raytraverse Documentation

Release 1.2.2

Stephen Wasilewski

May 24, 2021

COMMAND LINE INTERFACE

raytraverse is a complete workflow for climate based daylight modelling, simulation, and 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.

- 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 .
```

note that on first run the skycalc module may download some auxilary data which could take a minute, after that first run start-up is much faster.

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 the src/ directory for more.

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

GETTING STARTED

the following example script shows the basic workflow for a complete simulation it can be saved to a local file with:

```
raytraverse examplescript > example.py
```

or the file is located at raytraverse/example.py

```
# -*- coding: utf-8 -*-
# Copyright (c) 2020 Stephen Wasilewski, HSLU and EPFL
# ============================================================================
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
# ============================================================================


"""example script for raytraverse."""

import numpy as np
from raytraverse.mapper import PlanMapper, SkyMapper, ViewMapper
from raytraverse.scene import Scene
from raytraverse.renderer import Rtrace, Rcontrib
from raytraverse.sampler import SkySamplerPt, SamplerArea, SamplerSuns
from raytraverse.sky import SkyData, skycalc
from raytraverse.lightfield import LightResult
from raytraverse.evaluate import MetricSet


# -------------------
# update these values
# -------------------

# output directory where are simulation results are written
out = "outdir"
# the radiance scene files (all materials and geometries, no sources)
scene_files = "room.rad"
# a horizontal analysis plane, for this demo should be 2x2 (or change ptres)
zone = "plane.rad"
# an 8760 epw or wea file with location data. although note that wea files
# do not include dew-point, a potentially important parameter of the perez
# sky model.
epw = "weather.epw"
# an output file path for writing compressed binary results
output = "metrics.npz"


# NOTE: many of the setting overrides are for demonstration only
# and may not yield accurate or meaningful results, they are accuracy
```

(continues on next page)

(continued from previous page)

```
# reductions made in order for this script to run quickly (less than 1 minute on
# a reasonably fast laptop).
# In this example, only directions are dynamically sampled, but position and
# solar sources are sampled for 1 level only at a low resolution. to sample
# points dynamically, set SamplerArea(nlev=3, jitter=True).  To dynamically
# sample suns, change  SamplerSuns(nlev=3) or other appropriate level.
# refer to the documentation for adjusting the sampling scheme used in
# directional sampling (SamplerPt, SkySamplerPt, SunSamplerPt).
def main():
    loc = skycalc.get_loc_epw(epw)
    # Make octree, manage output file directory
    scn = Scene(out, scene_files)
    # initialize sampling schemes and boundaries for position and solar source
    # sampling
    pm = PlanMapper(zone, ptres=2.0)
    sm = SkyMapper(loc=loc)
    # initialize rendering engines (note settings are appended to
    # Renderer.defaultargs, which are different from rtrace/rcontrib defaults)
    rcontrib = Rcontrib("-ab 2 -ad 4 -c 1000", scene=scn.scene)
    rtrace = Rtrace("-ab 2 -c 1", scene=scn.scene)

    # initialize sky point sampler and then call an area sampler to simulate
    # sky contribution
    sk_engine = SkySamplerPt(scn, rcontrib, accuracy=2.0)
    skysampler = SamplerArea(scn, sk_engine, accuracy=2.0, nlev=1, jitter=False)
    skyfield = skysampler.run(pm)
    print(rcontrib)
    rcontrib.reset()

    # to modify parameters for sun/pt sampler pass arguments to the ptkwargs
    # argument of SamperSuns
    ptkwargs = dict(accuracy=2.0)
    areakwargs = dict(accuracy=2.0, nlev=1, jitter=False)
    sunsampler = SamplerSuns(scn, rtrace, nlev=1,
                             ptkwargs=ptkwargs, areakwargs=areakwargs)

    daylightfield = sunsampler.run(sm, pm, specguide=skyfield)
    rtrace.reset()
    # calculate sky patch and sun contributions
    sd = SkyData(epw)

    # make a set of points to evaluate (here a regular grid at the final
    # sampling level (assuming nlev=2)
    pts = pm.point_grid(False, 1)

    # The raytraverse.lightfield.DayLightPlaneKD object holds the complete
    # sampling results. individual point data can be loaded by querying with
    # a 6 element solar position and plan cooordinate
    # the first sun in skydata:
    sun = sd.sun[0, 0:3]
    # the first point in our grid:
    pt = pts[0]
    i, d = daylightfield.query((*sun, *pt))
    sun_lightpoint = daylightfield.data[i[0]]
    # we can also query for the closest skypoint:
    j, d = daylightfield.skyplane.query(pt)
    sky_lightpoint = daylightfield.skyplane.data[j]
    # for energy conserving operations (avg luminance, illuminance, an image)
    # we can evaluate the lightpoints seperately and then add, but for
    # general analysis we need to combiine the points first:
    sun_sky_point = sky_lightpoint.add(sun_lightpoint)
```

(continues on next page)

(continued from previous page)

```
    # because the sources are combined now we need to concatenate the solar
    # value onto the rest of the skyvector:
    skyvec = np.concatenate((sd.smtx[0], sd.sun[0, 3]), axis=None)

    # and now we can make an hdr image:
    vm = ViewMapper((0, -1, 0), viewangle=180)
    # file named by hour of year
    outf = f"hour_{sd.maskindices[0]:04d}.hdr"
    sun_sky_point.make_image(outf, skyvec, vm)

    # or calculate DGP and UGP:
    vol = sun_sky_point.evaluate(skyvec, vm)
    metrics = MetricSet(*vol, vm)
    print(f"DGP: {metrics.dgp} UGP: {metrics.ugp}")

    # for larger sets of metric evaluations, DaylightPlaneKD has its' own
    # evaluate function to do bulk processing (also will use multiprocessing)
    # Note that you can mask the SkyData object to limit the evaluation
    # such as to only the first day of the year:
    sd.mask = np.arange(24)

    # view directions
    vdirs = np.array([[1, 0, 0], [0, 1, 0], [-1, 0, 0], [0, -1, 0]],dtype=float)
    # choose from ["illum", "avglum", "gcr", "ugp", "dgp", "tasklum", "backlum",
    # "dgp_t1", "log_gc", "dgp_t2", "ugr", "threshold", "pwsl2", "view_area",
    # "backlum_true", "srcillum", "srcarea", "maxlum"] or inherit
    # raytraverse.evaluate.MetricSet and pass metricclass=YourMetricClass
    metrics = ['illum', 'dgp', 'ugp']
    result = daylightfield.evaluate(sd, pts, vdirs, metrics=metrics)
    result.write(output)

    # to reload these results:
    result = LightResult(output)

    # this result object stores a 4D array:
    # (sky, point, view, metric)
    # to reshape/slice results for viewing or analysis, use the pull method:
    # here we get  the south facing view for the first point in our grid:
    data, axes, names = result.pull("sky", "metric", findices=[[0], [3]])
    print("hour", *axes[2])
    for d, h in zip(data[0], axes[1]):
        print(h, *d)


if __name__ == '__main__':
    main()
```

CHAPTER

FOUR

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. To duplicate the run shown in the example script above save the following to options.cfg:

```
[raytraverse_scene]
out = outdir
scene = room.rad

[raytraverse_area]
ptres = 2.0
zone = plane.rad

[raytraverse_suns]
loc = weather.epw
epwloc = True

[raytraverse_skydata]
wea = weather.epw

[raytraverse_skyengine]
accuracy = 2.0
rayargs = -ab 2 -ad 4 -c 1000

[raytraverse_sunengine]
accuracy = 2.0
rayargs = -ab 2 -c 1

[raytraverse_skyrun]
accuracy = 2.0
jitter = True
nlev = 2
overwrite = False

[raytraverse_sunrun]
accuracy = 2.0
nlev = 2
srcaccuracy = 2.0
```

and then from the command line run:

