampler needs

property levels

sampling scheme

Getter
Returns the sampling scheme

Setter
Set the sampling scheme

Type
np.array

sampling_scheme(**args*)

calculate sampling scheme

run(*mapper*, *name*, *levels*, *plotp=False*, *log='err'*, *pfish=True*, ***kwargs*)

trigger a sampling run. subclasses should return a LightPoint/LightField from the executed object state (first call this method with super().run(...)

Parameters

- **mapper** (*raytraverse.mapper.Mapper*) – mapper to sample
- **name** (*str*) – output name
- **levels** (*np.array*) – the sampling scheme
- **plotp** (*bool, optional*) – plot weights, detail and vectors for each level
- **log** (*str, optional*) – whether to log level sampling rates can be 'scene', 'err' or None 'scene' - logs to Scene log file 'err' - logs to stderr anything else - does not log incremental progress
- **pfish** (*bool, optional*) – if True and plotp, use fisheye projection for detail/weight/vector images.
- **kwargs** – unused

draw(*level*)

draw samples based on detail calculated from weights

Returns

- **pdraws** (*np.array*) – index array of flattened samples chosen to sample at next level
- **p** (*np.array*) – computed probabilities

sample_to_uv(*pdraws*, *shape*)

generate samples vectors from flat draw indices

Parameters

- **pdraws** (*np.array*) – flat index positions of samples to generate
- **shape** (*tuple*) – shape of level samples

Returns

- **si** (*np.array*) – index array of draws matching samps.shape
- **vecs** (*np.array*) – sample vectors

sample(*vecs*)

call rendering engine to sample rays

Parameters
vecs (*np.array*) – sample vectors (subclasses can choose which to use)

Returns
lum – array of shape (N,) to update weights

Return type
np.array

detailfunc = 'wav'

filter banks for calculating detail choices:

'haar': [[1 -1]]/2, [[1] [-1]]/2, [[1, 0] [0, -1]]/2

'wav': [[-1 2 -1]] / 2, [[-1] [2] [-1]] / 2, [[-1 0 0] [0 2 0] [0 0 -1]] / 2

3.6.3 Sensor

class raytraverse.sampler.**Sensor**(*engine*, *dirs=(0.0, 0.0, 1.0)*, *offsets=(0.0, 0.0, 0.0)*, *name='sensor'*, *sunview=False*)

Bases: object

for use as engine in area sampler, holds collection of multiple sensor directions and offsets

Parameters

- **engine** (*craytraverse.renderer.Rennderer*) – fully initialized renderer class instance
- **dirs** (*Sequence, optional*) – array like shape (N, 3) sensor directions
- **offsets** (*Sequence, optional*) – array like shape (N, 3) offsets from sample position to include (for example mulitple z-heights)
- **sunview** (*bool, optional*) – NOT IMPLEMENTED if True, dirs are treated as candidate reflection normals, a value of (0, 0, 0) is prepended to hold the direct view.

property nproc

run(**args*, ***kwargs*)

alias for call, for consistency with SamplerPt classes for nested dimensions of evaluation

stack_rays(*r*)

3.6.4 ISamplerArea

class raytraverse.sampler.**ISamplerArea**(*scene*, *engine*, *accuracy=1.0*, *nlev=3*, *jitter=True*, *edgemode='constant'*, *t0=0.1*, *t1=0.9*, ***kwargs*)

Bases: *SamplerArea*

wavelet based area sampling class using Sensor as engine

Parameters

- **scene** (raytraverse.scene.Scene) – scene class containing geometry and formatter compatible with engine
- **engine** (raytraverse.sampler.Sensor) – renderer
- **accuracy** (*float, optional*) – parameter to set threshold at sampling level relative to final level threshold (smaller number will increase sampling, default is 1.0)
- **nlev** (*int, optional*) – number of levels to sample
- **jitter** (*bool, optional*) – jitter samples
- **edgemode** (*{'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional*) – default: 'constant', if 'constant' value is set to -self.t1, so edge is always seen as detail. Internal edges (resulting from PlanMapper borders) will behave like 'nearest' for all options except 'constant'

srcn

number of sources return per vector by run

Type
int

run(*mapper*, *plotp=False*, ***kwargs*)

adapively sample an area defined by mapper

Parameters

- **mapper** (raytraverse.mapper.PlanMapper) – the pointset to build/run
- **plotp** (*bool, optional*) – plot weights, detail and vectors for each level
- **kwargs** – passed to self.run()

Return type
raytraverse.lightplane.SensorPlaneKD

repeat(*guide*, *stype*)

repeat the sampling of a guide SensorPlane (to match all points)

Parameters

- **guide** (LightPlaneKD)
- **stype** (*str*) – alternate stype name. raises a ValueError if it matches the guide.

Return type
raytraverse.lightfield.SensorPlaneKD

sample(*vecs*)

call rendering engine to sample rays

Parameters
vecs (*np.array*) – sample vectors (subclasses can choose which to use)

Returns
lum – array of shape (N,) to update weights

Return type
np.array

3.6.5 ISamplerSuns

class raytraverse.sampler.**ISamplerSuns**(*scene*, *engine*, *accuracy=1.0*, *nlev=3*, *jitter=True*, *areakwargs=None*, *t0=0.05*, *t1=0.125*)

Bases: *SamplerSuns*

wavelet based sun position sampling class

Parameters

- **scene** (raytraverse.scene.Scene) – scene class containing geometry and formatter compatible with engine
- **engine** (raytraverse.sampler.Sensor) – with initialized renderer instance (with scene loaded, no sources)
- **accuracy** (*float, optional*) – parameter to set threshold at sampling level relative to final level threshold (smaller number will increase sampling, default is 1.0)
- **nlev** (*int, optional*) – number of levels to sample
- **jitter** (*bool, optional*) – jitter samples
- **ptkwargs** (*dict, optional*) – kwargs for raytraveerse.sampler.SunSamplerPt initialization
- **areakwargs** (*dict, optional*) – kwargs for raytravrse.sampler.SamplerArea initialization
- **metricset** (*iterable, optional*) – subset of samplerarea.metric set to use for sun detail calculation.

get_existing_run(*skymapper*, *areamapper*)

check for file conflicts before running/overwriting parameters match call to run

Parameters

- **skymapper** (raytraverse.mapper.SkyMapper) – the mapping for drawing suns
- **areamapper** (raytraverse.mapper.PlanMapper) – the mapping for drawing points

Returns

conflicts –

a tuple of found conflicts (None for each if no conflicts:

- suns: np.array of sun positions in vfile
- ptfiles: existing point files

Return type
tuple

run(*skymapper*, *areamapper*, ***kwargs*)

adaptively sample sun positions for an area (also adaptively sampled)

Parameters

- **skymapper** (raytraverse.mapper.SkyMapper) – the mapping for drawing suns
- **areamapper** (raytraverse.mapper.PlanMapper) – the mapping for drawing points
- **kwargs** – passed to self.run()

Return type
raytraverse.lightlplane.LightPlaneKD

3.6.6 SamplerSuns

class raytraverse.sampler.**SamplerSuns**(*scene*, *engine*, *accuracy=1.0*, *nlev=3*, *jitter=True*, *ptkwargs=None*, *areakwargs=None*, *metricset=('avglum', 'loggcr')*, *t0=0.05*, *t1=0.125*)

Bases: *BaseSampler*

wavelet based sun position sampling class

Parameters

- **scene** (raytraverse.scene.Scene) – scene class containing geometry and formatter compatible with engine
- **engine** (*raytraverse.renderer.Rtrace*) – initialized renderer instance (with scene loaded, no sources)
- **accuracy** (*float, optional*) – parameter to set threshold at sampling level relative to final level threshold (smaller number will increase sampling, default is 1.0)
- **nlev** (*int, optional*) – number of levels to sample
- **jitter** (*bool, optional*) – jitter samples
- **ptkwargs** (*dict, optional*) – kwargs for raytraveerse.sampler.SunSamplerPt initialization
- **areakwargs** (*dict, optional*) – kwargs for raytravrse.sampler.SamplerArea initialization

- **metricset** (*iterable, optional*) – subset of samplerarea.metric set to use for sun detail calculation.

ub = 8

upper bound for drawing from pdf

sampling_scheme(*mapper*)

calculate sampling scheme

get_existing_run(*skymapper*, *areamapper*)

check for file conflicts before running/overwriting parameters match call to run

Parameters

- **skymapper** (raytraverse.mapper.SkyMapper) – the mapping for drawing suns
- **areamapper** (raytraverse.mapper.PlanMapper) – the mapping for drawing points

Returns

conflicts –

a tuple of found conflicts (None for each if no conflicts:

- suns: np.array of sun positions in vfile
- ptfiles: existing point files

Return type
tuple

run(*skymapper*, *areamapper*, *specguide=None*, *recover=True*, ***kwargs*)

adaptively sample sun positions for an area (also adaptively sampled)

Parameters

- **skymapper** (raytraverse.mapper.SkyMapper) – the mapping for drawing suns
- **areamapper** (raytraverse.mapper.PlanMapper) – the mapping for drawing points
- **specguide** (*Union[*raytraverse.lightfield.LightPlaneKD*, Bool]*) – sky source lightfield to use as specular guide for sampling
- **recover** (*continue run on top of existing files, if false, overwrites*) – previous run.
- **kwargs** – passed to self.run()

Return type
raytraverse.lightlplane.LightPlaneKD

draw(*level*)

draw on condition of in_solarbounds from skymapper. In this way all solar positions are presented to the area sampler, but the area sampler is initialized with a weighting to sample only where there is variance between sun position. this keeps the subsampling of area and solar position independent.

Returns

- **pdraws** (*np.array*) – index array of flattened samples chosen to sample at next level
- **p** (*np.array*) – computed probabilities

sample_to_uv(*pdraws*, *shape*)

generate samples vectors from flat draw indices

Parameters

- **pdraws** (*np.array*) – flat index positions of samples to generate

- **shape** (*tuple*) – shape of level samples

Returns

- **si** (*np.array*) – index array of draws matching samps.shape
- **vecs** (*np.array*) – sample vectors

sample(*vecs*)

call rendering engine to sample rays

Parameters
vecs (*np.array*) – sample vectors

Returns
lum – array of shape (N,) to update weights

Return type
np.array

idxvecs()

3.6.7 SamplerArea

class raytraverse.sampler.**SamplerArea**(*scene*, *engine*, *accuracy=1.0*, *nlev=3*, *jitter=True*, *edgemode='constant'*, *metricclass=<class 'raytraverse.evaluate.samplingmetrics.SamplingMetrics'>*, *metricset=('avglum'*, *'loggcr'*, *'xpeak'*, *'ypeak')*, *t0=0.1*, *t1=0.9*, ***kwargs*)

Bases: *BaseSampler*

wavelet based area sampling class

Parameters

- **scene** (raytraverse.scene.Scene) – scene class containing geometry and formatter compatible with engine
- **engine** (raytraverse.sampler.SamplerPt) – point sampler
- **accuracy** (*float, optional*) – parameter to set threshold at sampling level relative to final level threshold (smaller number will increase sampling, default is 1.0)
- **nlev** (*int, optional*) – number of levels to sample
- **jitter** (*bool, optional*) – jitter samples
- **edgemode** (*{'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional*) – default: 'constant', if 'constant' value is set to -self.t1, so edge is always seen as detail. Internal edges (resulting from PlanMapper borders) will behave like 'nearest' for all options except 'constant'
- **metricclass** (raytraverse.evaluate.BaseMetricSet, *optional*) – the metric calculator used to compute weights
- **metricset** (*iterable, optional*) – list of metrics (must be recognized by metricclass. metrics containing "lum" will be normalized to 0-1)

ub = 100

upper bound for drawing from pdf

metricclass

raytraverse.evaluate.BaseMetricSet

metricset

iterable

features
int:

property edgemode

sampling_scheme(*mapper*)
calculate sampling scheme

run(*mapper*, *specguide=None*, *plotp=False*, ***kwargs*)
adapively sample an area defined by mapper

