G-Divergence Simulation: Technical Brief

Roland Schregle

Revised 24/01/22

Introduction and Scope

This document summarises technical details of the Python script suite developed for the G-divergence project to analyse and correct divergence of measured solar heat gain coefficients (SHGC, a.k.a. g-value) from fenestrations. The scripts are intended to supplement the measurements conducted at ISE's solar simulator laboratory. They consist of the following modules as shown in Figure 1:

- ***pf2pic.py***: converts luminance map (from HDR camera) to measured radiance map.
- ***gdivmeas.py***: serialises measured radiance map pixels & extracts contrib. coefficients.
- ***gdivsim.py***: generates simulated radiance map & extracts contrib. coefficients
- ***gdivclust.py***: clusters contribution coefficients from the above by symmetry criteria.
- ***gdivsolve.py***: solves for sought parallel SHGCs.
- ***gdivmeas-check.py***: verifies image-based irradiance from *gdivmeas.py* against rtrace.

pf2pic.py

Synopsis: *pftopic.py* ***pfFile*$_1$** *..* ***pfFile*$_n$**

Batch converts HDR luminance camera images *pfFile*$_i$ in PF format to RADIANCE HDR format using the *pftopic* utility. The latter accepts a luminous efficacy parameter, thereby converting from luminance to radiance. This term is defined as a constant *LUMEFFIC* in the script, and obtained as a one-time calibration from pyranometer and luxmeter readings in the solar simulator, since this parameter does not change with varying lamp configurations or sensor positions. The script embeds a fisheye view definition *VIEW* in the RADIANCE HDR file's header (again, via *pftopic*) to coincide with that of the simulated solar simulator model.

As a convenience, the script autodetects the horizontal and vertical extents of the circular fisheye image by extracting the maximum/minimum coordinates of runs of zero pixels using *pvalue* and *total.* These extents are then used to automatically crop the RADIANCE HDR image via *pcompos.*

Finally, the script also converts the RADIANCE HDR image (henceforth referred to as radiance map) to falsecolour TIFF via the *falsecolor* and *ra_tiff* tools.

gdivmeas.py

Synopsis: *gdivmeas_fulllres.py* ***hdrFile*$_1$** *..* ***hdrFile*$_n$**

This is one of the core scripts in the suite, as it forms the basis for the divergence evaluation from measurements. Its main function is to output normalised contribution coefficients for each pixel in the radiance maps obtained from the HDR camera.