```
raytraverse -c options.cfg skyrun sunrun
```

4.1 raytraverse

```
raytraverse [OPTIONS] COMMAND1 [ARGS]... [COMMAND2 [ARGS]...]...
```

the raytraverse executable is a command line interface to the raytraverse python package for running and evaluating climate based daylight models. sub commands of raytraverse can be chained but should be invoked in the order given.

the easiest way to manage options is to use a configuration file, to make a template:

```
raytraverse --template > run.cfg
```

after adjusting the settings, than each command can be invoked in turn and any dependencies will be loaded with the correct options, a complete run and evaluation can then be called by:

```
raytraverse -c run.cfg skyrun sunrun
```

as all required precursor commands will be invoked automatically as needed.

Options

VALUE OPTIONS:

-config, -c <PATH>
: path of config file to load

-n <INTEGER>
: sets the environment variable RAYTRAVERSE_PROC_CAP set to 0 to clear (parallel processes will use cpu_limit)

-out <DIRECTORY>

FLAGS (DEFAULT FALSE):

--template, --no-template
: write default options to std out as config

  Default False

HELP:

-opts, --opts
: check parsed options

  Default False

--debug
: show traceback on exceptions

  Default False

--version
: Show the version and exit.

  Default False

Commands

scene
define scene files for renderer and output...

area
define sampling area

suns
define solar sampling space

skydata
define sky conditions for evaluation

skyengine
initialize engine for skyrun

sunengine
initialize engine for sunrun

skyrun
run scene under sky for a set of points...

sunrun
run scene for a set of suns (defined by...

evaluate
evaluate metrics and/or make hdr for a...

examplescript
print an example workflow for script...

4.1.1 scene

```
raytraverse scene [OPTIONS]
```

define scene files for renderer and output directory

Effects

- creates outdir and outdir/scene.oct

Options

VALUE OPTIONS:

-out <DIRECTORY>

-scene <TEXT>
space separated list of radiance scene files (no sky) or precompiled octree

FLAGS (DEFAULT TRUE):

--log, --no-log
log progress to stderr

Default True

--reload, --no-reload
if a scene already exists at OUT reload it, note that if this is False and overwrite is False, the program will abort

Default True

FLAGS (DEFAULT FALSE):

--overwrite, --no-overwrite
Warning! if set to True all files inOUT will be deleted

Default False

HELP:

-opts, --opts
check parsed options

Default False

--debug
show traceback on exceptions

Default False

--version
Show the version and exit.

Default False

4.1.2 area

```
raytraverse area [OPTIONS]
```

define sampling area

Effects

- None

Options

VALUE OPTIONS:

-name <TEXT>
name for zone/point group (impacts file naming)

Default plan

-ptres <FLOAT>
initial sampling resolution for points

Default 1.0

-rotation <FLOAT>
positive Z rotation for point grid alignment

Default 0.0

-static_points <TEXT>
points to simulate, this can be a .npy file, a whitespace seperated text file or entered as a string with commas between components of a point and spaces between points. points should either all be 3 componnent (x,y,z) or 6 component (x,y,z,dx,dy,dz) but the dx,dy,dz is ignored

-zheight <FLOAT>
replaces z in points or zone

-zone <TEXT>
zone boundary to dynamically sample. can either be a radiance scene file defining a plane to sample or an array of points (same input options as -static_points). Pointsare used to define a convex hull with an offset of 1/2*ptres in which to sample. Note that if static_pointsand zone are both give, static_points is silently ignored

HELP:

-opts, --opts
check parsed options

Default False

--debug
show traceback on exceptions

Default False

--version
Show the version and exit.

Default False

4.1.3 suns

```
raytraverse suns [OPTIONS]
```

define solar sampling space

Effects

- None

Options

VALUE OPTIONS:

-loc <TEXT>
can be a number of formats:

1. a string of 3 space seperated values (lat lon mer) where lat is +west and mer is tz*15 (matching gendaylit).
2. a string of comma seperated sun positions with multiple items seperated by spaces: “0,-.7,.7 .7,0,.7” following the shape requirements of 3.
3. a file loadable with np.loadtxt) of shape (N, 2), (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.

4. path to an epw or wea formatted file: solar positions are generated and used as candidates unless –epwloc is True.

5. None (default) all possible sun positions are considered

in the case of a location, sun positions are considered valid when in the solar transit for that location. for candidate options (2., 3., 4.), sun positions are drawn from this set (with one randomly chosen from all candidates within adaptive grid.

-name <TEXT>
name for solar sourcee group (impacts file naming)

Default suns

-skyro <FLOAT>
counterclockwise sky-rotation in degrees (equivalent to clockwise project north rotation)

Default 0.0

-sunres <FLOAT>
initial sampling resolution for suns

Default 30.0

FLAGS (DEFAULT FALSE):

--epwloc, --no-epwloc
if True, use location from epw/wea argument to -loc as a transit mask (like -loc option 1.) instead of as a list of candidate sun positions.

Default False

HELP:

-opts, --opts
check parsed options

Default False

--debug
show traceback on exceptions

Default False

--version
Show the version and exit.

Default False

4.1.4 skydata

```
raytraverse skydata [OPTIONS]
```

define sky conditions for evaluation

Effects

- Invokes scene
- write outdir/name.npz (SkyData initialization object)

Options

VALUE OPTIONS:

-ground_fac <FLOAT>
ground reflectance

Default 0.15

-loc <FLOATS>
location data given as 'lat lon mer' with + west of prime meridian overrides location data in wea

-minalt <FLOAT>
minimum solar altitude for daylight masking

Default 2.0

-mindiff <FLOAT>
minumum diffuse horizontal irradiance for daylight masking

Default 5.0

-name <TEXT>
output file name for skydata

Default skydata

-skyres <FLOAT>
approximate square patch size in degrees (must match argument given to skyengine)

Default 10.0

-skyro <FLOAT>
angle in degrees counter-clockwise to rotate sky (to correct model north, equivalent to clockwise rotation of scene)

Default 0.0

-wea <TEXT>
path to epw, wea, .npy file or np.array, or .npz file,if loc not set attempts to extract location data (if needed).

FLAGS (DEFAULT TRUE):

--reload, --no-reload
reload saved skydata if it exists in scene directory

Default True

HELP:

-opts, --opts
check parsed options

Default False

--debug
show traceback on exceptions

Default False

--version
Show the version and exit.

Default False

4.1.5 skyengine

```
raytraverse skyengine [OPTIONS]
```

initialize engine for skyrun

Effects

- Invokes scene
- creates outdir/scene_sky.oct

Options

VALUE OPTIONS:

-accuracy <FLOAT>
a generic accuracy parameter that sets the threshold variance to sample. A value of 1 will have a sample count at the final sampling level equal to the number of directions with a contribution variance greater than .25

Default 1.0

-fdres <INTEGER>
the final directional sampling resolution, yielding a grid of potential samples at 2^fdres x 2^fdres per hemisphere

Default 9

-idres <INTEGER>
the initial directional sampling resolution. each side of the sampling square (representing a hemisphere) will be subdivided 2^idres, yielding 2^(2*idres) samples and a resolution of 2^(2*idres)/(2pi) samples/steradian. this value should be smaller than 1/2 the size of the smallest view to an aperture that should be captured with 100% certainty

Default 5

-rayargs <TEXT>
additional arguments to pass to the rendering engine

-skyres <FLOAT>
approximate resolution for skypatch subdivision (in degrees). Patches will have (rounded) size skyres x skyres. So if skyres=10, each patch will be 100 sq. degrees (0.03046174197 steradians) and there will be 18 * 18 = 324 sky patches. Must match argument givein to skydata

Default 10.0

FLAGS (DEFAULT TRUE):

--default-args, --no-default-args
use raytraverse defaults before -rayargs, if False, uses radiance defaults

Default True

HELP:

-opts, --opts
check parsed options

Default False

--debug
show traceback on exceptions

Default False

--version
Show the version and exit.

Default False

4.1.6 sunengine