Parameters

- **mapper** (raytraverse.mapper.PlanMapper) – the pointset to build/run
- **specguide** (*Union[None, bool, str]*)
- **plotp** (*bool, optional*) – plot weights, detail and vectors for each level
- **kwargs** – passed to self.run()

Return type
raytraverse.lightlplane.LightPlaneKD

repeat(*guide*, *stype*)
repeat the sampling of a guide LightPlane (to match all rays)

Parameters

- **guide** (LightPlaneKD)
- **stype** (*str*) – alternate stype name for samplerpt. raises a ValueError if it matches the guide.

draw(*level*)
draw samples based on detail calculated from weights

Returns

- **pdraws** (*np.array*) – index array of flattened samples chosen to sample at next level
- **p** (*np.array*) – computed probabilities

sample_to_uv(*pdraws*, *shape*)
generate samples vectors from flat draw indices

Parameters

- **pdraws** (*np.array*) – flat index positions of samples to generate
- **shape** (*tuple*) – shape of level samples

Returns

- **si** (*np.array*) – index array of draws matching samps.shape
- **vecs** (*np.array*) – sample vectors

sample(*vecs*)
call rendering engine to sample rays

Parameters
vecs (*np.array*) – sample vectors (subclasses can choose which to use)

Returns
lum – array of shape (N,) to update weights

Return type
np.array

idxvecs()

3.6.8 SamplerPt

class raytraverse.sampler.**SamplerPt**(*scene*, *engine*, *idres=32*, *nlev=5*, *accuracy=1.0*, *srcn=1*, *stype='generic'*, *features=1*, *samplerlevel=0*, ***kwargs*)

Bases: *BaseSampler*

wavelet based sampling class for direction rays from a point

Parameters

- **scene** (*raytraverse.scene.Scene*) – scene class containing geometry and formatter compatible with engine
- **engine** (*raytraverse.renderer.Renderer*) – should inherit from raytraverse.renderer.Renderer
- **idres** (*int, optional*) – initial direction resolution (as sqrt of samples per hemisphere)
- **nlev** (*int, optional*) – number of levels to sample (eeach lvl doubles idres)
- **accuracy** (*float, optional*) – parameter to set threshold at sampling level relative to final level threshold (smaller number will increase sampling, default is 1.0)
- **srcn** (*int, optional*) – number of sources return per vector by run
- **stype** (*str, optional*) – sampler type (prefixes output files)
- **srcdef** (*str, optional*) – path or string with source definition to add to scene
- **plotp** (*bool, optional*) – show probability distribution plots at each level (first point only)
- **features** (*int, optional*) – number of values evaluated for detail
- **engine_args** (*str, optional*) – command line arguments used to initialize engine
- **nproc** (*int, optional*) – number of processors to give to the engine, if None, uses os.cpu_count()

srcn

number of sources return per vector by run

Type
int

idres

initial direction resolution (as sqrt of samples per hemisphere (or view angle)

Type
int

sampling_scheme(*a*)

calculate sampling scheme

run(*point*, *posidx*, *mapper=None*, *lpargs=None*, ***kwargs*)

sample a single point, position index handles file naming

Parameters

- **point** (*np.array*) – point to sample
- **posidx** (*int*) – position index
- **mapper** (*raytraverse.mapper.ViewMapper*) – view direction to sample
- **lpargs** (*dict, optional*) – keyword arguments forwarded to LightPointKD construction
- **kwargs** – passed to BaseSampler.run()

Return type
LightPointKD

repeat(*guide*, *stype*)

3.6.9 SkySamplerPt

class raytraverse.sampler.**SkySamplerPt**(*scene*, *engine*, ***kwargs*)

Bases: *SamplerPt*

sample contributions from the sky hemisphere according to a square grid transformed by shirley-chiu mapping using rcontrib.

Parameters

- **scene** (raytraverse.scene.Scene) – scene class containing geometry, location and analysis plane scene: str, optional (required if not reload) space separated list of radiance scene files (no sky) or octree
- **engine** (*raytraverse.renderer.Rcontrib*) – initialized rendering instance

3.6.10 SunSamplerPt

class raytraverse.sampler.**SunSamplerPt**(*scene*, *engine*, *sun*, *sunbin*, *nlev=6*, *stype='sun'*, ***kwargs*)

Bases: *SamplerPt*

sample contributions from direct suns.

Parameters

- **scene** (raytraverse.scene.Scene) – scene class containing geometry, location and analysis plane
- **engine** (*raytraverse.renderer.Rtrace*) – initialized renderer instance (with scene loaded, no sources)
- **sun** (*np.array*) – shape 3, sun position
- **sunbin** (*int*) – sun bin

sunpos

sun position x,y,z

Type
np.array

run(*point*, *posidx*, *specguide=None*, ***kwargs*)

sample a single point, position index handles file naming

Parameters

- **point** (*np.array*) – point to sample
- **posidx** (*int*) – position index
- **mapper** (*raytraverse.mapper.ViewMapper*) – view direction to sample
- **lpargs** (*dict, optional*) – keyword arguments forwarded to LightPointKD construction
- **kwargs** – passed to BaseSampler.run()

Return type
LightPointKD

3.6.11 SunSamplerPtView

class raytraverse.sampler.**SunSamplerPtView**(*scene*, *engine*, *sun*, *sunbin*, ***kwargs*)

Bases: *SamplerPt*

sample view rays to a source.

Parameters

- **scene** (raytraverse.scene.Scene) – scene class containing geometry, location and analysis plane
- **sun** (*np.array*) – the direction to the source
- **sunbin** (*int*) – index for naming

ub = 1

deterministic sample draws

run(*point*, *posidx*, *vm=None*, *plotp=False*, *log=None*, ***kwargs*)

sample a single point, position index handles file naming

Parameters

- **point** (*np.array*) – point to sample
- **posidx** (*int*) – position index
- **mapper** (*raytraverse.mapper.ViewMapper*) – view direction to sample
- **lpargs** (*dict, optional*) – keyword arguments forwarded to LightPointKD construction
- **kwargs** – passed to BaseSampler.run()

Return type
LightPointKD

3.6.12 ImageSampler

class raytraverse.sampler.**ImageSampler**(*scene*, *vm=None*, *scalefac=None*, *method='linear'*, *color=False*, *uv=False*, ***kwargs*)

Bases: *SamplerPt*

sample image (for testing algorithms).

Parameters

- **scene** (raytraverse.scene.ImageScene) – scene class containing image file information
- **scalefac** (*float, optional*) – by default set to the average of non-zero pixels in the image used to establish sampling thresholds similar to contribution based samplers

3.6.13 DeterministicImageSampler

class raytraverse.sampler.**DeterministicImageSampler**(*scene*, *vm=None*, *scalefac=None*, *method='linear'*, *color=False*, *uv=False*, ***kwargs*)

Bases: *ImageSampler*

ub = 1

upper bound for drawing from pdf passed to raytraverse.sampler.draw.from_pdf()

run(*point*, *posidx*, *mapper=None*, *lpargs=None*, ***kwargs*)

sample a single point, position index handles file naming

Parameters

- **point** (*np.array*) – point to sample
- **posidx** (*int*) – position index
- **mapper** (*raytraverse.mapper.ViewMapper*) – view direction to sample
- **lpargs** (*dict, optional*) – keyword arguments forwarded to LightPointKD construction
- **kwargs** – passed to BaseSampler.run()

Return type
LightPointKD

3.7 raytraverse.lightpoint

3.7.1 LightPointKD

class raytraverse.lightpoint.**LightPointKD**(*scene*, *vec=None*, *lum=None*, *vm=None*, *pt=(0, 0, 0)*, *posidx=0*, *src='sky'*, *srcn=1*, *srcdir=(0, 0, 1)*, *calcomega=True*, *write=True*, *omega=None*, *filterviews=True*, *srcviews=None*, *parent=None*, *srcviewidxs=None*, *features=1*)

Bases: `object`

light distribution from a point with KDtree structure for directional query

Parameters

- **scene** (raytraverse.scene.BaseScene)
- **vec** (*np.array, optional*) – shape (N, >=3) where last three columns are normalized direction vectors of samples. If not given, tries to load from scene.outdir
- **lum** (*np.array, optional*) – reshapeable to (N, srcn). sample values for each source corresponding to vec. If not given, tries to load from scene.outdir
- **vm** (*raytraverse.mapper.ViewMapper, optional*) – a default viewmapper for image and metric calculations, should match viewmapper of sampler.run() if possible.
- **pt** (*tuple list np.array*) – 3 item point location of light distribution
- **posidx** (*int, optional*) – index position of point, will govern file naming so must be set to avoid clobbering writes. also used by spacemapper for planar sampling
- **src** (*str, optional*) – name of source group. will govern file naming so must be set to avoid clobbering writes.

- **srcn** (*int, optional*) – must match lum, does not need to be set if reloading from scene.outdir
- **calcomega** (*bool, optional*) – if True (default) calculate solid angle of rays. This is unnecessary if point will be combined before calculating any metrics. setting to False will save some computation time.
- **write** (*bool, optional*) – whether to save ray data to disk.
- **omega** (*np.array, optional*) – provide precomputed omega values, if given, overrides calcomega

vm

raytraverse.mapper.ViewMapper

scene

raytraverse.scene.Scene

posidx

index for point

Type
int

pt

point location

Type
np.array

src

source key

Type
str

file

relative path to disk storage

Type
str

srcdir

direction to source(s)

load()

dump()

property vec

direction vector (N,3)

property lum

luminance (N,srcn)

property d_kd

kd tree for spatial query

Getter
Returns kd tree structure

Type
scipy.spatial.cKDTree

property omega

solid angle (N)

Getter
Returns array of solid angles

Setter
sets soolid angles with viewmapper

Type
np.array

set_srcviews(*srcviews*, *idxs=None*)

calc_omega(*write=True*)

calculate solid angle

Parameters
write (*bool, optional*) – update/write kdtree data to file

apply_coef(*coefs*)

apply coefficient vector to self.lum

Parameters
coefs (*np.array int float list*) – shape (N, self.srcn) or broadcastable

Returns
alum – shape (N, self.vec.shape[0])

Return type
np.array

add_to_img(*img*, *vecs*, *mask=None*, *skyvec=1*, *interp=False*, *idx=None*, *interpweights=None*, *order=False*, *omega=False*, *vm=None*, *rnd=False*, *engine=None*, *srcrnd=False*, ***kwargs*)

add luminance contributions to image array (updates in place)

Parameters

- **img** (*np.array*) – 2D image array to add to (either zeros or with other source)
- **vecs** (*np.array*) – vectors corresponding to img pixels shape (N, 3)
- **mask** (*np.array, optional*) – indices to img that correspond to vec (in case where whole image is not being updated, such as corners of fisheye)
- **skyvec** (*int float np.array, optional*) – source coefficients, shape is (1,) or (srcn,)
- **interp** (*Union[bool, str], optional*) –
  - if "precomp", use index and interpweights
  - if True and engine is None, linearinterpolation
  - if "fastc" and engine: uses content_interp (best after sampling w/o detail)
  - if "highc" and engine: uses content_interp_wedge (best after sampling w/o detail)
  - if "fast": use interp_fast (pair with sampling w/ detail)
  - if "high": use interp_wedge (pair with sampling w/ detail)
- **idx** (*np.array, optional*) – precomputed query/interpolation result
- **interpweights** (*np.array, optional*) – precomputted interpolation weights
- **omega** (*bool*) – if true, add value of ray solid angle instead of luminance
- **vm** (*raytraverse.mapper.ViewMapper, optional*)

- **rnd** (*bool, optional*) – use random values as contribution (for visualizing data shape)
- **engine** (*raytraverse.renderer.Rtrace, optional*) – engine for content aware interpolation
- **kwargs** (*dict, optional*) – passed to interpolationn functions

evaluate(*skyvec*, *vm=None*, *idx=None*, *srconly=False*, *blursun=False*, *includeviews=True*)

return rays within view with skyvec applied. this is the analog to add_to_img for metric calculations

Parameters

- **skyvec** (*int float np.array, optional*) – source coefficients, shape is (1,) or (srcn,)
- **vm** (*raytraverse.mapper.ViewMapper, optional*)
- **idx** (*np.array, optional*) – precomputed query_ball result
- **srconly** (*bool, optional*) – only evaluate direct sources (stored in self.srcviews)
- **includeviews** (*bool, optional*) – include src views in returned results

Returns

- **rays** (*np.array*) – shape (N, 3) rays falling within view
- **omega** (*np.array*) – shape (N,) associated solid angles
- **lum** (*np.array*) – shape (N,) associated luminances

query_ray(*vecs*)

return the index and distance of the nearest ray to each of vecs

Parameters

vecs (*np.array*) – shape (N, 3) normalized vectors to query, could represent image pixels for example.