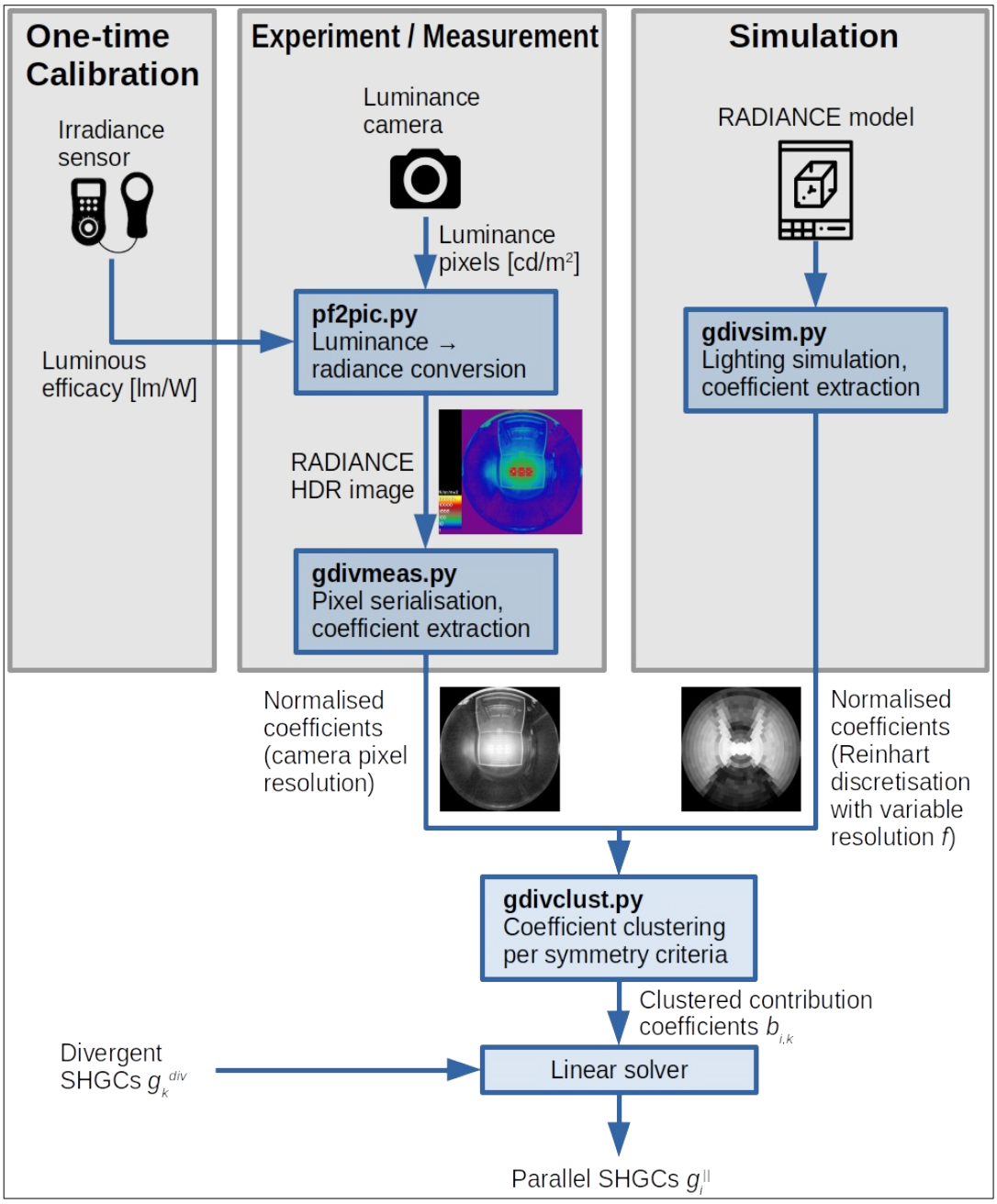

Figure 1: Gdiv script suite overview

The input consists of a set of radiance maps $hdrFile_i$ in RADIANCE HDR format generated by *pf2pic.py*, which are consecutively batch processed for convenience. The coefficients for each radiance map $hdrFile_i$ are output to a separate file $coeff/hdrFile_i.dat$.

gdivmeas.py expects a *view* string in each HDR header defining the fisheye projection used in acquiring the radiance map. It extracts the radiance map radius *r* from the image dimensions reported by *getinfo*; this is assumed to be the effective radius (i.e. no spurious dark peripheral pixels) as the image was cropped by *pf2pic.py*.

The pixels and their coordinates are then extracted from the radiance map via *pvalue.* Each pixel's 2D coordinates (*x, y*) are mapped to a 1D serial index j_p in raster fashion, i.e. starting at the top of the image and extending left to right towards the bottom. The index corresponds to the area *A* of the

circular segment[1] formed by the pixel's vertical offset *v,* plus the chord length *a* formed by the pixel's horizontal offset *u* (see Figure 2):

$$j_p(u,v,r) = \begin{cases} \lfloor A(v+1,r) + 0.5 \rfloor + \left\lfloor \frac{a(v+1,r)}{2} + (u+1) + 0.5 \right\rfloor \\ -1 \quad \text{if } u^2+v^2 > r \end{cases}$$

$$A(v,r) = \frac{r^2(\alpha(v,r) - \sin\alpha(v,r))}{2}$$

$$\alpha(v,r) = \begin{cases} 2\cos^{-1}\left(\frac{v}{r}\right) \\ 0 \quad \text{if } v > r \end{cases}; \quad a(v,r) = \begin{cases} 2\sqrt{r^2-v^2} \\ 0 \quad \text{if } v > r \end{cases}$$

Equation 1: Camera pixel index calculation

where:

- $u = x - r$ and $v = y - r$ are the pixel's horizontal and vertical offsets from the circle's origin (corresponding to the fisheye view's centre of projection); note that these offsets can be negative,
- r is the fisheye projection's radius (in pixels),
- $A(v, r)$ is the area (in pixels) of the circular segment at vertical offset v,
- $a(v, r)$ is the length (again, in pixels) of the chord corresponding to the pixel's raster line at vertical offset v,
- and $\alpha(v, r)$ is the central angle (in radians) subtended by the chord with length $a(v, r)$; note this covers the interval [0, 2π], and thus the entire fisheye image, with 0 corresponding to the fisheye view's up vector.