```
raytraverse sunengine [OPTIONS]
```

initialize engine for sunrun

Effects

- Invokes scene

Options

VALUE OPTIONS:

-accuracy <FLOAT>
a generic accuracy parameter that sets the threshold variance to sample. A value of 1 will have a sample count at the final sampling level equal to the number of directions with a contribution variance greater than .25

Default 1.0

-fdres <INTEGER>
the final directional sampling resolution, yielding a grid of potential samples at 2^fdres x 2^fdres per hemisphere

Default 10

-idres <INTEGER>
: the initial directional sampling resolution. each side of the sampling square (representing a hemisphere) will be subdivided 2^idres, yielding 2^(2*idres) samples and a resolution of 2^(2*idres)/(2pi) samples/steradian. this value should be smaller than 1/2 the size of the smallest view to an aperture that should be captured with 100% certainty

  Default 5

-maxspec <FLOAT>
: the maximum value in the specular guide considered as a specular reflection source. Above this value it is assumed that these are direct view rays to the source so are not sampled. in the case of low vlt glazing, this value should be reduced. In mixed (high-low) vlt scenes the specular guide will either over sample (including direct views) or under sample (miss specular reflections) depending on this setting.

  Default 0.2

-rayargs <TEXT>
: additional arguments to pass to the rendering engine

-slimit <FLOAT>
: the minimum value in the specular guide considered as a potential specular reflection source, in the case of low vlt glazing, this value should be reduced.

  Default 0.01

-speclevel <INTEGER>
: at this sampling level, pdf is made from brightness of sky sampling rather than progressive variance to look for fine scale specular highlights, this should be atleast 1 level from the end and the resolution of this level should be smaller than the size of the source

  Default 9

FLAGS (DEFAULT TRUE):

--default-args, --no-default-args
: use raytraverse defaults before -rayargs, if False, uses radiance defaults

  Default True

HELP:

-opts, --opts
: check parsed options

  Default False

--debug
: show traceback on exceptions

  Default False

--version
: Show the version and exit.

  Default False

4.1.7 skyrun

```
raytraverse skyrun [OPTIONS]
```

run scene under sky for a set of points (defined by area)

Effects

- Invokes scene
- Invokes area (no effects)
- Invokes skyengine
- **creates outdir/area.name/sky_points.tsv**
  - contents: 5cols x N rows: [sample_level idx x y z]
- **creates outdir/area.name/sky/######.rytpt**
  - each file is a LightPointKD initialization object

Options

VALUE OPTIONS:

-accuracy <FLOAT>
parameter to set threshold at sampling level relative to final level threshold (smaller number will increase sampling)

Default 1.0

-nlev <INTEGER>
number of levels to sample (final resolution will be ptres/2^(nlev-1))

Default 3

FLAGS (DEFAULT TRUE):

--jitter, --no-jitter
jitter samples on plane within adaptive sampling grid

Default True

FLAGS (DEFAULT FALSE):

--overwrite, --no-overwrite
If True, reruns sampler when invoked, otherwise will first attempt to load results

Default False

HELP:

-opts, --opts
check parsed options

Default False

--debug
show traceback on exceptions

Default False

--version
Show the version and exit.

Default False

4.1.8 sunrun

```
raytraverse sunrun [OPTIONS]
```

run scene for a set of suns (defined by suns) for a set of points (defined by area)

Effects

- Invokes scene
- Invokes area (no effects)
- Invokes sunengine (no effects)
- invokes skyrun (if guided=True)
- **creates outdir/area.name/sun_####_points.tsv**
  - contents: 5cols x N rows: [sample_level idx x y z]
- **creates outdir/area.name/sky/sun_####/######.rytpt**
  - each file is a LightPointKD initialization object

Options

VALUE OPTIONS:

-accuracy <FLOAT>
parameter to set threshold at sampling level relative to final level threshold (smaller number will increase sampling)

Default 1.0

-nlev <INTEGER>
number of levels to sample (final resolution will be ptres/2^(nlev-1))

Default 3

-srcaccuracy <FLOAT>
parameter to set threshold at sampling level relative to final level threshold (smaller number will increase sampling)

Default 1.0

-srcnlev <INTEGER>
number of levels to sample (final resolution will be sunres/2^(nlev-1))

Default 3

FLAGS (DEFAULT TRUE):

--guided, --no-guided
If True, uses skysampling results to guide sun sampling this is necessary if the model has any specular reflections, will raise an error if skyrun has not been called yet.

Default True

--jitter, --no-jitter
jitter samples on plane within adaptive sampling grid

Default True

--srcjitter, --no-srcjitter
jitter solar source within adaptive sampling grid for candidate SkyMappers, only affects weighting of selecting candidates in the same grid true positions are still used

Default True

FLAGS (DEFAULT FALSE):

--recover, --no-recover
If True, recovers existing sampling

Default False

HELP:

-opts, --opts
check parsed options

Default False

--debug
show traceback on exceptions

Default False

--version
Show the version and exit.

Default False

4.1.9 evaluate

```
raytraverse evaluate [OPTIONS]
```

evaluate metrics and/or make hdr for a single sensor location

Prequisites

- skyrun and sunrun must be manually invoked prior to this

Effects

- Invokes scene
- Invokes skydata
- invokes area (no effects)
- invokes suns (no effects)
- if hdr=True, writes: <basename>_#####.hdr
- if metric=True, writes: <basename>.tsv

Options

VALUE OPTIONS:

-basename <TEXT>
basename for hdr outputs will write: <basename>_#####.hdr for each value in skymask (or range(len(skydata)) if skymask is None).

Default results

-metrics <TEXTS>
metrics to compute, choices: [“illum”, “avglum”, “gcr”, “ugp”, “dgp”, “tasklum”, “backlum”, “dgp_t1”, “log_gc”, “dgp_t2”, “ugr”, “threshold”, “pwsl2”, “view_area”, “backlum_true”, “srcillum”, “srcarea”, “maxlum”]

Default illum dgp ugp

-res <INTEGER>
image resolution

Default 800

-sensor <FLOATS>
the sensor location (6 component vector ‘x y z dx dy dz’)

-skymask <INTS>
mask to reduce output from full SkyData, enter as index rows in wea/epw file using space seperated list or python range notation:

- 370 371 372 (10AM-12PM on jan. 16th)
- 12:8760:24 (everyday at Noon)

-viewangle <FLOAT>
applies to both image and metrics, should be <= 180 or 360

Default 180.0

FLAGS (DEFAULT TRUE):

--metric, --no-metric
calculate metrics

Default True

FLAGS (DEFAULT FALSE):

--hdr, --no-hdr
make an image for every unmasked item in skydata

Default False

--interpolate, --no-interpolate
use linear iinterpolation in image output

Default False

HELP:

-opts, --opts
check parsed options

Default False

--debug
show traceback on exceptions

Default False

--version
Show the version and exit.

Default False

4.1.10 examplescript

