Modelling the Spectro-Angular Reflectance of Grass for Accurate Bifacial Photovoltaic Yield Calculation

27th June 2023

Leonie Mariëlle Horst
s2852497

Master Graduation Assignment
for the master's programme
Sustainable Energy Technology
University of Twente
06-02-2023 to 04-07-2023

Graduation Committee
Chair: Prof. dr. Rebecca Saive
Internal member: Prof. dr. Monica Morales Masis
External member: Dr. ir. Eli Shirazi

UNIVERSITY OF TWENTE.

Abstract

Bifacial solar panels are able to capture irradiance from both faces of the module and therefore the reflectance properties of the surface on which they stand influence the yield. Being able to perform an accurate yield calculation has many benefits: the system's configuration can be optimized, the investment risk decreases, the operation of the grid is supported and the understanding of the bifacial solar panel technology is improved. This thesis aims to contribute to a more accurate yield calculation by analysing the impact of grass reflection on the yield. Grass is one of the most commonly present materials surrounding a bifacial solar panel and exhibits retroreflective behaviour: incoming light is reflected back to the source. To model the spectro-angular reflectance, a Monte Carlo ray tracing software is developed which is used in combination with a reverse ray tracing software to study the effect of different grass structures on the energy yield. Specifically, the developed method is applied to a case study, the vertical bifacial solar park Aasen-Donaueschingen. It was found that accounting for retroreflective behaviour of grass results in a lower energy yield compared to assuming grass to reflect diffusely. Furthermore, the simulations showed that the morphology of the grass influences the energy yield. The presented work can be used for optimisation of an agrivoltaic system and more accurate calculation of the energy yield.

Contents

Nomenclature

Parameters

β	Azimuth and elevation angle between two pixel centres on the detection dome
γ	Rotation angle of the blade
λ	Wavelength
ϕ	Azimuth angle
ρ	Albedo
τ	Tilt of the top part of the blade as measured from the xy-normal (zenith angle)
θ	Elevation angle
ξ	Random number between 0 and 1 as generated by Matlabs *rand* function
A	Area
a	Ratio between radius specified by the subscript to the radius of the sample.
C	Matrix of pixels of the detection dome
d	Distance from the plane to the origin
E	Irradiance
f	Bidirectional reflectance distribution function (BRDF)
h	Height at which the rays are initially aimed
I	Current
j	Current density
L	Radiance
m	Magnitude of the ray
n	Number of rays
P	Power
q	Ratio between radius of the detection dome and the height at which the rays are shot.
R	Reflectance of a surface
r	Radius
t	Time
V	Voltage
w	Width
x	X-component
y	Y-component
z	Z-component

Subscripts

$_{blackbody}$ Referring to a blackbody

$_{blade}$ Grass blade

$_{b}$ Centre of the grass blade

$_{C}$ Centre of a detection dome pixel

$_{detected}$ Point at which the ray is detected on the detection dome

$_{dome}$ Detection dome

$_{final}$ Referring to the properties of a ray when it has been detected

$_{h}$ Height of a grass blade

$_{illuminated}$ Part of the sample on which the rays are initially incident

$_{in}$ Referring to incoming rays

$_{Lambertian}$ Referring to a Lambertian surface

$_{l}$ Point on a plane

$_{MPP}$ Maximum power point

$_{new}$ Referring to the new properties after the ray has intersected with a specific surface

$_{OC}$ Open circuit

$_{old}$ Referring to the old properties before the ray has intersected with a specific surface

$_{out}$ Referring to outgoing rays

$_{sample}$ Circular sample on the xy-plane

$_{SC}$ Short circuit

$_{source}$ Light source

$_{s}$ End point of a ray

Vectors

$\vec{c}$ Centre of the detection dome

$\vec{d}$ Direction of a ray

$\vec{e}$ Starting point of a ray.

$\vec{n}$ Surface normal

$\vec{p}$ Position of a ray in space

$\vec{s}$ End point of a ray

$\vec{v}$ Point is space

1 Introduction

Increasing the share of renewably generated energy in the global energy mix is an important step towards a greener future and can be realised by making renewable energy generators as efficient and cost-effective as possible. In solar cells, the energy from the sun is harvested by converting energy from photons into electricity.

There is a continuous effort from the scientific community to optimise solar cells. Bifacial solar cells (BSC) capture irradiance from both faces, the front and the rear. This increases the number of charge carriers generated compared to a monofacial solar cell. When in an optimal configuration, BSC therefore generate a higher yield than monofacial solar panels, while keeping costs relatively low [1]. A parameter which measures this, is the levelized cost of energy (LCOE). The LCOE is the ratio of the total costs incurred to the total energy generated over the lifetime [2]. Due to the property of absorbing sunlight from both sides, a bifacial solar panel can have about 2–6% lower LCOE than monofacial solar panels [3], depending on the configuration. By using about the same amount of resources but producing a higher yield, the resources are used more efficiently and sustainably. The market share of bifacial solar modules is currently (2023) 35% and is expected to increase to 70% within the next 10 years [4], as can be seen from figure 1. Bifacial solar panels are therefore a technology with a bright future.

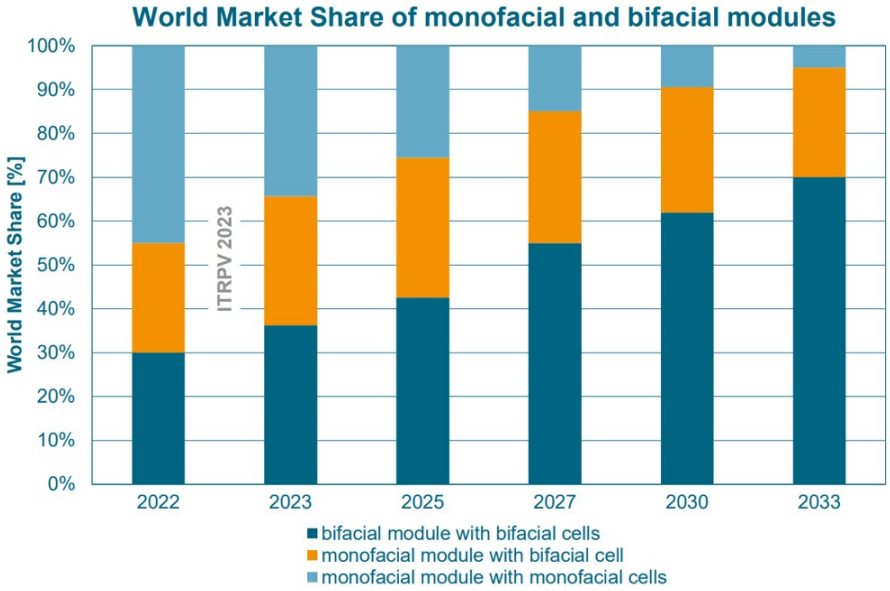


Figure 1: Market share of bifacial modules and monofacial modules. Figure from VDMA [4].

A promising application of bifacial solar panels is positioning them on fields used for agriculture. This could reduce the land competition between food and energy [5]. By optimising the density, elevation and tilt of the modules, sunlight is distributed between the panels and the crops. The modules can protect crops from adverse weather conditions (reducing thermal stress and soil leaching after rain) as well as reducing water evaporation. As solar panels are often cleaned with water, water irrigation can be further reduced [5]. Furthermore, combining photovoltaics with agriculture can improve the livelihood of farming communities and accelerate solar energy investments [5]. Bifacial vertical east-west facing panels were found to have a higher land productivity, higher spatial uniformity for sunlight and associated water distribution and more resilience to soiling loss compared to monofacial north-south oriented panels [5].

The agricultural crops also affect the solar panels. Specifically, the way in which they reflect light onto the solar panel influences the yield of the panel. The reflectance properties of the surfaces surrounding a bifacial solar panel influence its output [1]. The way a material reflects

light is both a function of the light spectrum and of the angle of the incident light, and research has shown that both of these parameters have an impact on the output of the solar panel [6, 7, 8, 9]. Modelling this reflection accurately is therefore important to compute the energy yield accurately.

1.1 Motivation for contributing to a more accurate energy yield calculation

The yield of a solar cell is influenced by geographical (e.g. the location of the panels, the time of the year, shade), environmental (e.g. clouds, aerosols, dust, precipitation, temperature, reflection of surroundings), technological (e.g. quantum efficiency, short circuit current and open circuit voltage), installation (e.g. sun-tracking, reflectors) and maintenance (e.g. cleaning interval and technique) factors [10, 1]. Furthermore, degradation of the system influences the yield as well [10]. Modelling the impact of all these individual factors on the cell performance is important to compute and predict the yield more accurately and optimise system configurations for a particular location. An accurate yield calculation comes with the following benefits:

- *Reduces investment risk barrier*
  The energy yield is vital information for making investment decisions. Inaccuracies in the yield will increase the investment risk [11] as the revenue stream has a larger error. This hinders market penetration [12]. An accurate yield will decrease the investment risk barrier, further enabling market penetration. Reduction in investment risk will therefore create the opportunity to exploit the aforementioned benefits of bifacial solar panels on a larger scale.

- *Supports grid operation*
  From an energy management perspective, an accurate yield calculation is important to make accurate forecasts. Power is a special commodity: power production and consumption must constantly match to meet customer demand and keep the grid balanced. On the electricity market, supply and demand are matched. When real-time supply and demand do not match, a shortage or surplus can occur. The grid operator activates balancing energy to prevent loss of quality of the electricity. This electricity often comes from non-renewable sources, as these have the shortest start-up time [13, 14]. To know to what extent future supply and demand match such that imbalance and ultimately loss of power quality can be avoided, both consumption and production are predicted. The accuracy of this forecast is important as an inaccurate prediction leads to imbalance. Moreover, it is required to predict reserves and efficiently schedule generation capacity [13]. Ultimately, better forecasting should lead to a system with less real-time price volatility, which benefits all stakeholders of the energy market [15].

  A more accurate energy yield calculation is beneficial for all renewable energy systems. Bifacial solar cells possess a property that is rare compared to monofacial panels: they can be put in a configuration which enables them to produce energy on moments of high energy demand. Whereas monofacial solar panels are traditionally placed facing south and therefore produce the most power around noon, east-west facing bifacial panels produce energy in the morning and the evening. These are moments of high energy demand. Supply and demand are thus matched better, which is beneficial for grid operation. This shines a unique light on an accurate yield calculation.

- *Reinforces the roots for scientific development*
  From a scientific perspective, being able to compute the yield more accurately is important for our fundamental understanding of the technology and its optimisation. For

example, understanding the impact of the reflection of materials on photovoltaic yield can be applied in agrivoltaics to design systems that optimise the energy yield and agricultural production. It also opens doors to optimise reflectors that boost the yield of solar panels. For example, in the urban environment, an idea that is currently being researched is the use of a reflector which takes in light from all directions and emits light in only one direction, thereby enabling a light beam to be aimed at a (bifacial) solar panel [16]. To arrive at a correct optimisation for this system, accurately determining how reflection impacts the yield of a bifacial solar panel is fundamental.

1.2 Goal of this thesis

The aim of this thesis is to model the influence of a very common reflecting surface, namely grass, on yield calculations. Grass is an important agricultural crop, as it is fed to ruminant animals and therefore lies at the basis of products like meat and milk [17]. As grass is an abundant crop, it is useful to be able to accurately calculate the yield of a bifacial solar panel surrounded by grass. An additional motivation to choose grass is that its structure is relatively simple compared to other crops.

The reflection of grass is known to vary with wavelength and therefore has a spectral dependence [18]. Furthermore, grass is known to exhibit retroreflective behaviour: light reflects back towards the source [19, 20]. The reflection of the environment is only superficially included in (agri)voltaic research: it is either neglected, given a constant value [5, 21, 22] or at its best spectrally taken into account [23, 18]. In this thesis, the *spectro-angular* reflection of grass will be modelled and its influence of the yield of a bifacial solar panel investigated.

To this end, two models are used. I developed a model in Matlab which simulates the reflection of grass. In the model, rays are shot from a position in the sky onto a sample of grass. When a ray hits a blade of grass, it is locally reflected. By performing this calculation many times, the global spectro-angular reflection can be computed for a certain angle of incidence of the rays. This calculation is repeated for various grass samples and angles of incident light to obtain a library of spectro-angular reflection. The obtained reflection is compared to experimental results. The information in the library is used as input for the model of Pal [9], which can be used to compute the energy yield of a bifacial solar panel taking into account spectro-angular reflection of a surface. Using this model in combination with additions from the work of Rikhof [24], the energy yield of a bifacial solar panel is computed. Lastly, this method, consisting of the combination of the reflection model and the energy yield model, is applied to a case study to analyse the impact of mowing on the energy yield for a bifacial solar farm in Germany.

The contributions of this thesis are:

- Development of a physical model to simulate the reflection of grass;
- Verification of the simulated reflection using experimental data;
- Calculation of the energy yield of a bifacial solar panel based on the reflection properties;
- Comparison of the energy yield of a bifacial solar panel between modelling the grass diffusely and as a retroreflector;
- Application of the calculation method to a case study.

1.3 Outline

This thesis is structured in the following way. Chapter 2 provides the theoretical background. It explains the concept of retroreflection, explores literature on the reflection of grass, elaborates on the need for a model that can simulate this reflection and provides more details on the model used to compute the energy yield. Chapter 3 introduces the model which is used to quantify the reflection of grass and explains it in detail. Chapter 4 presents the simulated reflection. This is compared to experimental results and applied to a case study to analyse the impact of the spectro-angular reflection of grass on the energy yield. Based on this work, recommendations are proposed in Chapter 5. Lastly, Chapter 6 provides the conclusion and the outlook.

2 Theoretical Background

This chapter presents the main concepts that are underlying this work. First, the physics of reflection is introduced. Literature about the reflection of grass specifically will be discussed, including existing models. Lastly, the model used to obtain the energy yield from the spectro-angular reflection is presented.

2.1 Reflection

When light strikes an interface between two media, part of the light is scattered backward, which is called *reflection* [25]. As reflection is a three dimensional phenomenon in space, a coordinate system has to be defined first.

In this thesis, the angles are defined in the *spherical coordinate system*, as shown in figure 2. Any point in the system can be represented by specifying three variables: the azimuth (ϕ), the elevation angle (θ) and the radius (r). In the coordinate system used in this thesis, the elevation angle runs from $\theta = 0°$ to $\theta = 90°$ and the azimuth from $\phi = -180°$ to $\phi = 180°$, as shown in figure 2. The position at which $\theta = 90°$ is called *nadir* position. When a plane is placed in a three dimensional space, the plane effectively splits the space in two: each half is called a *half-space*. Light reflected from a surface travels into the half-space from which the light was incident.

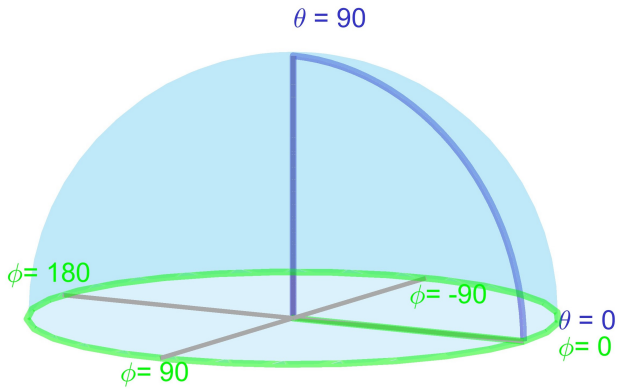
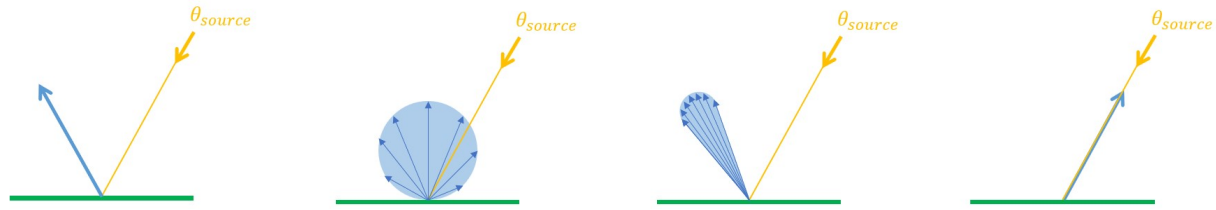


Figure 2: The main coordinate system used in this thesis. The azimuth ϕ runs from $\phi = -180°$ to $\phi = 180°$ (in green) and the elevation angle θ from $\theta = 0°$ to $\theta = 90°$ (in blue).

There are many ways a surface can reflect incoming light from a certain direction specified by θ_{source} and ϕ_{source}. To help characterise the reflection, the following terms are often used: specular reflection, diffuse reflection, glossy reflection and retroreflection, which are displayed in figure 3. A *specular* reflector reflects light into a unique direction (see figure 3a). This direction has the same elevation angle to the surface normal as the incoming light has. In contrast, a *diffuse reflector* has a rough surface and the incoming light is scattered in every direction with equal probability (see figure 3b)[9, 26, 25]. An ideal diffuse reflector, which reflects all incoming light, is called *Lambertian*. Both of these conditions are extremes. A *glossy* reflector represents the in between case - the reflectance does have an angle but is not perfectly specular (see figure 3c).

Another type of reflection considered less often is *retroreflection* or *backscattering* (see figure 3d). Light incident on a retroreflective surface will exit from the incident direction. Man-made retroreflective surfaces are often used in situations where safety is important. For example, retroreflective surfaces can be found in safety clothes. There are also retroreflective surfaces in nature, for example in biological tissues [27].



(a) Specular reflection. (b) Diffuse reflection. (c) Glossy reflection. (d) Retroreflection.

Figure 3: Two dimensional schematic of four types of reflection: a) specular, b) diffuse, c) glossy and d) retroreflection. The incoming beam at elevation angle θ_{source} is reflected as illustrated by the blue area.

2.1.1 Quantifying reflection

Figure 3 shows that the reflection may differ per angle from which it is observed - it depends on the solid angle. A *solid angle* or *viewing angle* is the area of a patch on a sphere divided by the squared radius of the sphere [28]. A hemisphere as depicted in figure 2 has a total solid angle of 2π. With the notion of solid angle, *radiance* can then be defined as the radiant flux reflected by a surface, per unit solid angle per unit projected area. As the flux has a spectral component, the radiance has a spectral component as well. The dependence on solid angle ensures that the reflection can vary in three dimensional space. Furthermore, the radiance may change as the angle of incidence varies. This can easily be seen from specular reflection: if the angle of incidence changes, the angle angle at which the light leaves the surface also changes and thus the radiance has changed. Radiance is defined with respect to a surface perpendicular to the direction from which the light is coming [28].

The albedo (ρ) of a surface, the ratio between the power of the reflected light and the power of the incoming light, is a measure of how light is reflected by a surface [6]. The albedo is, like radiance, in principle spectro-angular: the power of the reflected light depends on the radiance. However, in literature its many-parameters nature is often reduced to spectral albedo (integrated over all angles) or simply the albedo (integrated over all angles and the spectrum). Spectral albedo is the ratio of the spectral *exitance* ($E_{out}(\lambda)$) over the spectral *irradiance* ($E_{in}(\lambda)$). The exitance and irradiance are the leaving and incident flux per unit projected area, respectively. Compared to radiance, exitance and irradiance are not a function of direction. Spectral albedo is thus defined as follows:

$$\rho(\lambda) = \frac{E_{out}(\lambda)}{E_{in}(\lambda)} \tag{1}$$

For a Lambertian surface, the albedo at every wavelength is $\rho(\lambda) = 1$: the flux incident on the surface is just as large as the flux leaving a surface.

A useful concept to quantify the spectro-angular albedo is the bidirectional reflectance distribution function (BRDF). Introduced by Nicodemus et al. [29], this function relates light falling upon a surface from a certain angle of incidence $(\phi_{source}, \theta_{source})$ to light reflected by that surface. The reflection varies along the direction $(\theta_{out}, \phi_{out})$ and along the spectrum (λ). The BRDF is defined to be the ratio between the total reflected intensity in direction $(\theta_{out}, \phi_{out})$ to the energy incident per unit time and per unit area onto the surface from direction $(\phi_{source}, \theta_{source})$ [30]:

$$f(\theta_{source}, \phi_{source}, \theta_{out}, \phi_{out}, \lambda) = \frac{L(\theta_{source}, \phi_{source}, \theta_{out}, \phi_{out}, \lambda)}{E_{in}(\lambda)} \tag{2}$$

The BRDF can range from simple to complex based on the type of reflection that the surface

has. For example, the BRDF of a Lambertian surface ($f_{Lambertian}$) is a constant:

$$f_{Lambertian} = \frac{1}{\pi} \tag{3}$$

In contrast, the BRDF of surfaces which have a specular or retroreflective component will also depend on the angle of incidence of the light (θ_{source} and ϕ_{source}), and the angle from which the reflection is observed (θ_{out} and ϕ_{out}). Furthermore, as the surface may absorb certain wavelengths, the BRDF has a spectral dependence. The first objective of this thesis is to obtain the BRDF of grass.

2.2 The reflectance of grass

There are several scientific communities interested in the BRDF of surfaces. There is interest coming from computer graphics community, as they try to optimize the graphics of computer games and simulations. For example, Shah, Kontinnen and Pattanaik [31] constructed a spatially varying BRDF (also known as *bidirectional texture function*) to simulate the texture of grass for given a viewing angle and illumination conditions. Another research branch of science that looks into reflection of surfaces is Earth observation. The BRDF is relevant as it carries information about the vegetation [32]. For example, Zheng et al. [33] used spatial, temporal, and spectral variations in albedo to research vegetation changes in China's grasslands. Biogeoengineers are interested in altering the reflection of a surface on a large scale, so-called *albedo management*. The idea is to increase the albedo of e.g. agricultural land to reduce the regional warming and preserve soil moisture. Seneviratne et al. [34] found that increasing surface albedo by 0.1 could reduce the mean annual temperature by 1°C and the annual maximum daytime temperature by 2-3°C. Knowing which properties of the grass contribute to the BRDF is thus relevant for multiple science fields.

With all these fields also comes knowledge about the BRDF of grass. In this section, the focus will be on insights from Earth observation science, as their interest (i.e. retrieving information on vegetation using the reflection) is essentially the exact opposite from one of the subgoals of this thesis: retrieving the reflection based on the vegetation.

2.2.1 Grass as canopy

A way to look at grass is to consider it a canopy. A canopy is a term for the collection of the tops of multiple plants. Several common parameters for characterising a canopy are [32]:

- *Leaf Area Index* (LAI): the total one-sided area of photosynthetic tissue per unit ground surface area [35] [36].
- *Leaf Angle Distribution* (LAD): the probability of the leaf normal falling within an unit interval of inclination angle. Often, a mathematical function is used to model this probability. Some are used so commonly that they have their own name. In planophile canopies, horizontal leaves dominate, while in erectophile canopies, vertical leaves dominate [37].
- Leaf geometric parameters such as relative leaf size and shape.

Canopy has been observed to exhibit a retroreflective component. This retroreflective component is also known as the *hot spot*, *Heiligenschein* or *opposition effect* [38, 39]. In Earth observation science, *hot spot* is the most commonly used word for this phenomenon. In the case of grass, as the blades cast shadows, the shadows cannot be seen along the direction of the incident light as the light will be screened by the blades [39]. Research from this science

branch shows that leaf canopy BRDFs around the retroreflective region are known to depend strongly on leaf geometry [32].

2.2.2 Experimental measurements reflection of grass

The reflection of grass has been measured in several studies. One way to measure the reflection of grass is using a goniometer. A *goniometer* is a measurement device where a light source is aimed at a sample, after which the reflected light is measured at several angles using a detector. A disadvantage of this setup is that measuring at the hot spot itself is impossible, as the detector then blocks the light source.

There are ways to measure closer to the hot spot. For example, Belcour et al. [40] used a beam splitter to be able to measure at the same place as the light is coming in. Roosjen et al. [20] also circumvented this problem, using an industrial robot-arm. They measured lawn grass of the species *Lolium perenne L.* They observe a retroreflective component, which they attribute to the erectrophile LAD which causes more internal shadow casting within the canopy. They noticed that the reflectance decreases as the elevation angle increases, which they attribute to the soil as the grass is not fully covering the surface - especially at nadir position.

The reflection of the same grass species has also been analysed by Sandmeier et al. [19] in the form of anisotropy factors. The *anisotropy factor* is defined as the portion of the reflected radiance relative to the nadir reflectance, sometimes also called relative reflectance [41]. To compute the anisotropy factor, the reflectance is measured using a goniometer. Close to the hot spot, they measured an anisotropy factor of about 2 for $\lambda = 550nm$, which indicates that the retroreflection lobe can be rather sharp.

Besides the structural properties of grass by considering it a canopy, the properties of the individual grass blades are also an important component in the overall reflectance. For example, Carter [42] found that as leaf water content decreases in the grass species *Arundinaria tecta*, leaf reflectance increases over the entire spectrum from 400 - 2500 nm. These grass properties might be affected by for example mowing. Clark, Prioul and Couderc [43] observed that the relative water content in a leafs of Italian ryegrass decreased by about 15% in the first 30 minutes after cutting. Dyer, Turner and Seastedt [44] hypothesize that mowing could alter physiological processes in the plant, increasing its reflectance. So, physiological properties of the grass also play a role in grass reflection.

2.2.3 Numerical methods to simulate the reflection of grass

Models to investigate the canopy are called *canopy reflectance models* (CRM). Four main classes, grouped by their approach and complexity, are [45]:

- *Geometrical models* treat the canopy as translucent geometric shapes;
- *Turbid medium models* describe the canopy as a horizontally uniform plane-parallel layer with absorbing and scattering particles;
- *Hybrid models* use a combination of geometric models and turbid models;
- *Monte-Carlo ray tracing models* trace individual rays from the source to the receiver. A *ray* is defined as line drawn in space which represents the direction of flow of radiant energy [25]. The chain of scattering events encountered on the path of each ray are simulated by modelling only the single scattering properties (Monte-Carlo).

A popular model is the PROSAIL model, which consists of a leaf optical properties model PROSPECT and the turbid medium canopy model SAIL. Leaf optical properties obtained

from PROSPECT are fed into SAIL, which gives the reflectance of the canopy [45]. The current PROSAIL model computes the BRDF from 400-2500 nm in increments of 1 nm as a function of sixteen inputs, including water content, canopy architecture and solar diffusivity. The model makes the assumption that leaves are broad and behave like a Lambertian reflector. Furthermore, the canopy is modelled as being homogeneous and gaps in the canopy can therefore not be modelled accurately. Ray tracing models are therefore more accurate [46].

There are various Monte Carlo ray tracer models to model a canopy. For example, the model of North [47] does not model the leaves individually, but computes the intersection of the ray with the blade by random sampling of the distance until collision. This probability is a function of the previous direction of the photon and the total leaf surface area per unit volume of space. They arrive at an expression for the probability that a leaf at a certain depth in the canopy is both illuminated and viewed, and use Monte Carlo simulation to sample this probability.

In the 1980's, Juhan and Marshak [48] developed a Monte Carlo method to analyse the influence of leaf orientation and the specular component of leaf reflectance on the BRDF. They found that the BRDF of a canopy with more electrophile leaves differs the most from a Lambertian surface for low angles of incidence. They also found that considering the specular component of leaf reflectance significantly impacted the BRDF, especially for high elevation angles as the leaf's orientation becomes more and more horizontal [49].

Qin and Goel [50] compared hot spot models for canopies and found that the retroreflective lobe is broader for high elevation angles compared to lower elevation angles. Furthermore, for a given elevation angle, the retroreflective lobe is broader when the LAI is high, the leaves are more square-like and the LAD is planophile compared to a low LAI, rectangular leaves and erectophile LAD. As the ratio between the mean leaf width to the length increases (and the grass becomes more square-like), the width of the retroreflective lobe increases. Qin et al. [32] found that the retroreflective effect is more sensitive to changes in leaf dimension when the elevation angle is low.

2.2.4 Motivation for the development of a Monte Carlo ray tracing model

I developed a Monte Carlo ray tracing model to obtain the spectro-angular BRDF of grass, which I use to eventually compute the energy yield. This model is introduced in chapter 3.

A model has some pronounced advantages over obtaining the BRDF experimentally. Firstly, the model is free from the challenges of a physical goniometer: light can be measured in three dimensions and there is no detector blocking the source. The sample can have the desired dimensions which might not physically fit in a goniometer. Furthermore, experimenting with different types of grass and soil can be done easily without having to change the samples in the goniometer, which could contaminate the machine. The model introduces control over many parameters, including but not limited to the light source, detection mechanism and sample.

Compared to other numerical methods to obtain the BRDF of grass, a main advantage of choosing a Monte Carlo ray tracing method is that it gives the most insight in how the sample parameters influence the BRDF compared to the other CRM introduced above.

2.3 Incorporating the reflectance in photovoltaic yield

Knowing the BRDF is important information for calculating the yield and optimizing the system, as argued in section 1.1. Several studies showed that taking the spectral aspect into account is important for an accurate yield calculation [6, 7, 8]. Pal and Saive [51] simulated and Van Loenhout [52] experimentally verified that the different angular reflection behaviours affect

the current density of a solar panel. The spectro-angular reflection properties of a reflector thus impact the albedo and ultimately the yield that is obtained.

In this thesis, the simulated BRDF of grass will be used as input in the model of Pal [9] to calculate the resulting yield. This is an optical model: the irradiance that reaches the front and rear side of the bifacial solar panel is computed [53]. It is also a *reverse ray tracing* (RRT) model. In a RRT model, the path of the rays from the module to the sun is followed.

The model of Pal [9] consists of two surfaces, namely the bifacial solar panel and the reflector. Both surfaces are divided into pixels. Furthermore, the location of a light source can be specified, as well as its spectrum. The geometry of the setup can be adapted. For example, the solar panel can be tilted as desired.

In the model, the flux of photons incident on the bifacial solar cell consists of three parts:

1. the flux that reaches the bifacial solar panel directly from the source;
2. the flux that reaches the front of the bifacial solar panel indirectly, i.e. by first being reflected by the reflector;
3. the flux that reaches the rear of the bifacial solar panel indirectly.

The software computes the short circuit current density in every module pixel due to each of these three fluxes. The *short circuit current density* (j_{SC}) is the maximum current that the solar cell can obtain in a certain area [54]. A summation over the short circuit current density per pixel is performed to compute the *short circuit current* (I_{SC}).