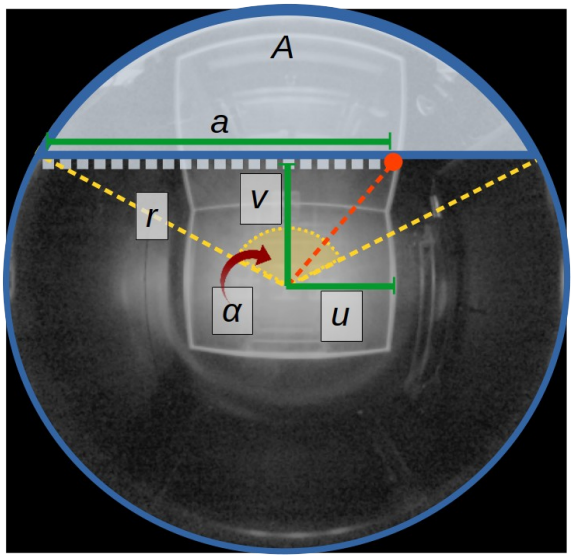

Figure 2: gdivmeas pixel indexing scheme

1 http://mathworld.wolfram.com/CircularSegment.html

Note that pixel coordinates outside the radius are mapped to an invalid index -1 and ignored.

Each pixel with a valid index (≥0) is mapped to its original view vector $\boldsymbol{r_p}$ by reversing the equiangular fisheye projection defined in the *VIEW* string embedded in the radiance map's header by *pf2pic.py*:

$$\boldsymbol{r_p}(\hat{u}, \hat{v}) = \begin{bmatrix} \hat{u} \dfrac{\sin \theta_p}{\|\hat{u}, \hat{v}\|} \\ \hat{v} \dfrac{\sin \theta_p}{\|\hat{u}, \hat{v}\|} \\ \cos \theta_p \end{bmatrix}$$

$$\cos \theta_p = \cos\left(\|\hat{u}, \hat{v}\| \frac{\pi}{2}\right); \quad \sin \theta_p = \frac{\sqrt{1 - \cos^2 \theta_z}}{\|\hat{u}, \hat{v}\|}; \quad \|\hat{u}, \hat{v}\| = \sqrt{\hat{u}^2 + \hat{v}^2}$$

Equation 2: Inverse equiangular fisheye projection

where

- $\hat{u} = u/r$ and $\hat{v} = v/r$ are the pixel's normalised coordinates in the range [-1, 1] relative to the centre of projection
- θ_p is the angle between $\boldsymbol{r_p}$ and the z-axis (corresponding to the fisheye lens' optical axis).

Note this inverse mapping assumes a *left-handed* coordinate system (with the z-axis pointing *into* the radiance map), and that the view vectors pass through each pixel's centre.

Each pixel's contribution is proportional to its discrete solid angle $\Delta\omega_p$, which accounts for the spherical excess[2] of the pixel's projected area onto the unit sphere's surface due to curvature (see Figure 3):

$$\Delta\, \omega_p(x, y, r) = \sum_{i=0}^{n-1} \beta_i(x, y, r) - (n-2)\,\pi = \sum_{i=0}^{n-1} \cos^{-1}\left(\hat{\boldsymbol{N}}_i \cdot \hat{\boldsymbol{N}}_{i+1 \bmod n}\right) - (n-2)\,\pi$$

$$\boldsymbol{N_i} = \boldsymbol{e_i} \times \tilde{\boldsymbol{e}}_{i+1 \bmod n}$$

$$\boldsymbol{e_i} = \boldsymbol{r_p}(\hat{u}_i, \hat{v}_i)$$

$$\tilde{\boldsymbol{e}}_i = \boldsymbol{e}_{i+1 \bmod n} - \boldsymbol{e_i}$$

$$\hat{u}_i = \frac{x_i}{2r}; \quad \hat{v}_i = \frac{y_i}{2r}$$

$$x_i = \left\{x - \frac{1}{2}, x + \frac{1}{2}, x + \frac{1}{2}, x - \frac{1}{2}\right\}; \quad y_i = \left\{y - \frac{1}{2}, y - \frac{1}{2}, y + \frac{1}{2}, y + \frac{1}{2}\right\}$$

Equation 3: Calculation of spherical excess and pixel solid angle

where:

- $n = 4$ is the number of projected vertices
- $\boldsymbol{e}_i$ are vertex vectors defined by reversing the fisheye projection for each pixel corner, under the assumption that the coordinates (*x, y*) refer to the pixel's centre; collectively, the vertices subtend a frustum bounding the pixel's spherical projection, and converging at the origin

2 https://en.wikipedia.org/wiki/Spherical_trigonometry#Area_and_spherical_excess

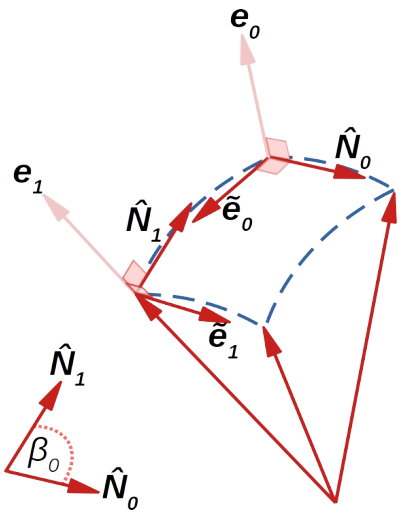

Figure 3: Calculation of spherical excess and pixel solid angle.