```
raytraverse examplescript [OPTIONS]
```

print an example workflow for script based/api level access to raytraverse

Options

HELP:

-opts, --opts
check parsed options

Default False

--debug
show traceback on exceptions

Default False

--version
Show the version and exit.

Default False

4.2 raytraverse.scene

4.2.1 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

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)

4.2.2 Scene

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

Bases: raytraverse.scene.basescene.BaseScene

container for radiance scene description

Parameters

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

4.2.3 ImageScene

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

Bases: raytraverse.scene.basescene.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)

4.3 raytraverse.mapper

4.3.1 Mapper

class raytraverse.mapper.**Mapper**(*dxyz=0.0, 0.0, 1.0*, *sf=1, 1*, *bbox=0, 0, 1, 1*, *aspect=None*, *name='mapper'*, *origin=0, 0, 0*)

Bases: object

translate between world and normalized UV space. do not use directly, instead use an inheriting class.

Parameters

- **sf** (*tuple np.array, optional*) – scale factor for each axis (array of length(2)
- **bbox** (*tuple np.array, optional*) – bounding box for mapper shape (2, 2)
- **name** (*str, optional*) – used for output file naming

property aspect

property dxyz
(float, float, float) central view direction

property bbox
bounding box of view

Type np.array of shape (2,2)

view2world(*xyz*)
rotate vectors from view direction to world Z

world2view(*xyz*)
rotate vectors from world Z to view direction

xyz2uv(*xyz*)
transform from world xyz space to mapper UV space

uv2xyz(*uv*, *stackorigin=False*)
transform from mapper UV space to world xyz

static idx2uv(*idx*, *shape*, *jitter=True*)

Parameters

- **idx** (*flattened index*) –
- **shape** – the shape to unravel into
- **jitter** (*bool, optional*) – randomly offset coordinates within grid

Returns **uv** – uv coordinates

Return type np.array

static uv2idx(*uv*, *shape*)

xyz2vxy(*xyz*)
transform from world xyz to view image space (2d)

vxy2xyz(*xy*, *stackorigin=False*)
transform from view image space (2d) to world xyz

framesize(*res*)

pixels(*res*)
generate pixel coordinates for image space

pixelrays(*res*)
world xyz coordinates for pixels in view image space

ray2pixel(*xyz*, *res*, *integer=True*)
world xyz to pixel coordinate

pixel2ray(*pxy*, *res*)
pixel coordinate to world xyz vector

pixel2omega(*pxy*, *res*)
pixel area

in_view(*vec*, *indices=True*)
generate mask for vec that are in the field of view

header(***kwargs*)

init_img(*res=512*, ***kwargs*)
Initialize an image array with vectors and mask

Parameters

- **res** (*int, optional*) – image array resolution
- **kwargs** – passed to self.header

Returns

- **img** (*np.array*) – zero array of shape (res, res)
- **vecs** (*np.array*) – direction vectors corresponding to each pixel (img.size, 3)
- **mask** (*np.array*) – indices of flattened img that are in view
- **mask2** (*np.array None*) –

  if ViewMapper has inverse, mask for opposite view, usage:

```
add_to_img(img, vecs[mask], mask)
add_to_img(img[res:], vecs[res:][mask2], mask2
```

- **header** (*str*)

add_vecs_to_img(*img*, *v*, *channels=1, 0, 0*, *grow=0*, ***kwargs*)

plot(*xyz*, *outf*, *res=1000*, *grow=1*, ***kwargs*)

4.3.2 AngularMixin

class raytraverse.mapper.angularmixin.**AngularMixin**
Bases: object

includes overrides of transformation functions for angular type mapper classes. Inherit before raytraverse.mapper.Mapper eg:

```
NewMapper(AngularMixin, Mapper)
```

initialization of NewMapper must include declarations of:

```
self._viewangle = viewangle
self._chordfactor = chordfactor
self._ivm = ivm
```

xyz2uv(*xyz*)
transform from world xyz space to mapper UV space

uv2xyz(*uv*, *stackorigin=False*)
transform from mapper UV space to world xyz

xyz2vxy(*xyz*)
transform from world xyz to view image space (2d)

vxy2xyz(*xy*, *stackorigin=False*)
transform from view image space (2d) to world xyz

static framesize(*res*)

pixelrays(*res*)
world xyz coordinates for pixels in view image space

pixel2omega(*pxy*, *res*)
pixel solid angle

in_view(*vec*, *indices=True*)
generate mask for vec that are in the field of view (up to 180 degrees) if view aspect is 2, only tests against primary view direction

header(*pt=0, 0, 0*, ***kwargs*)

init_img(*res=512*, *pt=0, 0, 0*, ***kwargs*)
Initialize an image array with vectors and mask

Parameters

- **res** (*int, optional*) – image array resolution
- **pt** (*tuple, optional*) – view point for image header

Returns

- **img** (*np.array*) – zero array of shape (res*self.aspect, res)
- **vecs** (*np.array*) – direction vectors corresponding to each pixel (img.size, 3)
- **mask** (*np.array*) – indices of flattened img that are in view
- **mask2** (*np.array None*) –

  if ViewMapper is 360 degree, include mask for opposite view to use:

```
add_to_img(img, vecs[mask], mask)
add_to_img(img[res:], vecs[res:][mask2], mask2)
```

- **header** (*str*)

add_vecs_to_img(*img*, *v*, *channels=1, 0, 0*, *grow=0*, *fisheye=True*)

property viewangle
view angle

property ivm
viewmapper for opposite view direction (in case of 360 degree view

ctheta(*vec*)
cos(theta) (dot product) between view direction and vec

radians(*vec*)
angle in radians betweeen vieew direction and vec

degrees(*vec*)
angle in degrees betweeen vieew direction and vec

4.3.3 ViewMapper

class raytraverse.mapper.**ViewMapper**(*dxyz=0.0, 1.0, 0.0*, *viewangle=360.0*, *name='view'*)
Bases: *raytraverse.mapper.angularmixin.AngularMixin*, raytraverse.mapper.mapper.Mapper

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

Parameters

- **dxyz** (*tuple, optional*) – central view direction
- **viewangle** (*float, optional*) – if < 180, the horizontal and vertical view angle, if greater, view becomes 360,180

property aspect

property dxyz
(float, float, float) central view direction

4.3.4 SkyMapper

class raytraverse.mapper.**SkyMapper**(*loc=None*, *skyro=0.0*, *sunres=20.0*, *name='sky'*)

Bases: *raytraverse.mapper.angularmixin.AngularMixin*, raytraverse.mapper.mapper.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 sunres

property loc

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

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

4.3.5 PlanMapper

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

Bases: raytraverse.mapper.mapper.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.
- **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
- **str, optional** (*name*) – plan mapper name used for output file naming

ptres = None
point resolution for area look ups and grid

Type float

property dxyz
(float, float, float) central view direction

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

borders()
world coordinate vertices of planmapper boundaries

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

shape(*level=0*)

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

Returns shape (N, 3)

Return type np.array

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

Returns shape (N, 2)

Return type np.array

4.4 raytraverse.formatter

4.4.1 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 add_source(*scene*, *src*)
add source files to compiled scene

static get_skydef(*color*, *ground=True*, *name='skyglow'*)
assemble sky definition

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

static extract_sources(*srcdef*, *accuracy*)
scan scene file for sun source definitions

4.4.2 RadianceFormatter

class raytraverse.formatter.**RadianceFormatter**
Bases: raytraverse.formatter.formatter.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 add_source(*scene*, *src*)
add source files to compiled scene

static get_skydef(*color*, *ground=True*, *name='skyglow'*)
assemble sky definition

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

static extract_sources(*srcdef*, *accuracy*)
scan scene file for sun source definitions

4.5 raytraverse.renderer

4.5.1 Renderer

class raytraverse.renderer.**Renderer**
Bases: object

virtual singleton renderer class. the Renderer is implemented as a singleton as specific subclasses (rtrace, rcontrib) have many global variables set at import time. This ensures the python object is connected to the current state of the engine c++-class.

All renderer classes are callable with with a numpy array of shape (N,6) representing the origin and direction of ray samples to calculate.

args = None

instance = <raytraverse.renderer.renderer._VirtEngine object>

scene = None

classmethod set_args(*args*, *nproc=None*)

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

4.5.2 RadianceRenderer

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

Bases: raytraverse.renderer.renderer.Renderer

Virtual class for wrapping c++ Radiance renderer executable classes

Do not use directly, either subclass or use existing: Rtrace, Rcontrib

name = 'radiance_virtual'

engine = <cRtrace>
raytraverse.crenderer.cRtrace

srcn = 1

defaultargs = ''

args = None

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.

4.5.3 Rtrace

class raytraverse.renderer.**Rtrace**(*rayargs=None*, *scene=None*, *nproc=None*, *default_args=True*, *direct=False*)

Bases: raytraverse.renderer.radiancerenderer.RadianceRenderer

singleton wrapper for c++ raytrraverse.crenderer.cRtrace class

this class sets default arguments, helps with initialization and setting cpu limits of the cRtrace instance. see raytrraverse.crenderer.cRtrace for more details.

Parameters

- **rayargs** (*str, optional*) – argument string (options and flags only) raises ValueError if arguments are not recognized by cRtrace.
- **scene** (*str, optional*) – path to octree
- **nproc** (*int, optional*) – if None, sets nproc to cpu count, or the RAYTRAVERSE_PROC_CAP environment variable
- **default_args** (*bool, optional*) – if True, prepend default args to rayargs parameter
- **direct** (*bool, optional*) – if True use Rtrace.directargs in place of default (also if True, sets default_args to True.

Examples

Basic Initialization and call:

```
r = renderer.Rtrace(args, scene)
ans = r(vecs)
# ans.shape -> (vecs.shape[0], 1)
```

If rayargs include cache files (ambient cache or photon map) be careful with updating sources. If you are going to swap sources, update the arguments as well with the new paths:

```
r = renderer.Rtrace(args, scene)
r.set_args(args.replace("temp.amb", "temp2.amb"))
r.load_source(srcdef)
```

Note that if you are using ambient caching, you must give an ambient file, because without a file ambient values are not shared across processes or successive calls to the instance.

name = 'rtrace'

engine = <cRtrace>
raytraverse.crenderer.cRtrace

defaultargs = '-av 0 0 0 -aa 0 -ab 7 -ad 128 -as 0 -c 10 -as 0 -lw 1e-5'

directargs = '-av 0 0 0 -ab 0 -lr 0'

usedirect = False

classmethod get_default_args()
return default arguments of the class

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

Parameters

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

classmethod update_ospec(*vs*)
set output of cRtrace instance

Parameters **vs** (*str*) –

output specifiers for rtrace:: o origin (input) d direction (normalized) v value (radiance) V contribution (radiance) w weight W color coefficient l effective length of ray L first intersection distance c local (u,v) coordinates p point of intersection n normal at intersection (perturbed) N normal at intersection (unperturbed) r mirrored value contribution x unmirrored value contribution R mirrored ray length X unmirrored ray length

Returns **outcnt** – the number of output columns to expect when calling rtrace instance

Return type int

Raises **ValueError:** – when an output specifier is not recognized

classmethod load_source(*srcfile*, *freesrc=- 1*, *ambfile=None*)
add a source description to the loaded scene

Parameters

- **srcfile** (*str*) – path to radiance scene file containing sources, these should not change the bounding box of the octree and has only been tested with the "source" type.

- **freesrc** (*int, optional*) – the number of objects to unload from the end of the rtrace object list, if -1 unloads all objects loaded by previous calls to load_source
- **ambfile** (*str, optional*) – path to ambient file. if given, and arguments

4.5.4 Rcontrib

class raytraverse.renderer.**Rcontrib**(*rayargs=None*, *scene=None*, *nproc=None*, *skyres=10.0*, *modname='skyglow'*, *ground=True*, *default_args=True*)

Bases: raytraverse.renderer.radiancerenderer.RadianceRenderer

singleton wrapper for c++ raytrraverse.crenderer.cRcontrib class

this class sets default arguments, helps with initialization and setting cpu limits of the cRcontrib instance. see raytrraverse.crenderer.cRcontrib for more details.

Parameters

- **rayargs** (*str, optional*) – argument string (options and flags only) raises ValueError if arguments are not recognized by cRtrace.
- **scene** (*str, optional*) – path to octree
- **nproc** (*int, optional*) – if None, sets nproc to cpu count, or the RAYTRAVERSE_PROC_CAP environment variable
- **skyres** (*float, optional*) – approximate resolution for skypatch subdivision (in degrees). Patches will have (rounded) size skyres x skyres. So if skyres=10, each patch will be 100 sq. degrees (0.03046174197 steradians) and there will be 18 * 18 = 324 sky patches.
- **modname** (*str, optional*) – passed the -m option of cRcontrib initialization
- **ground** (*bool, optional*) – if True include a ground source (included as a final bin)
- **default_args** (*bool, optional*) – if True, prepend default args to rayargs parameter

Examples

Basic Initialization and call:

```
r = renderer.Rcontrib(args, scene)
ans = r(vecs)
# ans.shape -> (vecs.shape[0], 325)
```

name = 'rcontrib'

engine = <cRcontrib>
raytraverse.crenderer.cRcontrib

ground = True

side = 18

srcn = 325

modname = 'skyglow'

classmethod setup(*scene=None*, *ground=True*, *modname='skyglow'*, *skyres=10.0*)
set class attributes for proper argument initialization

Parameters

- **scene** (*str, optional*) – path to octree

- **ground** (*bool, optional*) – if True include a ground source (included as a final bin)
- **modname** (*str, optional*) – passed the -m option of cRcontrib initialization
- **skyres** (*float, optional*) – approximate resolution for skypatch subdivision (in degrees). Patches will have (rounded) size skyres x skyres. So if skyres=10, each patch will be 100 sq. degrees (0.03046174197 steradians) and there will be 18 * 18 = 324 sky patches.

Returns **scene** – path to scene with added sky definition

Return type str

classmethod get_default_args()
construct default arguments

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

Parameters

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

4.5.5 ImageRenderer

class raytraverse.renderer.**ImageRenderer**(*scene*, *viewmapper=None*, *method='linear'*)
Bases: raytraverse.renderer.renderer.Renderer

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 projecting matching ViewMapper
- **viewmapper** (raytraverse.mapper.ViewMapper, *optional*) – if None, assumes 180 degree angular fisheye (vta)
- **method** (*str, optional*) – passed to scipy.interpolate.RegularGridInterpolator

4.6 raytraverse.sky

4.6.1 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

Returns

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

Returns

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*)
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*) – (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*, ***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*) – sky patch subdivision n = jn^2
- **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

4.6.2 SkyData

class raytraverse.sky.**SkyData**(*wea*, *loc=None*, *skyro=0.0*, *ground_fac=0.15*, *skyres=10.0*, *minalt=2.0*, *mindiff=5.0*)

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
- **skyres** (*float, optional*) – approximate square patch size in degrees
- **minalt** (*float, optional*) – minimum solar altitude for daylight masking
- **mindiff** (*float, optional*) – minumum diffuse horizontal irradiance for daylight masking

property skyres

property skyro

sky rotation (in degrees, ccw)

property loc

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

property skydata

sun position and dirnorm diffhoriz

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

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) 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()
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*)

Parameters

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

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

Return type np.array

masked_idx(*i*)

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

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

4.7 raytraverse.sampler

4.7.1 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*)
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

4.7.2 BaseSampler

class raytraverse.sampler.**BaseSampler**(*scene*, *engine*, *accuracy=1.0*, *stype='generic'*, *samplerlevel=0*)
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)

t0 = 0.00390625
initial sampling threshold coefficient this value times the accuracy parameter is passed to raytraverse.sampler.draw.from_pdf() at level 0

t1 = 0.0625
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

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 = None
scene information

Type *raytraverse.scene.Scene*

accuracy = None
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 = None
sampler type

Type str

weights = None
holds weights for self.draw

Type np.array

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

4.7.3 SamplerSuns

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

Bases: raytraverse.sampler.basesampler.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.

t0 = 0.05
initial sampling threshold coefficient

t1 = 0.5
final sampling threshold coefficient

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*)
adapively 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** (raytraverse.lightfield.LightPlaneKD) – 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()

Returns

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, escaping dimensional curses.

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

4.7.4 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')*, *metricfunc=<function amax>*, ***kwargs*)
Bases: raytraverse.sampler.basesampler.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)
- **metricfunc** (*func, optional*) – takes detail array as an argument, shape: (len(metricset),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.

t0 = 0.1
initial sampling threshold coefficient

t1 = 0.9
final sampling threshold coefficient

ub = 100
upper bound for drawing from pdf

metricclass = None
raytraverse.evaluate.BaseMetricSet

metricset = None
iterable

features = None
int:

sampling_scheme(*mapper*)
calculate sampling scheme

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

Parameters

- **mapper** (raytraverse.mapper.PlanMapper) – the pointset to build/run if initialized with points runs a static sampler
- **name** (*str, optional*) –
- **specguide** (raytraverse.lightfield.LightPlaneKD) – sky source lightfield to use as specular guide for sampling (used by engine of type raytraverse.sampler.SunSamplerPt)
- **kwargs** – passed to self.run()

Returns

Return type raytraverse.lightlplane.LightPlaneKD

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

4.7.5 SamplerPt

class raytraverse.sampler.**SamplerPt**(*scene*, *engine*, *idres=5*, *fdres=9*, *accuracy=1.0*, *srcn=1*, *stype='generic'*, *bands=1*, *samplerlevel=0*, ***kwargs*)
Bases: raytraverse.sampler.basesampler.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 log2(res))
- **fdres** (*int, optional*) – final directional resolution given as log2(res)
- **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)
- **bands** (*int, optional*) – number of spectral bands returned by the engine
- **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()

bands = None
number of spectral bands / channels returned by renderer based on given renderopts (user ensures these agree).

Type int

srcn = None
number of sources return per vector by run

Type int

idres = None
initial direction resolution (as log2(res))

Type int

sampling_scheme(*a*)
calculate sampling scheme

run(*point*, *posidx*, *mapper=None*, *lpargs=None*, ***kwargs*)
sample a single point, poisition 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()

Returns

Return type *LightPointKD*

4.7.6 SkySamplerPt

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

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

4.7.7 SunSamplerPt

class raytraverse.sampler.**SunSamplerPt**(*scene*, *engine*, *sun*, *sunbin*, *speclevel=9*, *fdres=10*, *slimit=0.01*, *maxspec=0.2*, *stype='sun'*, ***kwargs*)
Bases: raytraverse.sampler.samplerpt.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
- **speclevel** (*int, optional*) – at this sampling level, pdf is made from brightness of sky sampling rather than progressive variance to look for fine scale specular highlights, this should be atleast 1 level from the end and the resolution of this level should be smaller than the size of the source
- **slimit** (*float, optional*) – the minimum value in the specular guide considered as a potential specular reflection source, in the case of low vlt glazing, this value should be reduced.
- **maxspec** (*float, optional*) – the maximum value inn the specular guide considered as a specular reflection source. above this value it is assumed that these are direct view rays to the source so are not sampled. in the case of low vlt glazing, this value should be reduced. In mixed (high-low) vlt scenes the specular guide will either over sample (including direct views) or under sample (miss specular reflections) depending on this setting.

specidx = None
index of level at which brightness sampling occurs

Type int

sunpos = None
sun position x,y,z

Type np.array

run(*point*, *posidx*, *vm=None*, *plotp=False*, *specguide=None*, ***kwargs*)