The short circuit current is used to compute the power produced by the solar cell. This calculation requires the values of two more parameters: the open circuit voltage and the fill factor. The *open circuit voltage* (V_{OC}) is the maximum voltage that can be obtained from a solar cell [54]. Operating the cell at either the short circuit current or the open circuit voltage does not yield any power, as there is either no voltage or no current, respectively. Instead, the cell is ideally operated at its *maximum power point* (MPP) [54]. At this voltage (V_{MPP}) and current (I_{MPP}), the most power is produced. This point is indicated on a sketch of the current voltage diagram in figure 4. It can be found by multiplying the short-circuit current and open circuit voltage by the *fill factor* (FF). Both the open circuit voltage and the fill factor depend on the solar cell type [54]. The maximum power (P_{MPP}) can thus be computed as follows:

$$P_{MPP} = FF \cdot I_{SC} \cdot V_{OC} \tag{4}$$

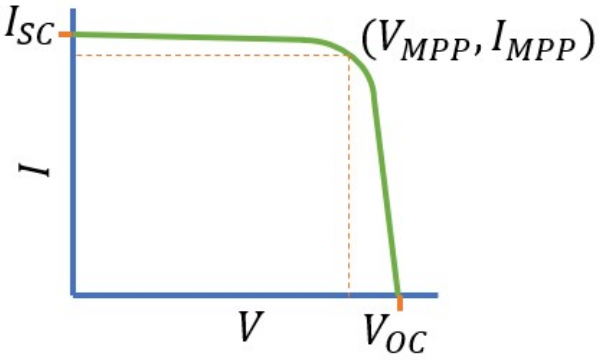


Figure 4: In this sketch of a current voltage (IV) plot, the short circuit current (I_{SC}), open circuit voltage (V_{OC}) and maximum power point (MPP) are displayed. The fill factor (FF) multiplied by the I_{SC} and V_{OC} is the area of the largest rectangle which fits under the IV curve.

Running the model of Pal [9] for multiple positions of the sun allows for computing the short circuit current at each of these positions. Multiplication by the time the sun spends in every position, the open circuit voltage (V_{OC}) and the fill factor (FF), allows for computation of the *power profile* ($P(t)$). This calculation has been implemented by Rikhof [24]. Integration of the power profile over time gives the *energy yield*.

An important property of this model for this work is that it allows for computing the energy yield while taking the spectro-angular reflection of the reflector into account. Furthermore, as the current I_{SC} can be traced back to the three types of fluxes, the model can be employed to get insight in how the energy yield is built up. For example, it can be used to analyse the contribution of each of the three fluxes of light to the power profile and the energy yield.

3 Ray tracing model for simulation of grass reflection

In this chapter, the ray tracing model that I developed will be explained. First, an overview of the model is given. Then, each main component of the model is described in detail. The introduced parameters are calibrated: their optimum value to reduce the error of the ray tracer is explored. Lastly, the use of the model is discussed.

3.1 Overview ray tracing software

A Monte Carlo ray tracing model has been designed to determine the BRDF of grass. The model has been made in Matlab. The code can be found in appendix D. A brief guide how use the code to reproduce the main figures in this report can be found in appendix C.

The main assumptions of this model are:

- A ray can be modelled as a vector with a certain direction, a magnitude and wavelength;
- A grass blade can be modelled as a vertical rectangle and exhibits diffuse reflection;
- The ground can be modelled as a horizontal plane and as a blackbody;
- Depending on the wavelength of the incoming wave, a portion is reflected, the rest of the wave magnitude is assumed to be absorbed and lost from the system;
- Rays whose magnitude at a all wavelengths is smaller than 1% of the initial magnitude are considered negligible and disappear from the system[1].

A bundle of parallel rays is generated which is directed at the volume of the space under examination. The space is filled with a horizontal surface (the soil) and vertical surfaces (grass blades). When a ray intersects with a surface, it is redirected in a random direction (diffuse reflection) and it loses spectral magnitude (energy) depending on the reflectance of the surface at that wavelength. Once it has been redirected, the ray may bump into another surface, after which its direction and magnitude are adapted again according to the reflectance. When the ray intersects with the detection dome, it is condidered to be detected and its spectral magnitude at the intersection point is stored. Figure 5 shows how a single ray is traced from the source (green dot on the dome). The ray is reflected by the grass and eventually detected (blue dot on the dome). By repeating this process for many rays, the total magnitude of the rays that intersected in a specific part of the detection dome grid is obtained. By normalizing this spectro-angular radiance, the BRDF is obtained.

[1]This assumption is used to avoid the model to trace light indefinitely. Many Monte Carlo ray tracers have such a condition, for example the ray tracer software SunSolveYield [55].

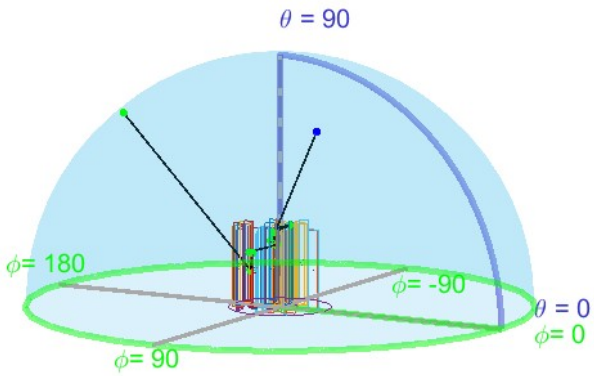


Figure 5: Visualisation of the ray tracing model. The same coordinate system is used as in figure 2. The hemisphere represents the detection dome. The sample is placed in the middle of the dome and consists in this case of a circular ground with vertical rectangular grass blades on top. A ray (black line) is shot from the detection dome (green dot), reflected between grass blades and eventually detected (blue dot).

Algorithm 1 shows the pseudocode for the ray tracing program. After ray and plane generation, the path of every ray is traced. The first task of the algorithm is to find the first surface (be it the ground, grass or the detection dome) that the ray intersects with.

To this end, first the intersection time of the ray with the detection dome and all surfaces (i.e. the ground and the grass) has to be computed. This serves two purposes: 1) it is used to check which surface is hit first and 2) it is needed to compute the intersection point of the ray with the surface. For the grass and the ground, the plane in which the surface is situated is used for this calculation. All surfaces (ground, grass and detection dome) are sorted on intersection time in ascending order for computational efficiency. For every surface in this list, the software computes if the ray hits this surface. If the ray intersects with the detection dome, the ray is detected and thereby the program breaks out of the outer *while* loop and starts tracing the next ray. If the surface is not the detection dome, the intersection point of the ray and the plane in which the surface lies is computed. If the ray falls within the boundaries of the surface, the ray changes direction and magnitude according to the refectance at that wavelength and the program breaks out of the inner *while* loop. Using the new ray properties, it proceeds to compute the intersection time with every surface again and checks if the ray hits any of the surfaces. If the ray does not hit the detection dome, nor the surface in the plane, the next surface in the list of sorted surfaces is considered, while the ray's properties remain unchanged. If the magnitude of the ray is lower than 1% for all considered wavelengths, the path of the ray is no longer traced.

In the following sections, the main components of the algorithm will be explained in detail.

Algorithm 1 Ray tracing algorithm to determine the BRDF of grass.

```
generate sample
generate a bundle of parallel rays
for every ray do
  while ray is not detected do
    if ray magnitude < 0.01 then
      break
    end if
    calculate intersection time of ray with detection dome
    for every plane do
      calculate intersection time of ray with plane
    end for
    sort surfaces on intersection time
    k_plane ← 1
    while k_plane < number of surfaces do
      if surface is the detection dome then
        detect ray
        break
      end if
      calculate interscetion point ray with plane
      if ray hits the surface then
        change ray direction
        change ray magnitude
        break
      end if
      increase k_plane
    end while
  end while
end for
```

3.2 Sample generation

The sample can consists of two types of surfaces: circular surfaces and rectangular surfaces. Figure 6 shows a sample for grass, where the ground is modelled as a circle and the blades are assumed to be vertical rectangles - a blade is considerably thin compared to its width and length. Therefore, the surfaces are two dimensional in an otherwise three dimensional model.

The properties attributed to these surfaces are elaborated upon below.

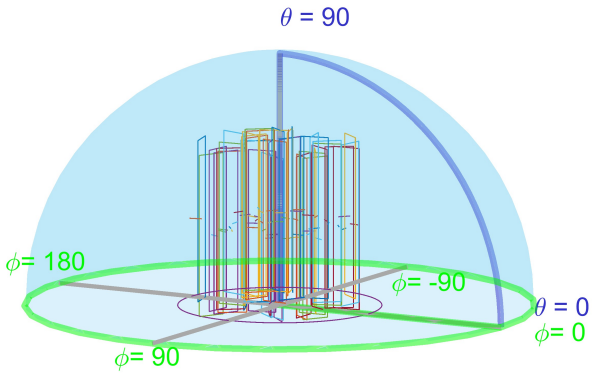


Figure 6: Example of a grass sample in the coordinate system, where the ground is modelled as a circle and the grass blades are modelled as vertical rectangles. The small lines perpendicular to the surfaces are the surface normals.

Circular surface properties
For a circular surface, the centre, radius (r_{sample}) and height of the surface are specified. For the sample grass, the ground is considered to be a circular surface whose center is in the middle of the detection dome.

Rectangular surface properties
For a rectangle, all four corners of the surface need to be specified. Furthermore, the centre of the surface is saved and the length and height can be set.

In the example of the grass sample, the grass blades are modelled as vertical rectangles, whose centres (ϕ_b, r_b) are randomly (but uniformly) distributed over the circular sample in the xy-plane. Therefore, the centres are chosen by uniformly sampling a disc.

$$\phi_b = 2\pi\xi_1 \qquad (rad) \tag{5}$$

$$r_b = \left(r_{sample} - \frac{1}{2}w_{blade}\right)\sqrt{\xi_2} \qquad (rad) \tag{6}$$

$$\xi_1, \xi_2 \in [0, 1] \tag{7}$$

where the random numbers ξ_i are generated by Matlab's *rand* function. This function returns uniformly distributed random numbers between 0 and 1. The grass blades therefore spawn randomly but uniformly in the sample. By restricting the radius in which the centre of the blades are allowed to spawn by half of the width of the blade, no blade falls outside of the

radius of the sample. Figure 7a shows the sample from nadir position and shows the randomly spawned points (in orange) in the allowed area (shaded in gray).

Now that the centres (x_b, y_b) of the samples projected on the xy-plane are known, choosing a height of the grass blades (z_{blade}) and width of the grass blades (w_{blade}) allows for determining the corners of the grass blades as illustrated in figure 7b:

$$x_1 = x_4 = x_b - \frac{1}{2} w_{blade} \tag{8}$$

$$x_2 = x_3 = x_1 + w_{blade} \tag{9}$$

$$y_1 = y_2 = y_3 = y_4 = y_b \tag{10}$$

$$z_1 = z_2 = z_{blade} \tag{11}$$

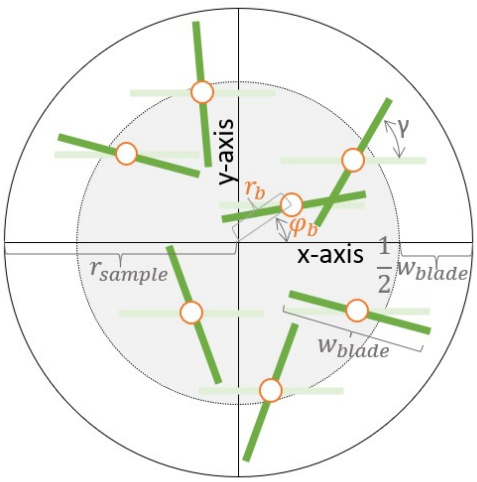
$$z_3 = z_4 = 0 \tag{12}$$

where subscripts 1,2 3 and 4 refer to the left top corner, right top corner, right bottom corner and left bottom corner, respectively. Now, the sample consists of rectangular blades that are randomly oriented on the sample, shown by the light green lines in figure 7a. Lastly, the grass blades are rotated randomly such that each blade is facing a random direction, where rotation angle γ is:

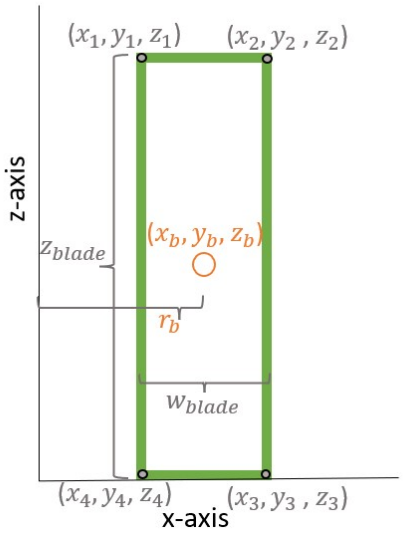
$$\gamma = 2\pi\xi_3 \tag{13}$$

$$\xi_3 \in [0, 1] \tag{14}$$

In figure 7a, the dark green lines represent the rotated grass blades as viewed from the top. As the centres and rotation of each grass blade depends on these equations only, grass blades can also grow through each other. This is illustrated by the two grass blades in the top right quarter in figure 7a.



(a) A grass sample as viewed from the nadir position. The centres of the grass blades (in orange) are randomly distributed over the allowed area (shaded in gray). First, the grass blades are all parallel to the x-axis (in light green). They are rotated by angle γ to make them face a random direction (in dark green).



(b) A grass blade as viewed along the y-axis. First, its centre is determined (in orange). The coordinates of the angles of the blade are determined based on this position. The orientation of the blade is randomized later by application of a random angle γ (see figure 7a).

Figure 7: Two dimensional illustrations of the main parameters relevant to the generation of the sample.

Non-sample area
The area of the bottom of the detection dome that is not a sample[2], is modelled to behave like a blackbody. A ray of light that intersects with this area is lost from the system.

3.3 Ray generation

Generating the first ray
A ray can be described by a vector: it has both a magnitude and a direction. From hereon, any vector is a three dimensional vector with three components as defined by the Cartesian coordinate system - e.g. any vector $\vec{v}$ has an x, y and z component. Consider a ray starting at point $\vec{e}$, travelling along the direction $\vec{d}$. The point $\vec{p}(t)$ where the ray is at time t can then be computed as follows [56]:

$$\vec{p}(t) = \vec{e} + t\vec{d} \tag{15}$$

Using this notation, a ray is created with the following properties:

- Starting point ($\vec{e}$), computed from the azimuth (ϕ_{source}) and elevation (θ_{source}) angle of the source and the radius of the detection dome.
- Direction ($\vec{d} = \vec{s} - \vec{e}$) where $\vec{s}$ is the end point. The first ray that is generated is aimed at the middle of the detection dome. The height h at which the rays are aimed can be chosen by the user. The choice for height influences the results and the height should therefore be chosen carefully, as described in section 3.7 ($(s_x, s_y, s_z) = (0, 0, h)$).
- Magnitude per wavelength $m(\lambda)$. The value of the magnitude can be chosen freely, as the magnitude measured will eventually be normalised to obtain the BRDF. Therefore, initially, $m(\lambda) = 1 \forall \lambda$.

Note that knowing the time t and the first two properties allows us to compute the position of the ray using equation 15. Figure 8 shows one generated ray. Its starting point corresponds to the green point and it ends at the middle of the detection dome. The magnitude per wavelength allows for measuring the spectral dependence of the radiance.

Generate a bundle of parallel rays
A bundle of rays that fall uniformly on the surface is generated based on the direction of the first ray. Figure 9 shows a bundle of 50 rays. The direction of all rays is the same ($\vec{d}$), but the end point (s_x and s_y, $s_z = h$) differs such that the rays are falling uniformly distributed on the surface, mimicking a beam. This beam has a circular cross section on the xy-plane at height h. When $h = 0$, the end points of the rays are uniformly distributed on the ground in a certain illuminated area (the area within the yellow circle in figure 9). Figure 10 is a sketch of the view from nadir position, where the illuminated area is shown by the yellow-shaded area.

The sample (which is a circle) can be divided into an illuminated and non-illuminated area. As the cross section of the beam is always circular on the ground, the illuminated area $A_{illuminated}$ can be determined with respect to the area of the sample A_{sample}:

$$A_{illuminated} = a^2_{illuminated} A_{sample} \tag{16}$$

$$r_{illuminated} = a_{illuminated} r_{sample} \tag{17}$$

[2] $A_{non-sample} = A_{dome} - A_{sample} = (a^2_{dome} - 1)A_{sample}$ according to equation 30.

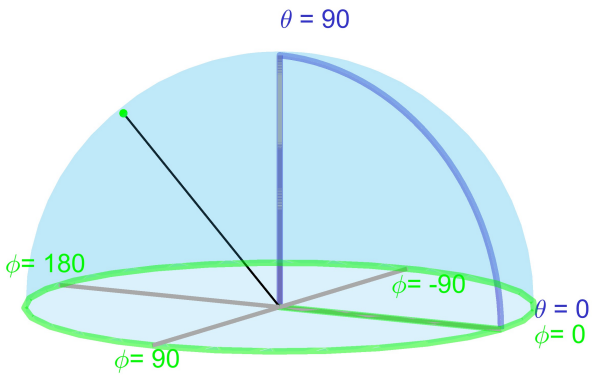


Figure 8: One ray with $\phi_{source} = 180°$ and $\theta_{source} = 45°$ is shot to the centre of the detection dome.

where $a^2_{illuminated}$ is the fraction of the area which is illuminated compared to the sample area. In principle, the value of $a_{illuminated}$, the ratio between the radius of the illuminated area $r_{illuminated}$ and the radius of the sample r_{sample}, can be any value: however, $a_{illuminated} > 1$ is not sensible as part of the rays will be shot next to the sample and thus lost immediately. The value of $a_{illuminated}$ has been optimised, as is further elaborated on in section 3.7.

The properties of the parallel rays are:

- Direction ($\vec{d}$) as computed for the first ray.
- Starting point ($\vec{e}$) is computed using equation 15. Here, $\vec{d}$ is as defined above. The end points $\vec{s}$ are chosen such that the end points are uniformly distributed in a circle on the chosen height h, which is the same height as for the first ray:

$$\phi_s = 2\pi\xi_4 \qquad (rad) \tag{18}$$

$$r_s = r_{illuminated}\sqrt{\xi_5} \qquad (rad) \tag{19}$$

$$\xi_4, \xi_5 \in [0, 1] \tag{20}$$

  These spherical coordinates are then transformed to Cartesian to obtain (s_x, s_y, s_z). Lastly, time t is the time it takes before this ray intersects with the detection dome (see equation 29) and using these inputs $\vec{e}$ is computed.
- Magnitude per wavelength $m(\lambda)$ as assigned to the first ray.

All rays (including the first ray) thus have the same direction and an end point that falls within the circle of radius $r_{illuminated}$, which can be seen from figure 9.

3.4 Ray intersection

A surface can be described as a plane with boundaries. For any point $\vec{p}$ that lies in the plane of point $\vec{p_l}$ with surface normal $\vec{n}$, the following relation holds true [56]:

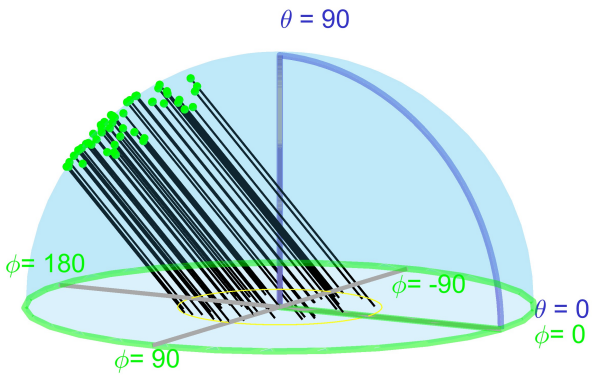


Figure 9: 50 parallel rays incident from $\phi_{source} = 180°$ and $\theta_{source} = 45°$ are shot into the illuminated area ($A_{illuminated}$), in yellow.

$$(\vec{p} - \vec{p_l}) \cdot \vec{n} = 0 \tag{21}$$

The intersection time of the ray with the plane can now be found by substituting equation 15 in equation 21 [56],

$$t = \frac{(\vec{p} - \vec{e}) \cdot \vec{n}}{\vec{d} \cdot \vec{n}} \tag{22}$$

Substituting equation 22 in equation 15 gives the intersection point of the ray with the plane, $\vec{p_i}$. This does not guarantee that the ray also intersects with the surface in the plane - the program checks if the computed intersection point falls on or within the boundaries of the surface. Only if this is the case, the surface is considered to be 'hit'.

The planes resembling the grass blades are two dimensional, but situated in a three dimensional world. If a ray is parallel to the surface and hits the surface from its infinitesimally thin side, $\vec{d} \cdot \vec{n} = 0$ [3] and no intersection time can be computed (equation 22). Therefore, the ray does not 'see' the grass blade and the ray effectively shines past it. This is the case when light shines from $\theta_{source} = 90°$ on a vertical grass blade, as well as when light shines from any ϕ_{source} to a blade whose tangent line is parallel to ϕ_{source}. In these situations, the light does not reflect on the blade.

As the starting point and direction of the ray are known, the number of surfaces that the ray can hit when it is initially incident on the surface is reduced to only the surfaces that are in the path of the ray. Computational time can be saved if only the intersection time of the blades in the path has to be computed. The initial position is specifically ideal to apply this technique, as the path of the parallel rays is almost the same - a list of surfaces that are present in this path only has to be made once. For a ray coming from the sky, the software only computes the

[3] $\vec{d} \cdot \vec{n} = 0$ or practically zero ($|\vec{d} \cdot \vec{n}| < 10^{15}$)

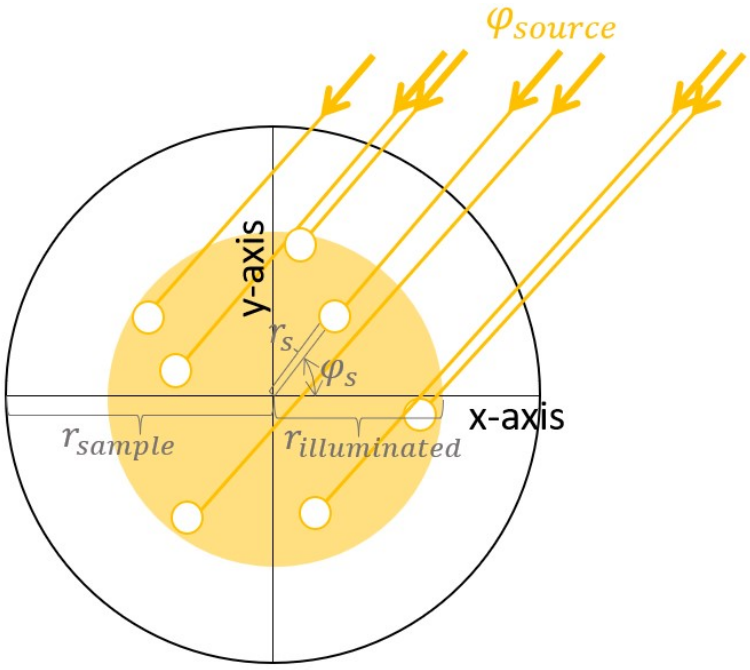


Figure 10: Two dimensional illustration of the top view of the sample (black circle). The illuminated area (shaded in yellow) lies within the sample. The light is incident from ϕ_{source}. The end points $\vec{s}$ of the light (yellow circles) have as coordinates (ϕ_s,r_s,h).

intersection time for the surfaces whose centres occur in the area $A_{illuminated,initial}$. This area is defined to have the width of $r_{illuminated} + w_{blade}$ and extends from one end of the detection dome to the other. This area is shaded in red in figure 11. Once the ray has hit a surface, all surfaces are taken into account again.

When a ray hits a surface, its direction and magnitude change.

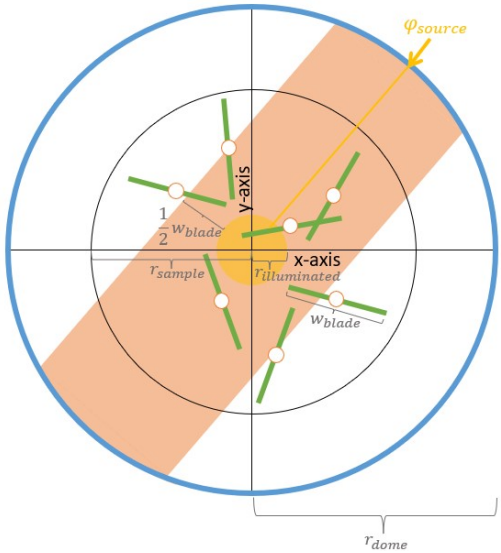


Figure 11: Two dimensional illustration of the top view of the detection dome (blue circle). The black circle represents the sample and the yellow shaded area is the illuminated area. As light is shot from azimuth ϕ_{source}, only blades (dark green lines) whose centres (orange points) fall within the initially illuminated area (shaded in red) are considered in the calculations, as these are the only blades that could potentially be hit. This reduces calculation time.

3.4.1 New direction

When a ray hits a surface, it changes direction. The method through which the new direction is chosen and the grid layout of the detection dome determine the number of photons detected in every pixel of the detection dome. Choosing a new direction and measuring the rays at the detection dome should be done such that upon measuring a Lambertian surface, the radiance measured is equal for every angle: this is the definition of a Lambertian surface (see equation 3).

The possible directions from which the ray can 'choose' depends on the orientation of the surface that the ray hits: the new direction should be physically possible. For example, if the surface is oriented horizontally on the ground, the new direction of the light has to be in the half-space above the surface. Figure 12a displays 1000 possible ray directions for one ray intersecting at (0,0) on the xy-plane. For a differently oriented surface, some directions are impossible as the ray is not allowed to travel through the surface. Figure 12b illustrates this principle, showing a half-space of possible directions in black and a half-space of invalid directions in red.

The situation described above is translated to mathematical equations in order to choose valid new ray directions. Mathematically, the new direction is chosen in the half-space of the surface normal. The valid polar angles of the new ray direction are sampled uniformly in the spherical coordinate system of the detection dome. To take the direction of the incoming light into account (and thus the side of the surface that is hit), the normal should be in the half-space of the incoming light ray. This ensures that the light reflects back into the half-space that it was coming from.

(a) (b)

Figure 12: Visualisation of possible reflection directions (1000 displayed) of one incoming ray for a horizontal (a) and tilted (b) surface. The directions in red are not valid, because they are not in the half-space of the surface normal in the direction where the light is coming from.

Ensuring that the surface normal is in the direction of the incoming ray
First, the program evaluates if the normal is in the half-space of the incoming ray. For a vector $\vec{v}$, surface normal $\vec{n}$ and related surface parameter d, the following relations are true:

$$\begin{cases} \vec{n} \cdot \vec{v} > d & (\vec{v} \text{ is in the same half-space as } \vec{n}) \\ \vec{n} \cdot \vec{v} < d & (\vec{v} \text{ is in the opposite half-space from } \vec{n}) \\ \vec{n} \cdot \vec{v} = d & (\ \vec{v} \text{ is in the plane}) \end{cases} \tag{23}$$

In these relations, the parameter d follows from the Cartesian equation which models a surface. It is is the distance from the plane to the origin:

$$d = \vec{p_i} \cdot \hat{n} \tag{24}$$

Knowing d, the relations are applied to ensure that the normal is in the direction of the incoming light. To this end, $\vec{v}$ is substituted by the direction of the incoming ray, $\vec{d}$, and the direction of the normal is flipped to $-\vec{n}$ if $\vec{n} \cdot \vec{d} < d$.

Determining a new ray direction in the half-space of the normal
Now that the surface normal is in the same half-space as the incoming ray, a new direction can be chosen. The new direction of the ray is determined by sampling a sphere uniformly in polar angles and based on random numbers generated by a normal distribution:

$$\phi_{new} = -\frac{\pi}{2} + \pi\xi_6 \qquad (rad) \tag{25}$$

$$\theta_{new} = -\pi + 2\pi\xi_7 \qquad (rad) \tag{26}$$

$$\xi_6, \xi_7 \in [0, 1] \tag{27}$$

This gives a possible new ray direction in the sphere. However, only the ray directions in the correct half-space are allowed. Therefore, after generating a new ray direction, it is checked whether this new direction $\vec{d_{new}}$ is in the same half-space as the surface normal. If this is the case, the new direction is a valid direction. If the new direction is not in the half-space of the surface normal, the new direction is rejected and a new direction is chosen according to equation 25.

The new direction of the ray is chosen such that the surface reflects diffusely, like a Lambertian reflector. Every surface that is part of the sample is modelled to reflect diffusely - the grass blades included. This is also how Juhan and Marshak [48] model the reflection of leafs.

The chosen new ray direction is transformed to Cartesian coordinates, which results in $\overrightarrow{d_{new}}$. Figure 12b illustrates that the new directions that are chosen are rotated correctly.

3.4.2 New magnitude

The new magnitude of the ray depends on the type of surface, which determines the percentage of reflectance per wavelength. The new magnitude of the ray after intersection (m_{new}) is simply the old magnitude of the ray (m_{old}) times the reflectance ($R_{surface}$) which depends on the material. The ray magnitude is thus adapted as follows:

$$m_{new}(\lambda) = m_{old}(\lambda) \cdot R_{surface}(\lambda) \tag{28}$$

The non-reflected part of the ray is assumed to be fully absorbed - in other words, lost from the system.

Included options for surface types are, among others, a blackbody (all wavelengths are absorbed, $R_{blackbody}(\lambda) = 0 \forall \lambda$) and a Lambertian reflector (all wavelengths are reflected, $R_{Lambertian}(\lambda) = 1 \forall \lambda$). Spectrally dependent reflectance data can also be used to model non-ideal surfaces. In this report, the reflectance data of grass comes from Russell et al. [6] and is shown in figure 13. This data is interpolated to get the reflectance at the input wavelengths that are set by the user at the start of the program. The soil is assumed to be a blackbody.

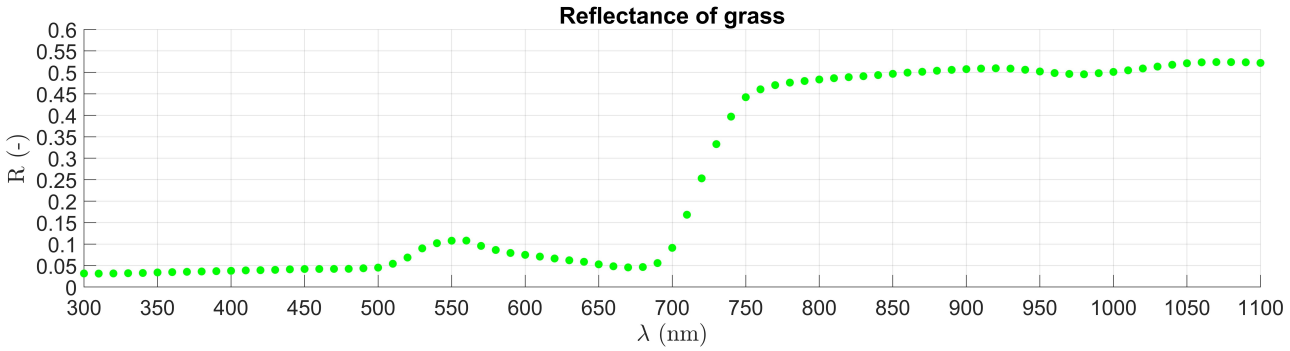


Figure 13: The reflectance of grass (R) as function of the wavelength (λ). Data from Russell et al. [6].

3.5 Ray detection

The ray is detected when it intersects with the detection dome. This is the hemisphere in which the system is contained. The dome is placed over the ground surface. To know the angular BRDF, it is important to know where the ray hits the dome - i.e. where the intersection point is between the ray and the dome.

A sphere can be described in vector notation where every point $\vec{p}$ lies on the sphere with radius r_{dome} and centre $\vec{c}$ [56]:

$$(\vec{p} - \vec{c}) \cdot (\vec{p} - \vec{c}) - r_{dome}^2 = 0 \tag{29}$$

where r_{dome} and correspondingly the area at the bottom of the hemisphere A_{dome} is determined with respect to the sample size according to:

$$A_{dome} = a_{dome}^2 A_{sample} \tag{30}$$

$$r_{dome} = a_{dome} r_{sample} \tag{31}$$

The value of a_{dome}, the ratio between the radius of the illuminated area r_{dome} and the radius of the sample r_{sample} is restricted to $a_{dome} \geq 1$ to ensure the entire sample falls within the dome. The value of a_{dome} has been optimised, as is further elaborated on in section 3.7 .