Returns

- **i** (*np.array*) – integer indices of closest ray to each query
- **d** (*np.array*) – distance (corresponds to chord length on unit sphere) from query to ray in lightpoint. use translate.chord2theta to convert to angle.

query_ball(*vecs*, *viewangle=180*)

return set of rays within a view cone

Parameters

- **vecs** (*np.array*) – shape (N, 3) vectors to query.
- **viewangle** (*int float*) – opening angle of view cone

Returns

i – if vecs is a single point, a list of vector indices of rays within view cone. if vecs is a set of point an array of lists, one for each vec is returned.

Return type

list np.array

make_image(*outf*, *skyvec*, *vm=None*, *res=1024*, *interp=False*, *showsample=False*)

direct_view(*res=512*, *showsample=False*, *showweight=True*, *rnd=False*, *order=False*, *srcidx=None*, *interp=False*, *omega=False*, *scalefactor=1*, *vm=None*, *fisheye=True*, *grow=1*, *srcrnd=False*)

create an unweighted summary image of lightpoint

add(*lf2*, *src=None*, *calcomega=True*, *write=False*, *sumsrc=False*)

add light points of distinct sources together results in a new lightpoint with srcn=self.srcn+srcn2 and vector size=self.vecsize+vecsize2

Parameters

- **lf2** (raytraverse.lightpoint.LightPointKD)
- **src** (*str, optional*) – if None (default), src is "{lf1.src}_{lf2.src}"
- **calcomega** (*bool, optional*) – passed to LightPointKD constructor
- **write** (*bool, optional*) – passed to LightPointKD constructor
- **sumsrc** (*bool, optional*) – if True adds matching source indices together (must be same shape) this assumes that the two lightpoints represent the same source but different components (such as direct/indirect)

Returns

will be subtyped according to self, unless lf2 is needed to preserve data

Return type

raytraverse.lightpoint.LightPointKD

update(*vec*, *lum*, *omega=None*, *calcomega=True*, *write=True*, *filterviews=False*)

add additional rays to lightpoint in place

Parameters

- **vec** (*np.array, optional*) – shape (N, >=3) where last three columns are normalized direction vectors of samples.
- **lum** (*np.array, optional*) – reshapeable to (N, srcn). sample values for each source corresponding to vec.
- **omega** (*np.array, optional*) – provide precomputed omega values, if given, overrides calcomega
- **calcomega** (*bool, optional*) – if True (default) calculate solid angle of rays. This is unnecessary if point will be combined before calculating any metrics. setting to False will save some computation time. If False, resets omega to None!
- **write** (*bool, optional*) – whether to save updated ray data to disk.
- **filterviews** (*bool, optional*) – delete rays near sourceviews

linear_interp(*vm*, *srcvals*, *destvecs*)

static apply_interp(*i*, *srcvals*, *weights=None*)

interp(*destvecs*, *bandwidth=10*, *rt=None*, *lum=True*, *angle=True*, *dither=False*, ***kwargs*)

3.7.2 SrcViewPoint

class raytraverse.lightpoint.**SrcViewPoint**(*scene*, *vecs*, *lum*, *pt=(0, 0, 0)*, *posidx=0*, *src='sunview'*, *res=64*, *srcomega=6.796702357283834e-05*)

Bases: object

interface for sun view data

static offset(*points*, *target*)

scene

raytraverse.scene.Scene

posidx

index for point

Type
int

pt

point location

Type
np.array

src

source key

Type
str

radius

source radius

Type
float

lum

source luminance (average)

Type
float

property vm

add_to_img(*img*, *vecs*, *mask=None*, *coefs=1*, *vm=None*)

evaluate(*sunval*, *vm=None*, *blursun=False*)

direct_view(*res=80*)

3.7.3 CompressedPointKD

class raytraverse.lightpoint.**CompressedPointKD**(*scene*, *vec=None*, *lum=None*, *write=True*, *src=None*, *dist=0.0981*, *lerr=0.01*, *plotc=False*, ***kwargs*)

Bases: *LightPointKD*

compressed data needs special methods for making images.

can be initialized either like LightPointKD (but with required omega argument), or if 'scene' is a LightPointKD then a compressed output is calculated from the input

Parameters

- **scene** (*BaseScene LightpointKD*)
- **src** (*str, optional*) – new name for src passed to LightPointKD constructor
- **dist** (*float, optional*) – translate.theta2chord(np.pi/32), primary clustering distance using the birch algorithm, for lossy compression of lf. this is the maximum radius of a cluster, preserving important directional information. clustering acts on ray direction and luminance, with weight of luminance dimension controlled by the lweight parameter.
- **lerr** (*float, optional*) – min-max normalized error in luminance grouping.

- **plotc** (*bool, optional*) – make directview plot of compressed output showing source vectors

add_to_img(*img*, *vecs*, *mask=None*, *skyvec=1*, *vm=None*, *fisheye=True*, *order=False*, *omega=False*, *rnd=False*, ***kwargs*)

add luminance contributions to image array (updates in place)

Parameters

- **img** (*np.array*) – 2D image array to add to (either zeros or with other source)
- **vecs** (*np.array*) – vectors corresponding to img pixels shape (N, 3)
- **mask** (*np.array, optional*) – indices to img that correspond to vec (in case where whole image is not being updated, such as corners of fisheye)
- **skyvec** (*int float np.array, optional*) – source coefficients, shape is (1,) or (srcn,)
- **vm** (*raytraverse.mapper.ViewMapper, optional*)

compress(*lp*, *src=None*, *dist=0.0981*, *lerr=0.01*)

A lossy compression based on clustering. Rays are clustered using the birch algoritm on a 4D vector (x,y,z,lum) where lum is the sum of contributions from all sources in the LightPoint. In the optional second stage (activated with secondary=True) sources are further grouped through agglomerative cluster using an average linkage. this is to help with source indentification/matching between LightPoints, but can introduce significant errors to computing non energy conserving metrics in cases where the applied sky vectors have large relative differences between adjacent patches (> 1.5:1) or if the variance in peak luminance above the lthreshold parameter is significant. These include cases where nearby transmitting materials is varied (example: a trans upper above a clear lower), or lthreshold is set too low. For this reason, it is better to use single stage compression for metric computation and only do glare source grouping for interpolation between LightPoints.

Parameters

- **lp** (LightPointKD)
- **src** (*str, optional*) – new name for src passed to LightPointKD constructor
- **dist** (*float, optional*) – translate.theta2chord(np.pi/32), primary clustering distance using the birch algorithm, for lossy compression of lf. this is the maximum radius of a cluster, preserving important directional information. clustering acts on ray direction and luminance, with weight of luminance dimension controlled by the lweight parameter.
- **lerr** (*float, optional*) – min-max normalized error in luminance grouping.
- **plotc** (*bool, optional*) – make directview plot of compressed output showing source vectors

Return type
arguments for initializing a CompressedPointKD

3.8 raytraverse.lightfield

3.8.1 LightField

class raytraverse.lightfield.**LightField**(*scene*, *vecs*, *pm*, *src*)

Bases: object

collection of light data with KDtree structure for spatial query

Parameters

- **scene** (raytraverse.scene.BaseScene)
- **vecs** (*np.array str*) – the vectors used to organizing the child data as array or file shape (N,3) or (N,4) if 3, indexed from 0
- **pm** (raytraverse.mapper.PlanMapper)
- **src** (*str*) – name of source group.

property samplelevel

the level at which the vec was sampled (all zero if not provided upon initialization

property vecs

indexing vectors (such as position, sun positions, etc.)

property data

light data

property kd

kdtree for spatial queries built on demand

property omega

solid angle or area

query(*vecs*)

return the index and distance of the nearest point to each of points

Parameters

vecs (*np.array*) – shape (N, 3) vectors to query.

Returns

- **i** (*np.array*) – integer indices of closest ray to each query
- **d** (*np.array*) – distance from query to point in spacemapper.

evaluate(**args*, ***kwargs*)

3.8.2 LightPlaneKD

class raytraverse.lightfield.**LightPlaneKD**(*scene*, *vecs*, *pm*, *src*)

Bases: *LightField*

collection of lightpoints with KDtree structure for positional query

property data

LightPointSet

property omega

representative area of each point

Getter

Returns array of areas

Setter

sets areas

Type

np.array

evaluate(*skyvec*, *points=None*, *vm=None*, *metricclass=<class 'raytools.evaluate.metricset.MetricSet'>*, *metrics=None*, *mask=True*, ***kwargs*)

make_image(*outf*, *vals*, *res=1024*, *interp=False*, *showsample=False*)

make an image from precomputed values for every point in LightPlane

Parameters

- **outf** (*str*) – the file to write
- **vals** (*np.array*) – shape (len(self.points),) the values computed for each point
- **res** (*int, optional*) – image resolution (the largest dimension
- **interp** (*bool, optional*) – apply linear interpolation, points outside convex hull of results fall back to nearest
- **showsample** (*bool, optionaal*) – color pixel at sample location red

direct_view(*res=512*, *showsample=True*, *vm=None*, *area=False*, *metricclass=<class 'raytools.evaluate.metricset.MetricSet'>*, *metrics=('avglum',)*, *interp=False*)

create a summary image of lightplane showing samples and areas

3.8.3 SunsPlaneKD

class raytraverse.lightfield.**SunsPlaneKD**(*scene*, *vecs*, *pm*, *src*)

Bases: *LightField*

collection of lightplanes with KDtree structure for sun position query

property vecs

indexing vectors (sx, sy, sz, px, py, pz)

property suns

property data

LightPlaneSet

property kd

kdtree for spatial queries built on demand

property sunkd

kdtree for sun position queries built on demand

query(*vecs*)

return the index and distance of the nearest vec to each of vecs

Parameters

vecs (*np.array*) – shape (N, 6) vectors to query.

Returns

- **i** (*np.array*) – integer indices of closest ray to each query
- **d** (*np.array*) – distance from query to point, positional distance is normalized by the average chord-length between level 0 sun samples divided by the average distance between level 0 pt samples.

query_by_sun(*sunvec*, *fixed_points=None*, *stol=10*, *minsun=1*)

for finding vectors across zone, sun vector based query

Parameters

- **sunvec** (*Sequence*) – sun direction vector (normalized, xyz)
- **fixed_points** (*Sequence, optional*) – 2d array like, shape (N, 3) of additional fixed points to return use for example with a matching sky query. Note that if point filter is to large not all of these points are necessarily returned.