- $\tilde{\boldsymbol{e}}_i$ is a vector that lies in the *i*-th frustum plane together with $\boldsymbol{e}_i$, which is found by subtracting $\boldsymbol{e}_i$ from its neighbour $\boldsymbol{e}_{i+1 \bmod n}$
- $\boldsymbol{N}_i$ is the vector perpendicular to the frustum plane defined by $\boldsymbol{e}_i$ and $\tilde{\boldsymbol{e}}_i$ with $\hat{\boldsymbol{N}}_i$ denoting vector normalisation (since a vector cross product is generally unnormalised)
- β_i is the angle between neighbouring $\boldsymbol{N}_i$, which is > π/2 due to curvature.

Finally, each pixel's contribution coefficient is obtained from its discrete radiance ΔL_p weighted by its projected solid angle $\Delta\omega_p \cos\theta_p$ (corresponding to its discrete irradiance ΔE_p), and then normalised:

$$\Delta \hat{E}_{p,j} = \frac{\Delta E_{p,j}}{\sum_j \Delta E_{p,j}}$$

$$\Delta E_{p,j} = \Delta L_{p,j} \Delta \omega_{p,j} \cos\theta_p$$

Equation 4: Normalised pixel irradiance

where $\Delta\hat{E}_{p,j}$ is the *j*-th pixel's normalised irradiance.

gdivsim.py

Synopsis: *gdivsim-reinhart.py* [no arguments]

This is a counterpart script to *gdivmeas.py* that generates simluated contribution coefficients from a model of the solar simulator, using RADIANCE as the underlying lighting simulation engine.

The script takes no input, and instead relies on built-in definitions to batch process simulations for various lamp and sensor configurations of the solar simulator. These definitions and their defaults are as follows and can be found in lines 30–40 of the script:

- LAMPCONF = *['3h']* : lamp configuration string (e.g. triplet, quartet)

- THETAIN = *[0, 30, 45, 60]* : incidence angles (lamp elevation, in degrees)
- VIEWS = *['SP1.vf', 'SP3.vf', 'SP5.vf', 'SP7.vf', 'SP10.vf', 'SP11.vf'] :* RADIANCE view file for each sensor position on the receiving plane of the solar simulator.

gdivsim.py iterates over each lamp configuration, incidence (lamp) angle and sensor position, and expects to find a corresponding RADIANCE scene description *setup-[conf][theta].rad,* where *conf* ∈ LAMPCONF and *theta* ∈ THETAIN. A RADIANCE octree is then generated for each such file with *oconv.*

The script uses a Reinhart discretisation of adjustable resolution to "bin" the contributions from the solar simulator at each sensor position. To register these, a dummy "bubble" is placed around each viewpoint to only register contributions from rays passing through this modifier, ignoring those contributions incidence on other surfaces in the solar simulator model). Its position is instantiated for each view/sensor point and fed into *oconv* alongside the RADIANCE scene description of the solar simulator model. Since *rcontrib* would otherwise also register contributions incident from the back of the sensor plane, the sensor bubble is modified by a *brightfunc* that crops all rays incident outside a 180° field of view. This sensor bubble is defined as follows:

```
void brightfunc cropfunc
2 cropS crop180.cal
0
0

cropfunc trans sensormat
0
0
7 1 1 1 0 0 1 1

sensormat sphere sensor
0
0
4 vPntx vPnty vPntz 1e-4
```

where $vPnt_x$, $vPnt_y$, $vPnt_z$ is the current view/sensor point as obtained from VIEWS, and *crop180.cal* contains preset cropping orientations (*cropS* being used in the above case) and is defined as follows:

```
crop(Nx, Ny, Nz) = if (-Dx*Nx - Dy*Ny - Dz*Nz, 1, 0);
cropN = crop(0, -1, 0);
cropE = crop(-1, 0, 0);
cropS = crop(0, 1, 0);
cropW = crop(1, 0, 0);
```

Once an octree *octFile* of the solar simulator is generated for the current sensor position (including sensor bubble), the following mother-of-all RADIANCE commands is run (with binning parameters highlighted in green):

*rsensor -h -rd <**NUMSAMP**> <**vPnt**> <**vDir**> <**vUp**> <**SENSFILE**> .* |