Substituting equation 15 in equation 29, the intersection time with the sphere is [56]:

$$t = \frac{-\vec{d} \cdot (\vec{e} - \vec{c}) \pm \sqrt{(\vec{d} \cdot (\vec{e} - \vec{c}))^2 - (\vec{d} \cdot \vec{d})((\vec{e} - \vec{c}) \cdot (\vec{e} - \vec{c}) - r^2)}}{\vec{d} \cdot \vec{d}} \tag{32}$$

and equation 15 can then be used to compute the intersection point. Figure 14 shows an incoming ray being reflected by the sample and detected at the blue point.

The detection dome is discretized to measure the detected ray. Therefore, it is divided into pixels. The detection dome is considered to be a spherical coordinate system, resembling a

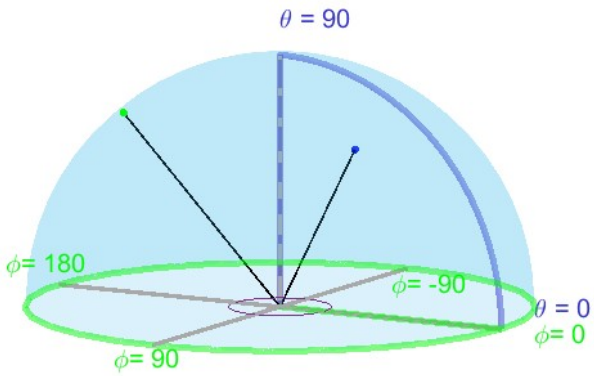


Figure 14: An incoming ray is reflected by the sample and detected at the detection point, displayed in blue.

spherical geographic coordinate system. The grid pixel size is thus uniform in terms of θ and ϕ. That is, if an accuracy of β degrees is chosen, the matrix $C(i,j)$ containing the centre of every pixel looks as follows:

$$C(i,j) = C(\theta,\phi) \tag{33}$$
$$C(i+1,j) = C(\theta+\beta,\phi) \tag{34}$$
$$C(i,j+1) = C(\theta,\phi+\beta) \tag{35}$$
$$C(i+1,j+1) = C(\theta+\beta,\phi+\beta) \tag{36}$$

In this way, the entire C matrix is filled, where each value belongs to the centre of a pixel on the detection dome. Figure 15 shows the pixels on the detection dome and their centres when $\beta = 15°$. The sum of the final ray magnitude (m_{final}) at every wavelength of all rays hitting the pixel with center C(i,j) pixel is saved in the matrix grid pixel C(i,j).

When a ray hits the detection dome, it is checked which pixel was hit by looking for the smallest difference between the detection point ($\theta_{detected}$ and $\phi_{detected}$) and the centres of the pixels (θ_C and ϕ_C). Essentially, the program finds in which pixel the detection point (blue dot) falls, as illustrated in figure 15.

Then, the final magnitude of the ray at each wavelength (m_{final}) is added up to the spectral value that this pixel already had. In this way, the spectro-angular radiance is obtained: the radiant flux reflected by a surface, per unit solid angle per unit projected area. So, for each bundle of rays shot at a specific angle of incidence (θ_{source}, ϕ_{source}), the spectro-angular radiance ($L(C,\lambda)$) can be determined. The formula below describes how the radiance for a single pixel and wavelength is built up: the magnitudes of the rays (m_{final}) at a wavelength (λ) are summed for all rays ($n(C)$) that hit a pixel C.

$$L(C,\lambda) = \sum_{n(C)} m_{final}(\lambda, C) \tag{37}$$

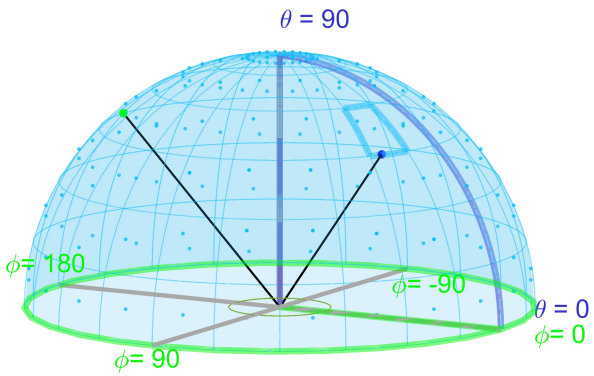


Figure 15: An incoming ray is reflected by the sample and detected at the detection point (in blue), which falls in the highlighted pixel. Here, the angle between the pixel centres $\beta = 15°$.

Or, in terms of θ and ϕ:

$$L(\theta, \phi, \lambda) = \sum_{n(\theta,\phi)} m_{final}(\lambda, C) \tag{38}$$

So, the radiance for a certain angle of incidence is a three dimensional matrix: tabulated spectro-angular radiance.

3.6 From radiance to BRDF

Clearly, the BRDF will depend on the shape of the radiance - but they are not the same quantity. To transform the reflected spectro-angular radiance (L) to the spectro-angular BRDF ($f(\theta, \phi, \lambda)$) -which will be a *tabulated* BRDF (TBRDF) - the radiance has to be normalized [57]. This is done by applying the law of energy conservation.

The law of energy conservation states that the number of photons coming out of the system should be equal to the number of photons entering the system when there are no loss mechanisms. This principle of energy conservation can be mathematically described as follows:

$$\sum_{\phi=0}^{2\pi} \sum_{\theta=0}^{\frac{1}{2}\pi} f(\theta_{dome}, \phi_{dome}, \lambda) cos(\frac{\pi}{2} - \theta_{dome}) sin(\frac{\pi}{2} - \theta_{dome}) d\theta_{dome} d\phi_{dome} = 1 \tag{39}$$

The cosine takes into account that every pixel has a different viewing angle with respect to the centre of the sample, and therefore sees a different amount of radiance. The sine takes into account the tilt of each detection dome pixel.

However, in the detection dome, energy is not always conserved as there is absorption in the system. In the case of a simple horizontal circular sample, energy might be lost if the sample absorbs ($R(\lambda) < 1$ in equation 28). Such a loss mechanism can cause less number of photons to

come out out of the system than were initially coming in. This imbalance is to be accounted for in the BRDF, otherwise all samples would always have an albedo of 1. This effect is accounted for by the spectral albedo ($\rho(\lambda)$):

$$\rho(\lambda) = \frac{E_{out}(\lambda)}{E_{in}(\lambda)} \tag{40}$$

where E_{out} and E_{in} are the exitance (radiant flux leaving a surface per unit area) and incoming spectral irradiance (radiant flux received by a surface per unit area), respectively:

$$E_{out}(\lambda) = \sum_{\phi=0}^{2\pi} \sum_{\theta=0}^{\frac{\pi}{2}} L(\theta, \phi, \lambda) \tag{41}$$

$$E_{in}(\lambda) = \sum_{n=0}^{n} m_{initial}(\lambda, C) \tag{42}$$

In summary, the BRDF is computed as follows:

$$f(\theta_{dome}, \phi_{dome}, \lambda) = \frac{L(\theta, \phi, \lambda)}{\sum_{\phi=0}^{2\pi} \sum_{\theta=0}^{\frac{1}{2}\pi} L(\theta, \phi, \lambda) cos(\frac{\pi}{2} - \theta) sin(\frac{\pi}{2} - \theta) d\theta d\phi} \rho(\lambda) \tag{43}$$

For a given wavelength, the BRDF has the shape of the reflected radiance. The radiance has to be normalised, which is done by the sums in the denominator. To account for energy loss in the BRDF, the first division is multiplied by the spectral albedo. The BRDF as shown in equation 43 can be obtained for every angle of incidence of the light, which makes the BRDF five dimensional: $f(\theta_{source}, \phi_{source}, \theta_{dome}, \phi_{dome}, \lambda)$.

3.7 Calibration of the parameters

The ray tracer software makes use of many parameters, whose values affect the resulting BRDF. This section aims to create insight in the effects of these parameters with as goal to achieve a BRDF which is as accurate as possible. First, the simplest sample is considered: a horizontal Lambertian surface. Then, the parameters are calibrated for grass, modelled by multiple diffuse reflectors.

3.7.1 Calibrating the horizontal Lambertian reflector

To ensure the model is working, the TBRDF of a Lambertian reflector (as illustrated in figure 12a) is measured and compared to the analytical Lambertian. The BRDF of the latter is $\frac{1}{\pi}$ for all directions of the hemisphere (equation 3).

As the model is run for a Lambertian reflector, some simplifications are made. The model is run by simply only generating a horizontal plane in the detection dome, which does not decrease the magnitude of the ray after collision:

$$m_{new} = m_{old} \tag{44}$$

Thereby, the spectral dependence of the TBRDF is taken out entirely. Furthermore, as the reflector is Lambertian, the BRDF does not depend on the angle of the incident light. Consequently, the TBRDF becomes two dimensional as it only depends on θ_{dome} and ϕ_{dome}. Fur-

thermore, as it is known that the BRDF for a Lambertian surface is a constant, it can be deduced that the obtained TBRDF should not vary with θ_{dome} and ϕ_{dome}.

All rays are directed at the origin of the xy-plane to increase calculation speed and such that the variables a_{dome} and a_{sample} do not play a role. Furthermore, the software is run ten times and the average radiance is taken such that local errors are evened out.

Figure 22e shows the BRDF of a horizontal Lambertian surface, evaluated for 100,000 rays.

As the magnitude of the rays never decreases, the energy entering the dome should also leave the dome to fulfil the law of energy conservation. In other words, the integration shown in equation 39 should evaluate to 1 for an ideal diffuse reflector, which has been verified.

Comparison TBRDF to analytical Lambertian: Number of rays and detection dome grid accuracy

Comparing the obtained TBRDF to the analytical Lambertian, the root mean squared error (RMSE) follows:

$$RMSE = \sqrt{\frac{\sum_{\phi=0}^{2\pi} \sum_{\theta=0}^{\frac{\pi}{2}} \left(\frac{1}{\pi} - f(\theta,\phi)\right)^2}{\sum_{\phi=0}^{2\pi} \sum_{\theta=0}^{\frac{\pi}{2}} 1}} \tag{45}$$

The simulation is run for n rays and a detection dome accuracy measured in β, the amount of degrees between detection dome pixel centers. Figure 16 shows how for $\beta = 0.5°$ an increase in the number of rays decreases the RMSE - which is expected as a larger sample size will decrease the error.

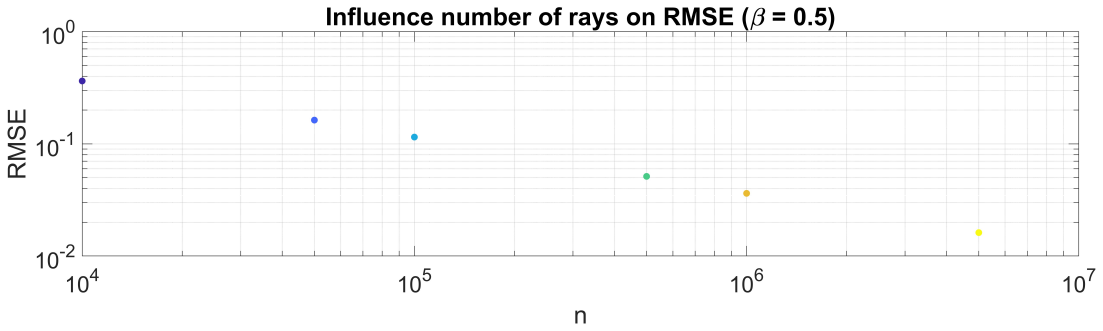


Figure 16: Root mean squared error (RMSE) as function of the number of rays (n) for $\beta = 0.5°$.

Figure 17 shows the results of a simulation where the BRDF has been saved for n from 10^4 to 10^9 and the detection dome pixel size varies $(\beta)^4$. For each value of the number of rays n, there is a minimum at a certain value of the detection dome pixel size parameter β. Two main effects can be seen:

1. As the pixel size increases, the RMSE decreases as the errors on individual pixels are averaged;
2. As the pixel size increases further, the RMSE increases. Because of the large pixel size, the detection dome shape is made up of less faces and is therefore different from a hemisphere - in that case, measuring diffuse reflection becomes less accurate.

[4]The chosen values of β are multiples of $\beta = 0.5°$ such that the TBRDF for larger pixel sizes (e.g.$\beta = 1°$, $\beta = 1.5°$ etc.) can be computed after running the ray tracer for only one pixel size. Some multiples of $\beta = 0.5°$ are skipped because the detection dome hemisphere cannot be divided into equal parts anymore (e.g., $\beta = 3.5°$ does not fit an integer number of times in the azimuth (360°) nor in the elevation angle (90°))

The first effect is more prominent for a low number of rays, whereas the second effect is more prominent for a larger number of rays. This can be intuitively explained: when there are more rays, the error per pixel will be smaller already. Therefore, to minimise the RMSE, a high number of rays needs to be detected by a smaller pixel size compared to a low number of rays.

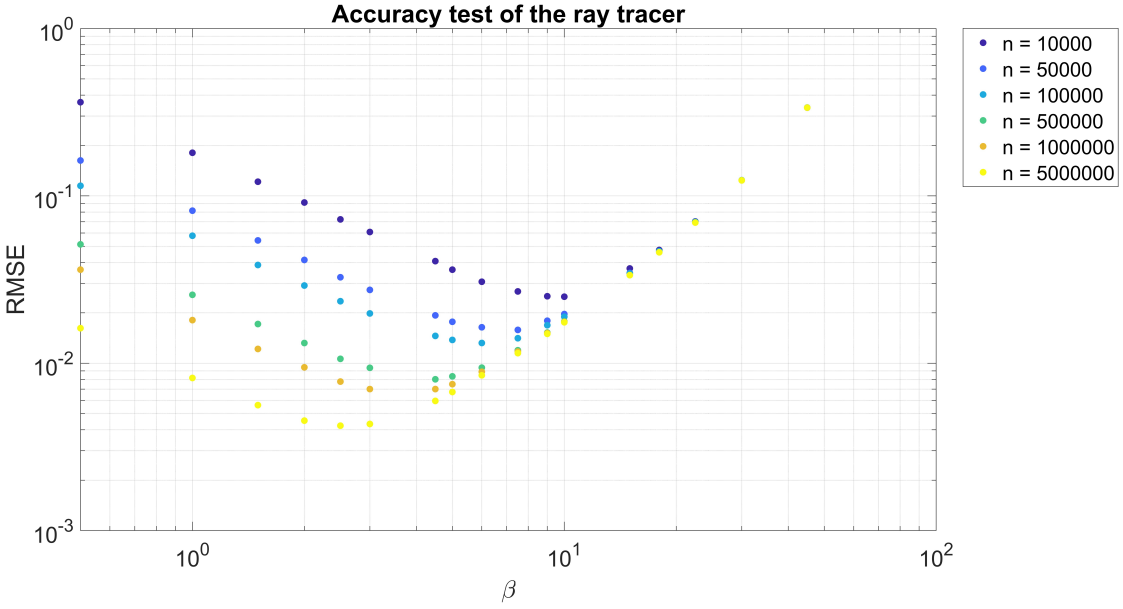


Figure 17: Root mean squared error (RMSE) as function of the detection dome accuracy β in degrees for various number of rays (n).

The ray tracer runs by a *for* loop over the number of rays. Therefore, the running time proportionally increases with the number of rays as can be seen from figure 18. Furthermore, a high value of β means a low number of pixels, which reduces the memory requirements.

Comparison TBRDF to analytical Lambertian: Measured area relative to the dome
In the section above, all incoming n rays are shot towards the middle of the detection dome. However, if they were to be shot to a point at a certain radius away from the centre of the dome, the rays don't fall equally distributed on the grid anymore and the resulting BRDF gets distorted. Therefore, the rays should be aimed close to the centre of the detection dome. The impact of the size of the detection dome with respect to the size of the illuminated area has been evaluated. The illuminated area is set to be the same as the sample size, and the ratio between the radius of the detection dome and the radius of the sample a_{dome} is varied.

Figure 19 shows the relation between the RMSE and a_{dome}. The plot is for $n = 10^6$ and $\beta = 3°$ (as deduced from figure 17). It can be seen that the error is minimised when the detection

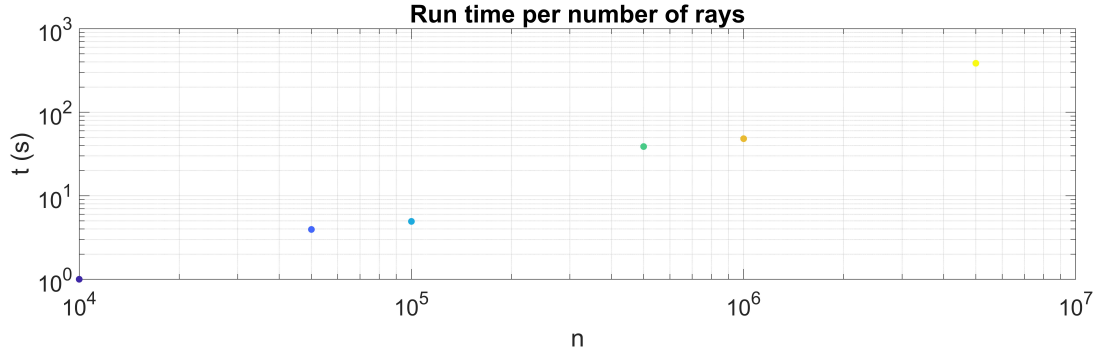


Figure 18: The run time t in seconds against the number of rays for $\beta = 0.5°$.

dome is larger than one thousand times the size of the sample.

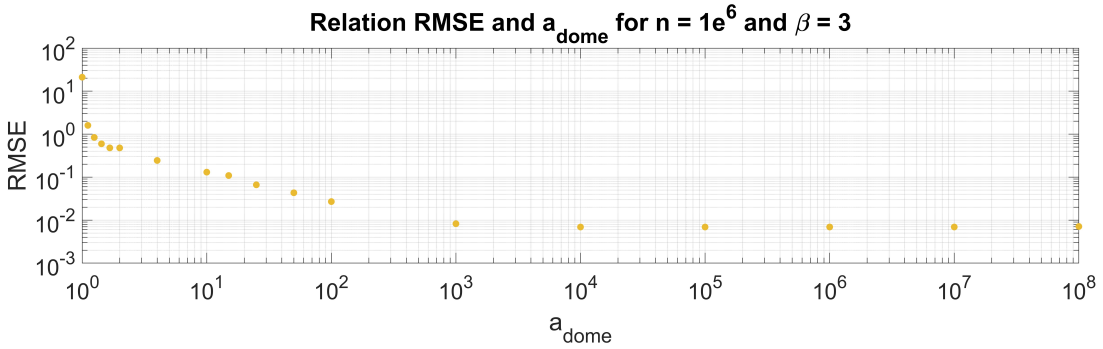


Figure 19: Root mean squared error (RMSE) as function of a_{dome}, the ratio between the radius of the detection dome and the sample (which is the spot size as here, $a_{illuminated} = 1$. Case for $n = 10^6$ and $\beta = 3°$.

So far, the incident rays have been aimed at height $h = 0$. When the sample height is increased and the rays are aimed higher ($h > 0$), the lower pixels of the detection dome will receive less radiance. To analyse this effect, the parameter q is introduced. This parameter is the ratio between the dome size and the height:

$$q = \frac{r_{dome}}{h} \tag{46}$$

Note that $q > 1$ to make physical sense. Figure 20 shows the resulting RMSE when shooting the rays at $(x, y, z) = (0, 0, h)$ and varying h. At $q = 1$, the rays are shot at the top of the detection dome. The error is minimised when the dome is about a thousand times larger than the height at which the rays are shot ($q = 10^3$).

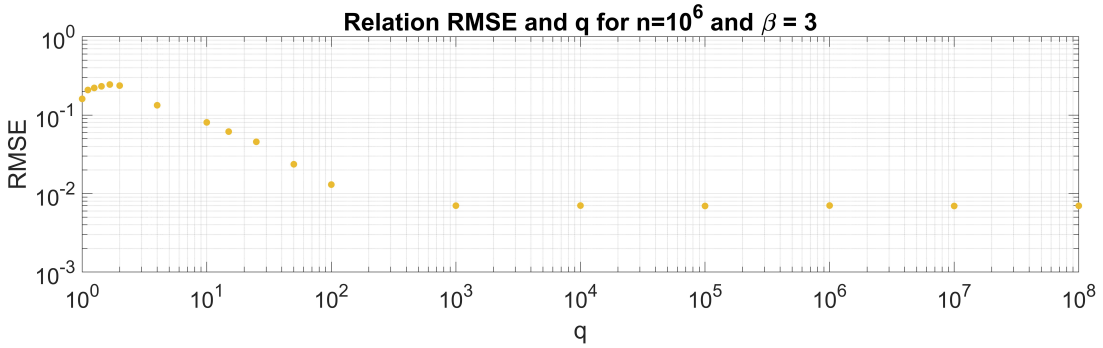
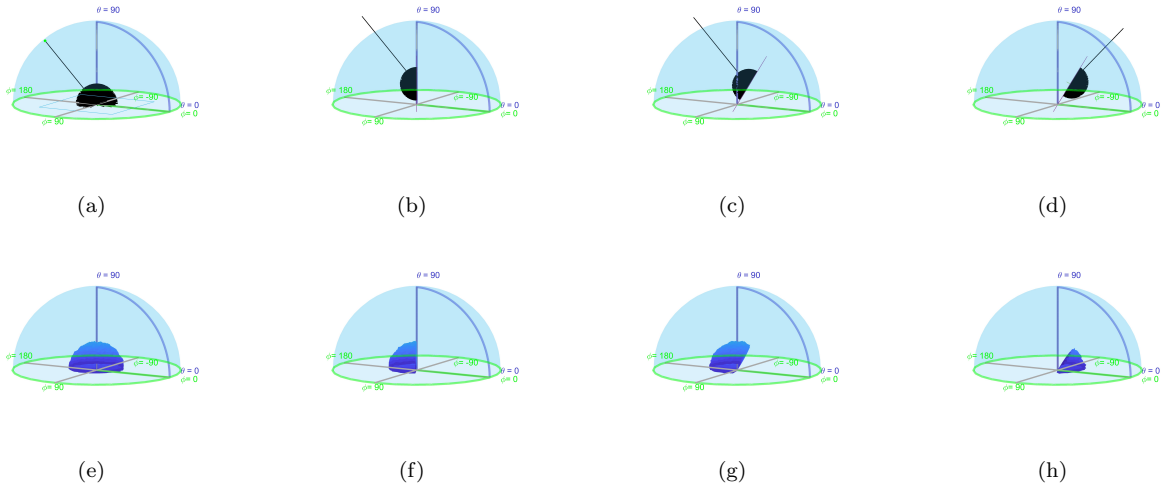


Figure 20: RMSE as function of q, the ratio between the radius of the detection dome and the height h that the rays are shot at. Case for $n = 1,000,000$ and $\beta = 3°$.

Since a larger detection dome is not computationally more expensive, a detection dome is chosen with $a_{dome} = 10^8$. Since a_{dome} determines r_{dome}, this means that h may not exceed $h = 10^5$ to retain accuracy.

Other orientations of a Lambertian reflector

The TBRDF of differently oriented surfaces is simulated like the Lambertian surface. A Lambertian surface should always give an equal radiance - and since the shape of the BRDF depends on the radiance, this can be checked by looking at the BRDF. Figure 21 shows four different orientations of a Lambertian surface, and the resulting shape of the BRDF. Note that here, $a_{dome} = 10^8$ is used. It can be seen that the BRDF is equally large to all directions that are available, thereby validating that the BRDF of a Lambertian reflector in any orientation can be simulated accurately (i.e. with the limitations according to figure 17).



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 21: Possible reflection directions for a horizontal Lambertian (a), vertical (b) and tilted Lambertian surface (c and d) and their corresponding BRDF (e,f,g, and h, respectively). The simulation has been run for $n = 10^6$ and $\beta = 6$.

3.7.2 Calibrating the grass reflector

The simplest grass sample consists of a circular surface, which is the ground, and rectangular vertical surfaces which represent the grass. To focus only on the effect of the grass geometry, the ground is assumed to be a blackbody. Each grass blade reflects diffusely, like a Lambertian reflector - but for the grass, spectral reflectivity $R(\lambda)$ is taken into account.

The simulations are run for ten different randomly oriented samples such that the local error of a single sample is reduced.

Considerations for the height h

When the sample is changed for grass modelled as explained above, other parameters become important: $a_{illuminated}$ and $r_{illuminated}$. These parameters determine in part the realism of the obtained BRDF.

The goal of the ray tracer is to produce a BRDF for grass as if it were a large patch. However, in the ray tracer, there is only a circular sample of grass - it has distinct boundaries. Depending on the height that the rays are shot at, some rays will hit the grass at the edge of the sample - especially for low elevation angles. These blades act like a vertical Lambertian (see figures 22b and 22f) and therefore influence to great extent the reflectance and thus the BRDF. This influence has to be minimized. This problem can be solved by aiming the rays at height $h = z_{blade}$ - now, by definition, the light will not hit the blade closest to the edge of the sample as much from the side.

However, aiming the rays at $h = z_{blade}$ also introduces constraints on the height of the blades. If the blades are very high, the RMSE is distorted (see figure 20). Furthermore, the valid angles for the source (the sun) are reduced. Consider a vertical Lambertian surface with a height 5 cm, which stands in the middle of the detection dome with a radius of 10 cm (figure 22b). If the source is at elevation angle $\theta_{source} = 0$ and the ray is shot towards $h = 5cm$, then the ray direction has a positive z component: the ray is shot towards the sky instead of towards the ground. Place some more grass blades in the sample, and it can be deduced that the initial rays from the sun will have to go through grass blades to be able to follow their path towards the sky. Clearly, this is not a physically possible situation.

With the help of geometry, the problem can be solved. The elevation angle of the source should

be defined such that rays are never directed towards the sky. So, the ray direction is either horizontal or aimed down. Related to this problem is that light which is shot at very low elevation angles, will be detected on the opposite site of the detection dome. To avoid both problems, the elevation angle of the source should always be higher than $\theta_{source,min}$:

$$\theta_{source,min} = arctan\left(\frac{h(1+\frac{r_{dome}+r_{illuminated}}{r_{dome}-r_{illuminated}})}{r_{dome}}\right) \tag{47}$$

Only at angles $\theta_{source} > \theta_{source,min}$ the ray tracer will produce a valid result. If a_{dome} is small, $\theta_{source,min}$ will be large and the ray tracer can be used for less incident elevation angles. Again, a large detection dome thus proves to be useful as it reduces $\theta_{source,min}$. For example, aiming rays at a sample with $r_{illuminated} = 2.5cm$ at a height $h = z_{blade} = 8cm$ and a detection dome of $r_{dome} = 5000km$ gives $\theta_{source,min} = 1.83 \cdot 10^{-6°}$. This angle should be computed to know the lower limit at which the model is valid. The Matlab code throws an error when angles of incidence are chosen which force the ray direction towards the sky.

Another option is to place the sample *under* the detection dome by height h. This configuration carries the inherent assumption that there is a blackbody at the edge of the sample, as reflection coming from the side of the sample is not measured and considered to be lost. For this reason, this configuration was not chosen, although it might be worth re-evaluating this option in the future as it avoids the error introduced by $\theta_{source,min}$.

Considerations for the blade density
The number of blades hit by the beam and their orientation matters for the obtained BRDF. Consider a sample illuminated by a light beam so small that all light falls on one grass blade in the middle. The orientation of this grass blade has a profound impact on the resulting BRDF - in the initial intersection of all rays with this blade, there is only one half-space of possible directions (see figure 12b). If the beam were to be aimed at a larger area of the sample, there are initially more half-spaces of directions to choose from. So, the number of blades that are illuminated influences the BRDF. In this thesis, I take $r_{illuminated} = 2.5cm$ which consists of 45 blades - that gives a blade density of approximately 22,918 blades per square meter, based on artificial grass [58].

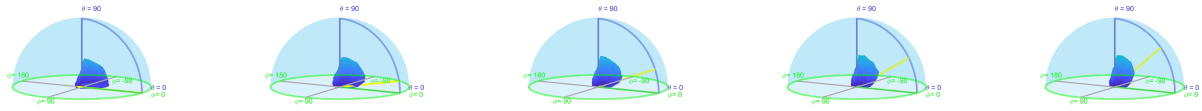
Considerations for $a_{illuminated}$
Consider a grass sample which is fully illuminated. The light rays hitting the grass blades at the edge have an equal chance to be reflected upwards (get a positive z-component to $\vec{d_{new}}$) and to reflect towards the ground (a negative z-component). If the sample size is increased while the illuminated area stays the same, upward directed rays which would otherwise have been detected have an increased probability of hitting an extra blade. The same is true for downward directed rays.

To analyse if and how these effects affect the albedo, the ray tracing model has been run by shooting $n = 10^5$ rays on the sample. The sample consists of a blackbody ground and rectangular grass blades with height $z_{blade} = h = 4cm$ and width $w_{blade} = 0.5cm$. The accuracy of the detection dome is $\beta = 10°$. The size of the illuminated area (determined by $r_{illuminated}$) is kept constant, while the sample size (determined by r_{sample}) is increased. In other words, $a_{illuminated}$ is changed, but in such a way that the spot size will stay the same. To analyse the effect of decreasing $a_{illuminated}$, the model is run for four situations:

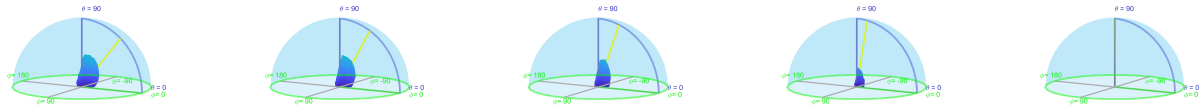
- $r_{illuminated} = 2.5$ cm, $r_{sample} = 2.5$ cm, $a_{illuminated} = 1$
- $r_{illuminated} = 2.5$ cm, $r_{sample} \approx 3.3$ cm, $a_{illuminated} = 0.75$

- $r_{illuminated} = 2.5$ cm, $r_{sample} = 5$ cm, $a_{illuminated} = 0.5$
- $r_{illuminated} = 2.5$ cm, $r_{sample} = 10$ cm, $a_{illuminated} = 0.25$

Figure 22 presents the BRDF of grass for ten elevation angles of the source. There is no reflection at $\theta_{source} = 90°$, as the rays directly hit the ground which is a blackbody and absorbs all the light.