- **stol** (*Union[float, int], optional*) – maximum angle (in degrees) for matching sun vectors
- **minsun** (*int, optional*) – if atleast these many suns are not returned based on stol, directly query for this number of results (regardless of sun error)

Returns

- **vecs** (*np.array*) – shape (N, 6) final vectors, because of fixed_points, this may not match exactly with self.vecs[i] so this array mus be used in further processing
- **i** (*np.array*) – integer indices of the closest rays to each query
- **d** (*np.array*) – angle (in degrees) between queried sunvec and returned index

query_by_suns(*sunvecs*, *fixed_points=None*, *stol=10*, *minsun=1*)

parallel processing call to query_by_sun for 2d array of sunvecs

Parameters

- **sunvecs** (*np.array*) – shape (N, 3) sun direction vectors (normalized, xyz)
- **fixed_points** (*Sequence, optional*) – 2d array like, shape (N, 3) of additional fixed points to return use for example with a matching sky query. Note that if point filter is to large not all of these points are necessarily returned.
- **stol** (*Union[float, int], optional*) – maximum angle (in degrees) for matching sun vectors
- **minsun** (*int, optional*) – if atleast these many suns are not returned based on stol, directly query for this number of results (regardless of sun error)

Returns

- **vecs** (*list*) – list of np.array, one for each sunvec (see query_by_sun)
- **idx** (*list*) – list of np.array, one for each sunvec (see query_by_sun)
- **d** (*list*) – list of np.array, one for each sunvec (see query_by_sun)

3.8.4 SensorPlaneKD

class raytraverse.lightfield.**SensorPlaneKD**(*scene*, *vecs*, *pm*, *src*)

Bases: *LightPlaneKD*

collection of sensor results with KDtree structure for positional query

data has shape (pts, sensors, sources, bands)

property sensors

property vecs

indexing vectors (such as position, sun positions, etc.)

property data

light data

static apply_coef(*data*, *coefs*)

apply coefficient vector to data

Parameters

- **data** (*np.array*) – ndims should match self.data (N, sensors, nsrcs, nfeatures)
- **coefs** (*np.array int float list*) – shape (L, self.srcn) or broadcastable

Returns

alum – shape (L, N, sensors, nfeatures)

Return type
np.array

evaluate(*skyvec*, *points=None*, *sensoridx=None*, *mask=True*, ***kwargs*)

direct_view(*res=512*, *showsample=True*, *area=False*, *interp=False*, *sensoridx=None*, ***kwargs*)
create a summary image of lightplane showing samples and areas

3.8.5 SunSensorPlaneKD

class raytraverse.lightfield.**SunSensorPlaneKD**(*scene*, *vecs*, *pm*, *src*)
Bases: *SunsPlaneKD*

collection of sensorplanes with KDtree structure for sun position query

data has shape (pts * suns, sensors, sources, bands)

property sensors

property suns

property data
LightPlaneSet

static apply_coef(*data*, *coefs*)
apply coefficient vector to data

Parameters

- **data** (*np.array*) – ndims should match self.data (N, M, nsrcs, nfeatures)
- **coefs** (*np.array int float list*) – shape (L, self.srcn) or broadcastable

Returns
alum – shape (L, N, M, nfeatures)

Return type
np.array

3.8.6 LightResult

class raytraverse.lightfield.**LightResult**(*data*, **axes*, *boundary=None*)
Bases: object

a dense representation of lightfield data analyzed for a set of metrics

this class handles writing and loading results to disk as binary data and intuitive result extraction and reshaping for downstream visualisation and analysis using one of the "pull" methods. axes are indexed both numerically and names for increased transparency and ease of use.

Parameters

- **data** (*np.array str*) – multidimensional array of result data or file path to saved LightResule
- **axes** (*Sequence[raytraverse.lightfield.ResultAxis]*) – axis information

property boundary

property data

property axes

property names

property file

axis(*name*)

load(*file*)

write(*file*, *compressed=True*)

merge(**lrs*, *axis='sky'*)

create merged lightresult from lightresults, must match on all axes except axis. does not sort but culls duplicates

pull(**axes*, *preserve=1*, ***kwargs*)

arrange and extract data slices from result.

Integrators construct a light result with these axes:

0. sky
1. point
2. view
3. metric

Parameters

- **axes** (*Union[int, str]*) – the axes (by name or integer index) to reorder output, list will fill with default object order.
- **preserve** (*int, optional*) – number of dimensions to preserve (result will be N+1).
- **kwargs** – keys with axis names will be used to filter output.

Returns

- **result** (*np.array*) – the result array, will have 1+len(axes) dims, with the shaped determined by axis size and any indices argument.
- **labels** (*Sequence*) – list of labels for each axis, for flattened axes will be a tuple of broadcast axis labels.
- **names** (*Sequence*) – list of strings of returned axis names

static row_labels(*labels*)

static fmt_names(*name*, *labels*)

pull_header(*names*, *labels*, *rowlabel=True*)

print(*col*, *header=True*, *rowlabel=True*, *file=None*, *skyfill=None*, ***kwargs*)

first calls pull and then prints 2d result to file

sky_percentile(*metric*, *per=(50,)*, ***kwargs*)

print_serial(*col*, *basename*, *header=True*, *rowlabel=True*, *skyfill=None*, ***kwargs*)

print 3d result to series of 2d files

rebase(*points*)

pull2hdr(*basename*, *col='metric'*, *skyfill=None*, *spd=24*, *pm=None*, *res=480*, *showsample=False*, ***kwargs*)

info()

3.8.7 ZonalLightResult

class raytraverse.lightfield.**ZonalLightResult**(*data*, **axes*, *boundary=None*)

Bases: *LightResult*

a semi-dense representation of lightfield data analyzed for a set of metrics

this class handles writing and loading results to disk as binary data and intuitive result extraction and reshaping for downstream visualisation and analysis using one of the "pull" methods. axes are indexed both numerically and names for increased transparency and ease of use.

property data

load(*file*)

write(*file*, *compressed=True*)

merge(**lrs*, *axis='sky'*)

create merged lightresult from lightresults, must match on all axes except axis. does not sort but culls duplicates

pull2hdr(*basename*, *showsample=False*, *pm=None*, *res=480*, ***kwargs*)

rebase(*points*)

3.8.8 sets

LightSet

class raytraverse.lightfield.sets.**LightSet**(*dataclass*, *scene*, *points*, *idx*, ***kwargs*)

Bases: object

LightPointSet

class raytraverse.lightfield.sets.**LightPointSet**(*scene*, *points*, *idx*, *src*, *parent*)

Bases: *LightSet*

a collection of LightPoints, initialized by getitem

MultiLightPointSet

class raytraverse.lightfield.sets.**MultiLightPointSet**(*scene*, *points*, *idx*, *src*, *parent*)

Bases: *LightSet*

SensorPointSet

class raytraverse.lightfield.sets.**SensorPointSet**(*data*, *idx*, ***kwargs*)

Bases: *LightSet*

3.8.9 RaggedResult

class raytraverse.lightfield.**RaggedResult**(*a*)

Bases: tuple

has a shape parameter and indexing similar to a np.array, but with varying shape along the second axis. composed of a list of np.arrays whose shape match after the first dimension.

3.8.10 ResultAxis

class raytraverse.lightfield.**ResultAxis**(*values*, *name*, *cols=None*)

Bases: object

value_array()

index(*i*)

property cols

3.9 raytraverse.integrator

3.9.1 Integrator

class raytraverse.integrator.**Integrator**(**lightplanes*, *includesky=True*, *includesun=True*, *sunviewengine=None*, *ds=False*, *dv=False*)

Bases: object

collection of lightplanes with KDtree structure for sun position query

Parameters
lightplanes (*Sequence[raytraverse.lightfield.LightPlaneKD]*)

make_images(*skydata*, *points*, *vm*, *viewangle=180.0*, *res=512*, *interp=False*, *prefix='img'*, *namebyindex=False*, *suntol=10.0*, *blursun=False*, *resamprad=0.0*, *bandwidth=10*)

see namebyindex for file naming conventions

Parameters

- **skydata** (*raytraverse.sky.Skydata*)
- **points** (*np.array*) – shape (N, 3)
- **vm** (*Union[raytraverse.mapper.ViewMapper, np.array]*) – either a predefined ViewMapper (used for all points) or an array of view directions (will use a 180 degree view angle when initializing ViewMapper)
- **viewangle** (*float, optional*) – view opening for sensor (0-180,360) when vm is given as an array of view directions.
- **res** (*int, optional*) – image resolution
- **interp** (*bool, optional*) – interpolate image
- **prefix** (*str, optional*) – prefix for output file naming
- **namebyindex** (*bool, optional*) – if False (default), names images by: <prefix>_sky-<row>_pt-<x>_<y>_<z>_vd-<dx>_<dy>_<dz>.hdr if True, names images by: <prefix>_sky-<row>_pt-<pidx>_vd-<vidx>.hdr, where pidx, vidx are refer to the order of points, and vm.

Return type
np.array of out_files shape (skies, points, views)

evaluate(*skydata*, *points*, *vm*, *viewangle=180.0*, *metricclass=<class 'raytools.evaluate.metricset.MetricSet'>*, *metrics=None*, *datainfo=False*, *srconly=False*, *suntol=10.0*, *blursun=False*, *coercesumsafe=False*, *stol=10*, *minsun=1*, *emax=10000*, ***kwargs*)

apply sky data and view queries to daylightplane to return metrics parallelizes and optimizes run order.

Parameters

- **skydata** (*raytraverse.sky.Skydata*)
- **points** (*np.array*) – shape (N, 3)
- **vm** (*Union[raytraverse.mapper.ViewMapper, np.array]*) – either a predefined ViewMapper (used for all points) or an array of view directions (will use 'viewangle' when initializing ViewMapper)
- **viewangle** (*float, optional*) – view opening for sensor (0-180,360) when vm is given as an array of view directions, note that for illuminance based metrics, a value of 360 may not make sense as values behind will be negative.
- **metricclass** (raytraverse.evaluate.BaseMetricSet, *optional*)
- **metrics** (*Sized, optional*)
- **srconly** (*bool, optional*) – sun only calculations
- **suntol** (*float, optional*) – if Integrator has an engine, resample sun views when actual sun position error is greater than this many degrees.
- **blursun** (*bool, optional*) – apply human PSF to small bright sources
- **coercesumsafe** (*bool, optional*) – attempt to calculate sumsafe metrics
- **datainfo** (*Union[Sized[str], bool], optional*) – include information about source data as additional metrics. Valid values include: ["pt_err", "pt_idx", "src_err", "src_idx"]. If True, includes all.
- **stol** (*Union[float, int], optional*) – maximum angle (in degrees) for matching sun vectors (zonal)
- **minsun** (*int, optional*) – if atleast these many suns are not returned based on stol, directly query for this number of results (regardless of sun error) (zonal)

Return type

raytraverse.lightfield.LightResult

zonal_evaluate(*skydata*, *pm*, *vm*, *viewangle=180.0*, *metricclass=<class 'raytools.evaluate.metricset.MetricSet'>*, *metrics=None*, *srconly=False*, *suntol=10.0*, *blursun=False*, *coercesumsafe=False*, *stol=10*, *minsun=1*, *datainfo=False*, *calcarea=True*, *emax=10000*, ***kwargs*)

apply sky data and view queries to daylightplane to return metrics parallelizes and optimizes run order.