*rcontrib -faa -h -n <**NPROC**> -ds 0.02 -dc 1 -dt 0 -dj 0.5 -st 0 -ss 64 -ab 5 -aa 0.01 -ar 1024 -ad 4096 -as 1024 -lw 1e-4 -c <**NUMSAMP**> -V+ -f reinhartb-gdiv.cal -e 'MF:<**MF**>; rNx=<**vDir**$_x$>; rNy=<**vDir**$_y$>; rNz=<**vDir**$_z$>; Ux=<**vUp**$_x$>;Uy=<**vUp**$_y$>; Uz=<**vUp**$_z$>' -bn Nrbins -b rbin -m <**MOD**> <**octFile**>* |

rcollate -h -oc 1 | cut -d\ -f1

where:

- ***NUMSAMP*** is a predefined number of sample rays (default 100k)
- ***vPnt***, ***vDir*** and ***vUp*** are the view point, direction and up vector read from the current sensor position (see ***VIEWS*** above). Note that ***vDir*** coincides with the sensor plane normal.
- ***SENSFILE*** defines a sensor with a uniform hemispherical response (default 'sensor.dat')
- ***NPROC*** is the number of parallel *rcontrib* processes (default 4)
- ***MF*** is the Reinhart discretisation distribution for contributions incident in the hemisphere centered at the sensor position (default 4)
- ***MOD*** is the modifier of the sensor bubble whose contributions as sought (default 'sensormat', see sensor bubble definition above).

In this command, *rsensor* (red text) sends uniformly distributed rays from the sensor position into the model, while *rcontrib* (blue text) accumulates the radiance from each such ray into a discrete "bin" corresponding to its incidence direction as defined by the Reinhart subdivision of the hemisphere (green text). These bins are then output by *rcontrib* and passed via a pipe to *rcollate* (yellow text), where they are reshaped into a column vector so they can be read by the script through the pipe as a *numpy* vector for postprocess the radiance contributions.

Similar to *gdivmeas.py,* the binned radiance contributions $L_{Rh,j}$ are weighted by the solid angle $\Delta\omega_{Rh,j}$ and cosine $cos\ \theta_{Rh,j}$ of each Reinhart region j as returned by a companion script, *reinhartSolidAngles.py* (note this takes the cosine at the midpoint of each region), before being subsequently normalised:

$$\Delta \hat{E}_{Rh,j} = \frac{\Delta E_{Rh,j}}{\sum_j \Delta E_{Rh,j}}$$

$$\Delta E_{Rh,j} = \frac{\Delta L_{Rh,j}}{\Delta \hat{\omega}_{Rh,j}} \Delta \omega_{Rh,j} \cos \theta_{Rh,j}$$

$$\Delta \hat{\omega}_{Rh,j} = \frac{\Delta \omega_{Rh,j}}{\sum_k \Delta \omega_{Rh,k}}$$

Equation 5: Simulated contribution coefficient calculation

Note that the RADIANCE $L_{Rh,j}$ must be biased by the normalised solid angle $\Delta\omega_{Rh,j}$ subtended by its corresponding Reinhart region. This compensates for the fact that *rcontrib* averages binned contributions if >1 sample rays is used, in which case the contributions would not be proportional to their solid angle.

The coefficients are dumped to separate files *coeff/setup-[conf][theta][view].dat* where *conf* ∈ LAMPCONF, *theta* ∈ THETAIN and *view* ∈ VIEWS are the current lamp configuration, incidence angle and view / sensor position. Similarly, the radiance contributions $L_{Rh,j}$ are read saved to *radcontrib/setup-[conf][theta][view].dat*, and the integrated irradiance (sum of all $\Delta E_{Rh,j}$) is dumped as a scalar to *irrad/setup-[conf][theta][view].dat* for verification if necessary.

gdivsim.py can optionally also render an image of the solar simulator model as seen from each sensor position, in both HDR and falsecolour. This is triggered if RENDER is enabled and uses a combination of *vwrays* and *rcontrib*:

*vwrays -ff -c <**NUMSAMP**> -vf <**viewFile**> -pj 0.5 -x <**RES**> -y <**RES**>* |

*rcontrib -fac `vwrays -ff -c <**NUMSAMP**> -vf <**viewFile**> -pj 0.5 -x <**RES**> -y <**RES**> -d` -n <**NPROC**> -ds 0.02 -dc 1 -dt 0 -dj 0.5 -st 0 -ss 64 -ab 5 -aa 0.01 -ar 1024 -ad 4096 -as 1024 -lw 1e-4 -c <**NUMSAMP**> -V+ -fo -o <**hdrFile**> -m <**MOD**> <**octFile**>*

where:

- ***viewFile*** is the view definition for from the current sensor position in VIEWS
- ***RES*** is the rendering resolution in each axis, assuming a 1:1 aspect ratio suitable for a fisheye view (default 400x400 pixels)
- ***hdrFile*** is the output file for the rendering in RADIANCE HDR format. This filename has the format *hdr/setup-[conf][theta][view].rad,* where *conf* ∈ LAMPCONF and *theta* ∈ THETAIN and *view* ∈ VIEWS, as for the numeric output.

gdivclust.py

Synopsis: *gdivclust.py* <***coeffFile***$_0$> [<***coeffFile***$_1$> *...* <***coeffFile***$_{n-1}$>] <***outFile***>

This script accepts sets of files each *coeffFile*$_j$ containing a column vector of contribution coefficients for each incidence (lamp) angle $k \in [0, n\text{-}1]$, as output by *gdivmeas.py* (at full HDR camera resolution) or *gdivsim.py* (discretised via Reinhart subdivision); as such, the script is agnostic to the source of the contribution coefficients, be it measurement or simulation.