(a) $\theta_{source} = 0.1°$ (b) $\theta_{source} = 10°$ (c) $\theta_{source} = 20°$ (d) $\theta_{source} = 30°$ (e) $\theta_{source} = 40°$



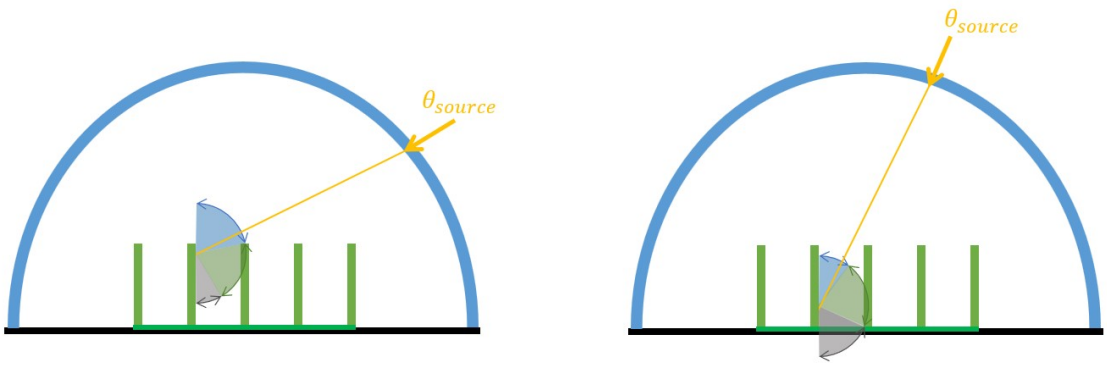
(f) $\theta_{source} = 50°$ (g) $\theta_{source} = 60°$ (h) $\theta_{source} = 70°$ (i) $\theta_{source} = 80°$ (j) $\theta_{source} = 90°$

Figure 22: The BRDF of vertical grass for ten elevation angles of the source (θ_{source}). Light (yellow line) is incident from $\phi_{source} = 0°$. Here, $r_{illuminated} = 2.5cm$, $r_{sample} = 5cm$ and $a_{illuminated} = 0.5$.

Figure 25 shows the value for the albedo (ρ) integrated over the detection dome (θ_{dome} and ϕ_{dome}) for ten angles of incidence for each of the four situations for one wavelength ($\lambda = 900$ nm). Overall, the albedo for this wavelength varies between $\rho = 0.064$ and $\rho = 0.294$. Given that upon first intersection, a ray at $\lambda = 900$ nm reflects 50.76% of the incoming light (see figure 13), the albedo can be maximally 50.76%. Indeed it can be seen in figure 25 that the albedo stays well below this limit.

In figure 25, compare the albedo for different elevation angles when the entire sample is illuminated, i.e. for $r_{illuminated} = r_{sample} = 2.5cm$. The albedo is higher when the angle of incidence is lower. Light coming in from a low elevation angle hits the grass blades at the top of the blade (see figure 23a), whereas light that comes in at a higher elevation angle penetrates deeper into the grass (see figure 23b). Light hitting the grass deeper has less chance to reflect towards the sky, as illustrated by the blue arrow in figure 23b. Therefore, the simulated albedo of grass is lower at high elevation angles of incidence. This effect will from hereon be called *higher elevation angle - lower albedo.*

There are two main effects when the sample size increases while the illuminated area stays constant. Consider again a fully illuminated sample ($r_{illuminated} = r_{sample} = 2.5cm$). After intersection with a grass blade, there is a 50% probability that the new ray direction will have a negative z-component. The reflected ray is travelling down and is absorbed by the non-sample area. When the sample size is increased while the illuminated area stays the same, essentially an extra ring of blades is added around the illuminated area. A ray which would have otherwise been absorbed by the non-sample area, now has a chance to hit the grass blades in this ring. Therefore, these rays get a new opportunity to be reflected towards the detection dome. So, as the sample size increases, the albedo increases. For example, when the sample is fully illuminated ($r_{illuminated} = r_{sample} = 2.5cm$), the albedo for $\theta_{source} = 50°$ is $\rho_{\lambda=900nm,\theta_{source}=50°,a_{illuminated}=1} = 0.135$, whereas when the sample increases two times in radius, $\rho_{\lambda=900nm,\theta_{source}=50°,a_{illuminated}=0.5} = 0.179$. When the elevation angle of the source is low, the first intersection of the ray with the blade will take place relatively on the top of the blade (see

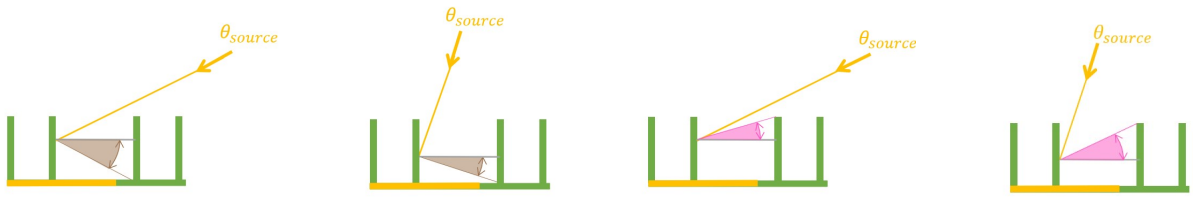


(a) Low elevation angle.

(b) High elevation angle.

Figure 23: Two dimensional illustration of how elevation angle (θ_{source}) influences the chance to reflect towards the elevation dome. The incoming ray (yellow) is shot from a certain angle of incidence on the grass blades, in green. The ray then chooses a new direction. The blue angle represents the directions which will be detected at the detection dome. The green angle represents the directions for which the light will hit another grass blade. The grey angle represents the directions for which the ray is aimed at the ground. The angles are an indication of the probability of this event happening. The figures show how changing (θ_{source}) influences these probabilities.

figure 23a). Therefore, a reflected ray travelling towards the ground from the top of the blade (see figure 24a) has a larger probability of intersecting with a grass blade than a ray which departed from the bottom of the blade (see figure 24b). Consequently, the increase in albedo is more prominent at low elevation angles (see figure 25).



(a) Downward ray: Low elevation angle.

(b) Downward ray: High elevation angle.

(c) Upward ray: Low elevation angle.

(d) Upward ray: High elevation angle.

Figure 24: Two dimensional illustration of how elevation angle (θ_{source}) influences the probability for a downward ray to intersect with a grass blade (represented by the brown arrow) and the probability of an upward ray to intersect with a grass blade (represented by the pink arrow). The yellow line represents the illuminated area.

Secondly, after intersection with a grass blade, there is also a 50% probability that the new ray direction will have a positive z-component. When the sample size has increased, these rays have an increased probability to hit a grass blade. Hitting yet another blade means that the magnitude of the ray decreases. This effect is most present when light is incident from a high elevation angle, as for light penetrating deeper into the grass the probability that an upward directed ray hits another grass blade (see figure 24d) is larger than for rays hitting the blades near the top (see figure 24c) .

These two effects counteract each other, and as a result, it can be seen in figure 25 that for every elevation angle, the sample size at which the albedo is at its maximum is different. For example, for $\theta_{source} = 80°$, the figure shows that the maximum albedo for this angle is obtained at $r_{illuminated} = 2.5$cm, $r_{sample} \approx 3.3$ cm, $a_{illuminated} = 0.75$. For $\theta_{source} = 20°$, the maximum albedo is not reached yet at $r_{illuminated} = 2.5$ cm, $r_{sample} = 10.0$ cm, $a_{illuminated} = 0.25$. Each elevation angle thus has a sample size related to it at which its albedo is at its maximum.

It is expected that the albedo will converge for a large sample size. However, the convergence

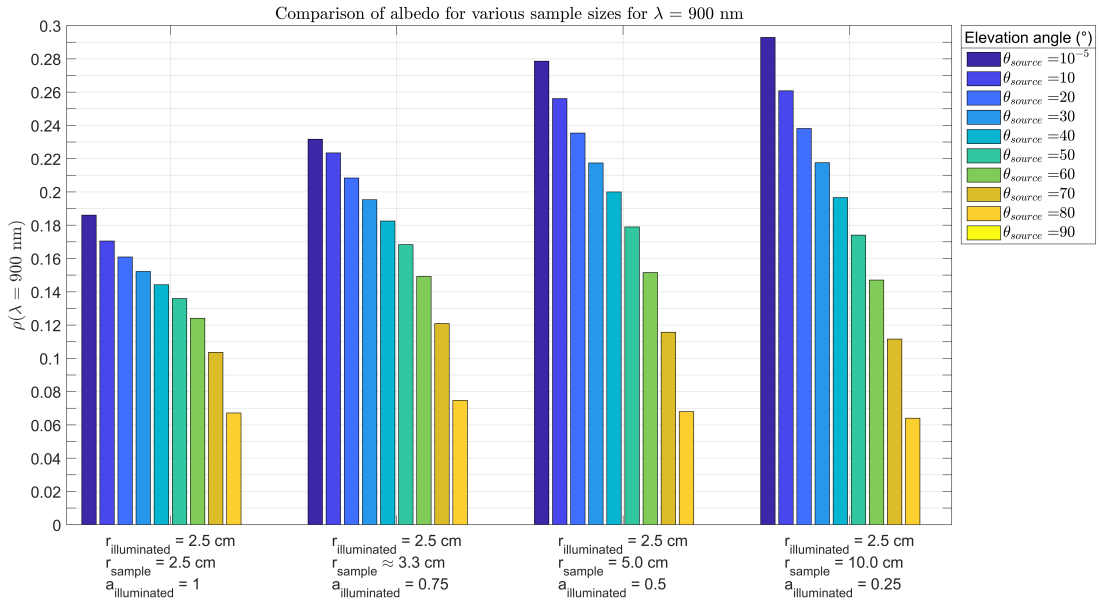


Figure 25: The albedo (ρ) at wavelength $\lambda = 900$nm for four values of the ratio of the illuminated radius over the sample radius ($a_{illuminated}$) and ten elevation angles (θ_{source}). The illuminated area is kept the same, but the sample around the area increases from $r_{sample} = 2.5cm$ (left case) to $r_{sample} = 10cm$ (right case). Note that the albedo at $\theta_{source} = 90°$ is zero in all four cases.

cannot be clearly seen from figure 25 - likely the sample size has to be increased even further. Increasing $a_{illuminated}$ comes with a computational cost (see figure 26), as the number of blades on the sample has to increase to retain the blade density. Therefore, in the continuation of the report, $a_{illuminated} = 0.5$ will be used.

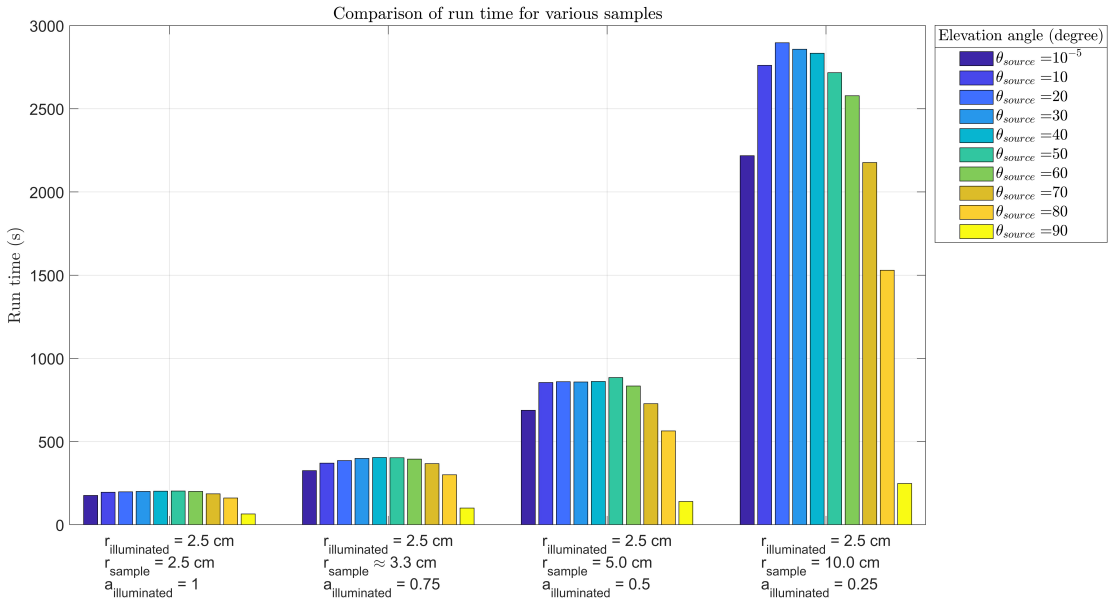


Figure 26: The average run time of the ten runs in seconds for four values of $a_{illuminated}$ and ten elevation angles (θ_{source}).

3.8 Usage of the ray tracing model

In principle, the described ray tracing software can be used to generate a BRDF for any position of the sun. Considering that the sun's position varies a lot throughout a day and the year, running the ray tracer software for each and every possible position of the sun is computationally expensive. Computational time can be saved by simply checking which positions the sun takes in the sky: there are many combinations of ϕ_{source} and θ_{source} that the sun will never be in for a certain location when modelling direct light. For example, for the city of Aasen in the Germany, there is a $\theta_{source,max}$ elevation angle in which the sun can be. This restricts the number of positions to compute the BRDF for. When grass is assumed to be azimuthal symmetric on the macro scale, after retrieving the BRDF for $\phi_{source} = 0°$ and multiple values of θ_{source}, the BRDF data can be simply rotated to retrieve the BRDF data for the desired ϕ_{source}. Thus, a library containing the BRDF at $\phi_{source} = 0°$ and θ_{source} between $\theta_{source,min}$ and $\theta_{source,max}$ is enough to be able to model the BRDF at every possible position of the sun at a given location.

From the library of BRDF's, the BRDF corresponding to a certain θ_{source} and ϕ_{source} can be retrieved and used in the reverse ray tracing code of Pal [9] to compute the energy yield.

4 Results and discussion

In this section, the BRDF that is obtained from the developed ray tracing model will be compared to experimental data. Furthermore, it will be demonstrated that the program can be used to compute the influence of different grass geometries on the yield of a bifacial solar panel. To this end, a case study of a vertical bifacial solar farm in Aasen, Germany, is performed.

4.1 Introduction grass samples

The following grass samples are investigated (see figure 27):

- **Sample 1: Vertical grass 4 cm**
  Parameters as displayed in table 1.
- **Sample 2: Vertical grass 8 cm**
  Parameters as displayed in table 1, but:
  - Blade length is $z_{blade} = 8cm$
- **Sample 3: Vertical grass 12 cm**
  Parameters as displayed in table 1, but:
  - Blade length is $z_{blade} = 12cm$
- **Sample 4: Varying height**
  Parameters as displayed in table 1, but:
  - The type of blade is changed to "VerticalVaryingHeight"
  - Blade length z_{blade} is a random number between $z_{blade,min} = 4cm$ and $z_{blade,max} = 12cm$ as follows:

$$z_{blade} = z_{blade,min} + (z_{blade,max} - z_{blade,min})\xi_8 \tag{48}$$
$$\xi_8 \in [0, 1] \tag{49}$$

- **Sample 5: Tilted top 20° bottom 4cm top 4cm**
  Parameters as displayed in table 1, but:
  - The type of blade is changed to "VerticalAndTiltedTop"
  - $z_{blade,top} = 4cm$
  - An extra variable, the blade tilt, is introduced. This variable gives a range for the minimum (τ_{min}) and maximum angle (τ_{max}) that the tilt may have as seen from the xy-plane normal. Here, $\tau_{min} = -20°$, $\tau_{max} = 20°$. The blade tops are assigned a random tilt angle τ between these two angles:

$$\tau = \tau_{min} + (\tau_{max} - \tau_{min})\xi_9 \tag{50}$$
$$\xi_9 \in [0, 1] \tag{51}$$

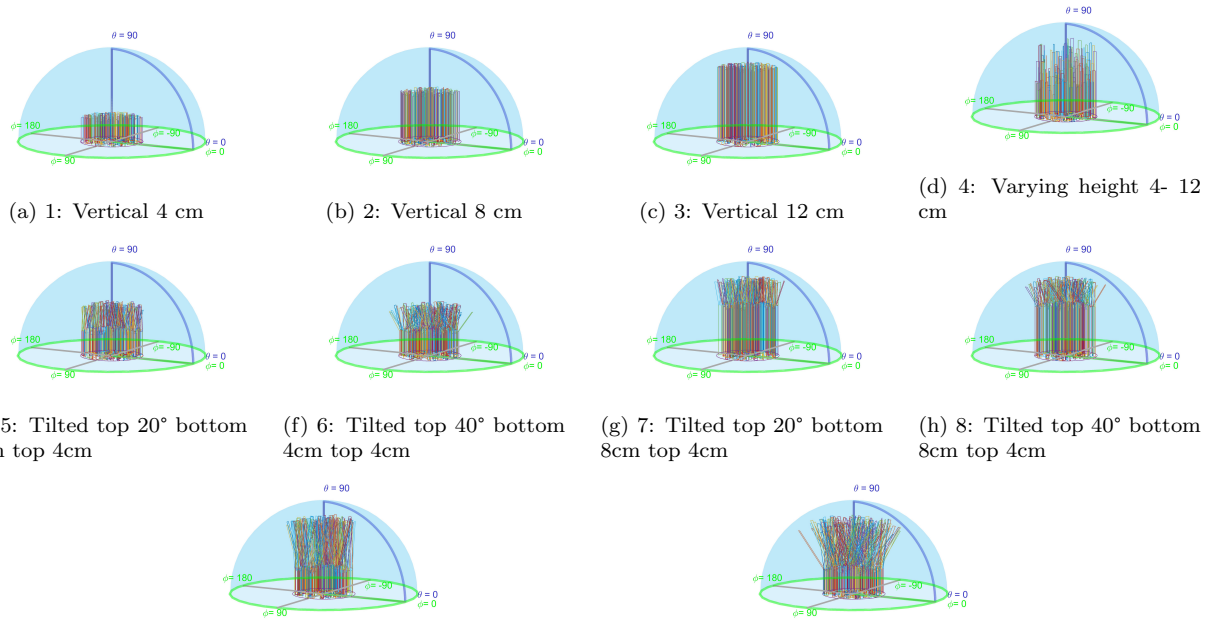
- **Sample 6: Tilted top 40° bottom 4cm top 4cm**
  Parameters as displayed in table 1 and with corrections as for sample 5, but:
  - $\tau_{min} = -40°$, $\tau_{max} = 40°$.
- **Sample 7: Tilted top 20° bottom 8cm top 4cm**
  Parameters as displayed in table 1 and with corrections as for sample 5, but:

- $z_{blade} = 8cm$

- **Sample 8: Tilted top** $40°$ **bottom 8cm top 4cm**
  Parameters as displayed in table 1 and with corrections as for sample 7, but:
  - $\tau_{min} = -40°$, $\tau_{max} = 40°$.

- **Sample 9: Tilted top** $20°$ **bottom 4cm top 8cm**
  Parameters as displayed in table 1 and with corrections as for sample 5, but:
  - $z_{blade,top} = 8cm$

- **Sample 10: Tilted top** $40°$ **bottom 4cm top 8cm**
  Parameters as displayed in table 1 and with corrections as for sample 9, but:
  - $\tau_{min} = -40°$, $\tau_{max} = 40°$.



(a) 1: Vertical 4 cm

(b) 2: Vertical 8 cm

(c) 3: Vertical 12 cm

(d) 4: Varying height 4- 12 cm

(e) 5: Tilted top 20° bottom 4cm top 4cm

(f) 6: Tilted top 40° bottom 4cm top 4cm

(g) 7: Tilted top 20° bottom 8cm top 4cm

(h) 8: Tilted top 40° bottom 8cm top 4cm

(i) 9: Tilted top 20° bottom 4cm top 8cm

(j) 10: Tilted top 40° bottom 4cm top 8cm

Figure 27: The analysed samples.

As the model is run ten times for every sample, the individual peculiarities introduced by all parameters determined by random numbers (e.g. low probability events) are evened out.

The latter six samples consisting are considered to be more true to the real morphology of grass than the first four. Nevertheless, the first four grass sample help to explain the effects seen for the other six grass samples.

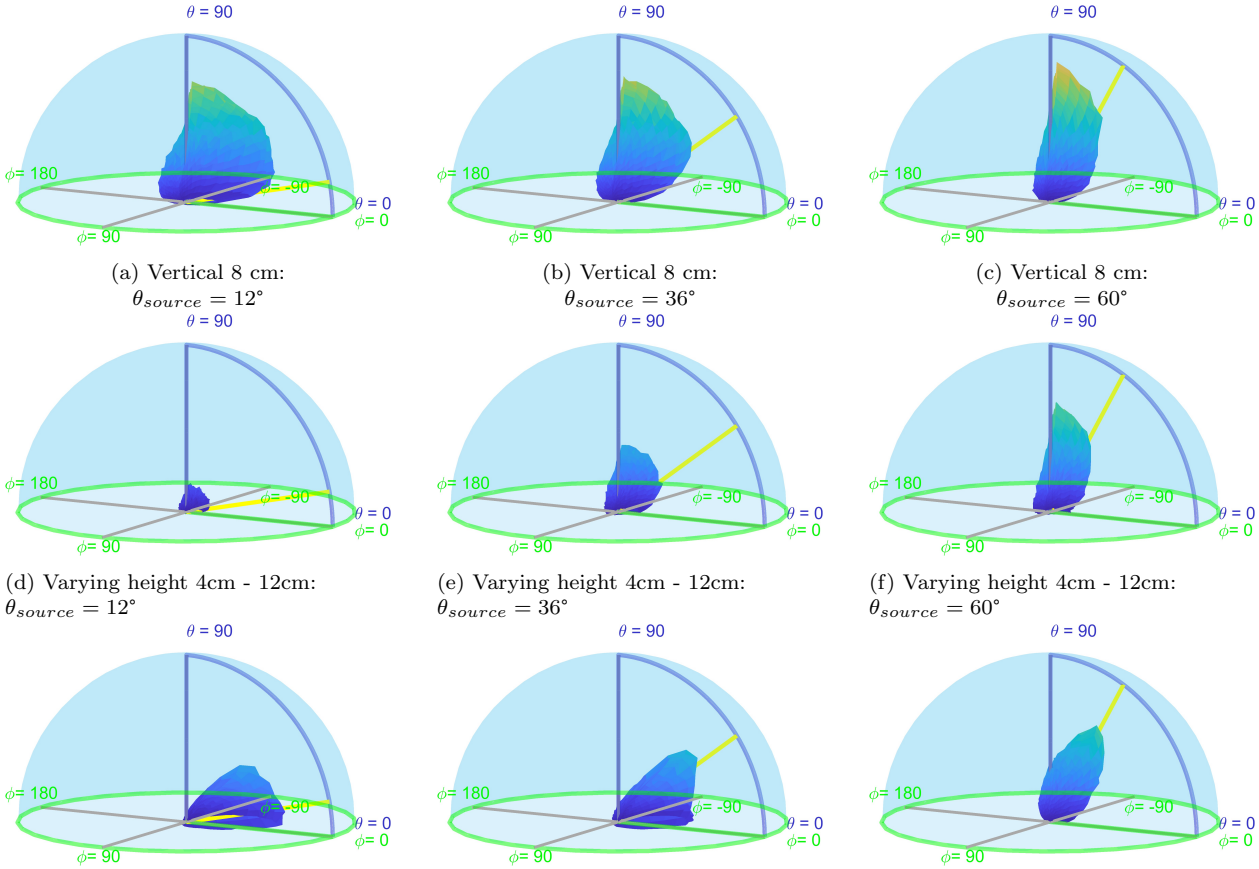
Table 1: The parameters used to run the ray tracing model.

	Parameter	Symbol in report	Value	Notes
Sample	Radius	r_{sample}	5 cm	
	Reflectance ground	$R_{ground}(\lambda)$	0	Blackbody
	Number of blades	-	180	
	Type of blades	-	"VerticalSameHeight"	All blades have a height z_{blade}
	Blade width	w_{blade}	0.5 cm	
	Blade length	z_{blade}	4 cm	
	Reflectance grass	$R_{grass}(\lambda)$	Values from [6]	
Detection	Accuracy dome grid	β	6°	
	Size dome relative to sample	a_{dome}	10^8	$r_{dome} = 5000$ km
Light	Azimuth angle source	ϕ_{source}	0°	
	Elevation angle source	θ_{source}	Varied	$[\theta_{source,min}, \beta : \beta : \theta_{source,max}]$°
	Wavelengths	λ	[300 : 10 : 1100] nm	
	Number of rays	n	10^5	
	Size illuminated area relative to sample	$a_{illuminated}$	0.5	$r_{illuminated} = 2.5$ cm
Other	Number of runs	-	10	

4.2 Comparison BRDF and albedo for various types of grass

In this section, the spectro-angular albedo of the different samples are analysed. Furthermore, the albedo variation from sample to sample is investigated.

4.2.1 An angular dependent BRDF



(a) Vertical 8 cm: $\theta_{source} = 12°$

(b) Vertical 8 cm: $\theta_{source} = 36°$

(c) Vertical 8 cm: $\theta_{source} = 60°$

(d) Varying height 4cm - 12cm: $\theta_{source} = 12°$

(e) Varying height 4cm - 12cm: $\theta_{source} = 36°$

(f) Varying height 4cm - 12cm: $\theta_{source} = 60°$

(g) Tilted top 40° bottom 4cm top 8 cm: $\theta_{source} = 12°$

(h) Tilted top 40° bottom 4cm top 8 cm: $\theta_{source} = 36°$

(i) Tilted top 40° bottom 4cm top 8 cm: $\theta_{source} = 60°$

Figure 28: The BRDF of vertical grass for three values of the elevation angle θ_{source}. Figures a to c belong to sample 2 "Vertical 8 cm", d to f to sample 4 "Varying height 4cm - 12cm" and figures g to i to sample 10 "Tilted top 40° bottom 4cm top 8 cm".

Figure 28 shows the angular BRDF for $\lambda = 900nm$ for several samples. All values of the BRDF for the ten samples can be found in appendix A. From these plots, there appears to be azimuthal retroreflection for all ten samples: the light reflects in the azimuthal direction from which it came ($\phi_{dome} = \phi_{source} = 0°$).

For the samples consisting only of vertical blades (i.e. "Vertical grass 8 cm" and "Varying height 4cm - 12 cm"), quite some light is reflected straight up into the sky: there is a peak at $\theta_{dome} = 90°$. This can both be seen in subfigures a-f in figure 28 and in figures 43,44,45 and 46 in appendix A. This behaviour can be attributed to the vertical geometry of the grass. Upon first incidence, the light can only escape in the available directions as displayed in figures 23b and 23a. Furthermore, the possible escape directions at $\theta_{dome} = 90°$ after first incidence overlap with the possible escape directions after hitting a blade on the other side again, as is illustrated in figure 29. These effects result in a peak around $\theta_{dome} = 90°$.

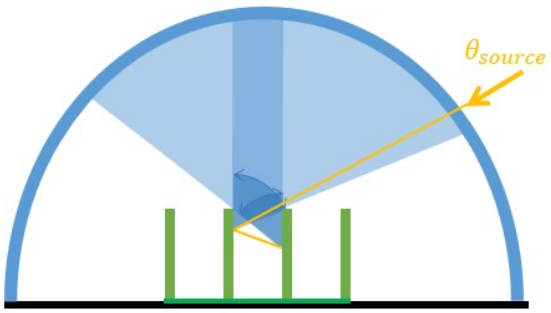


Figure 29: Two dimensional sketch of the detection dome (blue hemisphere) and the sample. When light is reflected more than once, the probability that the new ray direction is towards the detection dome (blue arrow) overlaps, resulting in an overall higher chance to reflect towards θ_{source}.

Retroreflection over the elevation angle (i.e. the light reflects in the elevation angle direction from which it came, $\theta_{dome} = \theta_{source}$) only occurs when there is a tilted top involved (see figures 47,48, 49,50, 51 and 52 in appendix A).

In figure 30, the shape[5] of the obtained retroreflection lobe for "Tilted top 40° bottom 4cm top 8cm" is compared to experimentally obtained data for the wavelength $\lambda = 550nm$. The experimental data is obtained by goniometer measurements on grass performed by Jelle Westerhof. The figure shows the cross-section of the retroreflection lobe: a cut through the detection dome from $\phi_{dome} = 0°$ to $\theta_{dome} = 180°$. As the simulation has been run with an accuracy of the detection dome of $\beta = 6°$, the elevation angles that are compared are not exactly the same.

For elevation angles $\theta_{source} = 30°$, the shape of the simulated BRDF does not agree with the measured reflection for $\theta_{dome} < 15°$. Interestingly, the measurements do not show this effect. For the highest elevation angles shown in figure 30, the shape seems to agree - a lobe is obtained that is not 'attached' to the ground.

[5]Comparing the numerical values is challenging, as the detector in the goniometer has another viewing angle on the sample than is the case in the simulation.

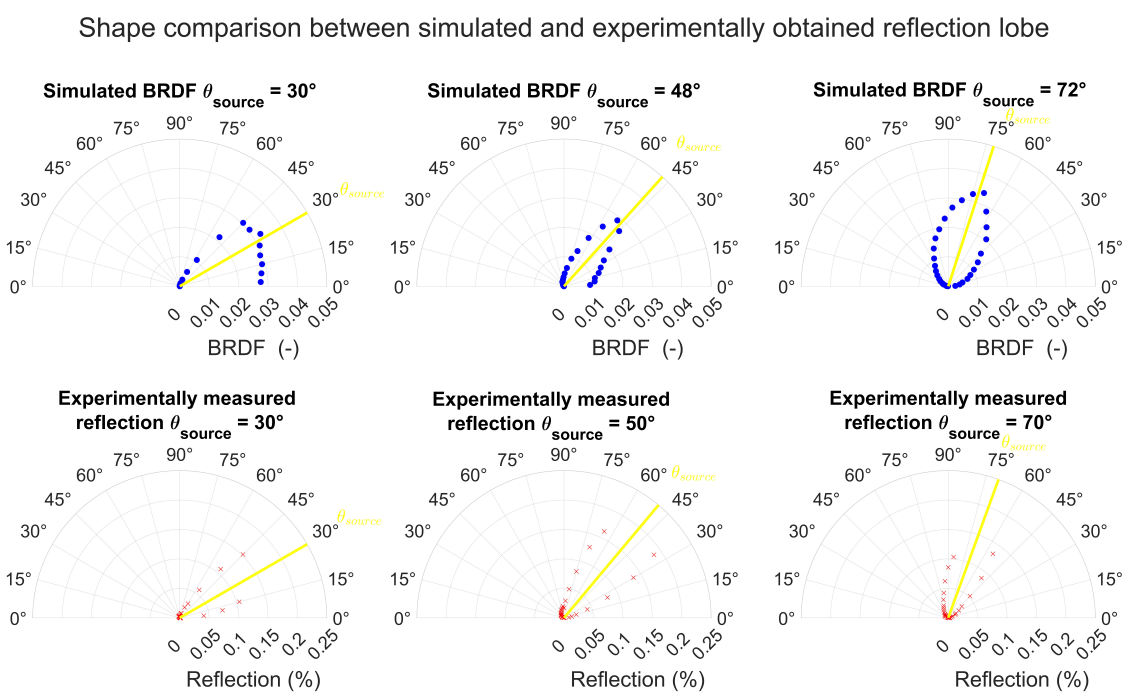


Figure 30: Comparison of the shape of the reflection. The yellow line represents the elevation angle of the incident light (θ_{source}). The top three figures show the simulated BRDF of "Tilted top 40° bottom 4 cm top 8 cm" for three elevation angles of the source (θ_{source}). The bottom three figures show the measured reflection of grass. Measurements performed with a goniometer by Jelle Westerhof (unpublished).