Parameters

evaluate (*see*)

Return type

raytraverse.lightfield.ZonalLightResult

3.9.2 SensorIntegrator

class raytraverse.integrator.**SensorIntegrator**(**lightplanes*, *ptype=None*, *factors=None*, *scale=179.0*, ***kwargs*)

Bases: *Integrator*

collection of sensorplanes with evaluation routines

Parameters

- **lightplanes** (*Sequence[raytraverse.lightfield.SensorPlaneKD]*)
- **ptype** (*Sequence[str]*) – matching order of lightplanes, requires one for each:
  - "sky": represents sky with nsrcs = skydata
  - "skysun": represents sky+sun with nsrcs = skydata
  - "patch": represents sun contribution as patch
  - "sun": sun contribution
  - "fixed": does not respond to skydata (electric lighting)
- **factors** (*Sequence[int], optional*) – values, for each light plane to scale contribution of eeach light plane for example, provide (1, -1, 1) for ptype: ("skysun", "patch", "sun") for 2-phase DDS calculation. If not give, all are set to 1
- **scale** (*float, optional*) – default output in lux (179)

make_images(**args*, ***kwargs*)

see namebyindex for file naming conventions

Parameters

- **skydata** (*raytraverse.sky.Skydata*)
- **points** (*np.array*) – shape (N, 3)
- **vm** (*Union[raytraverse.mapper.ViewMapper, np.array]*) – either a predefined ViewMapper (used for all points) or an array of view directions (will use a 180 degree view angle when initializing ViewMapper)
- **viewangle** (*float, optional*) – view opening for sensor (0-180,360) when vm is given as an array of view directions.
- **res** (*int, optional*) – image resolution
- **interp** (*bool, optional*) – interpolate image
- **prefix** (*str, optional*) – prefix for output file naming
- **namebyindex** (*bool, optional*) – if False (default), names images by: <prefix>_sky-<row>_pt-<x>_<y>_<z>_vd-<dx>_<dy>_<dz>.hdr if True, names images by: <prefix>_sky-<row>_pt-<pidx>_vd-<vidx>.hdr, where pidx, vidx are refer to the order of points, and vm.

Return type

np.array of out_files shape (skies, points, views)

evaluate(*skydata*, *points=None*, *vm=None*, *datainfo=False*, *stol=10*, *minsun=1*, ***kwargs*)

apply sky data and view queries to daylightplane to return metrics parallelizes and optimizes run order.

Parameters

- **skydata** (*raytraverse.sky.Skydata*)
- **points** (*np.array, optional*) – shape (N, 3), if None evaluates zone
- **vm** (*ignored*)

- **datainfo** (*Union[Sized[str], bool], optional*) – include information about source data as additional metrics. Valid values include: [“pt_err”, “pt_idx”, “src_err”, “src_idx”]. If True, includes all. zonal evaluation will only include src_err and src_idx
- **stol** (*Union[float, int], optional*) – maximum angle (in degrees) for matching sun vectors
- **minsun** (*int, optional*) – if atleast these many suns are not returned based on stol, directly query for this number of results (regardless of sun error)

Return type
raytraverse.lightfield.LightResult

zonal_evaluate(*skydata*, *pm*, *vm=None*, *datainfo=False*, *stol=10*, *minsun=1*, *calcarea=True*, ***kwargs*)

Parameters
evaluate (*see*)

Return type
raytraverse.lightfield.ZonalLightResult

3.9.3 helpers

parallelization functions for integration

raytraverse.integrator.helpers.**evaluate_pt**(*lpts*, *skyvecs*, *suns*, *vm=None*, *vms=None*, *metricclass=None*, *metrics=None*, *srconly=False*, *sumsafe=False*, *suntol=1.0*, *svengine=None*, *blursun=False*, *refl=None*, *resamprad=0.0*, ***kwargs*)

point by point evaluation suitable for submitting to ProcessPool

raytraverse.integrator.helpers.**img_pt**(*lpts*, *skyvecs*, *suns*, *vms=None*, *combos=None*, *qpts=None*, *skinfo=None*, *res=512*, *interp=False*, *prefix='img'*, *suntol=1.0*, *svengine=None*, *refl=None*, *resamprad=0.0*, ***kwargs*)

point by point evaluation suitable for submitting to ProcessPool

raytraverse.integrator.helpers.**prep_ds**(*lpts*, *skyvecs*)

raytraverse.integrator.helpers.**evaluate_pt_ds**(*lpts*, *skyvecs*, *suns*, ***kwargs*)

raytraverse.integrator.helpers.**img_pt_ds**(*lpts*, *skyvecs*, *suns*, ***kwargs*)

raytraverse.integrator.helpers.**evaluate_pt_dv**(*lpts*, *skyvecs*, *suns*, ***kwargs*)

raytraverse.integrator.helpers.**img_pt_dv**(*lpts*, *skyvecs*, *suns*, ***kwargs*)

raytraverse.integrator.helpers.**prep_resamp**(*lpts*, *refl=None*, *resamprad=0.0*)

raytraverse.integrator.helpers.**update_src_view**(*engine*, *lpt*, *sun*, *vm=None*, *tol=1.0*, *refl=None*, *resampvecs=None*, *reflarea=None*, *resamprad=0.0*)

raytraverse.integrator.helpers.**apply_dsky_patch**(*skp*, *skd*, *skyvecs*, *skdir*, *dirlum=None*)

3.10 raytraverse.evaluate

3.10.1 BaseMetricSet

class raytraverse.evaluate.**BaseMetricSet**(*vec*, *omega*, *lum*, *vm*, *metricset=None*, *scale=179.0*, *omega_as_view_area=True*, *guth=True*, *warn=False*, ***kwargs*)

Bases: object

object for calculating metrics based on a view direction, and rays consisting on direction, solid angle and luminance information

by encapsulating these calculations within a class, metrics with redundant calculations can take advantage of cached results, for example dgp does not need to recalculate illuminance when it has been directly requested. all metrics can be accessed as properties (and are calculated just in time) or the object can be called (no arguments) to return a np.array of all metrics defined in "metricset"

Parameters

- **vm** (*raytools.mapper.ViewMapper*) – the view direction
- **vec** (*np.array*) – (N, 3) directions of all rays in view
- **omega** (*np.array*) – (N,) solid angle of all rays in view
- **lum** (*np.array*) – (N,) luminance of all rays in view (multiplied by "scale")
- **metricset** (*list, optional*) – keys of metrics to return, same as property names
- **scale** (*float, optional*) – scalefactor for luminance
- **omega_as_view_area** (*bool, optional*) – take sum(omega) as view area. if false corrects omega to vm.area
- **warnings** (*bool, optional*) – if False, suppresses numpy warnings (zero div, etc…) when accessed via __call__
- **kwargs** – additional arguments that may be required by additional properties

allmetrics = ['illum', 'avglum', 'loggcr', 'gcr', 'pwgcr', 'logpwgcr', 'density', 'avgraylum', 'pwavglum', 'maxlum']

safe2sum = {'avglum', 'density', 'illum'}

defaultmetrics = ['illum', 'avglum', 'loggcr']

available metrics (and the default return set)

classmethod check_metrics(*metrics*, *raise_error=False*)

returns list of valid metric names from argument if raise_error is True, raises an Atrribute Error

classmethod check_safe2sum(*metrics*)

checks if list of metrics is safe to compute for seperate sources before adding

property vec

property lum

property omega

property ctheta

cos angle between ray and view

property radians

angle between ray and view

property pos_idx

property pweight

property pweighted_area

property illum

illuminance

property avglum

average luminance

property maxlum

average luminance

property pwavglum

position weighted average luminance

property avgraylum

average luminance (not weighted by omega

property gcr

a unitless measure of relative contrast defined as the average of the squared luminances divided by the average luminance squared

property pwgcr

a unitless measure of relative contrast defined as the average of the squared luminances divided by the average luminance squared weighted by a position index

property logpwgcr

a unitless measure of relative contrast defined as the log of gcr

property loggcr

a unitless measure of relative contrast defined as the log of gcr

property density

average vector density of view representation

3.10.2 MultiLumMetricSet

class raytraverse.evaluate.**MultiLumMetricSet**(*vec*, *omega*, *lum*, *vm*, *metricset=None*, *scale=179.0*, *omega_as_view_area=True*, ***kwargs*)

Bases: *BaseMetricSet*

object for calculating metrics based on a view direction, and rays consisting on direction, solid angle and luminance information

by encapsulating these calculations within a class, metrics with redundant calculations can take advantage of cached results, for example dgp does not need to recalculate illuminance when it has been directly requested. all metrics can be accessed as properties (and are calculated just in time) or the object can be called (no arguments) to return a np.array of all metrics defined in "metricset"

Parameters

- **vm** (*raytraverse.mapper.ViewMapper*) – the view direction
- **vec** (*np.array*) – (N, 3) directions of all rays in view
- **omega** (*np.array*) – (N,) solid angle of all rays in view
- **lum** (*np.array*) – (N, M) luminance of all rays in view (multiplied by "scale")
- **metricset** (*list, optional*) – keys of metrics to return, same as property names

- **scale** (*float, optional*) – scalefactor for luminance
- **kwargs** – additional arguments that may be required by additional properties

property illum

illuminance

property avglum

average luminance

property avgraylum

average luminance (not weighted by omega)

property gcr

a unitless measure of relative contrast defined as the average of the squared luminances divided by the average luminance squared

3.10.3 MetricSet

class raytraverse.evaluate.**MetricSet**(*vec*, *omega*, *lum*, *vm*, *metricset=None*, *scale=179.0*, *threshold=2000.0*, *guth=True*, *tradius=30.0*, *omega_as_view_area=False*, *lowlight=False*, ***kwargs*)

Bases: *BaseMetricSet*

object for calculating metrics based on a view direction, and rays consisting on direction, solid angle and luminance information

by encapsulating these calculations within a class, metrics with redundant calculations can take advantage of cached results, for example dgp does not need to recalculate illuminance when it has been directly requested. all metrics can be accessed as properties (and are calculated just in time) or the object can be called (no arguments) to return a np.array of all metrics defined in "metricset"

Parameters

- **vm** (*raytools.mapper.ViewMapper*) – the view direction
- **vec** (*np.array*) – (N, 3) directions of all rays in view
- **omega** (*np.array*) – (N,) solid angle of all rays in view
- **lum** (*np.array*) – (N,) luminance of all rays in view (multiplied by "scale")
- **metricset** (*list, optional*) – keys of metrics to return, same as property names
- **scale** (*float, optional*) – scalefactor for luminance
- **threshold** (*float, optional*) – threshold for glaresource/background similar behavior to evalglare '-b' paramenter. if greater than 100 used as a fixed luminance threshold. otherwise used as a factor times the task luminance (defined by 'tradius')
- **guth** (*bool, optional*) – if True, use Guth for the upper field of view and iwata for the lower if False, use Kim
- **tradius** (*float, optional*) – radius in degrees for task luminance calculation
- **kwargs** – additional arguments that may be required by additional properties

defaultmetrics = ['illum', 'avglum', 'loggcr', 'ugp', 'dgp']

available metrics (and the default return set)

allmetrics = ['illum', 'avglum', 'loggcr', 'gcr', 'pwgcr', 'logpwgcr', 'density', 'avgraylum', 'pwavglum', 'maxlum', 'ugp', 'dgp', 'tasklum', 'backlum', 'dgp_t1', 'log_gc', 'dgp_t2', 'ugr', 'threshold', 'pwsl2', 'view_area', 'backlum_true', 'srcillum', 'srcarea', 'maxlum']

safe2sum = {'avglum', 'density', 'illum', 'pwsl2', 'srcillum'}

property src_mask

boolean mask for filtering source/background rays

property task_mask

property sources

vec, omega, lum of rays above threshold

property background

vec, omega, lum of rays below threshold

property source_pos_idx

property threshold

threshold for glaresource/background similar behavior to evalglare '-b' paramenter

property pwsl2

position weighted source luminance squared, used by dgp, ugr, etc sum(Ls^2*omega/Ps^2)

property srcillum

source illuminance

property srcarea

total source area

property maxlum

peak luminance

property backlum

average background luminance CIE estimate (official for some metrics)

property backlum_true

average background luminance mathematical

property tasklum

average task luminance

property dgp

property dgp_t1

property log_gc

property dgp_t2

property ugr

property ugp

//dx.doi.org/10.1016/j.buildenv.2016.08.005

Type
http

3.10.4 FieldMetric

class raytraverse.evaluate.**FieldMetric**(*vec*, *omega*, *lum*, *vm=None*, *scale=1.0*, *npts=360*, *close=True*, *sigma=0.05*, *omega_as_view_area=True*, ***kwargs*)

Bases: *BaseMetricSet*

calculate metrics on full spherical point clouds rather than view based metrics.