Each set of coefficients $b_{j,k}$ for a given incidence angle k is grouped (clustered) and accumulated in symmetric regions $b_{i,k}$ with a mapping

$$f_c : \{b_{j,k}\} \rightarrow \{b_{i,k}\},\ b_{i,k} = \sum_{\{j\,:\,j_k < j \le j_{k+1} \vee j = j_k = 0\}} b_{j,k}$$
$$j \in [0,\ n-1],\ i \in [0,\ m-1],\ m \le n$$

Equation 6: Coefficient clustering

where j_k are the coefficient indices the boundaries of each cluster *as* defined by the clustering function, and the mapping domain [0, *m*-1] is typically smaller than the range [0, *n*-1]. The output of *gdivclust.py* is therefore a matrix containing rows of cluster vectors $\boldsymbol{b_k}$ for each incidence angle k, which is output to *outFile* as a *numpy* array.

The clustering function f_c is defined as an instance of a Python class *SymmetryBase* in a separate module *gdivsymm.py*. The base class implements a function *SymmetryBase.clusterCoeffs*() which accumulates the contribution coefficients into the clusters as described above. Furthermore, an overloadable function *SymmetryBase.clusters*() defines the specific cluster regions in terms of the input coefficient indices *j*. As in Equation 6, the clusters are returned as a list of cluster vectors [[0 .. j_0], [j_0+1 .. j_1] .. [j_{m-2}+1 .. j_{m-1}]] where each vector $\boldsymbol{b_k}$ contains the coefficient indices *j* that map to cluster *k.* Each cluster is assumed to exhibit symmetric behaviour in its SHGC divergence. Predefined clustering functions for specific symmetry types are implemented as derived classes which overload the *SymmetryBase.clusters*(). Currently, these include:

- ***KlemsSymmetry***: Maps the 145 Klems regions to four rotationally symmetric regions grouped by elevation angle θ of [0°, 30°, 45°, 60°] (w.r.t. each region's midpoint).
- ***ReinhartSymmetry***: 1:1 mapping to a Reinhart subdivision of resolution *MF* = √((n-1) / 144), i.e. the number of Reinhard regions is approximately equal to the number of input coefficients. This is a basic definition for derived classes.

- ***ReinhartCurveSymmetry***: Derived from *ReinhardSymmetry,* this class maps to g-divergence curves with symmetric regions at incidence angles of [0°, 3°, 6°, 9°, 10°, 20°, 30°, 40°, 44°, 47°, 50°, 53°, 56°, 60°, 70°].
- ***ReinhartRotSymmetry***: Derived from *ReinhardSymmetry,* this class maps to bands of rotational symmetry grouped by incidence angles obtained from the base class at the resolution *MF* adapted to the number of coefficients.
- ***ReinhartProfSymmetry***: Again derived from *ReinhartSymmetry*, this class maps coefficients to 2305 regions of a Reinhart MF:4 subdivision, and accumulates by profile symmetry into 13 regions.
- ***FullResSymmetry***: Clusters coefficients acquired from HDR camera pixels via *gdivmeas.py* into regions of rotational symmetry with incidence angles θ_p [0°, 30°, 45°, 60°] (w.r.t. each region's midpoint). The incidence angle θ_p is obtained by mapping each coefficient index *j* to its original pixel coordinates (*u, v*) by reversing the pixel index calculation in Equation 1, and then reversing the equiangular fisheye projection described in Equation 2. Consequently, this class imports *gdivmeas.py* to access the relevant functions.

gdivsolve.py

Synopsis: *gdivsolve.py* <***coeffFile***> <***gvalFile***>

This core script solves for the vector of unknown parallel SHGCs $\boldsymbol{g}^{\parallel}$ based on the clustered contribution coefficients $b_{i,k}$ read from *coeffFile* (as output by *gdivclust.py*), and a vector of divergent (measured/simulated) SHGCs $\boldsymbol{g}^{\boldsymbol{div}}$ read from *gvalFile.* The latter are assumed to be a linear decomposition of the former weighted by the clustered coefficients $b_{i,k}$:

$$g_k^{\text{div}} = \frac{\sum_i g^{\parallel} \Delta L_{i,k} \cos(\theta_i) \omega_i}{\sum_i \Delta L_{i,k} \cos(\theta_i) \omega_i} = ¿ \sum_i g^{\parallel} \Delta \hat{E}_{i,k} = \sum_i g^{\parallel} b_{i,k}$$

$$\Leftrightarrow \boldsymbol{g}_{\mathbf{div}} = \boldsymbol{B}\ \boldsymbol{g}_{\parallel}$$