sample a single point, poisition 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()

Returns

Return type *LightPointKD*

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

Returns **pdraws** – index array of flattened samples chosen to sample at next level

Return type np.array

4.7.8 SunSamplerPtView

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

Bases: raytraverse.sampler.samplerpt.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, poisition 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()

Returns

Return type *LightPointKD*

4.7.9 ImageSampler

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

Bases: raytraverse.sampler.samplerpt.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

4.7.10 DeterministicImageSampler

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

Bases: raytraverse.sampler.imagesampler.ImageSampler

ub = 1

4.8 raytraverse.lightpoint

4.8.1 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*)

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 = None
raytraverse.mapper.ViewMapper

scene = None
raytraverse.scene.Scene

posidx = None
index for point

Type int

pt = None
point location

Type np.array

src = None
source key

Type str

file = None
relative path to disk storage

Type str

srcdir = None
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

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*, *omega=False*, *vm=None*, *rnd=False*)
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** (*bool, optional*) – for linear interpolation (falls back to nearest outside of convexhull
- **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)

evaluate(*skyvec*, *vm=None*, *idx=None*, *srcvecoverride=None*, *srconly=False*)
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
- **srcvecoverride** (*np.array, optional*) – replace source vector of any source views with this value. For example, by giving the actual sun position, this will improve irradiance calculations (and yield more consistent results when the sampled sun position over an area varies) compared with using the sampled ray direction directly.
- **srconly** (*bool, optional*) – only evaluate direct sources (stored in self.srcviews)

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*, *srcidx=None*, *interp=False*, *omega=False*, *scalefactor=1*, *vm=None*, *fisheye=True*)
create an unweighted summary image of lightpoint

add(*lf2*, *src=None*, *calcomega=True*, *write=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

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

4.8.2 SrcViewPoint

class raytraverse.lightpoint.**SrcViewPoint**(*scene*, *vecs*, *lum*, *pt=0, 0, 0*, *posidx=0*, *src='sunview'*, *res=64*, *blursun=1.0*, *srcomega=6.796702357283834e-05*)

Bases: object

interface for sun view data

static offset(*points*, *target*)

scene = None
raytraverse.scene.Scene

posidx = None
index for point

Type int

pt = None
point location

Type np.array

src = None
source key

Type str

raster = None
individual vectors that hit the source (pixels)

Type np.array

lum = None
source luminance (average)

Type float

radius = None
source radius

Type float

property vm

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

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

direct_view(*res=80*)

4.8.3 CompressedPointKD

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

Bases: raytraverse.lightpoint.lightpointkd.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*, ***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

Returns

Return type arguments for initializing a CompressedPointKD

4.9 raytraverse.lightfield

4.9.1 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.

query_ball(*pts*, *dist=1.0*)
return set of vectors within a distance

Parameters

- **pts** (*np.array*) – shape (N, 3) vectors to query.
- **dist** (*int float*) – radius

Returns **i** – if vecs is a single vector, a list of indices within radius. if vecs is a set of points an array of lists, one for each is returned.

Return type list np.array

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

4.9.2 LightPlaneKD

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

collection of lightpoints with KDtree structure for positional query

property data
LightPointSet

property omega
represeentative area of each point

Getter Returns array of areas

Setter sets areas

Type np.array

evaluate(*skyvec*, *points=None*, *vm=None*, *metricclass=<class 'raytraverse.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 'raytraverse.evaluate.metricset.MetricSet'>*, *metrics=('avglum',)*, *interp=False*)
create a summary image of lightplane showing samples and areas

add(*lf2*, *src=None*, *calcomega=True*, *write=False*, *compress=False*)
add light planes of distinct sources together

4.9.3 DayLightPlaneKD

class raytraverse.lightfield.**DayLightPlaneKD**(*scene*, *vecs*, *pm*, *src*, *includesky=True*)
Bases: raytraverse.lightfield.lightfield.LightField

collection of lightplanes with KDtree structure for sun position query

Parameters

- **scene** (raytraverse.scene.BaseScene) –
- **vecs** (*np.array str*) – suns as array or file shape (N,3), (N,4) or (N,5) if 3, indexed from 0
- **pm** (raytraverse.mapper.PlanMapper) –
- **src** (*str*) – name of sun sources group.

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

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

property data
LightPlaneSet

property kd
kdtree for spatial queries built on demand

property skyplane
LightPlaneKD of sky results

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 leveel 0 sun samples divided by the PlanMapper ptres * sqrt(2).

evaluate(*skydata*, *points*, *vm*, *metricclass=<class 'raytraverse.evaluate.metricset.MetricSet'>*, *metrics=None*, *logerr=True*, *datainfo=False*, *hdr=False*, *res=1024*, *interp=False*, *prefix='img'*, ***kwargs*)

Parameters

- **skydata** (*raytraverse.sky.Skydata*) –
- **points** (*np.array*) – shape (N, 3)
- **vm** (*Union[raytraverse.mapper.ViewMapper, np.array]*) –
- **metricclass** (raytraverse.evaluate.BaseMetricSet, *optional*) –
- **metrics** (*Sized, optional*) –
- **logerr** (*bool, optional*) –
- **datainfo** (*Union[Sized[str], bool], optional*) – include information about source data as additional metrics. Valid values include: [“sun_pt_err”, “sun_pt_bin”, “sky_pt_err”, “sky_pt_bin”, “sun_err”, “sun_bin”]. If True, includes all. “err” is distance from queried vector to actual. “bin” is the unraveled idx of source vector at a 500^2 resolution of the mapper. order is ignored, info is always in order listed above after the last metric.

- **hdr** (*bool, optional*) – whether to write a 180 degree fisheye hdr image for each point/sky combo (be careful with large sets), named by img_row(skydata)_pt_vdir.hdr
- **res** (*int, optional*) – image resolution
- **interp** (*bool, optional*) – interpolate image
- **prefix** (*str, optional*) – prefix for output file naming

Returns

Return type *raytraverse.lightfield.LightResult*

4.9.4 LightResult

class raytraverse.lightfield.**LightResult**(*data*, **axes*)

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 data

property axes

property names

static load(*file*)

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

pull(**axes*, *aindices=None*, *findices=None*, *order=None*)

arrange and extract data slices from result

Parameters

- **axes** (*Union[int, str]*) – the axes (by name or integer index) to maintain and order the returned result (where axes will be the last N axes of result)
- **aindices** (*Sequence[array_like], optional*) – sequence of returned axis indices, up to one per each of axes to return a subset of data along these axes.
- **findices** (*Sequence[array_like], optional*) – sequence of indices for pre-flattened axes to be flattened. give in order matching "order"
- **order** (*Sequence*) – the remainder of the axes in the order in which they should be arranged prior to flattening. by default uses their original order in self.data

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

pull2pandas(*ax1*, *ax2*, ***kwargs*)
returns a list of dicts suitable for initializing pandas.DataFrames

Parameters

- **ax1** (*Union[int, str]*) – the output row axis
- **ax2** (*Union[int, str]*) – the output column axis
- **kwargs** (*dict*) – additional parameters for self.pull()

Returns

- **panda_args** (*Sequence[dict]*) –

  list of keyword arguments for initializing a pandas DataFrame:

  ```
  frames = [pandas.DataFrame(**kw) for kw in panda_args]
  ```

  keys are ['data', 'index', 'columns']

- **frame_info** (*Sequence[dict]*) –

  information for each data frame keys:

  – name: the summary name of the frame, a concatenation of the flattened axes (for example: "point_view" implies the frame is extracted for a particular point and view direction)

  – item: the values of the frame from each of the flatten axes (for example: for a name "point_view" this item = [(x, y, z), (vx, vy, vz)]

  – axis0: the name of the row axis (for example: "sky")

  – axis1: the name of thee column axis (for example: "metric")

4.10 raytraverse.evaluate

4.10.1 BaseMetricSet

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

allmetrics = ['illum', 'avglum', 'gcr', 'density', 'avgraylum']

safe2sum = {'avglum', 'illum'}

defaultmetrics = ['illum', 'avglum', 'gcr']
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 if 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
cos angle between ray and view

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

property density

4.10.2 MultiLumMetricSet

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

Bases: raytraverse.evaluate.basemetricset.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

4.10.3 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*, ***kwargs*)
Bases: raytraverse.evaluate.basemetricset.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,) 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', 'gcr', 'ugp', 'dgp']
available metrics (and the default return set)

allmetrics = ['illum', 'avglum', 'gcr', 'density', 'avgraylum', 'ugp', 'dgp', 'task

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

4.10.4 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: raytraverse.evaluate.basemetricset.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

4.10.5 SamplingMetrics

class raytraverse.evaluate.**SamplingMetrics**(*vec*, *omega*, *lum*, *vm*, *scale=1.0*, *peakthreshold=0.0*, *lmin=0*, ***kwargs*)

Bases: raytraverse.evaluate.basemetricset.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 0-2)

property ypeak
y-component of avgvec as positive number (in range 0-2)

property loggcr
log of global contrast ratio

4.10.6 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** (raytraverse.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.arrray

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

4.10.7 retina

raytraverse.evaluate.retina.**rgcf_density_on_meridian**(*deg*, *mi*)
retinal ganlgion cell field density along a meridian as a functional best fit.

the field density accounts for the input region of the ganglion cell to account for displaced ganglion cells. This value is estimate from cone density and the inferred density of midget ganglion cells. see Watson (2014) for important caveats.

Parameters

- **deg** (*np.array*) – eccentricity in degrees along merdian
- **mi** (*int*) – meridian index. [0, 1, 2, 3] for Temporal, Superior, Nasal, Inferior.

Returns 1d array of retinal ganglion cell density along a merdian

Return type np.array

raytraverse.evaluate.retina.**rgc_density_on_meridian**(*deg*, *mi*)
retinal ganglion cell density along a merdian as a linear interpolation between non-zero measurements

As opposed to the field density this estimate the actual location of ganglion cells, which could be important to consider for intrinsically photosensitive cells. These are (partially?) responsible for pupillary response. However, even iprgc (may?) receive signals from rods/cones

Parameters

- **deg** (*np.array*) – eccentricity in degrees along merdian
- **mi** (*int*) – meridian index. [0, 1, 2, 3] for Temporal, Superior, Nasal, Inferior.

Returns 1d array of retinal ganglion cell density along a merdian

Return type np.array

raytraverse.evaluate.retina.**rgcf_density_xy**(*xy*, *func=<function rgcf_density_on_meridian>*)
interpolate density between meridia, selected by quadrant

Parameters

- **xy** (*np.array*) – xy visual field coordinates on a disk in degrees (eccentricity 0-90 from fovea)

- **func** (*callable*) – density function along a meridian, takes r in degrees and an axes index: [0, 1, 2, 3] for Temporal, Superior, Nasal, Inferior.

Returns 1d array of single eye densities

Return type np.array

raytraverse.evaluate.retina.**binocular_density**(*xy*, *func=<function rgcf_density_on_meridian>*)

average denisty between both eyes.

Parameters

- **xy** (*np.array*) – xy visual field coordinates on a disk (eccentricity 0-1 from fovea)
- **func** (*callable*) – density function along a meridian, takes r in degrees and an axes index: [0, 1, 2, 3] for Temporal, Superior, Nasal, Inferior. coordinates are for the visual field.

Returns 1d array of average binocular densities

Return type np.array

raytraverse.evaluate.retina.**rgcf_density**(*xy*)

retinal ganglion cell field density

Parameters **xy** (*np.array*) – xy visual field coordinates on a disk (eccentricity 0-1 from fovea)

Returns 1d array retinal ganglion cell field density according to model by Watson

Return type np.array

raytraverse.evaluate.retina.**rgc_density**(*xy*)

retinal ganglion cell density (includes displaced ganglion cells)

Parameters **xy** (*np.array*) – xy visual field coordinates on a disk (eccentricity 0-1 from fovea)

Returns 1d array retinal ganglion cell density according to measurements by Curcio

Return type np.array

4.11 raytraverse.craytraverse

4.12 raytraverse.io

functions for reading and writing

raytraverse.io.**get_nproc**(*nproc=None*)

raytraverse.io.**set_nproc**(*nproc*)

raytraverse.io.**unset_nproc**()

raytraverse.io.**np2bytes**(*ar*, *dtype='<f'*)

format ar as bytestring

Parameters

- **ar** (*np.array*) –
- **dtype** (*str*) – argument to pass to np.dtype()

Returns

Return type bytes

raytraverse.io.**np2bytefile**(*ar*, *outf*, *dtype='<f'*, *mode='wb'*)
save vectors to file

Parameters

- **ar** (*np.array*) – array to write
- **outf** (*str*) – file to write to
- **dtype** (*str*) – argument to pass to np.dtype()

raytraverse.io.**bytes2np**(*buf*, *shape*, *dtype='<f'*)
read ar from bytestring

Parameters

- **buf** (*bytes, str*) –
- **shape** (*tuple*) – array shape
- **dtype** (*str*) – argument to pass to np.dtype()

Returns

Return type np.array

raytraverse.io.**bytefile2np**(*f*, *shape*, *dtype='<f'*)
read binary data from f

Parameters

- **f** (*IOBase*) – file object to read array from
- **shape** (*tuple*) – array shape
- **dtype** (*str*) – argument to pass to np.dtype()

Returns necessary for reconstruction

Return type ar.shape

raytraverse.io.**version_header**()
generate image header string

raytraverse.io.**array2hdr**(*ar*, *imgf*, *header=None*)
write 2d np.array (x,y) to hdr image format

Parameters

- **ar** (*np.array*) – image array
- **imgf** (*str*) – file path to right
- **header** (*list*) – list of header lines to append to image header

Returns

Return type imgf

raytraverse.io.**carray2hdr**(*ar*, *imgf*, *header=None*)
write color channel np.array (3, x, y) to hdr image format

Parameters

- **ar** (*np.array*) – image array
- **imgf** (*str*) – file path to right
- **header** (*list*) – list of header lines to append to image header

Returns

Return type imgf

raytraverse.io.**hdr2array**(*imgf*, *stdin=None*)
read np.array from hdr image

Parameters

- **imgf** (*file path of image*) –
- **stdin** – passed to Popen (imgf should be "")

Returns **ar**

Return type np.array

raytraverse.io.**rgb2rad**(*rgb*)

raytraverse.io.**rgb2lum**(*rgb*)

raytraverse.io.**rgbe2lum**(*rgbe*)
convert from Radiance hdr rgbe 4-byte data format to floating point luminance.

Parameters **rgbe** (*np.array*) – r,g,b,e unsigned integers according to: http://radsite.lbl.gov/radiance/refer/filefmts.pdf

Returns **lum**

Return type luminance in cd/m^2

4.13 raytraverse.translate

functions for translating between coordinate spaces and resolutions

raytraverse.translate.**norm**(*v*)
normalize 2D array of vectors along last dimension

raytraverse.translate.**norm1**(*v*)
normalize flat vector

raytraverse.translate.**uv2xy**(*uv*)
translate from unit square (0,1),(0,1) to disk (x,y) http://psgraphics.blogspot.com/2011/01/improved-code-for-concentric -map.html.

raytraverse.translate.**uv2xyz**(*uv*, *axes=0, 1, 2*, *xsign=- 1*)
translate from 2 x unit square (0,2),(0,1) to unit sphere (x,y,z) http://psgraphics.blogspot.com/2011/01/improved-code-for-concentric -map.html.

raytraverse.translate.**xyz2uv**(*xyz*, *normalize=False*, *axes=0, 1, 2*, *flipu=True*)
translate from vector x,y,z (normalized) to u,v (0,2),(0,1) Shirley, Peter, and Kenneth Chiu. A Low Distortion Map Between Disk and Square. Journal of Graphics Tools, vol. 2, no. 3, Jan. 1997, pp. 45-52. Taylor and Francis+NEJM, doi:10.1080/10867651.1997.10487479.

raytraverse.translate.**xyz2skybin**(*xyz*, *side*, *tol=0*, *normalize=False*)

raytraverse.translate.**skybin2xyz**(*bn*, *side*)
generate source vectors from sky bins

Parameters

- **bn** (*np.array*) – bin numbers
- **side** (*int*) – square side of discretization

Returns **xyz** – direction to center of sky patches

Return type np.array

raytraverse.translate.**xyz2xy**(*xyz*, *axes=0, 1, 2*, *flip=True*)
xyz coordinates to xy mapping of angular fisheye proejection

raytraverse.translate.**tpnorm**(*thetaphi*)
normalize angular vector to 0-pi, 0-2pi

raytraverse.translate.**tp2xyz**(*thetaphi*, *normalize=True*)
calculate x,y,z vector from theta (0-pi) and phi (0-2pi) RHS Z-up

raytraverse.translate.**xyz2tp**(*xyz*)
calculate theta (0-pi), phi from x,y,z RHS Z-up

raytraverse.translate.**tp2uv**(*thetaphi*)
calculate UV from theta (0-pi), phi

raytraverse.translate.**uv2tp**(*uv*)
calculate theta (0-pi), phi from UV

raytraverse.translate.**aa2xyz**(*aa*)
calculate altitude (0-90), azimuth (-180,180) from xyz

raytraverse.translate.**xyz2aa**(*xyz*)
calculate xyz from altitude (0-90), azimuth (-180,180)

raytraverse.translate.**chord2theta**(*c*)
compute angle from chord on unit circle

Parameters **c** (*float*) – chord or euclidean distance between normalized direction vectors

Returns **theta** – angle captured by chord

Return type float

raytraverse.translate.**theta2chord**(*theta*)
compute chord length on unit sphere from angle

Parameters **theta** (*float*) – angle

Returns **c** – chord or euclidean distance between normalized direction vectors

Return type float

raytraverse.translate.**uv2ij**(*uv*, *side*, *aspect=2*)

raytraverse.translate.**uv2bin**(*uv*, *side*)

raytraverse.translate.**bin2uv**(*bn*, *side*, *offset=0.5*)

raytraverse.translate.**resample**(*samps*, *ts=None*, *gauss=True*, *radius=None*)
simple array resampling. requires whole number multiple scaling.

Parameters

- **samps** (*np.array*) – array to resample along each axis
- **ts** (*tuple, optional*) – shape of output array, should be multiple of samps.shape
- **gauss** (*bool, optional*) – apply gaussian filter to upsampling
- **radius** (*float, optional*) – when gauss is True, filter radius, default is the scale ratio - 1

Returns to resampled array

Return type np.array

raytraverse.translate.**rmtx_elem**(*theta*, *axis=2*, *degrees=True*)

raytraverse.translate.**rotate_elem**(*v*, *theta*, *axis=2*, *degrees=True*)

raytraverse.translate.**rmtx_yp**(*v*)
generate a pair of rotation matrices to transform from vector v to z, enforcing a z-up in the source space and a y-up in the destination. If v is z, returns pair of identity matrices, if v is -z returns pair of 180 degree rotation matrices.

Parameters **v** (*array-like of size (3,)*) – the vector direction representing the starting coordinate space

Returns ymtx, pmtx – two rotation matrices to be premultiplied in order to reverse transform, swap order and transpose. Forward: (pmtx@(ymtx@xyz.T)).T Backward: (ymtx.T@(pmtx.T@xyz.T)).T

Return type (np.array, np.array)

CHAPTER

FIVE

TUTORIALS

5.1 Directional Sampling Overview

(starting at 4:56:25)

5.1.1 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.

5.2 History

5.2.1 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

5.2.2 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

5.2.3 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.

5.2.4 0.1.0 (2020-05-19)

- First release on PyPI.

5.3 Index

5.4 Search

5.5 Todo

5.6 Git Info

this project is hosted in two places, a private repo (master branch) at:

https://gitlab.enterpriselab.ch/lightfields/raytraverse

and a public repo (release branch) at:

https://github.com/stephanwaz/raytraverse

the repo also depends on two submodules, to initialize run the following:

```
git clone https://github.com/stephanwaz/raytraverse
cd raytraverse
git submodule init
git submodule update --remote
git -C src/Radiance config core.sparseCheckout true
cp src/sparse-checkout .git/modules/src/Radiance/info/
git submodule update --remote --force src/Radiance
```

after a "git pull" make sure you also run:

```
git submodule update
```

to track with the latest commit used by raytraverse.

CHAPTER

SIX

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

Additionally, please cite this conference paper for a description of the directional sampling and integration method:

> 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.

CHAPTER

SEVEN

LICENCE

CHAPTER

EIGHT

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

NINE

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

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y