Parameters

- **vec** (*np.array*) – (N, 3) directions of all rays
- **omega** (*np.array*) – (N,) solid angle of all rays
- **lum** (*np.array*) – (N,) luminance of all rays (multiplied by "scale")
- **metricset** (*list, optional*) – keys of metrics to return, same as property names
- **scale** (*float, optional*) – scalefactor for luminance
- **npts** (*int, optional*) – for equatorial metrics, the number of points to interpolate
- **close** (*bool, optional*) – include npts+1 duplicate to draw closed curve
- **sigma** (*float, optional*) – scale parameter of gaussian for kernel estimated metrics
- **omega_as_view_area** (*bool, optional*) – set to true when vectors either represent a whole sphere or a subset that does not match the viewmapper. if False, corrects boundary omega to properly trim to correct size.
- **kwargs** – additional arguments that may be required by additional properties

property tp

vectors in spherical coordinates

property phi

interpolated output phi values

property eq_xyz

interpolated output xyz vectors

property avg

overall vector (with magnitude)

property peak

overall vector (with magnitude)

property eq_lum

luminance along an interpolated equator with a bandwidth=sigma

property eq_density

ray density along an interpolated equator

property eq_illum

illuminiance along an interpolated equator

property eq_gcr

cosine weighted gcr along an interpolated equator

property eq_loggc

property eq_dgp

3.10.5 SamplingMetrics

class raytraverse.evaluate.**SamplingMetrics**(*vec*, *omega*, *lum*, *vm*, *scale=1.0*, *peakthreshold=0.0001*, *lmin=0*, *gcrnorm=8*, ***kwargs*)

Bases: *BaseMetricSet*

default metricset for areasampler

defaultmetrics = ['avglum', 'loggcr', 'xpeak', 'ypeak']

available metrics (and the default return set)

allmetrics = ['avglum', 'loggcr', 'xpeak', 'ypeak']

property peakvec

average vector (with magnitude) for peak rays

property xpeak

x-component of avgvec as positive number (in range 0-1)

property ypeak

y-component of avgvec as positive number (in range 0-1)

property loggcr

log of global contrast ratio

3.10.6 PositionIndex

class raytraverse.evaluate.**PositionIndex**(*guth=True*)

Bases: object

calculate position index according to guth/iwata or kim

Parameters

guth (*bool*) – if True, use Guth for the upper field of view and iwata for the lower if False, use Kim

positions(*vm*, *vec*)

calculate position indices for a set of vectors

Parameters

- **vm** (*raytools.mapper.ViewMapper*) – the view/analysis point, should have 180 degree field of view
- **vec** (*np.array*) – shape (N,3) the view vectors to calculate

Returns

posidx – shape (N,) the position indices

Return type

np.array

positions_vec(*viewvec*, *srcvec*, *up=(0, 0, 1)*)

3.10.7 retina

3.10.8 hvsgsm

3.10.9 GSS

class raytraverse.evaluate.**GSS**(*view=None*, *age=40*, *f=16.67*, *scale=179*, *pigmentation=0.106*, *fwidth=10*, *psf=True*, *adaptmove=True*, *directmove=True*, *raw=False*)

Bases: `object`

calculate GSS for images with angular fisheye projection

application of model described in:

> A GENERIC GLARE SENSATION MODEL BASED ON THE HUMAN VISUAL SYSTEM Vissenberg, M.C.J.M., Perz, M., Donners, M.A.H., Sekulovski, D. Signify Research, Eindhoven, THE NETHERLANDS gilles.vissenberg@signify.com DOI 10.25039/x48.2021.0P23

see methods for citations associated with each step in model.

the model requires the following steps:

Done when setting an image with a new resolution:

1. calculate solid angle of pixels
2. calculate eccentricity from guth position idx

Steps for applying model to an image:

1. calculate eye illuminance from image
2. mask non-glare source pixels (REMOVED, only masks to 180 degree incidence)
3. calculate pupil area and diameter
4. calculate global retinal irradiance
5. calculate incident retinal irradiance of glare sources (REMOVED, calculate on totial irradiance)
6. apply PSF to (4)
7. apply movement affecting adaptation to (6)
8. apply movement affecting direct response to (6)
9. calculate local adaptation using (7)
10. calculate V/V_m photoreceptor response (8)
11. calculate receptor field response to (10) as DoG
12. normalize field response with logistic
13. sum GSS and apply position weighting

Parameters

- **view** – can be None, a view file, a ViewMapper, or an hdrimage with a valid view specification (must be -vta)
- **age** – age of observer
- **f** – eye focal length
- **scale** – factor to apply to raw pixel values to convert to cd/m^2
- **pigmentation** –

  from Ijspeert et al. 1993:
  mean for blue eyes: 0.16 brown eyes: 0.106 dark brown eyes: 0.056

- **fwidth** (*Union[int, float], optional*) – the width of the frame for psf
- **psf** (*bool, optional*) – apply pointspread function for light arriving at retina
- **adaptmove** (*bool, optional*) – apply involuntary eye movement effect on local adaptation
- **directmove** (*book, optional*) – apply involuntary eye movement effect on direct cone response
- **raw** (*bool, optional*) – do not weight results, used for calibration

Notes

set self.lum, either by initializing with an image, or with the parameter setter, then compute:

```
gss = GSS("img.hdr")
gss.lum = "img.hdr"
score = gss.compute()
```

additional images can be loaded and computed with the parameter setter by calling images with the same resolution and view size on an initialized object, subsantial re-computation can be avoided.

Alternatively, to get access to process arrays or to override pupil adaptation and or isolating glare sources:

```
e_g, pupa, pupd = self.adapt(ev_eye)
r_g, parrays = self.glare_response(img_gs, e_g, pupa, pupd,
return_arrays=True)
```

For processing multiple images with the same GSS initialization in parallel, see hvsgsm.gss_compute()

emax = 0.12

emin = 0.009

fr_a = 22

fr_b = 0.25

fr_k = 0.67

norm = 4

contrast = 0.8

adapt(*ev_eye=None*)

step 1 in compute, adapt eye to image

glare_response(*img_gs*, *e_g*, *pupa*, *pupd*, *return_arrays=False*)

step 3 in compute, apply steps of Vissenberg et al. model

Parameters

- **img_gs** (*np.array*) – representing all glare sources
- **e_g** (*float*) – global retinal irradiance
- **pupa** (*float*) – pupil area (mm^2)
- **pupd** (*float*) – pupil diameter (mm)
- **return_arrays** (*bool, optional*) – if True returns second value with dict of process arrays else return r_w only

Returns

- **r_w** (*np.array*) – weighted glare response for entire retina as represented by image

- **parrays** (*dict, optional*) – with returned_arrays=True keys: retinal_irrad, psf, adapt_eye_movement, direct_eye_movement, local_adaptation, response_ratio, response_lin, response_log

`compute`(*save=None*, *ev_eye=None*)

apply glare sensation model to loaded image

Parameters

- **save** (*str*) – if given save response image to file specified (.hdr)
- **ev_eye** (*float, opttional*) – externally calculated Ev

Return type
float

`property ecc`

`property lum`

`property res`

resolution, set via lum

`property vecs`

directions, set via lum

`property omega`

solid angle, set via lum

`property mask`

view mask, set via lum

`property ctheta`

cos between vectors and view direction, set via lum

`property sigma_c`

position index scaled to eccentricity .009-.12 (used in field_response)

Note that this differs from the implementation dscribed by Vissenberg and incorporates KIM below the horizon

`property vm`

`pupil`(*ev*)

calculate pupil area

Based on: Donners, Maurice & Vissenberg, Michel & Geerdinck, L.M. & Broek-Cools, J. (2015). A PSYCHOPHYSICAL MODEL OF DISCOMFORT GLARE IN BOTH OUTDOOR AND INDOOR APPLICATIONS.

Parameters
ev – illumiance at eye (lux)

`retinal_irradiance`(*lum*, *pupa*)

adjust incident light on retina based on pupil size and focal-length

from Vissenberg et al. 2021 equation (1): (1) E_r = A_p * L / f^2 E_r: local retinal irrradiance L: field luminance

`prep_kernel`()

construct an array to hold a kernel scaled to image resolution

`psf_coef`(*pupd*)

age, pupil size and pigmentation adjusted PSF coefficients

PSF:
PSF(phi) = sum(c * f_b(phi)) f_b(phi) = b/(2 * (sin^2(phi) + b^2*cos^2(phi))^1.5) 1/steradian

LSF:
LSF(phi) = sum(c * l_b(phi)) l_b(phi) = b/(* (sin^2(phi) + b^2*cos^2(phi))) 1/rad

based on: J.K. Ijspeert, T.J.T.P. Van Den Berg, H. Spekreijse, An improved mathematical description of the foveal visual point spread function with parameters for age, pupil size and pigmentation, Vision Research, Volume 33, Issue 1, 1993,Pages 15-20, ISSN 0042-6989, https://doi.org/10.1016/0042-6989(93)90053-Y.

`apply_psf`(*e_r*, *pupd*)

apply human foveal point spread function

based on: J.K. Ijspeert, T.J.T.P. Van Den Berg, H. Spekreijse, An improved mathematical description of the foveal visual point spread function with parameters for age, pupil size and pigmentation, Vision Research, Volume 33, Issue 1, 1993,Pages 15-20, ISSN 0042-6989, https://doi.org/10.1016/0042-6989(93)90053-Y.

`apply_eye_movement_1`(*e_r*)

eye movement gaussian adaptation model to blur image at the time- scale of adaptation response.

based on: R. A. Normann, B. S. Baxter, H. Ravindra and P. J. Anderton, "Photoreceptor contributions to contrast sensitivity: Applications in radiological diagnosis," in IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-13, no. 5, pp. 944-953, Sept.-Oct. 1983, doi: 10.1109/TSMC.1983.6313090.