Equation 7: Decomposition of parallel SHGCs

where $\boldsymbol{B}$ is the matrix of clustered coefficients $b_{i,k}$. Solving this equation involves straighforward use of an optimisation routine from the *scipy* library:

scipy.optimize.minimize($f_{min}(\boldsymbol{x})$, $\boldsymbol{g}_{\parallel} = \boldsymbol{g}_{\boldsymbol{div}}$, *method* = 'L-BFGS-B', *bounds* = $[(0, 1)^m]$).

This invokes an optimisation using the limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm, with the vector of sought parallel SHGCs $\boldsymbol{g}_{\parallel}$ initialised to the divergent ones $\boldsymbol{g}_{\boldsymbol{div}}$ (a reasonable assumption for a starting point), and constained to lie in the interval [0, 1] in each incident direction $k \in [0, m\text{-}1]$. The optimisation function f_o to be minimised is defined as:

$$f_{min}(\boldsymbol{x}) = \boldsymbol{B}\ \boldsymbol{x} - \boldsymbol{g}_{\mathbf{div}}$$

Equation 8: Objective function for solving parallel SHGCs

The advantage of using *scipy.optimize.minimize* over the strictly linear solver *linalg.solve* is improved robustness due to the ability to constrain the parameter space.

gdivmeas-check.py

Synopsis: *gdivmeas-check.py* [no arguments]

This script is supplementary to the suite and performs a stand-alone verification of the pixel discretisation error incurred by gdivmeas.py when extracing SHGCs from full-resolution HDR camera images. To this end, the script compares the simulated irradiance integrated by *rtrace* from the solar simulator model for a fixed sensor position and varying lamp angles with that extracted from radiance maps rendered with *rpict*. The latter irradiance is obtained from *gdivmeas.py* by treating the rendering as if it had been acquired from the HDR camera. The rationale for doing so is that any errors in the underlying light transport simulation used by both *rtrace* and *rpict* are expected to cancel, leaving the discretisation error incurred by *gdivmeas.py* when calculating the pixel solid angles to integrate the irradiance.

By default, *gdivmeas-check.py* uses the RADIANCE photon mapping extension to accelerate the light transport simulation[3]. Furthermore, it launches parallel instances of *rtrace* and *rpict* for multiple lamp angles as background processes using the Python *subprocess* library. This is governed by the NPROCS preset, which defaults to 4.

These following options are preset in the script:

- DO_MKPMAP: Generate a RADIANCE photon map (will *rtrace/rpict* will use previously built if disabled).
- DO_BZIP: Compress photon map files between iterations to save disk space
- DO_RTRACE: Get “ground thruth” (reference) irradiance from photon map with *rtrace*
- DO_RPICT: Render radiance map with *rpict*
- DO_GDIVMEAS: Get irradiance from rendered radiance map via *gdivmeas.py*
- DO_ERROR: Calculate relative error between irradiance integrated by rtrace vs. irradiance integrated by *gdivmeas.py* from rendered radiance map.

Each irradiance calculation is iterated for a predefined list of lamp angles; this is defined (in degrees) per default as:

ANGLES=[0, 3, 6, 9, 12, 15, 18, 22, 25, 28, 31, 34, 37, 41, 44, 47, 50, 53, 56, 60].

A new photon map is generated for each angle with the predefined command:

*mkpmap -apg pm/setup-<**conf**><**theta**>.gpm 10m -apc pm/setup-<**conf**><**theta**>.cpm 100m -fo -t 2 -n 8 oct/setup-<**conf**><**theta**>.oct*

where ***theta*** is the current lamp angle $\in$ ANGLES, ***conf*** is the lamp configuration, and *setup-<**conf**><**theta**>.oct* is the octree containing the solar simulator model under these conditions.