4.2.2 A spectrally dependent BRDF

Figure 31 shows the spectral albedo of one grass sample "Tilted top 40° bottom 4 cm top 8 cm". The figure shows the spectral albedo for seven angle of incidences of the source (θ_{source}). The albedo varies over the wavelength in accordance to Russell et al. [6] (crosses in figure 31), as expected from applying the reflectance R in equation 28.

The albedo for all angles is lower than the reflectance. This is to be expected, as in the ray tracing model, rays are absorbed by the soil or multiplied multiple times with the reflectance by intersecting with multiple blades. Both these effects decreases the radiance on the detection dome and thus the albedo.

4.2.3 BRDF dependence on blade geometry

Figure 32 shows the albedo (ρ) at $\lambda = 900$ nm for thirteen elevation angles θ_{source} of all ten samples. The figure shows multiple trends in albedo, which will be discussed in this section.

Grass length

In the first three vertical samples, the *higher elevation angle - lower albedo* effect as explained by figures 23b and 23a can be clearly recognized, which agrees with the observations of Roosjen et al. [20].

As the vertical grass becomes taller, the albedo increases for every angle of incidence. For example, for an angle of incidence of $\theta_{source} = 60°$, the albedo of vertical grass of 4 cm tall is about $\rho = 0.165$ for $\lambda = 900$ nm. The albedo for 8 and 12 cm tall grass for the same parameters are higher: $\rho = 0.176$ and $\rho = 0.179$, respectively. When the grass is taller, the ray hits the blade relatively higher on the blade - recall how the height h that the rays are aimed at is related to the vertical height of the blade z_{blade}. A smaller portion of the ray directions is

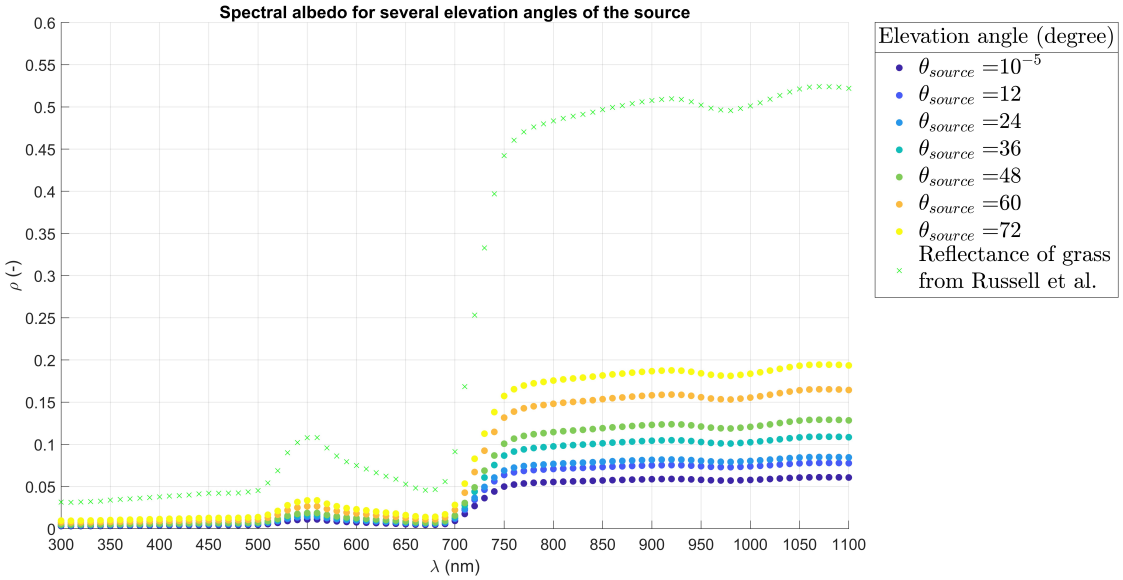


Figure 31: The albedo (ρ) as a function of the wavelength (λ) from $\lambda = 300$ nm to $\lambda = 1100$ nm of tilted grass top 40° bottom 4 cm top 8 cm for seven values of elevation angle (θ_{source}). The reflectance of grass from Russell et al. [6] is shown as well.

therefore pointed at the ground. Compare figure 33a to figure 33b: the angle that the gray area makes, which is an indication of the probability that the ray hits the ground, decreases as the grass becomes taller. So, taller grass increases the probability that the ray is reflected towards another grass blade, after which the ray is given another chance to be directed towards the sky and eventually detected - taller grass has a higher albedo for a given angle of incidence and wavelength.

At low elevation angles, the ray hits the tall and the short grass close to the top of the blade. The chance that the ray is reflected towards the ground is now more similar for the short and the tall grass. So, for lower elevation angles the difference in albedo of the three vertical grass samples is less.

These length effects are from hereon referred to as *taller grass at higher elevation angles - higher albedo*.

When grass of various heights is combined, the trend in albedo as the angle of incidence changes is different (see figure 31). Effect *higher elevation angle - lower albedo* still plays a role, and *taller grass at higher elevation angles - higher albedo* as well, be it locally for individual tall grass blades. However these effects do not explain the low albedo at low elevation angles - for $\theta_{source} = 10^{-5}°$, the albedo is even 0, which can also be seen in figure 46 in appendix A. Clearly, there is another effect turning up.

Recall that the incoming rays are aimed at $h = z_{blade,max}$. At all elevation angles, there are less blades that are tall enough to be in the path of the ray. For example, when the ray comes in at $\theta_{source} = 10^{-5}°$, only the grass blades that are taller than 11.99999869 cm will contribute to the reflection [6]. The probability that blades of this length even occur in the sample (related to Matlab's *rand* algorithm that generates ξ_8 (in equation 50)), is low. The probability is so small that over all ten runs with 180 blades no collision takes place. The rays shoot overhead

[6] In the most optimal situation, a ray is aimed at the edge of the illuminated area ($r_{illuminated} = 2.5$ cm). The ray then has $r_{sample} + r_{illuminated}$ cm to descend from $h = 12$ cm while having an elevation angle of $\theta_{source} = 10^{-}5$. Using geometry, the ray can maximally descend $\simeq 1.30899694 \times 10^{-6} cm$. To be hit, the blade has to be higher than $11.99999869 cm$

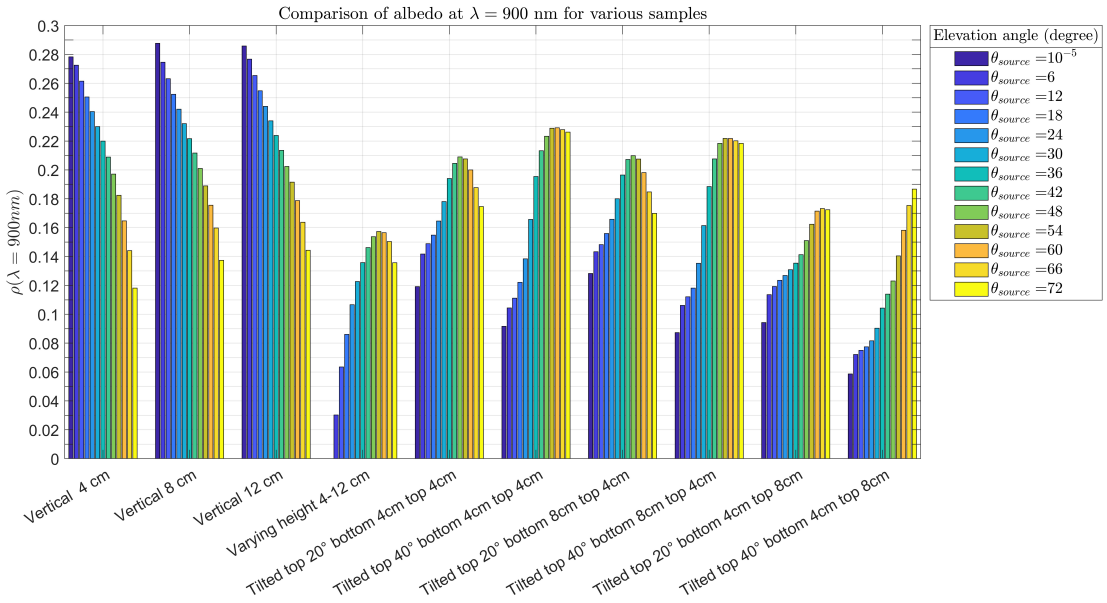
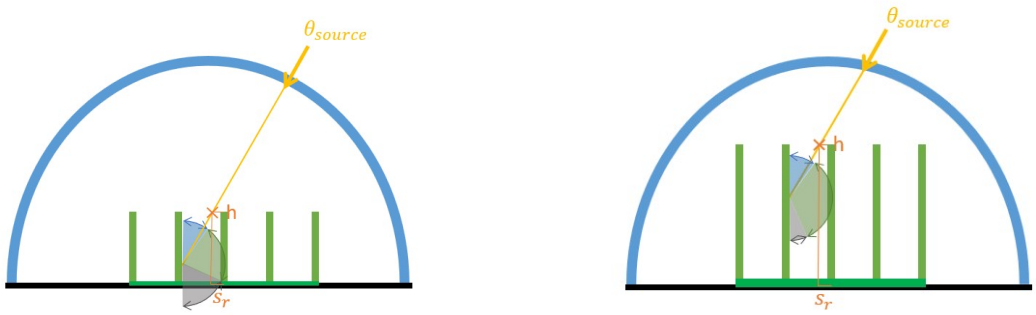


Figure 32: Comparison of albedo (ρ) for $\lambda = 900nm$ for the ten samples for thirteen elevation angles of incident light (θ_{source}). Note that for "Varying height 4-12 cm", the albedo at $\theta_{source} = 10^{-5}$° is zero.



(a) Short grass

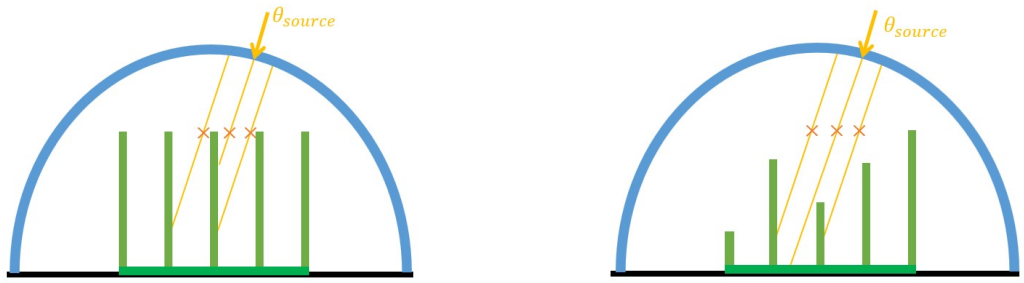
(b) Tall grass

Figure 33: Two dimensional illustration of how grass length (l_{blade}) influences the chance to reflect towards the ground. The incoming ray (yellow) is aimed at a certain radius from the centre of the detection dome (s_r) in both cases, but the height (h) at which it is aimed has changed. When the grass is taller, the probability that the ray hits the ground (related to the gray angle) has decreased, increasing the albedo.

and fall upon the non-sample surface. None of the rays are detected and the albedo is 0. This effect, *mixed grass lengths at low elevation angles - lower albedo*, also occurs at higher elevation angles, although with a less extreme effect on the albedo. At high elevation angles, some rays hit the ground immediately when the blade length is reduced, as can be seen by comparing 34b and 34a: *mixed grass lengths at high elevation angles - lower albedo*. These two effects allow for a maximum albedo when the elevation angle is neither too low for *mixed grass lengths at low elevation angles - lower albedo*, nor too high for *mixed grass lengths at high elevation angles - lower albedo*.

Introducing a tilted top

Four effects (*higher elevation angle - lower albedo*, *taller grass at higher elevation angles - higher albedo*, *mixed grass lengths at high elevation angles - lower albedo* and *mixed grass lengths at low elevation angles - lower albedo*) have been discussed so far. When introducing a tilted top, these effects are all still relevant. However, compared to the previously discussed samples, the effects will be less visible. Because of the introduced tilt, the grass will have various lengths: but the variation in length is much less extreme than is the case for sample "Varying height 4cm



(a) (b)

Figure 34: Two dimensional illustration of how mixed blade length (l_{blade}) influences the chance to reflect towards the ground. As rays of different lengths are mixed, the probability that the ray hits the ground increases, decreasing the albedo.

- 12cm". For example, for the sample "Tilted top 40° bottom 4cm top 8 cm", the height will vary between 9.14 cm and 12 cm. As both *mixed grass lengths at high elevation angles - lower albedo* and *mixed grass lengths at low elevation angles - lower albedo*) are less pronounced, by just looking at the reduced height difference, the albedo will be higher for the tilted top cases compared to "Varying height 4cm - 12cm".

When the grass blades have tilted tops, the architecture of the grass significantly changes. the tilted tops form a roof-like structure, consisting of the top blades and some gaps in between them. When rays come in from a high elevation angle, they hit the top of the blade on the side that faces the sky. As a result, the probability that the new ray direction will be towards the sky increases compared to vertical grass. When the tilt of the blades is more extreme ($\tau_{min} = -40°, \tau_{max} = 40°$), the area covered in blades that is visible from the position of the sun has increased - the LAI is larger. This increases the probability that a ray will hit a blade and thus the albedo. So, *more extreme tilted top at high elevation angles - higher albedo*. Together with the other effects, this results in an elevation angle at which the albedo is maximal.

At low elevation angles, more rays penetrate the grass. When the ray comes under the roof-like structure, it is more difficult to escape compared to the vertical grass, as a ray aimed at the sky might hit the top of the blades. When the tilt of the blades is becoming more extreme ($\tau_{min} = -40°, \tau_{max} = 40°$), the LAI increases which makes it even harder for rays that are under the roof to escape the grass structure and reach the detection dome. Therefore, *more extreme tilted top at low elevation angles - lower albedo*.

Comparing the samples where the bottom part of the grass is 4 cm tall to the samples where the bottom part of the grass is 8 cm tall, it can be seen that the albedo is very similar - changing the height of the bottom part of the blade does not affect the albedo as much as was the case for the vertical grass. The effect *taller grass at higher elevation angles - higher albedo* which is present for vertical grass is overshadowed by the effect of the tilted tops.

When the length of the tilted top is increased instead from 4 to 8 cm, the albedo for low elevation angles decreases as the LAI is higher - there is more roof compared to gaps making it difficult for light entering the grass to escape: *longer tilted top at low elevation angles - lower albedo*.

4.3 Influence spectro-angular albedo on the energy yield

The influence of taking the spectro-angular albedo of grass into account when computing the energy yield is investigated by using the computed BRDFs for all ten samples as an input to

the software developed by Pal [9] and adapted by Anne Rikhof. As the energy yield differs per specific situation, a case study is used to assess the influence of the spectro-angular BRDF of grass.

4.3.1 Introduction case study Donaueschingen-Aasen solar park

In the town Aasen in the south of Germany, there is a solar farm with vertically mounted bifacial solar panels, built by the company Next2Sun (see Figure 35). It consists of n-PERT bifacial solar cells, which are vertically aligned from east to west. The peak power is 4.1 MWp and its annual energy yield is about 4850 MWh. The solar panels are surrounded by grass, used for hay and silage [59].

Figure 35: The Donaueschingen-Aasen solar park consists of n-PERT bifacial solar panels which are vertically aligned, east to west. Figure from Next2Sun [59].

The company is interested in the influence of mowing the grass on the energy yield. They shared measurements of the energy yield and the albedo - the albedo in this case is a single number and does not have a spectro-angular dependency. It is known that a mowing event took place on the 5th of July, 2022. Hence, by comparing a day before to a day after the mowing, information can be obtained about the effect of the albedo on the yield.

As noted in the introduction, the yield of a bifacial solar panel depends on many factors, of which one is the cloud coverage. To take this factor out of the equation as much as possible, two clear-sky days are taken: days on which the direct irradiance (E_{direct}) is similar. Figure 36 shows the direct irradiance in Aasen from the first to the ninth of July. The irradiance was almost the same on the second and eight of July. Therefore, these two days will be compared. The temperature on these days was very similar too, which is relevant as the temperature also influences the performance of a solar panel as mentioned in the introduction.

Figure 36 shows the measured albedo on-site. The data has been filtered such that negative values and values larger than 1 are not considered. Moreover, only data between 09:00 and 18:00 is displayed, as albedo measurements under low light intensity in the night have a low

Figure 36: The direct normal beam irradiance (E_{direct}), temperature (T) and albedo (ρ) around the mowing day (the fifth of July, highlighted in green). From this data, the second and eighth of July have been chosen for comparison (highlighted in yellow). Data from Next2Sun.

signal-to-noise ratio. The graph shows that on days with less direct irradiance (i.e. cloudy days) the albedo fluctuates more: when there is less irradiance, the signal-to-noise ratio is low. So, the albedo of clear-sky days measured during daytime is the most accurate.

The albedo decreases throughout the day. This can have two possible explanations: either the change of angle of the sun is at the root of this, or the grass morphology itself changes throughout the day. In the first case, the albedo is expected to increase in the morning as the sun's elevation angle increases, and decrease in the afternoon as the sun's elevation angle decreases again. Although the albedo seems to have a small peak around midday, the overall trend is still decreasing. One can also imagine that on a warm day, the grass might lose some water, thereby rigidity of it structure, bending over a bit more. As the tilted top length increases, the albedo decreases. Another explanation could be that the wind gradually changes direction, which changes the overall tilt direction of the top of the grass blades which might also affect the albedo.

The albedo seems to have increased slightly as the grass became shorter. On the second of July, the albedo varies between 0.17 and 0.19, whereas on the 8th of July the albedo varies between 0.21 and 0.19. Next2Sun measured the energy yield on both of these days and found that the energy yield on the eighth of July was 6.61 % higher than the energy yield on the second of July. A factor that could explain part of this energy yield increase is the increase of the albedo of the ground. In section 4.2.3, it was shown that a change in grass morphology changes the spectro-angular albedo. In the continuation of this chapter, it is investigated if the various spectro-angular albedos lead to a different energy yield. For the specific introduced case study, the question is if the mowing of the grass could be responsible for the increase in energy yield that Next2Sun measured.

To this end, the ray tracing mode as elaborated upon in section 3 is used to obtain the BRDFs of the samples introduced and discussed in sections 4.1 and 4.2. These BRDFs are used as input

to the reverse ray tracing software of Pal [9], which was introduced in section 2.3. using this method, the energy yield of a single vertical bifacial solar panel is computed. When applying the RRT software, a silicon heterojunction cell (specifically the cell published in Saive, Russell and Atwater [60]) is assumed because of availability of this data. This determines the fill factor and open circuit voltage, as well as the external quantum efficiency. The radiance of the incoming light is taken to be the direct normal beam irradiance as obtained from the company Next2Sun (see figure 36).

4.3.2 Energy yield of a vertical bifacial solar panel surrounded by a diffuse reflector

Figure 37 sketches the used configuration for the vertical bifacial solar panel. The vertical panel receives direct and indirect light on the front and the rear of the module.

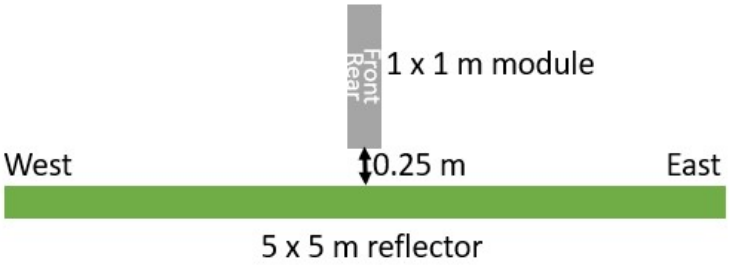


Figure 37: Two dimensional sketch of the three dimensional simulation used as input to the software of Pal [9]. A vertical module is placed 0.25 meter above a reflector.

Figure 39a shows the total power generated by the solar panel: the power due to the indirect light captured by the front of the module, the power due to the indirect light captured by the rear of the module and the power due to the captured direct light from the sky. The figure also shows the power due to only the direct light from the sky (in black). This power profile has two peaks, one in the morning and one in the afternoon. This is expected for east-west positioned bifacial panels, as the angle of incidence of the sun will be beneficial (i.e. perpendicular to one of the faces of the module) twice a day. The peak in the morning is higher as the irradiance received in the morning is slightly higher than in the afternoon.

The same figure shows that making the reflector Lambertian (in dark blue) contributes significantly to the power output. For example, at solar noon (at 13:30), the Lambertian reflector produces 89 W/m^2 whereas the direct light does not produce any power. The power contributed by the reflector varies throughout the day due to the combination of a changing angle of incidence of the sun, a changing irradiance and changing shade.

Figures 39b and 39c show the power generated by the indirect light that is captured by the front and rear of the module, respectively. At sun rise, the sun is positioned in the north-east. Throughout the morning, the irradiance coming from the sun increases, which increases the flux reflected by the reflector and thus the power output due to indirect light. Furthermore, the angle between the sun and the front face of the panel becomes less favourable as it start to deviate more and more from the module's normal. This decreases the power output towards the end of the morning.

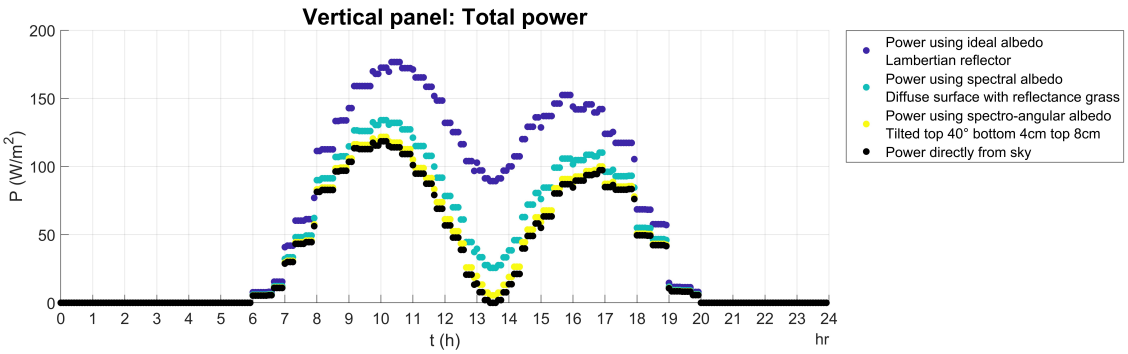
In the morning, the shadow is in the west and only plays a role for the power output of the rear. This is illustrated by figure 38a. Specifically, in the morning, a shadow is cast in the south-west of the module. As the sun rises and travels south throughout the morning, the shadow decreases and shifts to the north. The decreasing shadow size causes the power output of the rear to increase, as a larger area of the reflector now can contribute to the reflection. The

changing position of the shade also matters. At sunrise, the shade is in the south-west and the part of the reflector closest to the module contributes fully to the reflection. As the morning progresses and the sun moves to the east, the shadow casted by the module will be right in front of the rear of the module. This inhibits the part of the reflector closest to the module from contributing. The shift of the shadow decreases the power output until the sun has past the east (at 09:15), after which the part of the reflector close to the module is lit again. These effects can be seen back in the power profile of the rear (figure 39c) and this part of the module to produce another power output than the front (figure 39b).

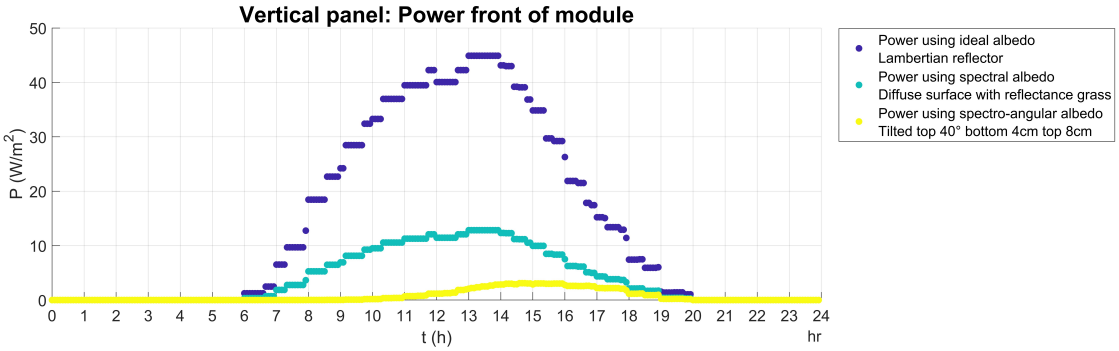
(a) Situation in the morning, when the sun is in the east.

(b) Situation in the afternoon, when the sun is in the west.

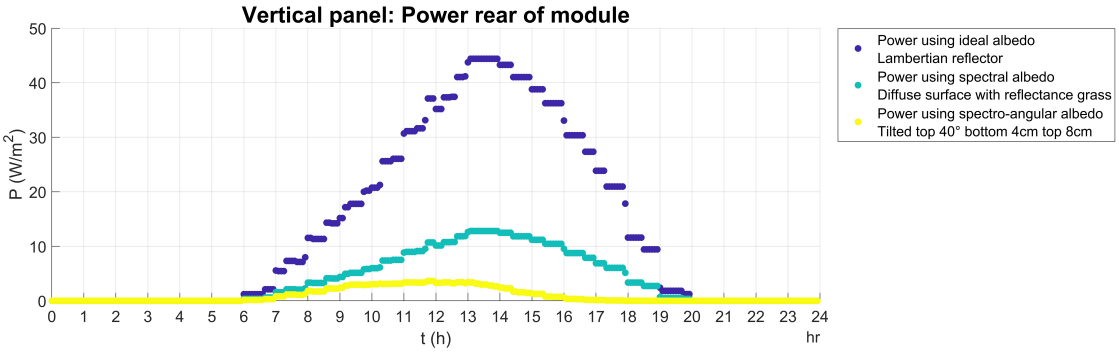
Figure 38: Two dimensional illustration of the incident light (in yellow) on the module (in gray) and the reflector (in green). The module casts a shadow on the reflector, shown by the black line. The diffuse reflection upon illumination of the reflector is shown in blue.



(a) Total power generated by the module (including power generated by direct light).



(b) Power generated due to indirect light on the front of the module.



(c) Power generated due to indirect light on the rear of the module.

Figure 39: Power generated on the second of July for a horizontal oriented bifacial solar panel. Figures a) and b) present the power generated by the rear and front of the module due to indirect (reflected) light, respectively. Figure c) presents the total power.

After solar noon, the described effects take place in reverse order. The generated power on the rear and the front of the panel due to indirect light decreases as the irradiance falling on the reflector decreases due to a combination of less favourable angle of the sun and solar irradiance. The shade is now on the east side of the module (see 38b). Therefore, less power is produced on the front of the module compared to the rear of the module. As the sun sets and its elevation angle decreases, the shadow increases in size, further reducing the produced power on the front module.

Changing the Lambertian reflector for a diffuse reflector with the reflectance of grass, the total output of the solar cell will be significantly less, as can be seen from figure 39a. For example, at solar noon the diffuse reflector with the reflectance of grass contributes 26 W/m^2, which is only 29% of the contribution of the Lambertian reflector at that time. This can be explained by the albedo, which is lower when the reflectance is taken into account. The flux coming from the reflector is thus less, reducing the number of photons that the solar cell can capture. The power profile trend throughout the day is the same as for a Lambertian reflector.

4.3.3 Energy yield of a vertical bifacial solar panel surrounded by grass

Compared to the Lambertian and diffuse reflector, grass, which exhibits retroreflective behaviour, produces a lot less power. In figures 39a, 39b and 39c, the power of "Tilted top 40° bottom 4 cm top 8 cm" is displayed in yellow. The power profiles of the other grass types can be found in appendix B. Returning to the solar noon example, the power produced by the grass "Tilted top 40° bottom 4 cm top 8 cm" is only 5 W/m^2. The first explanation comes from the albedo: the albedo of grass is less, as can be seen from figure 31. Furthermore, the albedo has received an angular component, as it has a retroreflective lobe.

In figures 39b and 39c, the angular, retroreflective component of the reflectance of grass can be seen in the power profile of the front and the rear of the module. In the morning, the sun shines from the east, creating a shadow on the west side of the module. The grass which is not in the shadow has a retroreflective lobe pointing towards the sun, so towards the east, as can be seen in figure 40a. Retroreflective lobes from the grass in the west of the module can be partly captured by the module's rear. In contrast, retroreflective lobes from the grass in the east of the module are not captured. As a result, in the early morning mostly the rear produces power. In figure 39c, it can be seen that the rear does generate power due to the indirect light received in the morning, whereas figure 39b shows that the front of the module barely produces any power in the early morning. Since sunrise, there is some power production in the front of the module as well as the BRDF is not entirely zero opposite to the retroreflection lobe (see figure 52 in appendix A, where the BRDF does not become zero at exactly $\phi_{dome} = -90°$ and $\phi_{dome} = 90°$).

For a reflector with retroreflective behaviour such as grass, the angular component of the reflection changes throughout the day. Specifically, the angle of the retroreflective lobe changes, as it follows the sun's path through the sky. For example, at solar noon the retroreflective lobes all point parallel to the module's surface. So, even though there is no shadow, the output of the module's front and rear is relatively small compared to a diffuse reflector, as can be seen in figure 39b and 39c. Together with the change in irradiance and the shadow effect as explained in section 4.3.2, this behaviour results in an optimum power output from the rear of the module in the morning, which lies around 12:00 for this sample.

(a) Situation in the morning, when the sun is in the east.

(b) Situation in the afternoon, when the sun is in the west.

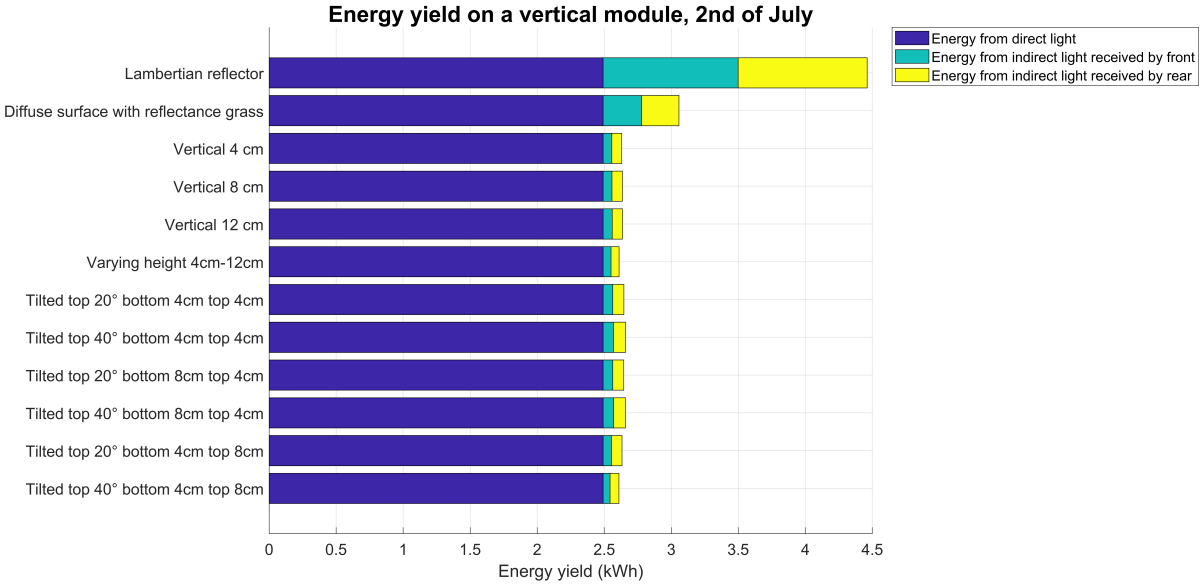
Figure 40: Two dimensional illustration of the incident light (in yellow) on the module (in gray) and the reflector (in green). The module casts a shadow on the reflector, shown by the black line. The retroreflective lobes upon illumination of the reflector are shown in blue.