Parameters
e_r (*np.array*) – retinal irradiance (optical correction)

Returns
retinal irradiance (with adaptation scale movement and optical correction)

Return type
adapt_eye_movement

`apply_eye_movement_2`(*e_r*, *e_g*)

blur image due to eye movement during direct response

from Vissenberg et al. 2021 equations (5) and (6): (5) = 100/(E_g * f^2)^0.12 ms tau (): cone integration time

(6) w = 2 * sqrt(D *) D = 30.0 arcmin^2 * s^-1 (occular drift) D = 250.0 (micro saccades)

Parameters

- **e_r** (*np.array*) – retinal irradiance (optical correction)
- **e_g** (*float*) – global retinal irrradiance

Returns
retinal irradiance (with movement and optical correction)

Return type
direct_eye_movement

`local_eye_adaptation`(*e_r*, *e_g*)

calculate locallized eye adaptation

from Vissenberg et al. 2021 equation (4): log_10(E_a) = p * log_10(E_r) + (1-p) * log_10(E_g) E_a: adaptation illuminance p: 0.8 (indoor / moderate) - 0.9 (outdoor / strong) contrast

Parameters

- **e_r** (*np.array*) – retinal irradiance (optical correction)
- **e_g** (*float*) – global retinal irrradiance

Return type
local_adaptation

static cone_response(*e_r*, *e_a*)

calculate local response as a fraction of maximum at current adaptation

from Vissenberg et al. 2021 equations (2) and (3): (2) V/V_m = E_r^n / (E_r^n + ^n) V: photoreceptor response V_m: maximum response E_r: local retinal illuminance (apply w to this E_r) n: 0.74

(3) = (5.701055^(1/2.55) + E_a^(1/2.55))^2.55 sigma (): half-saturation retinal illuminance value

Parameters

- **e_r** (*np.array*) – retinal irradiance (with movement and optical correction)
- **e_a** (*np.array*) – local adaptation

Return type
response_ratio

field_response(*vvm*)

receptive field response

from Vissenberg et al. 2021 equation (7):

```
R_RF(r) = e^(-r^2/(2_c^2)) / (2_c^2)
          - K * e^(-r^2/(2_s^2)) / (2_s^2)
```

R_RF: receptive field response r: distance to receptive field center (degrees) _c: gaussian width of center (0.009 (center) - 0.12 (edge FOV) degrees) _s: gaussian width of surround 3.5 * _c K: DoG balance factor 0.67

Parameters
vvm (*np.array*) – response_ratio (saturation)

Returns
linear, difference of gussians

Return type
response_lin

normalized_field_response(*r*)

normalized non-linear ganglion response

from Vissenberg et al. 2021 equation (8): R_G = 1 / (1 + e^(-a * (R_lin - b))) R_G: normalized non-linear ganglion response a: slope of logistic = 22 b: 0.25

Parameters
r (*np.array*) – response_lin

Returns
logistic

Return type
response_log

gss(*r_g*)

calculate minkowski sum on normalized response

from Vissenberg et al. 2021 equation (9):

(9) GSS = sum_i(R_G,i^m _i)^(1/m) GSS: glare sensation score m: minkowski norm (4) delta (): solid angle of pixel (steradians)

Notes

fit on guth data using BCD = 2843.58 * e^(x + 1.5 * x^2) / 179 with a 2.12 degree source and 34.26 cd/m^2 background:

```
numpy.polynomial.Polynomial.fit(ecc, fac, 7, window=[0, 1],
                                domain=[.009, 0.12])
# where x = eccentricity (.009 -.12 from 0 to 55 degree vertical
# angle and y = 1/unweighted GSS
```

results:

$$33.12797281707965 + 2.2872877726594725 \cdot x^{1} - 104.61419835147568 \cdot x^{2} - 275.45010218009116 \cdot x^{3} + 1587.8255352939432 \cdot x^{4} - 2570.6813747583033 \cdot x^{5} + 1837.1741161137818 \cdot x^{6} - 499.8491902780004 \cdot x^{7}$$

3.11 raytraverse.craytraverse

3.12 raytraverse.api

factory functions for easy api access raytraverse.

raytraverse.api.**get_config**(*config*, *com='raytraverse_'*)

load a config file into a dict

Parameters

- **config** (*str*) – path to .cfg file with sections raytarverse_*
- **com** (*str*) – basename of commands in .cfg file

Returns
result

Return type
dict

raytraverse.api.**auto_reload**(*scndir*, *area*, *areaname='plan'*, *skydata='skydata'*, *ptres=1.0*, *rotation=0.0*, *zheight=None*)

reload associated class instances from file paths

Parameters

- **scndir** (*str*) – matches outdir argument of Scene()
- **area** (*str np.array*) – radiance scene geometry defining a plane to sample, tsv file of points to generate bounding box, or np.array of points.
- **areaname** (*str, optional*) – matches name argument of PlanMapper()
- **skydata** (*str, optional*) – matches name argument of SkyData.write()
- **ptres** (*float, optional*) – resolution for considering points duplicates, border generation (1/2) and add_grid(). updateable
- **rotation** (*float, optional*) – positive Z rotation for point grid alignment
- **zheight** (*float, optional*) – override calculated zheight

Returns

- *Scene*
- *PlanMapper*

- *SkyData*

raytraverse.api.**load_lp**(*path*, *hasparent=True*)

load a lightpoint from a file

will try to get appropriate scene/zone information from file path but reverts to a lightpoint without correct meta-data if it does not have the appropriate nesting.

Parameters

- **path** (*str*) – relative path to .rytpt file
- **hasparent** (*bool, optional*) – was sampled within a zone (typical), set to false in the case that the lightpoint file is multiple folders deep from CWD but was not sampled within a zone (meaning you used a samplerpt class, not typical).

Return type
LightPointKD

raytraverse.api.**get_integrator**(*scn*, *pm*, *srcname='suns'*, *simtype='2comp'*, *sunviewengine=None*)

CHAPTER

FOUR

TUTORIALS

4.1 Directional Sampling Overview

(starting at 4:56:25)

4.1.1 Transcript

1. Title Slide

Hello, my name is Stephen Wasilewski and I am presenting some work I have prepared along with my co-authors. Raytraverse is a new method that guides the sampling process of a daylight simulation.

2. The Daylight Simulation Process

To understand how this method can enhance the daylight simulation process, it is useful to view the process by parts.

2.b

The model describes how geometry, materials, and light sources are represented.

2.c

Sampling determines how the analysis dimensions are subdivided into discrete points to simulate.

2.d

These views rays are solved for by a renderer, yielding a radiance or an irradiance value for each view ray.

2.e

This output is evaluated according to some metric or otherwise preparing the data for interpretation.

3. Assumptions

To make a viable workflow, each of these parts require (whether explicitly or implicitly) a number of assumptions that define the limitations and opportunities of the method. To explain this in practical terms, here are three examples of well known climate based modeling methods for visual comfort.

4. CBDM Methods for Visual Comfort: Ev based

Illuminance based methods, including DGPs (simplified Daylight Glare Probability), limit the directional sampling resolution to a single sample per view direction in order to efficiently sample a larger number of positions and sky conditions throughout a space.

Unfortunately: Even if the employed rendering method perfectly captures the true Illuminance, as a model for discomfort glare it fails to account for scenes where the dominant driver of discomfort is contrast based or due to small bright sources in an otherwise dim scene.

5. CBDM Methods for Visual Comfort: 3/5 Phase

The 3-phase and 5-phase methods focus on the model and render steps. These methods fix the implementations of the material and sky models by discretizing the transmitting materials and sky dome in order to replace some steps of the rendering process with a matrix multiplication.

6. CBDM Methods for Visual Comfort: eDGPs

Like the 5-phase method, The enhanced-simplified daylight glare probability method, developed to overcome the limitations of illuminance only metrics, uses separate sampling and rendering assumptions for the indirect contribution and direct view rays. The adaptation level is captured by an illuminance value, but glare sources are identified with an image calculated for direct view ray contributions only.

7. Existing Options For Sampling a Point

In all of these methods, the sampling is treated as a fixed assumption.

7.b

Either directional sampling is directly integrated into an illuminance by the renderer,

7.c

or a high resolution image is generated.

7.d

This is because at intermediate image resolutions the accuracy of the results can be worse than an illuminance sample, and are unreliable for capturing contrast effects due to small sources.

7.e

So unlike sampling positions or timesteps which can be set at arbitrary spacing and easily tuned to the needs of the analysis, directional sampling is much more of an all or nothing choice; where the additional insights offered by an image can require 1 million times more data than a point sample. But is this really necessary?

7.f

Whether through direct image interpretation or any of the commonly used glare metrics, the critical information embedded in an HDR image is usually simplified to a small set of sources and background, each with a size, direction and intensity. We cannot directly sample this small set of rays because we do not know these important directions ahead of time, but how close can we get?

7.g

The raytraverse method provides a means to bridge the gap between point samples and high resolution images, allowing for a tunable tradeoff between simulation time and accuracy.

Our approach is structured by a wavelet space representation of the directional sampling. It works by applying a set of filters to an image to locate these important details.

8. Wavelet Decomposition

To match our sampling space, we apply these filters to a square image space based on the Shirley-Chiu disk to square transform, which preserves adjacency and area, both necessary for locating true details.

8.b

For each level of the decomposition, The high pass filters, applied across each axis (vertical, horizontal, and in combination) isolate the detail in the image, and the low pass filter performs an averaging yielding an image of half the size. This process is repeated, applying the high pass filters to the approximation, down to some base resolution. Each level of the decomposition stores the relative change in intensity at a particular resolution (or frequency).

8.c

The total size of the output arrays is the same as the original, and can be used to perfectly recover the original signal through the inverse transform.

The benefit to compression comes from the fact that the magnitude of the detail coefficients effectively rank the data in terms of their contribution to the reconstruction. By thresholding the coefficients, less important data can be discarded.

8.d

Even after discarding over 99% of the wavelet coefficients, the main image details are recoverable and only some minor artifacts have been introduced.

This property, that the wavelet coefficients rank the importance of samples at given resolutions, makes detail coefficients useful for guiding the sampling of view rays from a point.

9. Reconstruction Through Sampling

This process works as follows:

Beginning with a low resolution initial sampling the large scale features of the scene are captured.

Mimicking the wavelet transform, We apply a set of filters to this estimate and then use the resulting detail coefficients both to find an appropriate number of samples, and as probability distribution for the direction of these samples.

The new sample results returned by the renderer are used to update the estimate, which is lifted to a higher resolution.

This process is repeated up to a maximum resolution, equivalent to (or higher than) what a full resolution image might be rendered at.

10. Component Sampling

There are some cases where the wavelet based sampling will not find important details, such as specular views and reflections of the direct sun. Fortunately, because our method uses sky-patch coefficients to efficiently capture arbitrary sky conditions (similar to 3 phase and others), we can structure the simulation process in such a way to compensate for these misses. I refer you to our paper for details on how this works.

11. Results

Instead, I'll spend my remaining time sharing a few examples of scenes captured with: our approach, a high resolution reference and a matching uniform resolution image to demonstrate the benefits of variable sampling.

In addition to image reconstructions, the relative deviation from the reference is shown for vertical illuminance (characterizing energy conservation) and UGR (Unified Glare Rating, characterizing contrast), relative errors greater than 10% are highlighted in red.

This very glary scene highlights the different paths that light takes from the sun to the eye, including direct views, rough specular and diffuse reflections of the sun and sky. While the deviation in the low resolution image is unlikely to change a prediction in this case, the large errors show a failure case for uniform low-res sampling.

11.b

A more complex, but also more likely scenario is that roller shades will be closed. While there are open questions on how to evaluate the specular transmission of such materials, raytraverse does not introduce any substantial new errors to this process.

11.c

Raytraverse performs similarly well for partially open venetian blinds.

11.d Including deeper in a space where the floor reflection dominates.

11.e

Raytraverse, without virtual sources or other rendering tricks, handles the case of specular reflections of the direct sun, a difficult problem for low resolution sampling.

11.f

One case that we would expect raytraverse to struggle with would be a high frequency pattern like the dot frit shown here. And while the sampling does miss parts of the pattern, especially the lower contrast areas, enough of the detail is caught to meaningfully understand the image and, because of the direct sun view sampling, maintains high accuracy.

11.g

In cases where more image fidelity is desired, raytraverse can be tuned to increase the sampling rate with a proportional increase in simulation time, but in our paper we show that the low sampling rates previously shown achieve a high level of accuracy for field of view metrics.

12. Thank you

Thank you for watching my presentation.