Once a photon map is generated, the scalar reference irradiance E_{ref} is integrated with *rtrace* over the hemisphere of incidence angles at a fixed sensor position by gathering the photon flux via one (implicit) ambient bounce with the following command:

*rtrace -I -h -ap pm/setup-<**conf**><**theta**>.gpm 200 -ap pm/setup-<**conf**><**theta**>.cpm 200 -ad 1024 -ar 256 -as 512 -aa 0.1 oct/setup-<**conf**><**theta**>.oct < recv.pos* | \

*rcalc -e ‘$1 = $1*0.265 + $2*0.670 + $3*0.065' > irrad/rtrace-setup-<**conf**><**theta**>.dat*

3 Renderings of comparable (high) quality took *day*s with RADIANCE Classic as opposed to hours with photon map.

where the file *recv.pos* currently contains the sensor position and orientation at the centre of the sample plane. Note the use of *rcalc* to convert the RGB irradiance to monochrome to simplify the comparison.

For comparison with the integrated irradiance from *rtrace,* the script generates a rendering of the radiance map as it would be captured by the HDR camera using the following *rpict* command:

> *rpict -I -h -vf view/recv.vf -x 780 -y 780 -ad 1024 -ar 256 -as 512 -aa 0.1*
> *-ap pm/setup-<**conf**><**theta**>.gpm 200 -ap pm/setup-<**conf**><**theta**>.cpm 200*
> *-o hdr/gdivmeas-setup-<**conf**><**theta**>.hdr oct/setup-<**conf**><**theta**>.oct*

Where *recv.vf* defines an equiangular fisheye view at the centre of the sample plane as seen by the HDR camera. Note that the resolution of 780x780 pixels corresponds to that of the camera sensor.

Each rendered radiance map *gdivmeas-setup-<**conf**><**theta**>.hdr* is processed by gdivmeas.py as if it were an HDR camera capture to obtain the contribution coefficients (which are ignored) and the integral irradiance $E_\Delta = \sum_j \Delta E_{j,p}$ (c.f. Equation 4), which is output to file *irrad/gdivmeas-setup-<**conf**><**theta**>.dat.*

Finally, the script calculates the relative error $E_{\Delta,k} / E_{ref} - 1$ for every lamp angle *k,* and dumps it to a file *gdivmeas-check.dat* suitable for plotting along with the respective irradiance values. A sample (scatter)plot can be seen in Figure 4 for the default range of angles ANGLES. To emphasise the irradiance falloff as a function of lamp elevation, and to mitigate outliers due to variance, a 3rd degree polynomial curve was fitted to each data set.

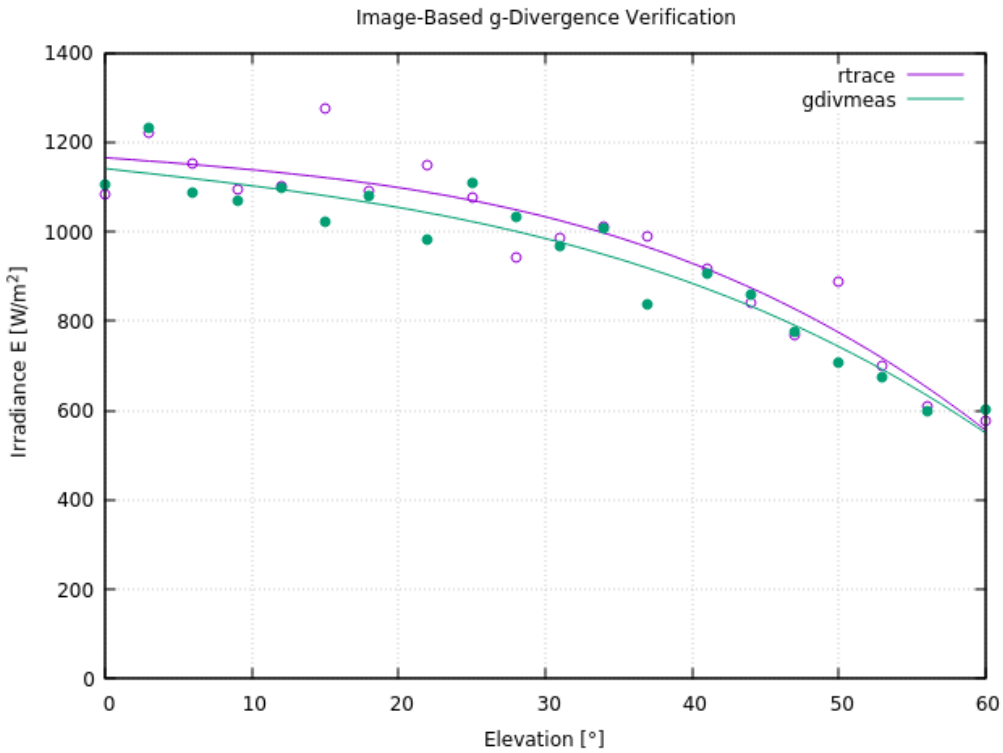

Figure 4: Sample plot generated from gdivmeas-check.py comparing integral irradiance from rtrace with that extracted from gdivmeas.py using a rendered radiance map.