The situation in the afternoon is sketched in figure 40b. After solar noon, the sun travels north and its elevation angle decreases. the shadow on the east of the module starts small and relatively north to the module, but increases and shifts to the south as the sun sets. The retroreflective lobes follow the path of the sun which results in an optimum power output for the front of the module around 14:45, as figure 39b shows.

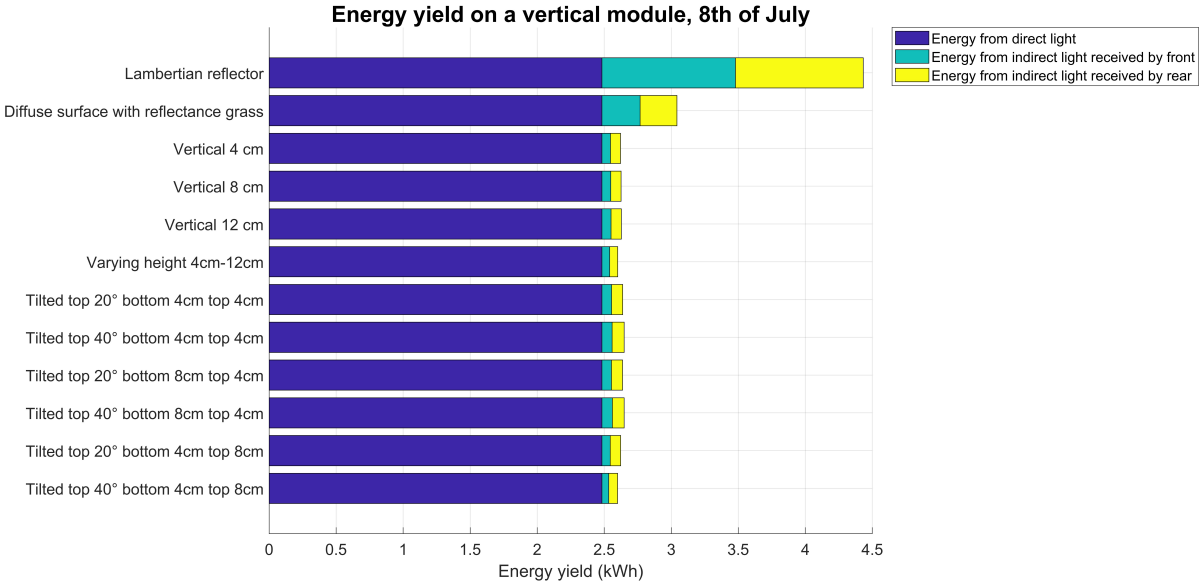
4.3.4 The influence of mowing on the energy yield

The influence of mowing on the energy yield can be analysed using the developed energy yield calculation method. The vertical grass types ("Vertical 4cm", "Vertical 8cm"," Vertical 12 cm" and "Varying height 4 cm - 12 cm") are not considered in this section, as they are considered to be less true to realistic grass morphology compared to the six tilted grass samples. This leaves six samples with a tilted top and various lengths of the bottom and top part of the leaf. These are divided into two categories: tall grass and short grass. The tall grass consists of the grass types where one part of the grass blade is 4 cm and the other part 8 cm (samples 7,8,9 and 10), the short grass consists of grass types whose top and bottom are 4 cm (samples 5 and 6).

Figure 41a presents the energy yield for the situation displayed in figure 37 in Aasen for the second of July. The energy from the direct light does not differ as the reflector is changed. However, changing the reflector impacts the energy yield. For the grass simulated using the ray tracing model the energy yield is less compared to a grass modelled as a Lambertian and a diffuse surface. Figure 41b shows the energy yield for the eight of July. It can be seen that the yield is similar to 41a - as expected, as the solar angles, irradiance and temperature are also similar for these two days.



(a) Energy yield on the second of July for a vertical panel per reflector type.

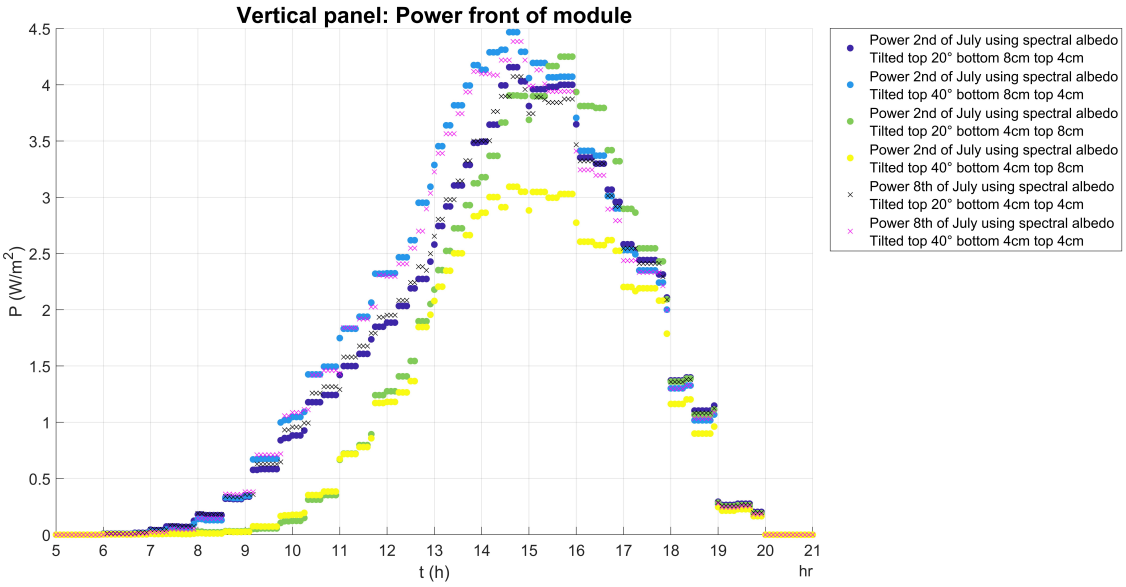


(b) Energy yield on the eight of July for a vertical panel per reflector type.

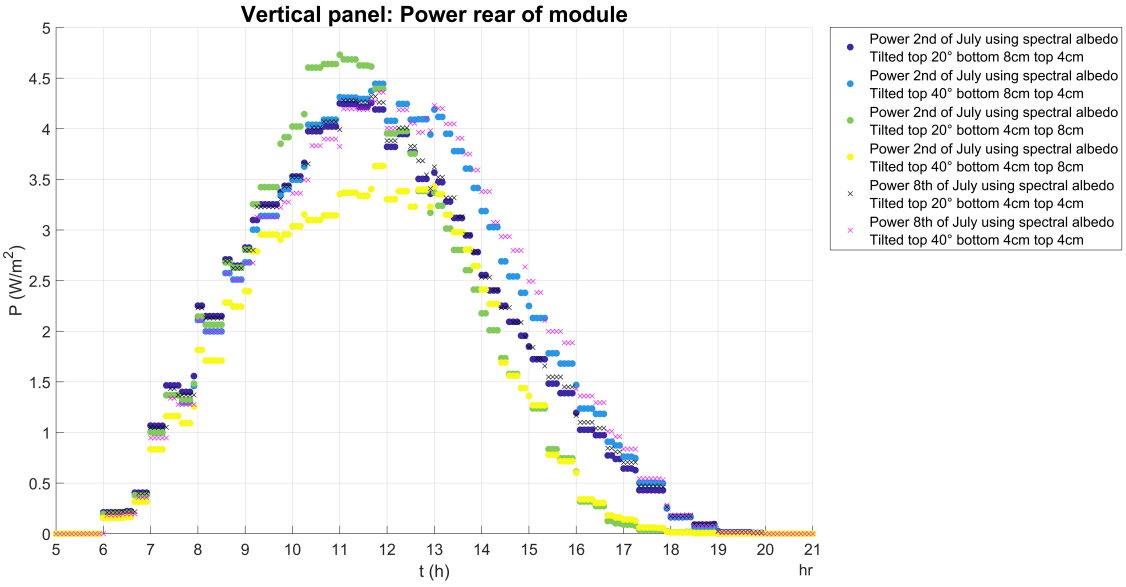
Figures 42a and 42b present the power profiles from the front and the rear of the module for tall grass on the second of July (circles), and the short grass on the eighth of July (crosses). The overall trend as described for "Tilted top 40° bottom 4 cm top 8 cm" in section 4.3.3 is the same for all these grass types. The different types of grass generate their peak power at slightly different times due to a combination of the albedo per elevation angle of incidence and shape of the retroreflective lobe.

An interesting case is the power profile of "Tilted top 40° bottom 4 cm top 8cm" (displayed in yellow), which reaches a maximum power that is about 1 W/m^2 lower on the front and on the rear compared to the other grass types. At the time of the maximum, the elevation angle of the sun is about $\theta_{source} = 60°$. From figure 32, it can be seen that the albedo of this grass sample at this elevation angle and $\lambda = 900$ nm is $\rho = 0.16$. Interestingly, the power output for "Tilted top 20° bottom 4 cm top 8 cm" (in green) at the same elevation angle and wavelength is significantly higher, even though the albedo is only 0.01 more ($\rho = 0.17$). This may be due to the shape of the reflection lobe, which for "Tilted top 40° bottom 4 cm top 8cm" is less wide

(i.e. extends to less azimuth angles and elevation angles) compared to the other tilted grass samples (see figure 52 in appendix A)



(a) Power generated due to indirect light on the front of the module.



(b) Power generated due to indirect light on the rear of the module.

Figure 42: Comparison of power P generated by indirect light throughout part of the day (time t from 05:00 until 21:00) on the second of July (tall grass) and the eight of July (short grass) for a vertical oriented bifacial solar panel.

Table 2 shows the percentage with which the energy yield changes compared from tall grass (second of July) to short grass (eight of July). The table shows that for the examined cases, the energy yield difference are in the range from -0.81% to $+1.46\%$.

Next2Sun measured the energy yield on both of these days and found that the energy yield on the eighth of July was +6.61 % higher than the energy yield on the second of July. The order of the energy yield change that they measured is higher than the energy yield change that was computed in this study (see table 2). This study does show that part of the increase in energy

Table 2: Influence of changing from tall grass (on the second of July) to short grass (on the eighth of July) on the energy yield.

Tall grass	**Short grass**	**Energy yield difference**
Tilted top 20° bottom 8 cm top 4 cm	Tilted top 20° bottom 4 cm top 4 cm	−0.33%
Tilted top 20° bottom 8 cm top 4 cm	Tilted top 40° bottom 4 cm top 4 cm	+0.08%
Tilted top 20° bottom 4 cm top 8 cm	Tilted top 20° bottom 4 cm top 4 cm	+0.18%
Tilted top 20° bottom 4cm top 8 cm	Tilted top 40° bottom 4 cm top 4 cm	+0.59%
Tilted top 40° bottom 8 cm top 4 cm	Tilted top 20° bottom 4 cm top 4 cm	−0.81%
Tilted top 40° bottom 8 cm top 4cm	Tilted top 40° bottom 4 cm top 4 cm	−0.40%
Tilted top 40° bottom 4 cm top 8 cm	Tilted top 20° bottom 4 cm top 4 cm	+1.04%
Tilted top 40° bottom 4 cm top 8 cm	Tilted top 40° bottom 4 cm top 4 cm	+1.46%

yield could be due to a change in grass morphology. For example, when the top of a tilted grass blade is cut, the energy yield increases.

To verify to what extent the change in grass morphology us responsible for the change in power output measured by Next2Sun, the grass morphology before and after mowing needs to be known. Furthermore, other factors that could cause a change in energy yield need to be monitored and taken into consideration in the energy yield calculation model. For example, mowing grass may change the reflectance properties of the individual blades as Dyer, Turner and Seastedt [44] hypothesised.

5 Recommendations

In this section, recommendation are given to improve the adopted method used to obtain the BRDF of grass and to subsequently to calculate the energy yield based on this BRDF.

5.1 Ray tracing model

Modelling ray direction
The model generates a beam of parallel rays which are shot towards the sample. However, realistic lighting conditions should also take into account diffuse light. Diffuse light has been scattered by other reflectors already (clouds, buildings, plants etc.) and therefore has a randomised direction. It can be taken into account by shooting rays from the detection dome with randomised ray directions. The fraction of diffuse to direct light (so rays with randomised ray direction to rays coming from the elevation angle of the source) is critical to correctly incorporate this effect. Overall, the reflection of a grass sample will show less retroreflective behaviour when taking into account diffuse light. For diffuse rays coming from a certain direction, the direction into which they will be reflected is likely to be the direction from which they were coming. Adding up these retroreflective lobes from all directions gives an overall diffuse effect. However, as long as more rays are coming from one angle than another, there will be a retroreflective lobe.

When, after the ray has intersected with a surface, a new ray direction is chosen, a new ray direction is sampled uniformly in polar angle. This has to be done because of the choice for a grid with a spherical coordinate system. However, it might be more intuitive to change to uniformly sampling a hemisphere instead, and take a coordinate system where every pixel on the detection dome has the same solid angle.

Modelling an individual blade
In the developed ray tracing model, surfaces can only reflect diffusely. Considering that Juhan and Marshak [49] found that the specular component of leaf reflectance significantly impacted the BRDF, analysing the effect of adding a specular component is recommended.

In the model, surfaces can only absorb or reflect light. However, surfaces can refract and transmit light as well: upon incidence on a grass blade, light is refracted (i.e. bend as it encounters a medium with a different index of refraction [25]), travels through the grass and finally refracted again when leaving the grass blade. Including these effects makes the model more accurate. As light which is now considered to be absorbed might be refracted and transmitted, including this effect is expected to increase the albedo.

The geometries of the blade which have been used in this work are simplistic. Samples consisting of only vertical blades are deemed unrealistic. Although the samples that include a tilted grass segment are already approaching reality a bit more, they are not representative for the geometries of a real grass blade. To make the blades more realistic, they need to be divided into more segments, such that the bends of grass can be modelled more accurately. Using the current calculation algorithm, this will increase the computational time drastically: recall how using more surfaces as input to the ray tracing model increased the computational time (figure 26). Related to the transmission and the geometry is the question to what extent a three dimensional grass blade can be represented by a two dimensional surface.

The leaf structure was assumed to be static. However, the orientation of leaves may exhibit spatial and temporal variability [61]. The LAD for various species is an active research area: for example, a recent development is the application of deep learning to monitor the temporal

variation of leaf angle distributions [62]. Implementing this will give a time-dependent BRDF, which can influence the energy yield results.

Furthermore, it should be considered to take temporal change in reflectance of the grass blades themselves into account, as for example Carter [42] found that leaf water content changes the albedo of the grass.

Realism of the sample
The samples investigated in this thesis are spawned homogeneously. However, in reality, grass is not as homogeneous - there might be spots where the grass is locally more dense. This will influence the albedo, as it changes the probability of a ray to be redirected towards the detection dome. Furthermore, in the current grass sample blades can cross each other, which is not the case in reality. An improved structure of the grass will benefit the accuracy of the obtained BRDF for grass.

The soil has been modelled as a blackbody to only study the impact of grass. However, the soil is still part of the reflector and to make a more accurate model, the reflectance of the soil should also be taken into account. This will increase the albedo overall, as light which would be lost from the system with blackbody soil, will now have a new chance to be redirected towards the sky. It will also impact the spectral distribution of the BRDF. Because only grass carries a reflectance, the entire spectrum is scaled to the spectrum of grass. When the reflectance of the soil is modelled, the spectral variation of the BRDF will not resemble the reflectance of grass as much anymore. Furthermore, in this model the soil is assumed to be a flat plane, whereas in reality it has a different geometry. When the soil is given a non-zero reflectance, the geometry of the soil will impact the albedo as well.

Validation and error analysis
The found BRDF can be validated by comparing it to more experimental data. Experimentally measured reflection quantities use different viewing angles, which poses a challenge: the ray tracing model has to be run for these specific viewing angles. This makes experimental validation time consuming. Nevertheless, it should be done to validate the found BRDFs as they form the bases of this study.

As randomness plays a significant role in the ray tracing model, the simulated BRDF can vary even though the model is run for the same input parameters. In this study, to limit this effect, the model is run ten times for each sample and the average BRDF is taken. However, it is important to know how much the BRDF can deviate from sample to sample such that the error on the computed energy yield can be estimated.

Computational efficiency
The computational time is a bottleneck for the performance of the ray tracing algorithm: with a faster code, the number of rays, the size of the grass sample and the illuminated part of the sample can be increased. This will benefit the accuracy of the simulated BRDF, as argued in section 3.7. Furthermore, it allows for running more complicated leaf geometries in considerable less time. There are various options to make the code faster. For example, the code can be re-written as to make it multiple-core. The algorithm at the foundation of the code can also be improved, for example by decreasing the number of planes that are checked for intersections to only planes that are in the half-space of the direction of the incoming ray.

5.2 Energy yield computation

Overall, there are many opportunities to make the energy yield calculation more exact - as mentioned in the introduction, the energy yield depends on many factors including e.g. the

temperature and the cloud coverage, which are not taken into account here. However, the model still serves the purpose of showing the influence of the spectro-angular albedo on the energy yield, and including a factor like temperature does not discredit this result.

Running the energy yield calculation at higher resolution, i.e. for a higher accuracy of possible sun positions will make the calculated yield more accurate, which means the impact of different grass types on the energy yield can be assessed more accurately. Furthermore, increasing the accuracy of the BRDF will also increase the energy yield calculation accuracy.

The performed energy calculation does not include diffuse light. To include this, the RRT model can be run with light from every possible solar angle, like proposed for the ray tracing model. When the BRDF is also taking into account diffuse light, the energy yield under diffuse conditions can be computed. This will change the computed energy yield.

Mismatch
The used reverse ray tracing model does not take into account electrical effects in computing the power and energy yield - the module is considered to be one large solar cell. However, electrical effects might impact the outcome of the analysis significantly. In a solar module, solar cells are often connected in series to increase the power and voltage obtained from a single cell. Some conditions will cause certain cells to generate more current than others. For example, a non-uniform irradiance on the module will cause some cells receive more photons and generate more current. However, the current through in-series connected cells must be the same. Due to this electrical effect, the total current drops to the lowest current generated by a cell. This phenomenon is called *mismatch* and can decrease the power output of the module as well as lead to deteroriation of the cell [63].

In this work, the angular component of the reflection is changed and this will have an impact on the distribution of irradiance over the cells of the module. The used reverse ray tracing model computes the short circuit current density for every pixel defined on the module and adds them up. However, it does not impose a mismatch condition, causing the energy yield to be overestimated. For a more accurate energy yield calculation, this condition should be applied. Future research should include an analysis on the effect of different grass types on the mismatch.

Validation of the calculation method
More case studies have to be performed to quantify to what extent a change in the spectro-angular albedo of grass impacts the energy yield. Availability of real-world data of the properties of grass (e.g. height and reflectance) as well as data on the solar cells in question is key to increase the accuracy of the yield calculation of the models and being able to compare this to energy yield measurements.

6 Conclusion

Research on grass reflection is widespread and therefore also disconnected. In the PV community, the phenomenon of retroreflection of grass impact on yield has not been accounted for to the author's knowledge. In this thesis, the impact of different grass structures on the spectro-angular reflection and impact on the energy yield has been investigated. A ray tracing software has been developed to simulate the reflection of different types of grass. This was used as input to a reverse ray tracing model to compute the energy yield. The tilt of the grass blade was found to introduce a retroreflective component as a function of the elevation angle of the source.

The simulated spectro-angular reflection has been applied to a case study to compute the energy yield. Taking into account the retroreflective reflection lobe of grass results in a lower energy yield of a vertical east-west bifacial solar panel compared to assuming grass to reflect diffusely. Moreover, it was shown that the morphology of the grass influences the energy yield. For example, cutting the tilted top part of the grass blade increases the energy yield in the order of 1% for the specific case study.

The presented work lays a foundation for more research into how the properties of the reflector impact the energy yield of solar panels. The research can be extended to other agricultural crops by modifying the sample examined by the ray tracing model. Knowing the energy yield more accurately comes with the benefits mentioned in the introduction: reduction of the investment risk barrier, support of grid operation and reinforcing the roots for scientific development.

Concerning the latter, the results can be used for optimisation of an agrivoltaic system: the solar panel can be positioned and oriented more accurately such as to capture more irradiance. Furthermore, it opens up a new dimension to albedo management: adjusting the reflection such as to increase the energy yield. For example, a farmer might choose a crop that reflects the light such as to maximise the energy yield, or adapt agricultural practices (e.g. mowing earlier) to enhance the amount of energy that can be harvested.

Overall, the photovoltaic community would benefit from incorporating the spectro-angular albedo in energy yield calculations.

Bibliography

[1] A. Cuevas et al. '50 Per cent more output power from an albedo-collecting flat panel using bifacial solar cells'. In: *Solar Energy* 29.5 (Jan. 1982), pp. 419–420. ISSN: 0038-092X. DOI: 10.1016/0038-092X(82)90078-0.

[2] A. Luque and S. Hegedus. *Handbook of photovoltaic science and engineering*. Wiley, 2011. ISBN: 9780470721698.

[3] M. Tahir Patel et al. 'A worldwide cost-based design and optimization of tilted bifacial solar farms'. In: *Applied Energy* 247 (Aug. 2019), pp. 467–479. ISSN: 0306-2619. DOI: 10.1016/J.APENERGY.2019.03.150.

[4] VDMA. *International Technology Roadmap for Photovoltaic (ITRPV)*. Tech. rep. 2023.

[5] M. H. Riaz et al. 'The optimization of vertical bifacial photovoltaic farms for efficient agrivoltaic systems'. In: *Solar Energy* 230 (Dec. 2021), pp. 1004–1012. ISSN: 0038092X. DOI: 10.1016/j.solener.2021.10.051.

[6] T.C.R. Russell et al. 'The Influence of Spectral Albedo on Bifacial Solar Cells: A Theoretical and Experimental Study'. In: *IEEE Journal of Photovoltaics* 7.6 (Nov. 2017), pp. 1611–1618. ISSN: 21563381. DOI: 10.1109/JPHOTOV.2017.2756068.

[7] M. P. Brennan et al. 'Effects of spectral albedo on solar photovoltaic devices'. In: *Solar Energy Materials and Solar Cells* 124 (May 2014), pp. 111–116. ISSN: 0927-0248. DOI: 10.1016/J.SOLMAT.2014.01.046.

[8] R. W. Andrews and J. M. Pearce. 'The effect of spectral albedo on amorphous silicon and crystalline silicon solar photovoltaic device performance'. In: *Solar Energy* 91 (May 2013), pp. 233–241. ISSN: 0038-092X. DOI: 10.1016/J.SOLENER.2013.01.030.

[9] S. Pal. *Tracing the light: designing reflectors for bifacial photovoltaic yield enhancement under outdoor irradiance*. 2022. ISBN: 9789464195972.

[10] R. R. Rao, M. Mani and P. C. Ramamurthy. 'An updated review on factors and their inter-linked influences on photovoltaic system performance'. In: *Heliyon* 4.9 (Sept. 2018), e00815. ISSN: 2405-8440. DOI: 10.1016/J.HELIYON.2018.E00815.

[11] A. Richter. 'Bankability'. In: Konstanz, Oct. 2017.

[12] J. Meydbray. 'Barriers to Financing Bifacial PV Projects'. In: bifiPV Workshop 2018, 2018. URL: https://www.bifipv-workshop.com/2018denverproceedings.

[13] G. Notton and C. Voyant. 'Forecasting of Intermittent Solar Energy'. In: *Advances in Renewable Energies and Power Technologies*. Ed. by Imene Yahyaoui. Madrid: Elsevier, 2018. Chap. 3.

[14] M. Saguan. 'L'Analyse économique des architectures de marché électrique. L'application au market design du temps réel'. PhD thesis. Paris: Université Paris Sud, Apr. 2007. URL: https://theses.hal.science/tel-00281131.

[15] S. Goodarzi, H. N. Perera and D. Bunn. 'The impact of renewable energy forecast errors on imbalance volumes and electricity spot prices'. In: *Energy Policy* 134 (Nov. 2019), p. 110827. ISSN: 0301-4215. DOI: 10.1016/J.ENPOL.2019.06.035.

[16] L. M. Einhaus et al. 'Free-Space Diffused Light Collimation and Concentration'. In: *ACS Photonics* (Feb. 2022). ISSN: 23304022. DOI: 10.1021/acsphotonics.2c01652.

[17] M. B. Jones and A. Lazenby. 'Preface'. In: *The Grass Crop: The Physiological basis of production*. Ed. by M. B. Jones and A. Lazenby. First edition. New York: Chapman and Hall Ltd., 1988.

[18] T.C.R. Russell, R. Saive and H.A. Atwater. 'Thermodynamic Efficiency Limit of Bifacial Solar Cells for Various Spectral Albedos'. In: *2017 IEEE 44th Photovoltaic Specialist Conference, PVSC 2017*. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 2236–2241. ISBN: 9781509056057. DOI: 10.1109/PVSC.2017.8366261.

[19] S. Sandmeier et al. 'Physical Mechanisms in Hyperspectral BRDF Data of Grass and Watercress'. In: *Remote Sensing of Environment* 66.2 (Nov. 1998), pp. 222–233. ISSN: 0034-4257. DOI: 10.1016/S0034-4257(98)00060-1.

[20] P. P. J. Roosjen et al. 'A laboratory goniometer system for measuring reflectance and emittance anisotropy'. In: *Sensors (Switzerland)* 12.12 (Dec. 2012), pp. 17358–17371. ISSN: 14248220. DOI: 10.3390/s121217358.

[21] B. Kirn, K. Brecl and M. Topic. 'A new PV module performance model based on separation of diffuse and direct light'. In: *Solar Energy* 113 (Mar. 2015), pp. 212–220. ISSN: 0038-092X. DOI: 10.1016/J.SOLENER.2014.12.029.

[22] U. A. Yusufoglu et al. 'Analysis of the annual performance of bifacial modules and optimization methods'. In: *IEEE Journal of Photovoltaics* 5.1 (Jan. 2015), pp. 320–328. ISSN: 21563381. DOI: 10.1109/JPHOTOV.2014.2364406.

[23] H. Ziar et al. 'A comprehensive albedo model for solar energy applications: Geometric spectral albedo'. In: *Applied Energy* 255 (Dec. 2019), p. 113867. ISSN: 0306-2619. DOI: 10.1016/J.APENERGY.2019.113867.

[24] A. Rikhof. 'Modelling the yield of building integrated photovoltaic systems employing free space luminescent solar concentrators'. Master thesis in preparation. Enschede: University of Twente, 2023.

[25] E. Hecht. *Optics*. 5th ed. Harlow: Pearson Education Limited, 2017.

[26] R. J. D. Tilley. *Colour and the optical properties of materials*. Chichester, West Sussex: John Wiley & Sons, 2000.

[27] K. M. Yoo, G. C. Tang and R. R. Alfano. *Coherent backscattering of light from biological tissues*. Tech. rep. 1990.

[28] D. Fleet and A. Hertzmann. *Radiometry and Reflection 12 Radiometry and Reflection*. 2005.

[29] F. E. Nicodemus et al. *Geometrical Considerations and Nomenclature for Reflectance*. Washington, D.C.: U.S. Department of commerce, Aug. 1977.

[30] R. Siegel and J. R. Howell. *Thermal Radiation Heat Transfer*. 2nd ed. New York: Hemisphere Publishing Corporation, 1981. Chap. 6.

[31] M. A. Shah, J. Kontinnen and S. Pattanaik. 'Real-time Rendering of Realistic-looking Grass'. In: *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. Association for Computing Machinery, Inc. 2005.

[32] W. Qin et al. *Characterizing leaf geometry for grass and crop canopies from hotspot observations: A simulation study*. Tech. rep. URL: www.elsevier.com/locate/rse.

[33] L. Zheng et al. 'Spatial, temporal, and spectral variations in albedo due to vegetation changes in China's grasslands'. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 152 (June 2019), pp. 1–12. ISSN: 0924-2716. DOI: 10.1016/J.ISPRSJPRS.2019.03.020.

[34] S. I. Seneviratne et al. 'Land radiative management as contributor to regional-scale climate adaptation and mitigation'. In: *Nature Geoscience* 11.2 (Feb. 2018), pp. 88–96. ISSN: 17520908. DOI: 10.1038/s41561-017-0057-5.

[35] J. M. Chen and T. A. Black. 'Defining leaf area index for non-flat leaves'. In: *Plant, Cell and Environment* 15 (1992), pp. 421–429.

[36] G. Zheng and L. M. Moskal. 'Retrieving Leaf Area Index (LAI) Using Remote Sensing: Theories, Methods and Sensors'. In: *Sensors* 9.4 (Apr. 2009), pp. 2719–2745. ISSN: 14248220. DOI: 10.3390/s90402719.

[37] R. Lemeur and B. L. Blad. 'A Critical Review of Light Models for Estimating the Shortwave Radiation Regime of Plant Canopies'. In: 1 (Jan. 1975), pp. 255–286. ISSN: 0166-2287. DOI: 10.1016/B978-0-444-41273-7.50025-8.

[38] S.A.W. Gerstl. 'The angular reflectance signature of the canopy hot spot in the optical regime'. In: *Conference: The angular reflectance signature of the canopy hot spot in the optical regime*. 1988.

[39] A. Kuusk. 'The Hot Spot Effect in Plant Canopy Reflectance'. In: *Photon-Vegetation Interactions*. 1991, pp. 139–160.

[40] L. Belcour et al. 'Bidirectional reflectance distribution function measurements and analysis of retroreflective materials'. In: *Journal of the Optical Society of America A* 31.12 (Dec. 2014), p. 2561. ISSN: 1084-7529. DOI: 10.1364/josaa.31.002561.

[41] R. D. Jackson et al. 'Bidirectional measurements of surface reflectance for view angle corrections of oblique imagery'. In: *Remote Sensing of Environment* 32.2-3 (May 1990), pp. 189–202. ISSN: 0034-4257. DOI: 10.1016/0034-4257(90)90017-G.

[42] G. A. Carter. 'Primary and Secondary Effects of Water Content on the Spectral Reflectance of Leaves'. In: *American Journal of Botany* 78.7 (1991), pp. 916–924. DOI: 10.1002/j.1537-2197.1991.tb14495.x.

[43] B. J. Clark, J.-L. Prioul and H. Couderc. 'The physiological response to cutting in Italian ryegrass'. In: *Grass and Forage Science* 32 (), pp. 1–5. DOI: https://doi.org/10.1111/j.1365-2494.1977.tb01405.x.

[44] M. I. Dyer, C. L. Turner and T. R. Seastedt. *Mowing and Fertilization Effect on Productivity and Spectral Reflectance in Bromus Inermis Plots*. Tech. rep. 4. 1991, pp. 443–452. DOI: 10.2307/1941901.