4.2 History

4.2.1 1.4.2 (2023-11-17)

- updated readthedocsyaml to handle deprecation
- fixed source view point image drawing that caused to large a solar source (only effects images, not metrics)

4.2.2 1.4.1 (2023-10-11)

- fixed bug when using 3 or 4 candidate points for static point samplers
- added benchmark testing to repository
- improved srcviewpoint for use with captured hdrs
- initial work on 3-channel color support for CBDM runs

4.2.3 1.4.0 (2023-01-12)

- updated image generation for comppressedpoint
- added option to directly sample imagerenderer without transformation
- reorganized code breaking out python libraries into raytools

4.2.4 1.3.10 (2022-12-20)

- paralellize sensorintegrator matrix multiplication
- include interpolated/represented area in zonallightresult.rebase() output
- fix scikit-learn in requiirements.txt/setup.py
- add random patch color view to lightpointkd.direct_view() for diagrams

4.2.5 1.3.9 (2022-12-08)

- improvements to more easily load lightpoints directly from files without context
- api.load_config for better compatibility between scripting/CLI
- change eccentricity model in hvsgsm
- fix bug in autorotate in planmapper when result should be 0 or 90

4.2.6 1.3.8 (2022-11-04)

- 0 variance bug in image interpolation
- added boundaries to LightResults
- accept dew_point in ttsv format to skydata
- outtput skydata with dewpoint using skydata_dew
- pull gridhdr no longer needs zone if it is embedded in LightResult

4.2.7 1.3.7 (2022-10-26)

- updated resonse fit in hsvgm
- resolution option for pull 2 hdr cli
- modules directly available from import raytraverse
- ensure parameters set correctly so sun is always resampled in 1compdv
- bug in integrator log showing too many chunks
- rebase method added to basic LightResult
- rewrote lightpoint image interpolation
- SrcSamplerPt no longer uses accuracy, instead, set t0, at t1 with physical units
- t0 and t1 now instance properties (settable from init)
- added direct view options to raytu lp2img (warning, non-fisheye color throws error)
- preserve -ss in direct view sampling
- clean up srcview sampling (always distant) and fix double counting image when rough specular gets re-samples
- new factoring with craytraverse / renderer inheritance fixes rcontrib reset

4.2.8 1.3.6 (2022-07-28)

- add scale to sensor integrator forr proper unit conversion (lux by default)
- parallel processing in zonallightresult.pull2hdr
- add lightresult.pull2planhdr to match signature of zonallightresult
- add zonallightresult.rebase to make standard lightresult from zonal
- fixed bug in sunsplanekd.query_by_sun that returned all points, not just best matches
- added index() function to resultaxis
- bug fixes in sensorintegrator, needed additional function overrides and index broadcasting
- avoid IndexError at the end of skydata.maskindices
- add lightresult.merge (and cli interface with raytu merge) for combining LightResults
- change chunking of large calls to evaluate for better performance and the save intermediate results
- pass jitterrate to MaskedPlanMapper constructor
- rewrote RadianceFormatter.get_scene() parser, not based on file extensions
- bug in SamplerArea when operating with MaskedPlanMapper, possible to have no samples, leading to IndexError, fixed at self._mask initialization so atleast one cell is True.
- added gss to raytu imgmetric (no options yet, uses standard observer)

4.2.9 1.3.5 (2022-07-05)

- better memory management in zonal sensorintegrator
- plot each weight in srcsamplerpt when using detail/color
- slight reorganization in Integrator to accommodate sensorintegrator changes
- fixed bug in pull with -skyfilter but no -skyfill
- allow skydata write without scene
- change default sunrun parameter to -ab 0
- updated installation instructions and Dockerfiles to include radiance installation
- added adpatch for better control over default args in Rcontrib
- 2x speedup in translate.calc_omega by checking for containment before intersection left commented code for pygeos method, but it is slower without better way to read in voronoi (creation with pygeos only uses small fraction of points).
- formatting change in CLI docstrings to avoid error with latest docutils

4.2.10 1.3.4 (2022-06-21)

- do not use srcview for local light sources, include atleast 1 level of clean-up
- make sure kd tree is rebuilt when lucky squirrel
- ambient file handling in rtrace
- better memory management in reflection_search (still a problem?)
- new example config with proper settings
- with minsamp > 0 make sure from_pdf returns something so sampling can complete

4.2.11 1.3.3 (2022-06-07)

- static light source sampler, directly samples electric lights at appropriate level, will use lots of extra samples with very long thin fixtures
- color support in lightPointKD and samplers, but for now only works with imagesampler and sourcesampler because need to update skydata to work with color (and handle mixed data)
- use scene detail in sampler (in this case image reconstruction works better WITHOUT scene detail, new interpolation keywords fastc and highc for context interpolation)
- consolidated integrator/zonalintegrator and special methods dv/ds into one class
- changed zonal sunplane query algorithm: filter suns, penalize, query instead of filter suns, sort, filter points
- removed ptfilter keyword for zonal evaluation (new process does not use)
- sunplane normalization based on level 0 distance of sampled suns and level 0 distance of areas for level 0 sampled suns
- SensorIntegrator to process sensorplane results
- manage stranded open OS file descriptors
- wait to calculate omega on demand in lightplaneKD
- removed img2lf in imagetools, creates circular reference, need to add to different module
- allow None vector argument for lightplane initialization (cconstructs filename)
- zero pad hour labels in lightresult for proper file name sorting

- calc_omega method now passes "QJ" to qhull which seems to reliable return regions for all points in case of failure, distributed area among points sharing region (moved from integrator.helpers to translate) so LightPointKD can share
- fixed mistakes in GSS implementation and recalibrated

4.2.12 1.3.2 (2022-04-28)

- force 'fork' for multiprocessing to ensure radiance state is copied to processes
- restructure radiancerrenderers - not singleton, just a stateful class, pickleable with get/set state
- dummy skydatamask class useful for intializing with lightresult axes to handle fill
- value_array method added to ResultAxis for easier syntax
- settable sigma_c method in hvsgsm
- make integrator.helpers public for overrides
- supress warnings from radiance during reflection search
- implement ZonalIntegratorDV

4.2.13 1.3.1 (2022-04-19)

- moved craytraverse to separate repository, now a requirement
- implemented glare sensation model, not yet available from CLI

4.2.14 1.3.0 (2022-04-01)

- first version compatible on linux systems
- changed skyres specification to int (defining side) for consistency with other resolution parameters

4.2.15 1.2.8 (2022-03-15)

- include radius around sun and reflections when resampling view. for 3comp, -ss should be 0 for skyengine
- handle stray hits when resampling radius around sun
- new simtype: 1compdv / integratordv

4.2.16 1.2.7 (2022-03-01)

- parametric search radius for specguide in sunsamplerpt
- integratorDS checks whether it is more memory efficient to apply skyvectors before adding points
- fixed double printing of 360 direct_views
- exposd lowlight and threshold parameter access to cli (both imgmetric and evaluate)
- changed to general precision formatting for lightresult printing
- fixed -skyfilter in pull, needs a skydata file to correctly index, otherwise based on array size
- new sampling metric normalizations, can now control logging and pbars with scene parameter

4.2.17 1.2.6 (2022-02-19)

- add hours when available to skydata
- proper masking of 360 images
- integratorDS handles stray roughness from direct patch
- planmapper, z set to median instead of max, added autorotation/alignment
- bugs/features/consistency in LightResult, need better usage documentation
- directviews from cli (only works with sky)

4.2.18 1.2.5 (2022-02-15)

- integrated zonal calcs in cli
- fall back to regular light result when possible (but keep area)
- fixed bugs in LightResult, ZonalLightResult
- added physically based point spread calculation that ~matches gregs gblur script, but using acutal lorentzian from reference
- added blur psf to sources in image evaluation

4.2.19 1.2.4 (2021-12-03) (not posted until 2022-02-10)

- organized command line code
- use process pool for sun sampler when raytracing is fast (such as -ab 0 runs with dcomp)
- propogate plotp to child sampler if sampling one level
- separated utility command line to own entry point. fixed ambiguity in coordinate handedness of some functions (changed kwarg defaults)

4.2.20 1.2.3 (2021-09-03)

- fixed rcontrib to work with Radiance/HEAD, radiance version string includes commit
- daylightplane - add indirect to -ab 0 sun run (daysim/5-phase style)
- lightpointkd - handle adding points with same sample rays
- sampler - add repeat function to follow an existing sampling scheme
- lightresult - added print function
- scene - remove logging from scene class
- **cli.py**
  - new command imgmetric, extract rays from image and use same metricfuncs
  - mew command pull, filter and output 2d data frames from lightresult
  - add printdata option to suns, to see candidates or border
- make TStqdm progress bar class public
- **include PositionIndex calculation in BaseMetricSet**
  - new metrics: loggcr and position weighted luminance/gcr
- skymapper: filter candidates by positive dirnorm when initialized with epw/wea

- **imagetools: parallel process image metrics, also normalize peak with some assumptions**
- lightresult: accept slices for findices argument
- **sunsamplerpt: at second and thrid sampling levels supplement sampling with spec_guide at 1/100 the threshold. helps with imterior spaces to find smaller patches of sun**
- positionindex: fix bug transcribed from evalglare with the positionindex below horizon

4.2.21 1.2.0/2 (2021-05-24)

- command line interface development

4.2.22 1.1.2 (2021-02-19)

- improved documentation

4.2.23 1.1.0/1 (2021-02-10)

- refactor code to operate on a single point at a time

4.2.24 1.0.4 (2020-11-18)

- create and manage log file (attribute of Scene) for run directories
- possible fix for bug in interpolate_kd resulting in index range errors
- protect imports in cli.py so documentation can be built without installing

4.2.25 1.0.3 (2020-11-10)

- new module for calculating position based on retinal features
- view specifications for directview plotting
- options for samples/weight visibility on directview plotting

4.2.26 0.2.0 (2020-09-25)

- Build now includes all radiance dependencies to setup multi-platform testing
- In the absence of craytraverse, sampler falls back to SPRenderer
- install process streamlined for developer mode
- travis ci deploys linux and mac wheels directly to pypi
- **release.sh should be run after updating this file, tests past locally and docs build.**

4.2.27 0.1.0 (2020-05-19)

- First release on PyPI.

4.3 Index

4.4 Search

CHAPTER

FIVE

CITATION

Either the latest or specific releases of this software are archived with a DOI at zenodo. See: https://doi.org/10.5281/zenodo.4091318

Please cite this journal article for a description and validation of the method:

> Stephen Wasilewski, Lars O. Grobe, Jan Wienold, Marilyne Andersen, *Efficient Simulation for Visual Comfort Evaluations*, Energy and Buildings, Volume 267, 2022, 112141, ISSN 0378-7788, https://doi.org/10.1016/j.enbuild.2022.112141.

Additional peer-reviewed references related to this software:

> Stephen Wasilewski, Lars O. Grobe, Roland Schregle, Jan Wienold, and Marilyne Andersen. 2021. *Raytraverse: Navigating the Lightfield to Enhance Climate-Based Daylight Modeling*. In 2021 Proceedings of the Symposium on Simulation in Architecture and Urban Design. https://infoscience.epfl.ch/record/290685?ln=en

> Quek, G., Wasilewski, S., Wienold, J., Andersen, M., 2021a. *Spatial evaluation of potential saturation and contrast effects of discomfort glare in an open-plan office*, in: BS2021. Presented at the Building Simulation 2021 Conference, Bruges, Belgium. https://infoscience.epfl.ch/record/288945

CHAPTER

SIX

LICENCE

CHAPTER

SEVEN

ACKNOWLEDGEMENTS

Thanks to additional project collaborators and advisors Marilyne Andersen, Lars Grobe, Roland Schregle, Jan Wienold, and Stephen Wittkopf

This software development was financially supported by the Swiss National Science Foundation as part of the ongoing research project "Light fields in climate-based daylight modeling for spatio-temporal glare assessment" (SNSF #179067).

CHAPTER

EIGHT

SOFTWARE CREDITS

- Raytraverse uses Radiance
- As well as all packages listed in the requirements.txt file, raytraverse relies heavily on the Python packages numpy, scipy, and for key parts of the implementation.
- C++ bindings, including exposing core radiance functions as methods to the renderer classes are made with pybind11
- Installation and building from source uses cmake and scikit-build
- This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

PYTHON MODULE INDEX

r

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z