[45] K. Berger et al. *Evaluation of the PROSAIL model capabilities for future hyperspectral model environments: A review study*. Jan. 2018. DOI: 10.3390/rs10010085.

[46] S. Jacquemoud et al. 'PROSPECT + SAIL models: A review of use for vegetation characterization'. In: *Remote Sensing of Environment* 113.SUPPL. 1 (Sept. 2009), S56–S66. ISSN: 0034-4257. DOI: 10.1016/J.RSE.2008.01.026.

[47] P. R. J. North. 'Three-Dimensional Forest Light Interaction Model Using a Monte Carlo'. In: *IEEE Transactions on Geoscience and Remote Sensing* 34.4 (1996). DOI: 10.1109/36.508411.

[48] R. Juhan and A. Marshak. 'Calculation of Canopy Bidirectional Reflectance Using the Monte Carlo Method'. In: *Remote Sensing of Environment* 24 (1988), pp. 213–225. DOI: 10.1016/0034-4257(88)90026-0.

[49] R. Juhan and A. Marshak. *The Influence of Leaf Orientation and the Specular Component of Leaf Reflectance on the Canopy Bidirectional Reflectance*. Tech. rep. 1989, pp. 251–260. DOI: 10.1016/0034-4257(89)90086-2.

[50] W. Qin and N. S. Goel. 'An evaluation of hotspot models for vegetation canopies'. In: *Remote Sensing Reviews* 13.1-2 (1995), pp. 121–159. ISSN: 02757257. DOI: 10.1080/02757259509532299.

[51] S. Pal and R. Saive. 'Output Enhancement of Bifacial Solar Modules under Diffuse and Specular Albedo'. In: *Conference Record of the IEEE Photovoltaic Specialists Conference*. Institute of Electrical and Electronics Engineers Inc., June 2021, pp. 1159–1162. ISBN: 9781665419222. DOI: 10.1109/PVSC43889.2021.9519093.

[52] F. Van Loenhout. 'Influence of Albedo on Bifacial Solar Module Output: An Experimental Approach'. Bachelor thesis. Enschede: University of Twente, July 2021.

[53] W. Gu et al. *A comprehensive review and outlook of bifacial photovoltaic (bPV) technology*. Nov. 2020. DOI: 10.1016/j.enconman.2020.113283.

[54] S. Bowden and C. Honsberg. *Photovoltaics Education Website - PVEducation*. URL: https://www.pveducation.org.

[55] PV Lighthouse. *SunSolve-Yield*. URL: https://www.pvlighthouse.com.au/sunsolve-yield.

[56] S. Marschner and P. Shirley. *Fundamentals of Computer Graphics*. 4th ed. Boca Raton, Florida, USA: CRC Press, 2016.

[57] J. De Young and A. Fournier. 'Properties of Tabulated Bidirectional Reflectance Distribution Functions'. In: *Proceedings of Graphics Interface '97*. Kelowna, British Columbia, Canada: Canadian Human-Computer Communications Society, 1997, pp. 47–55.

[58] Kunstgrasnet.nl. *Kunstgras Naturel 2.0 — Natuurlijk kunstgras*. URL: https://www.kunstgrasnet.nl/naturel.

[59] Next2Sun. *Testimonials Agri-PV Plants*. 2022. URL: https://next2sun.com/en/testimonials/agripv-systems/.

[60] R. Saive, T.C.R. Russell and H.A. Atwater. 'Light Trapping in Bifacial Solar Modules Using Effectively Transparent Contact (ETCs)'. In: *2018 IEEE 7th World Conference on Photovoltaic Energy Conversion (WCPEC)(A Joint Conference of 45th IEEE PVSC, 28th PVSEC 34th EU PVSEC)*. 2018. ISBN: 9781538685297. DOI: 10.1109/PVSC.2018.8547314.

[61] R. Wirth, B. Weber and R. J. Ryel. 'Spatial and temporal variability of canopy structure in a tropical moist forest'. In: *Acta Oecologica* 22.5-6 (Sept. 2001), pp. 235–244. ISSN: 1146-609X. DOI: 10.1016/S1146-609X(01)01123-7.

[62] T. Kattenborn et al. 'AngleCam: Predicting the temporal variation of leaf angle distributions from image series with deep learning'. In: *Methods in Ecology and Evolution* 13.11 (Nov. 2022), pp. 2531–2545. ISSN: 2041210X. DOI: 10.1111/2041-210X.13968.

[63] F. Spertino, P. D. Leo and F. Corona. 'Non-Idealities in the I-V Characteristic of the PV Generators: Manufacturing Mismatch and Shading Effect'. In: *Solar Cells-Silicon Wafer-Based Technologies* (2011). URL: www.intechopen.com.

Appendices

A Overview simulated BRDFs

The following figures present the BRDF of the ten simulated grass samples. Every rectangular plot shows the BRDF per pixel of the detection dome (ϕ_{dome},θ_{dome}) at a wavelength $\lambda = 900$ nm for a certain angle of incidence. The angle of incidence θ_{source} and $\phi_{source} = 0$. Note that in these plots, θ_{source} is rounded and the plot with $\theta_{source} = 0°$ shows the data of $\theta_{source} = 10^{-5}°$.

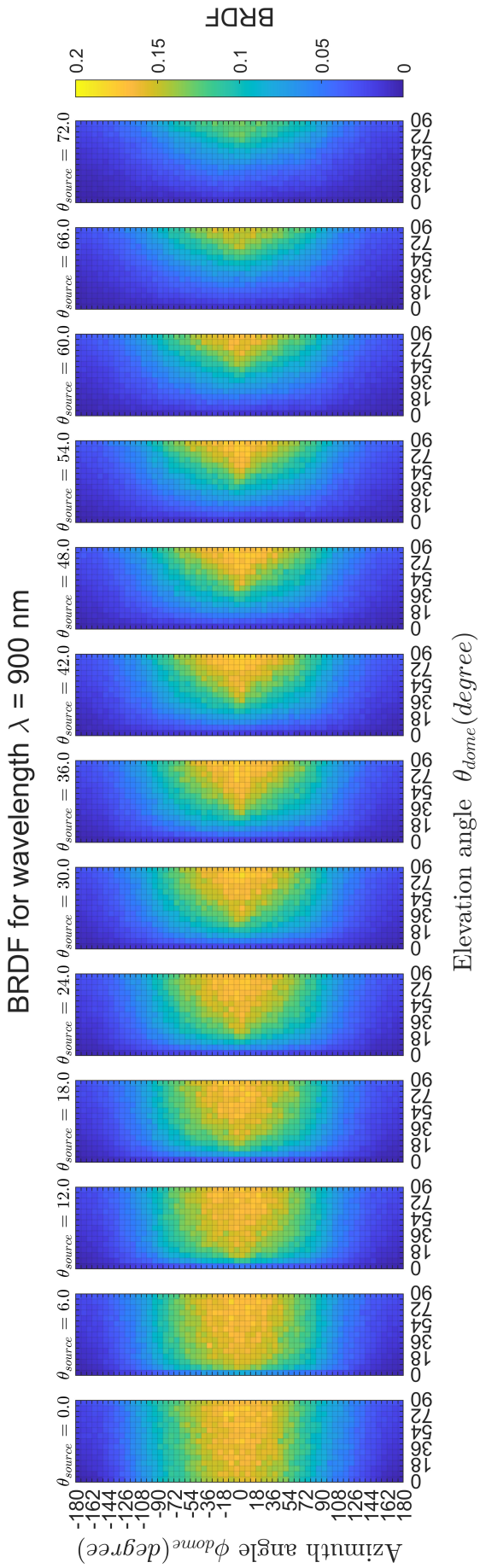


Figure 43: The BRDF obtained for the grass sample "Vertical 4 cm".

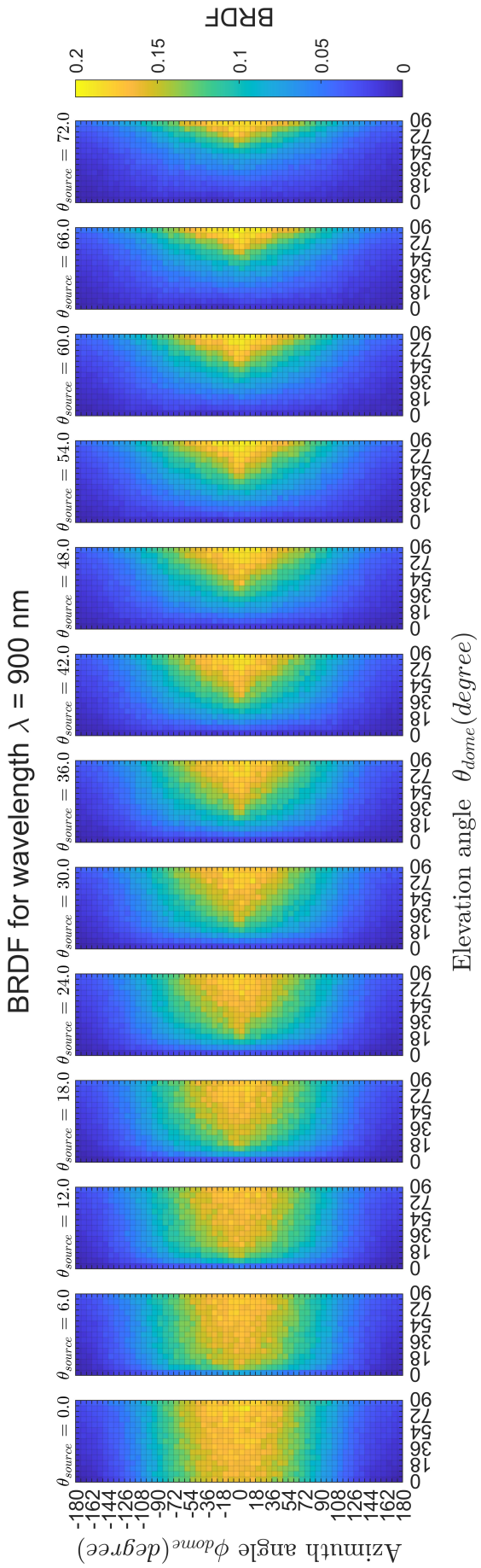


Figure 44: The BRDF obtained for the grass sample "Vertical 8 cm".

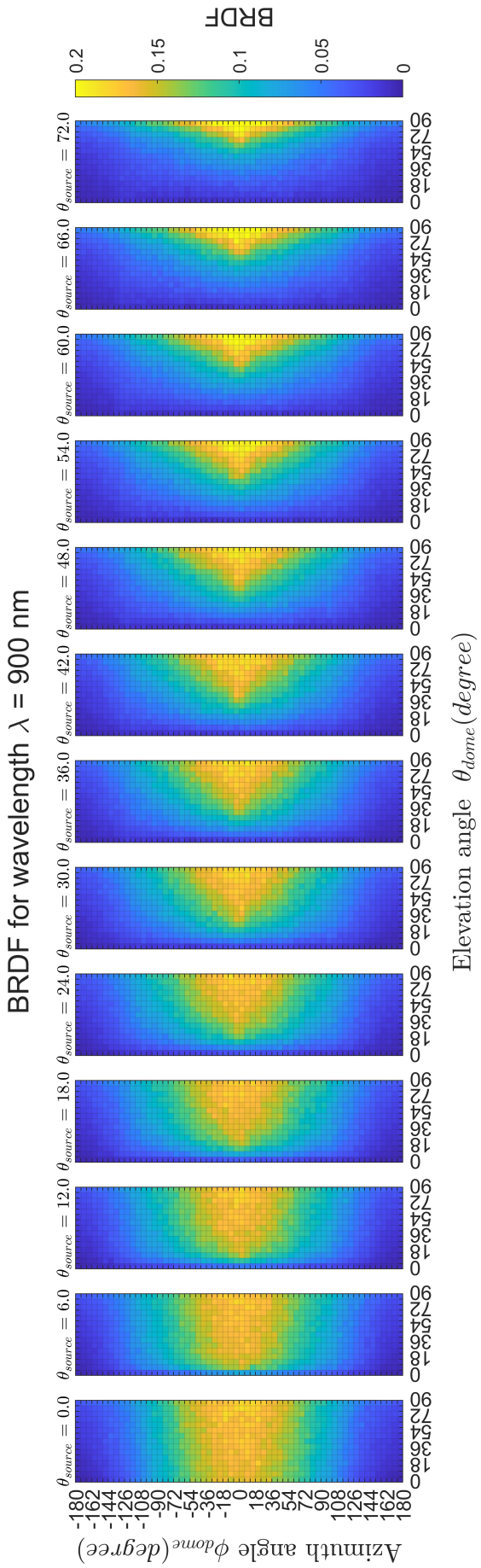


Figure 45: The BRDF obtained for the grass sample "Vertical 12 cm".

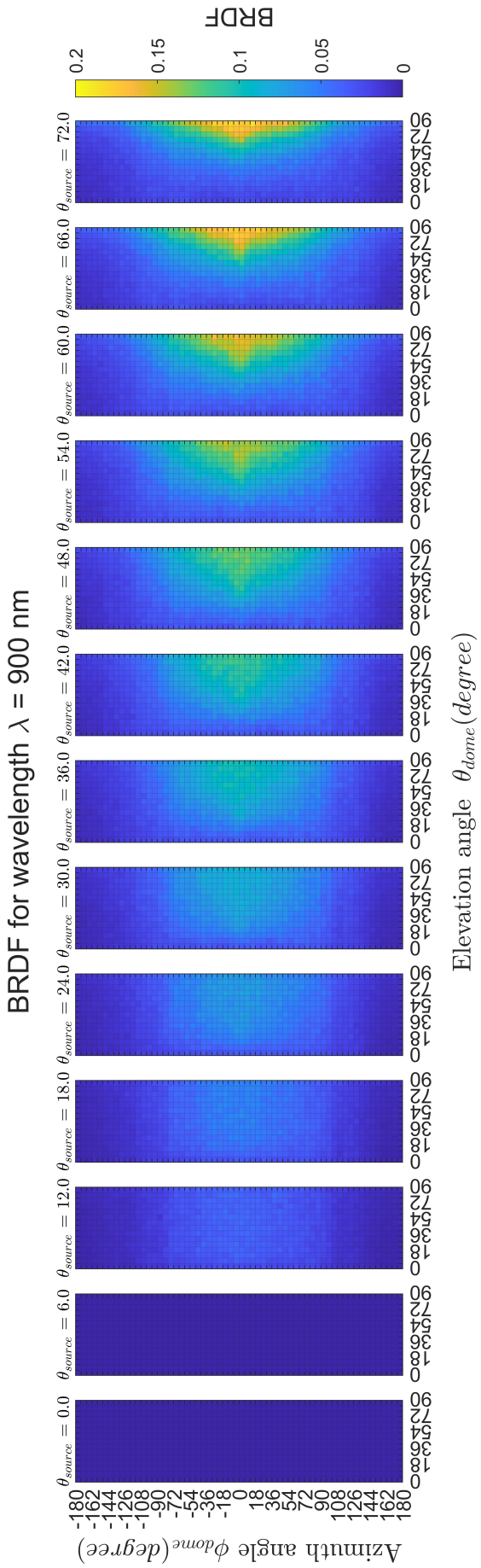


Figure 46: The BRDF obtained for the grass sample "Varying height 4cm - 12cm".

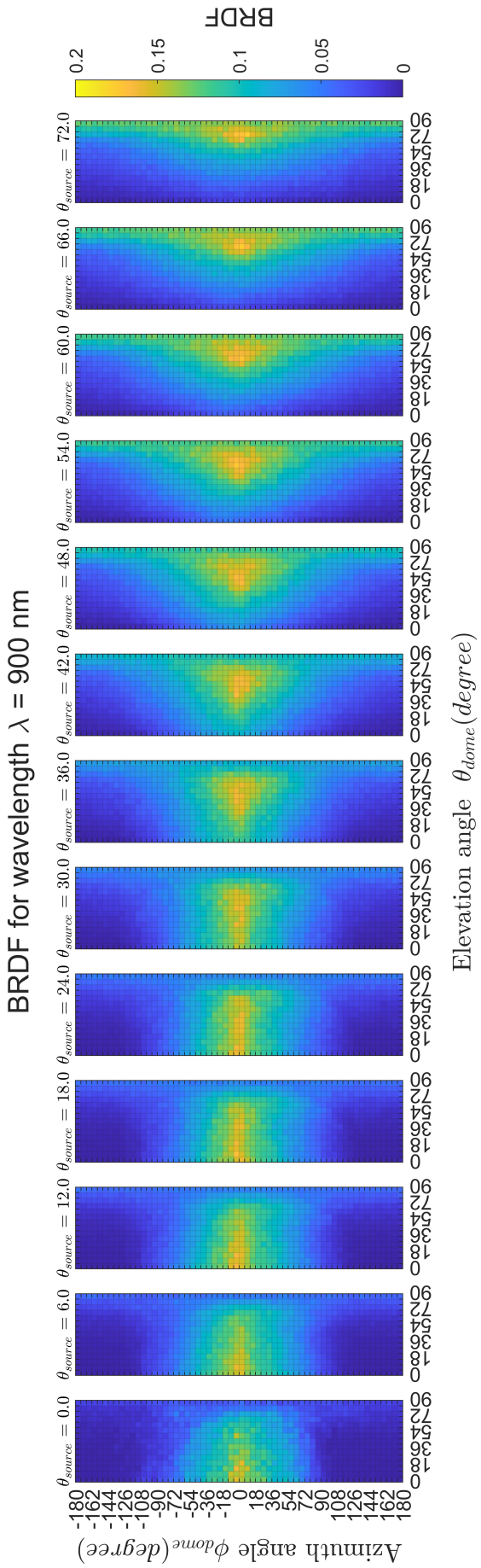


Figure 47: The BRDF obtained for the grass sample "Tilted top 20° bottom 4 cm top 4 cm".

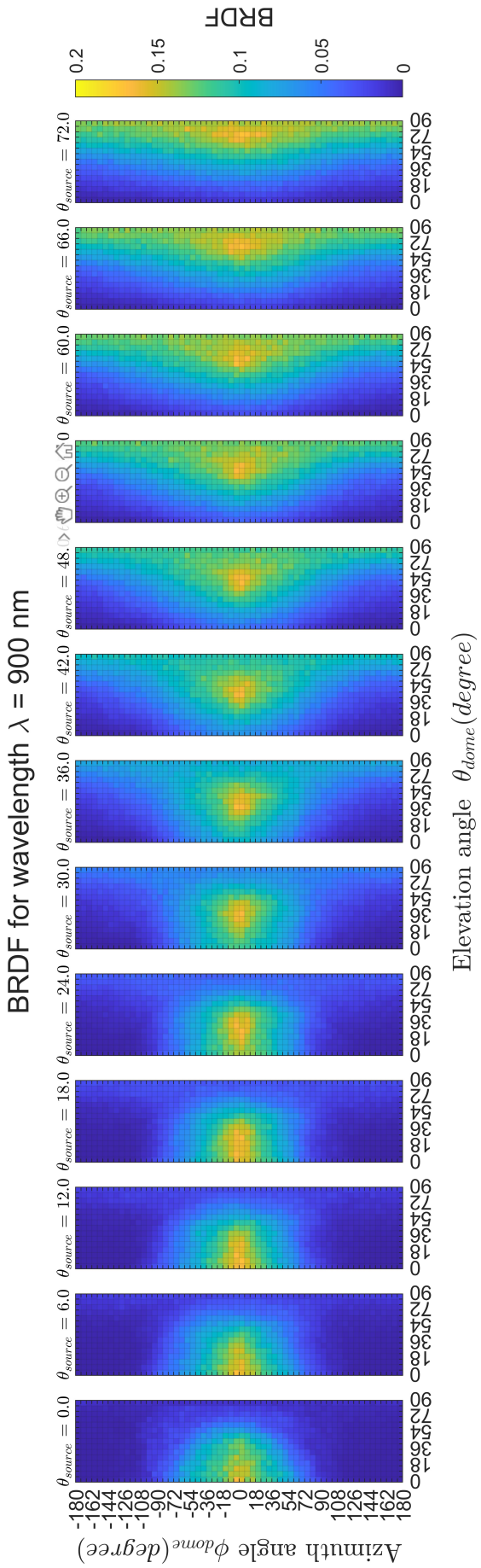


Figure 48: The BRDF obtained for the grass sample "Tilted top 40° bottom 4 cm top 4 cm".

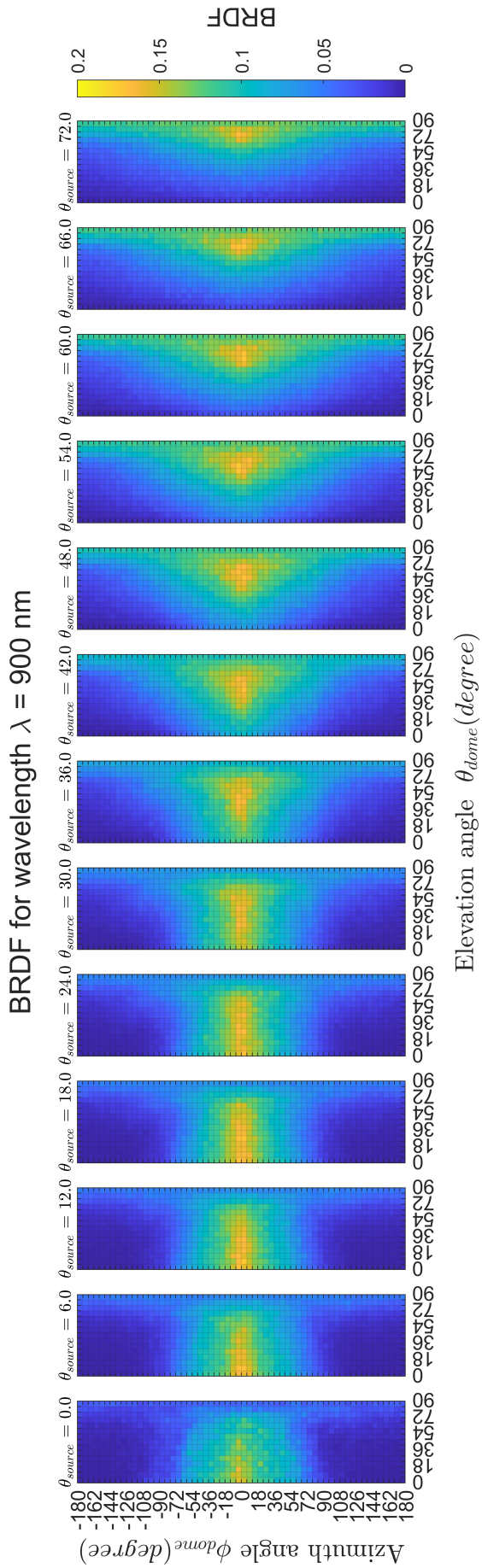


Figure 49: The BRDF obtained for the grass sample "Tilted top 20° bottom 8 cm top 4 cm".

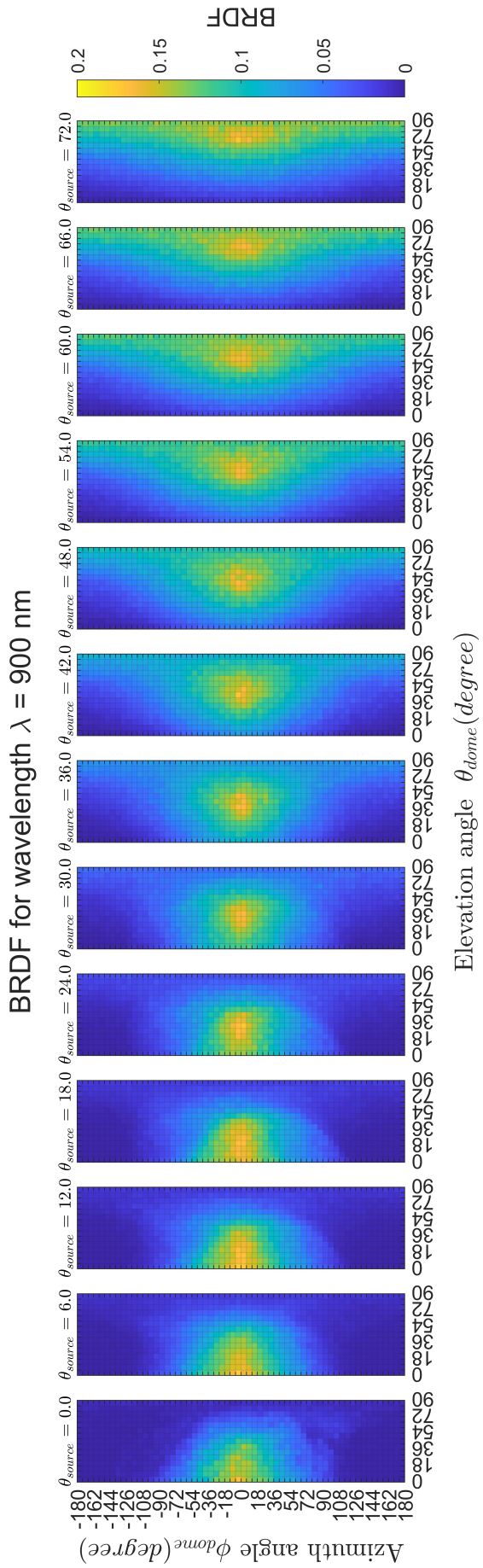


Figure 50: The BRDF obtained for the grass sample "Tilted top 40° bottom 8 cm top 4 cm".

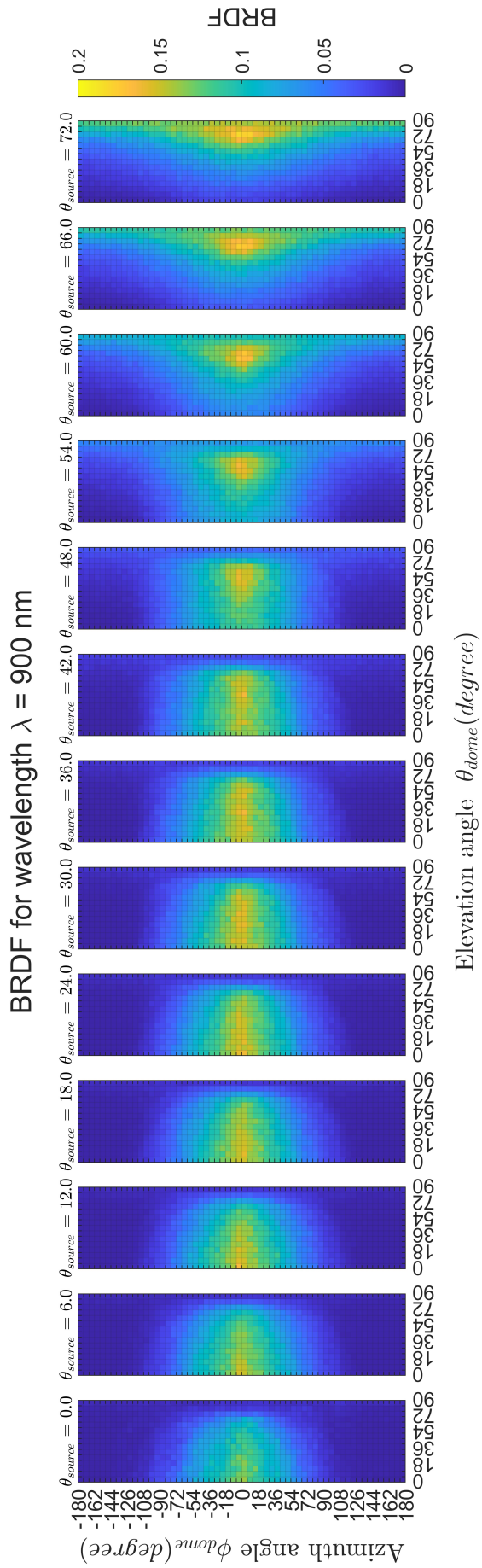


Figure 51: The BRDF obtained for the grass sample "Tilted top 20° bottom 4 cm top 8 cm".

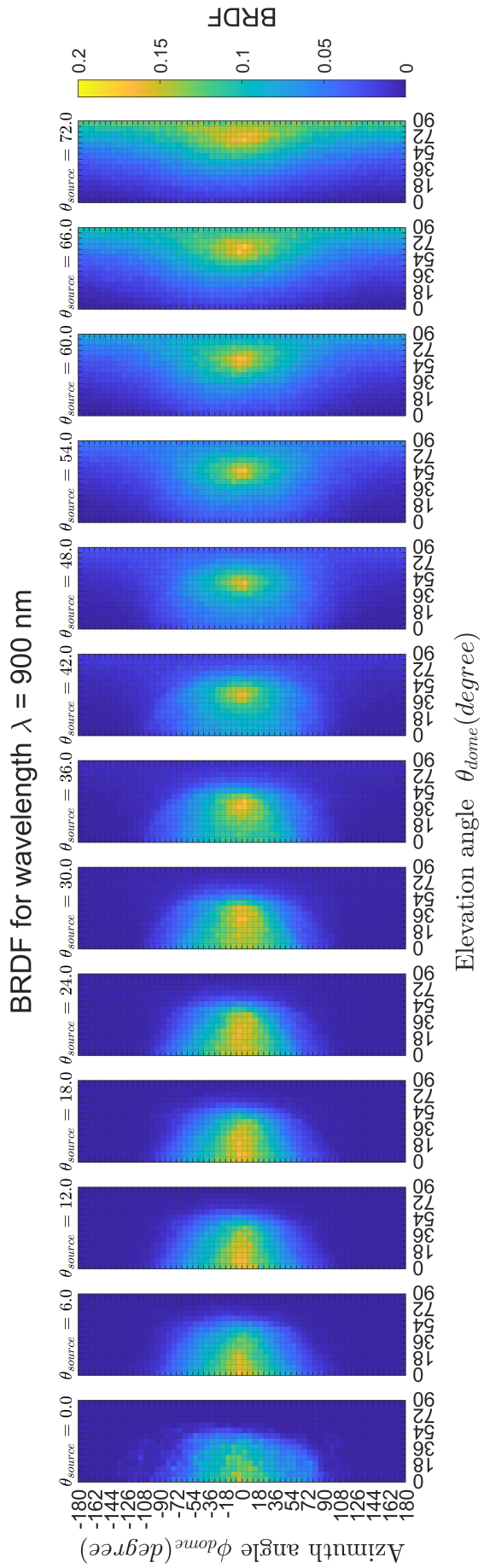
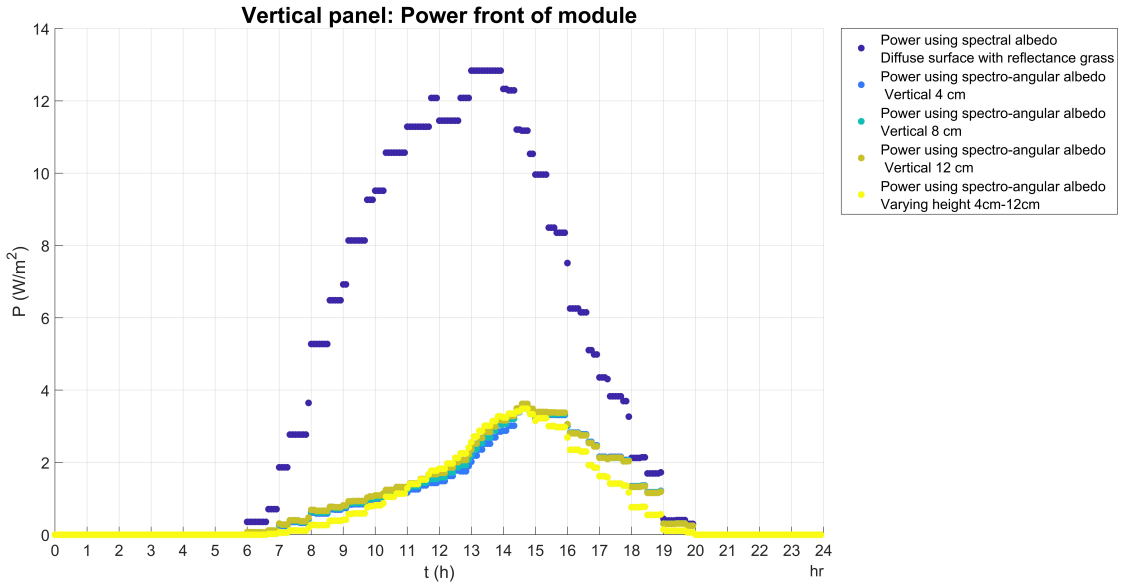
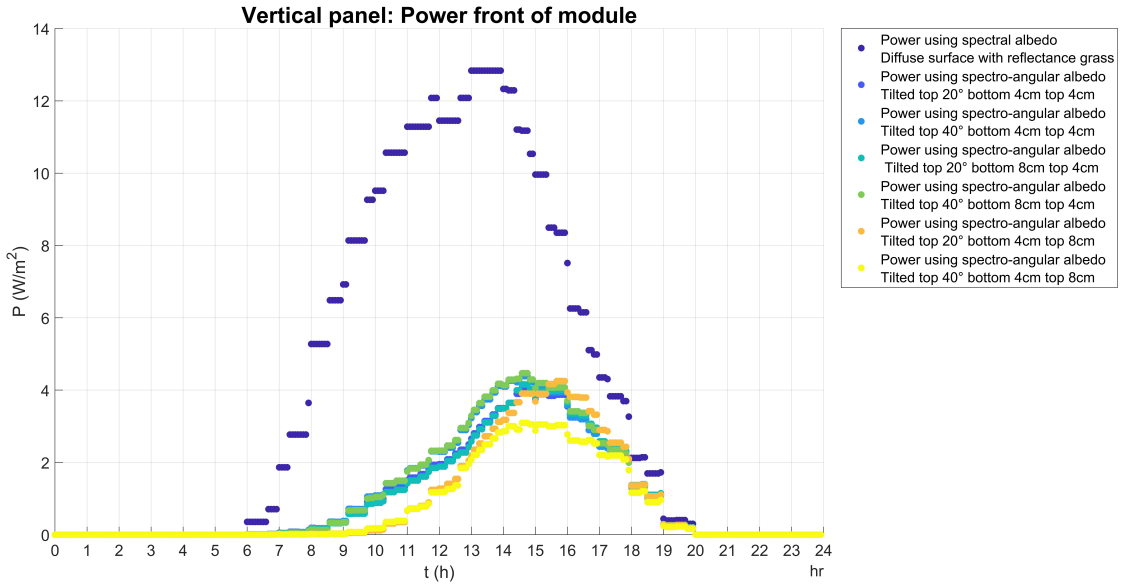


Figure 52: The BRDF obtained for the grass sample "Tilted top 40° bottom 4 cm top 8 cm".

B Overview power profiles

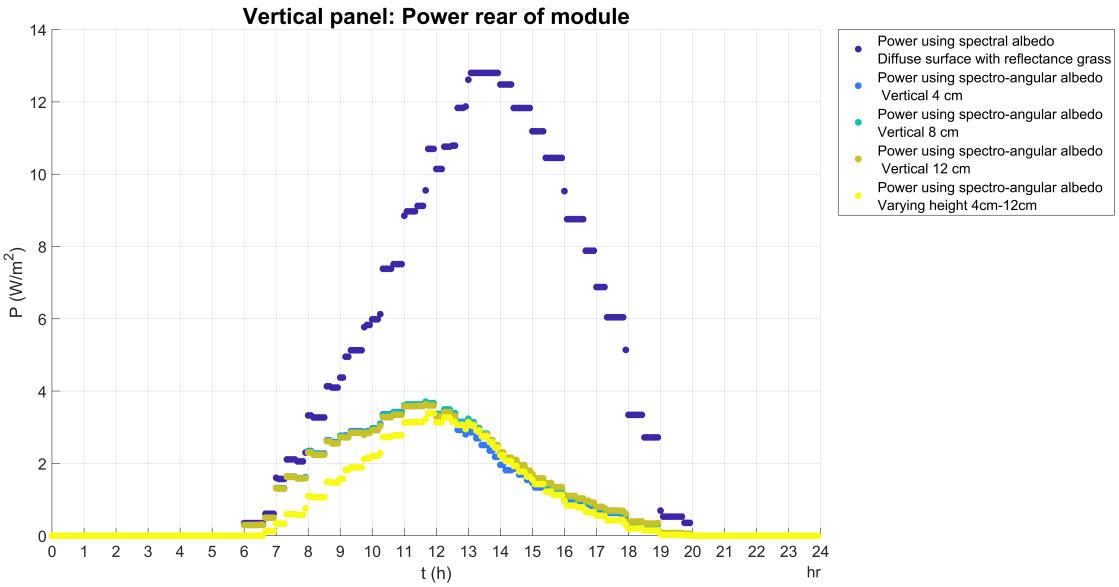


(a) Power output when surrounded by the vertical grass samples.

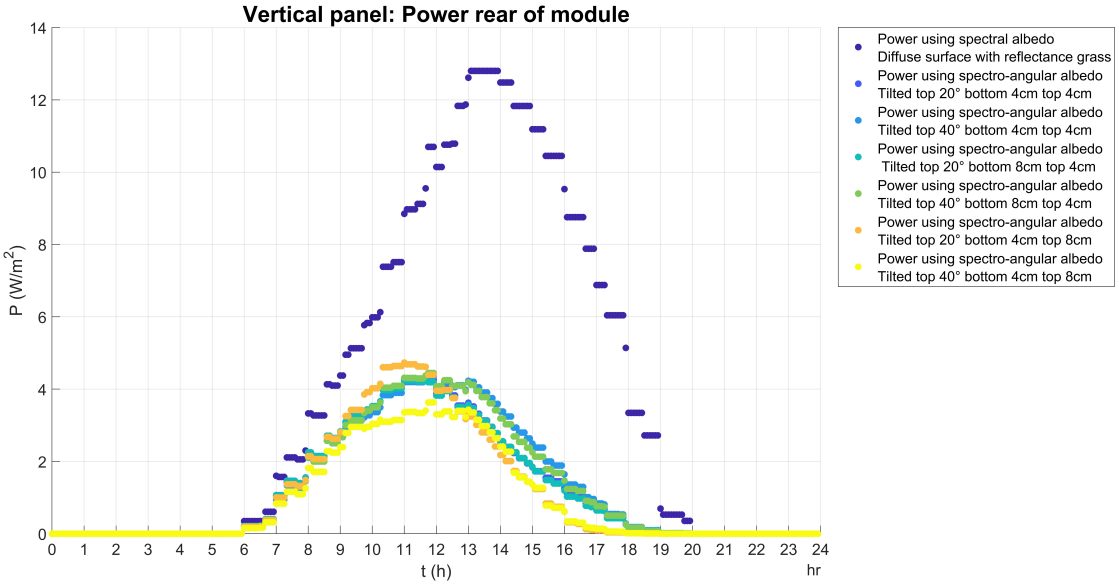


(b) Power output when surrounded by the tilted grass samples.

Figure 53: Power output due to indirect light on the front of a east-west vertically placed bifacial solar panel.

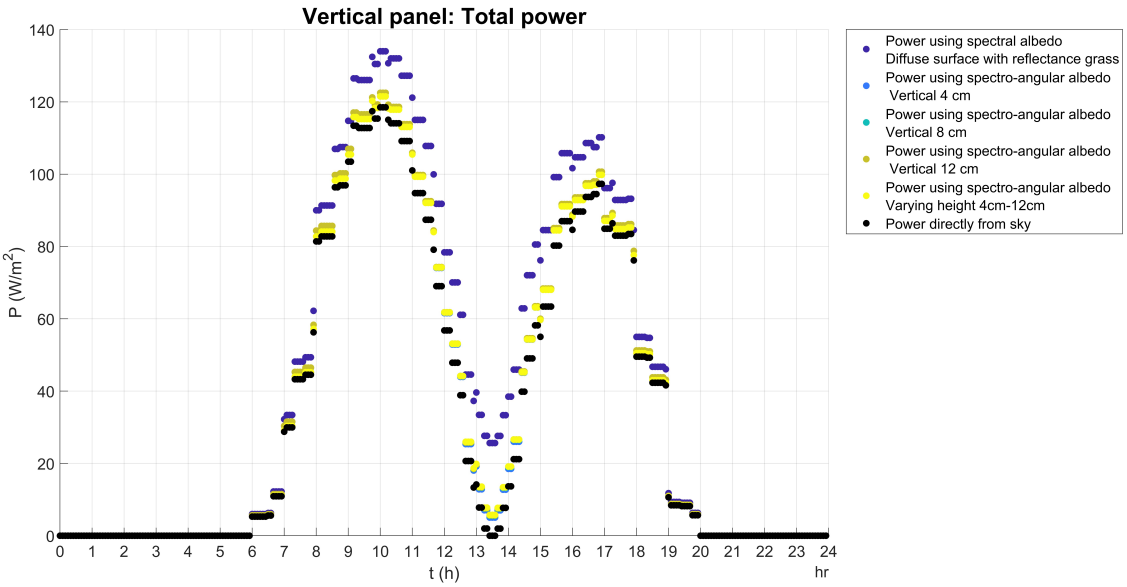


(a) Power output when surrounded by the vertical grass samples.

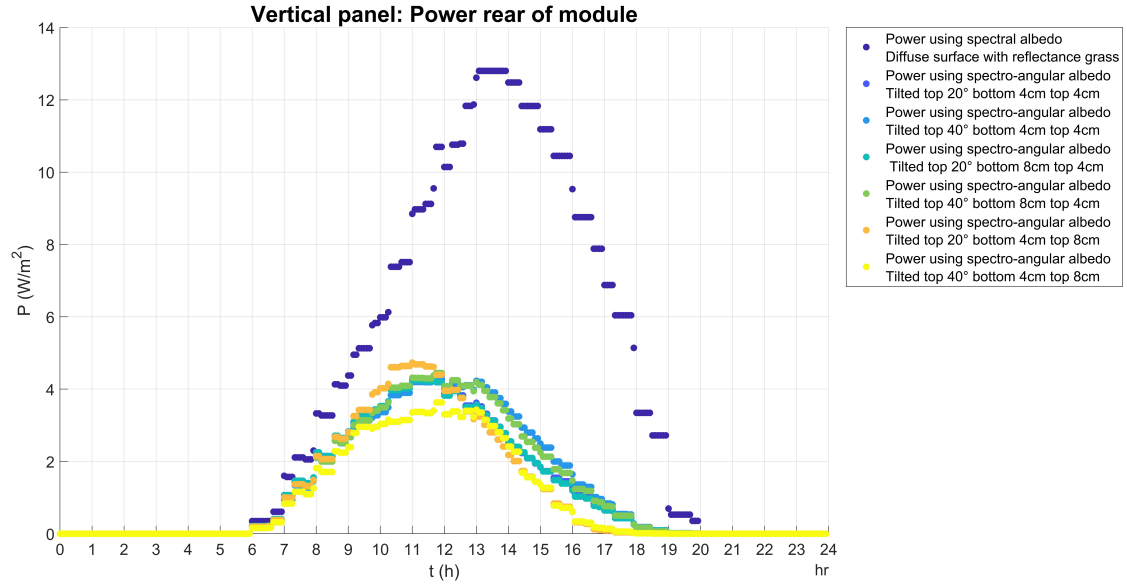


(b) Power output when surrounded by the tilted grass samples.

Figure 54: Power output due to indirect light on the rear of a east-west vertically placed bifacial solar panel.



(a) Power output when surrounded by the vertical grass samples.



(b) Power output when surrounded by the tilted grass samples.

Figure 55: Power output due to direct and indirect light on a east-west vertically placed bifacial solar panel.

C Guide code Monte Carlo ray tracing model

In this appendix, the focus lies on how the code can be used to reproduce the results in this report. With this information, the model can also be used to simulate the BRDF of new samples.

C.1 Overview functions and scripts

The code consists of two folders: *RayTracing* in which all scripts and functions to perform ray tracing are saved. With the code in this folder, a library of BRDFs can be generated using one of the *analyze* scripts in combination with *addToLibrary*. In the folder *AnalyzeLibrary*, the files in the library can be plotted. Figure 56 gives an overview of the functions and scripts in these two folders. The code can be found in appendix D.

Figure 56: Overview of all functions and scripts in the folders *RayTracing* and *AnalyzeLibrary*. The code can be found in appendix D.

C.2 Tracing a ray through the detection dome

Displaying one or multiple rays can be done by setting the parameteres in *setSettingsVisualisation* as desired. The parameters have the following effect:

- rays_visualised is a list specifying which ray number is plotted.
- show_diffuse_reflection shows, if true, the possible directions for the ray to go in at every intersection of the ray with a plane, like displayed in figure 21. By setting plot_invalid_ray_directions = true in *calculateNewRayDirection*, invalid directions are also shown like in figure 12b.
- show_ray_until_intersection traces, if true, the ray numbers in rays_visualised through the detection dome like in figure 5.
- show_incoming_light shows, if true, the beam of light falling on the sample like in figure 9.
- radius_detection_dome sets the radius that the detection dome has in the plots. This allows for seeing the sample better.

A ray can be traced step by step (from intersection to intersection) by putting two breakpoints below the following lines in the function *calculateBRDFgrass2*:

- if sorted_planes_index(k_plane) == 1 % if ray hits the detection dome
- if hit == true % if ray hits surface

C.3 Calibration figures

C.3.1 Ideal Lambertian

As explained in section 3.7.1, in the Lambertian case, some shortcuts can be taken - all rays can fall at $(x, y) = 0$. Therefore, the code can be much simpler than *calculateBRDFgrass* which saves computational time and makes it easier to check the foundation of the code.

Since some of the steps that are in the full ray tracer code are skipped, the input has to be checked with caution. For example, since the intersection point of the ray with the plane can be entered directly, it has to manually be adjusted to ensure the ray intersects at the right point. In *plotLambertianReflection*, a ray will always intersect - which is not the case in *calculateBRDFgrass*. Specifically, there are two situations for which *plotLambertianReflection* and *analyzeLambertianReflection* give radiance as output, but *calculateBRDFgrass* does not: 1) the plane is vertical and the light comes from $\theta = 90°$ 2) the light shines from any ϕ_{source} to a blade whose tangent line is parallel to ϕ_{source}. The function *calculateBRDFgrass* does not give output for these cases because it cannot compute an intersection time and thus the rays do not intersect with the plane (see section 3.4).

Recreating figures 16, 17 and 18

1. Run the function *analyzeLambertianReflection* for $accuracy\_detection\_deg = 0.5$ (β in this report) for the desired number of rays. The number of rays to run the program for can be specified in the list number_of_rays_list. Set $a\_dome\_inverse = 0$ ($\frac{1}{a_{dome}}$ in this report) to ensure the rays are shot to the middle of the detection dome. Set $number\_of\_repeating\_runs > 1$ to repeat the calculation and eventually average. Check that the results are saved.

2. Run the function *convergenceAnalysisLambertianReflector*. Ensure that the values of *accuracy_detection_deg* = 0.5, number_of_rays_list and *number_of_repeating_runs* > 1 are the same as the input to *analyzeLambertianReflection*. The function loads the radiance data generated by *analyzeLambertianReflection*. Therefore, check if it retrieves the right files. Set make_comparison_plots_n = true and make_comparison_plots_a_dome = false. The function then checks the energy conservation again and generates among other plots figures 16, 17 and 18.

Recreating figure 19 and 20

1. Run the function *analyzeLambertianReflection* for *accuracy_detection_deg* = 3 (β in this report) for *number_of_rays_list* = 1e6 rays. Set *a_dome_inverse* ($\frac{1}{a_{dome}}$ in this report) to a list of desired values. For example, *a_dome_inverse* = [0:1:10] shoots the rays in the middle of the detection dome when *a_dome_inverse* = 0 and to the middle of the detection dome when *a_dome_inverse* = 2 (since in the code, $r_{dome} = 1$). Set *number_of_repeating_runs* > 1 to repeat the calculation and eventually average. Set investigate_direction to either vertical or horizontal depending on whether figure 19 or 20 is to be reproduced. Check that the results are saved.
2. Run the function *convergenceAnalysisLambertianReflector*. Ensure that the values of *accuracy_detection_deg* = 0.5, number_of_rays_list, *number_of_repeating_runs* > 1 and *a_dome_inverse* are the same as the input to *analyzeLambertianReflection*. The function loads the radiance data generated by *analyzeLambertianReflection*. Therefore, check if it retrieves the right files. Set make_comparison_plots_n = false and make_comparison_plots_a_dome = true. The function then checks the energy conservation again and generates figure 19 or 20 depending on investigate_direction in *analyzeLambertianReflection*.

C.3.2 Grass reflector

The goal is to run *analyzeCalibrationGrass* and plot its results.

1. Set the appropriate settings in the files in *RayTracer >> set*. Used for this specific figure:
   - *setSettingsVisualisation*: rays_visualised and wavelengths_visualised empty lists and the other variables Boolean False.
   - *setSettingsSaveResults* save_detection_dome = False and save_results = True. Input name_to_save_brdf irrelevant because this is overwritten in *analyzeCalibrationGrass*.
   - *setSettingsSample* radius_sample = 1, ground_surface = "Blackbody", blade_geometry = "VerticalSameHeight", grass_type = "GrassRussellEtAl2017". The other variables are overwritten in *analyzeCalibrationGrass*.
   - *setSettingsReturnedData* All variables false.
   - *setSettingsRayDetection* accuracy_detection_deg = 6, ratio_sphere_over_sample_radius = 1e8, sphere_center = [0;0;0]
   - *setSettingsIncidentLight* sun_azimuth_deg = 0 , wavelengths = [300:10:1100], number_of_unique_rays = 1e5 , number_of_repeated_rays = 1 . The other variables are overwritten in *analyzeCalibrationGrass*.
2. Open *analyzeCalibrationGrass* and adapt the settings in the script. For figure 25 the following settings have been used:

- elevation_list = $[1e-6, 10:10:90]$
- a_illuminated_list = [1 0.75 0.5 0.25]
- r_illuminated = 2.5 (in cm)
- width_blade_set = 0.5
- height_blade_set = 4 (in cm)
- num_blades_set = 45 (in cm)
- run_number_of_times = 10

3. Check under what name the results are saved. The results are saved in the current folder (to check the current folder, enter *pwd* in the command window) with the filename as specified by the variable save_name.
4. Run *addToLibrary*, taking as input the resulting *.mat* files for all runs. The output is a *.mat* file in which the structure *data_library* contains the detection dome azimuth and elevation angles. Furthermore, for every angle, it contains the mean spectral albedo and mean BRDF of all the runs.
5. Open *AnalyzeLibrary >> plotFromLibrary*. Set the appropriate settings using the structure settings_plotter. Ensure that the variables in the structure settings_plotter are correct.

C.4 Filling the BRDF library

The goal is to obtain a library with for each sample type and desired elevation angle, a BRDF averaged over a certain number of runs.

1. Set the appropriate settings in the files in *RayTracer >> set*.
2. Open *analyzeBRDFMakeLibrary*. Check under what name the results are saved. The results are saved in the current folder (to check the current folder, enter *pwd* in the command window) with the filename as specified by the variable save_name.
3. Run *analyzeBRDFMakeLibrary*. For every elevation angle, a total number of number_of_runs will be created. The results are saved in the file *ResultsRayTracing*.
4. To average the BRDFs and albedos of the number_of_runs number of files, open *addToLibrary*. Ensure that the files created by *analyzeBRDFMakeLibrary* are retrieved correctly. Choose a location to save the averaged BRDFs and albedos by changing folder_to_add and name_individual_files as desired. Also ensure that elevation_angles and number_of_runs correspond to those of the file names.
5. Run *addToLibrary*. The output is a *.mat* file in which the structure *data_library* contains the detection dome azimuth and elevation angles. Furthermore, for every angle, it contains the mean spectral albedo and mean BRDF.

C.5 Analysing BRDF library

C.5.1 Plotting the samples

Recreating figure 27

1. Open the script *plotSample*. In the script, change data_to_plot such that the saved data of the desired file is loaded. For example, a file from the library can be loaded. In this data, the structure surfaces exists in which all surfaces are saved. Running the script for the right file gives a subfigure like figure 27 as output.

C.5.2 Plotting BRDF and albedo

1. Open *AnalyzeLibrary* >> *plotFromLibrary*. Set the appropriate settings using the structure settings_plotter. Ensure that the variables in the structure settings_plotter are correct.
2. Run *plotFromLibrary*. To not generate all figures at the same time, Matlabs 'run section' button can be used. The appropriate figures will be generated. An overview:
   - *plotFromLibraryBRDFperElevationAngle* plots the mean BRDF from the library per elevation able in a heatmap with elevation and azimuth angles of the dome on the axes. When in this function plot_3D = true, a three dimensional plot is generated. The BRDF is plotted for the wavelength specified by desired_wavelengthin the function. In this way, figure 28 is made.
   - *plotFromLibrarySpectralAlbedoPerElevationAngle* plots the albedo on the y-axis and the wavelengths on the x-axis. There is a line for every elevation angle of angles_to_run. This results in figure 31.
   - *plotFromLibraryAlbedoComparisonSamples* plots for one wavelength (specified by desired_wavelength in the function) for every sample and for every angle the albedo. To make this figure, the variable for_comparison needs to be filled. This results in figure 32.
   - *plotFromLibrary2DPolarPlot* plots a cross-section of a BRDF for all elevation angles of the source and for one wavelength (specified by desired_wavelength in the function). This gives figure 30.

D Code Monte Carlo ray tracing model

D.1 Ray tracing

D.1.1 Set

setSettingsSample

```
function settings_ray_tracer = ...
    setSettingsSample(settings_ray_tracer,variable_to_adapt, ...
    new_value,variable_to_adapt_2,new_value_2,...
variable_to_adapt_3,new_value_3,...
variable_to_adapt_4,new_value_4)


%% Ground material:
settings_ray_tracer.sample.radius_sample  = 1 ;

% Settings ground material:
%    -   Lambertian                  - All light is diffusely reflected
%    -   Blackbody                   - All light is absorbed
%    -   SoilAlfisolPaleustalf       - Light is spectrally reflected
%  important for reflectance data, see retrieveReflectanceSurface
settings_ray_tracer.sample.ground_surface = "Blackbody";

%% Blades:

settings_ray_tracer.sample.number_of_blades = 180;%4516; % 45 when 1r = 2.5 cm

% Settings blade geometry
% (For exact specifications see function generateSample)
%    -  "VerticalSameHeight"         - All blades point vertically upwards
%                                      They have the same height: X cm.
%    -  VerticalVaryingHeight       - All blades point vertically upwards
%                                      There is a variation in height.
%    -  VerticalAndTiltedTop         - Blades consist of 2 equal segments which
%                                       are equal in length. The lower half
%                                       is vertical. The top half is tilted.
settings_ray_tracer.sample.blade_geometry = "VerticalSameHeight";


settings_ray_tracer.sample.blade_width = 0.1; % measured in radius_sample

% Height
%   set to zero if reflector is at ground height.
settings_ray_tracer.sample.blade_height = 0.8; % measured in radius_sample ...
    (can be [0.1 1]) to vary height between 0.1r and 1r
settings_ray_tracer.sample.blade_tilt = [-10,10]; % measured in deg from ...
    xy-normal
settings_ray_tracer.sample.height_tilted_top = 0.8;

% Setting grass type
%    - GrassRussellEtAl2017         - Data from 10.1109/JPHOTOV.2017.2756068
%    - GrassKokalyEtAl2017          - Data from ...
    https://crustal.usgs.gov/speclab/data/HTMLmetadata/ ...
    LawnGrass_GDS91b_shifted_3nm_BECKa_AREF.html
% important for reflectance data, see retrieveReflectanceSurface
settings_ray_tracer.sample.grass_type = "GrassRussellEtAl2017";
```

```
%% Overwrite:
% simply overwrite the variable which has to be adapted
if nargin > 1 && nargin < 4
    settings_ray_tracer.sample.(variable_to_adapt) = new_value;
elseif  nargin > 3 && nargin < 6
    settings_ray_tracer.sample.(variable_to_adapt) = new_value;
    settings_ray_tracer.sample.(variable_to_adapt_2) = new_value_2;
elseif nargin > 5 && nargin < 8
    settings_ray_tracer.sample.(variable_to_adapt) = new_value;
    settings_ray_tracer.sample.(variable_to_adapt_2) = new_value_2;
    settings_ray_tracer.sample.(variable_to_adapt_3) = new_value_3;
elseif nargin == 9
    settings_ray_tracer.sample.(variable_to_adapt) = new_value;
    settings_ray_tracer.sample.(variable_to_adapt_2) = new_value_2;
    settings_ray_tracer.sample.(variable_to_adapt_3) = new_value_3;
    settings_ray_tracer.sample.(variable_to_adapt_4) = new_value_4;
end
end
```

setSettingsIncidentLight

```
function settings_ray_tracer = ...
    setSettingsIncidentLight(settings_ray_tracer,variable_to_adapt, ...
    new_value,variable_to_adapt_2,new_value_2)


% Settings incoming light
settings_ray_tracer.incident_light.sun_azimuth_deg = 0; % in degree
settings_ray_tracer.incident_light.sun_elevation_deg = 45; %%14.04;  % in degree
settings_ray_tracer.incident_light.wavelengths = [300 : 10 : 1100];
settings_ray_tracer.incident_light.number_of_unique_rays = 105;
settings_ray_tracer.incident_light.number_of_repeated_rays = 1;
settings_ray_tracer.incident_light.ratio_illuminated_over_sample_radius = 0.5;

% simply overwrite the variable which has to be adapted
if nargin > 1 && nargin <5
    settings_ray_tracer.incident_light.(variable_to_adapt) = new_value;
elseif  nargin == 5
    settings_ray_tracer.incident_light.(variable_to_adapt) = new_value;
    settings_ray_tracer.incident_light.(variable_to_adapt_2) = new_value_2;
end



end
```

setSettingsRayDetection

```
function settings_ray_tracer = ...
    setSettingsRayDetection(settings_ray_tracer,variable_to_adapt, new_value)


% Settings ray detection
settings_ray_tracer.ray_detection.accuracy_detection_deg = 4.5;
```

```
settings_ray_tracer.ray_detection.ratio_sphere_over_sample_radius = 1e8; %1e8
settings_ray_tracer.ray_detection.sphere_center =  [0;0;0];

% simply overwrite the variable which has to be adapted
if nargin > 1
    settings_ray_tracer.ray_detection.(variable_to_adapt) = new_value;
end



end
```

setSettingsReturnedData

```
function settings_ray_tracer = ...
    setSettingsReturnedData(settings_ray_tracer,variable_to_adapt, new_value)

% Detection dome
settings_ray_tracer.returned_data.return_centers_detection_dome = true;

% Radiance and BRDF
settings_ray_tracer.returned_data.return_brdf = true;
settings_ray_tracer.returned_data.return_radiance = true;

% Error calculation
settings_ray_tracer.returned_data.return_RMSE_lambertian = true;% compute ...
    the root mean squared compared to Lambertian reflector

% Albedo
settings_ray_tracer.returned_data.return_albedo = true;

% Settings
settings_ray_tracer.returned_data.return_settings = true;

% simply overwrite the variable which has to be adapted
if nargin > 1
    settings_ray_tracer.returned_data.(variable_to_adapt) = new_value;
end



end
```

setSettingsSaveResults

```
    function settings_ray_tracer = ...
        setSettingsSaveResults(settings_ray_tracer,variable_to_adapt, new_value)

% Settings save results
settings_ray_tracer.save_results.save_detection_dome = false;
settings_ray_tracer.save_results.save_results = true;
settings_ray_tracer.save_results.name_to_save_brdf = "LambertianNoBlades";

% simply overwrite the variable which has to be adapted
if nargin > 1
```

```
    settings_ray_tracer.save_results.(variable_to_adapt) = new_value;
end



end
```

setSettingsVisualisation

```
function settings_ray_tracer = ...
    setSettingsVisualisation(settings_ray_tracer,variable_to_adapt, new_value)


% Settings visualisation
settings_ray_tracer.visualisation.rays_visualised = []; % list of ray ...
    numbers that should be traced
settings_ray_tracer.visualisation.show_diffuse_reflection = false;
settings_ray_tracer.visualisation.show_ray_until_intersection = false;
settings_ray_tracer.visualisation.show_incoming_light = false;
settings_ray_tracer.visualisation.wavelengths_visualised = [];%[] = none

settings_ray_tracer.visualisation.radius_detection_dome = 2;%[] = none (only ...
    for visualisation)

% simply overwrite the variable which has to be adapted
if nargin > 1
    settings_ray_tracer.visualisation.(variable_to_adapt) = new_value;
end



end
```

D.1.2 Save

saveRaySpecifications

```
function [rays] = saveRaySpecifications(rays, ray_starting_point, ...
    ray_direction,ray_magnitude, specific_ray )
%=========================================================================
% Saves the starting point, direction and magnitude of a specific ray
% in the structure rays
%=========================================================================
% Input:  - rays                    structure       Contains information
%                                                   about the rays
%         - ray_starting_point      3x1 matrix      Starting point [x;y;z]
%         - ray_direction           3x1 matrix      Direction vector [x;y;z]
%         - ray_magnitude           1xn matrix      Magnitude ray for n
%                                                   wavelengths
%         - specific_ray            string          Ray ID
% ------------------------------------------------------------------------
% Output: - rays                    structure       Contains information
%                                                   about rays including
%                                                   the specific ray
%=========================================================================
```

```
rays.(specific_ray).start = ray_starting_point;
rays.(specific_ray).direction = ray_direction;
rays.(specific_ray).magnitude = ray_magnitude;

end
```

saveSurfaceSpecifications

```
function [surfaces] = saveSurfaceSpecifications(surfaces, ...
    surface_boundaries, surface_name, surface_normal, surface_type)
%==========================================================================
% Saves the corners, normal and surface type of a specific plane (plane
% name) in the structure planes
%==========================================================================
% Input:  - surfaces            structure        Contains information on ...
    surfaces
%         - surface_boundaries  struct           Contains at least the shape ...
    of the
%                                                surface.
%                                                - If the shape is a "Square",
%                                                  contains the corners of the
%                                                  surface.
%                                               - If the shape is a "Circle",
%                                                 contains the radius, center
%                                                 and height.
%         - surface_name       string           Plane ID
%         - surface_normal     3x1 matrix       Normal direction of the plane
%         - surface_type       string           Type of surface
% --------------------------------------------------------------------------
% Output: - surfaces        structure       Contains information
%                                           about surfaces including
%                                           the specific surface (surface name)
%==========================================================================
% Update log:
%   04/05/2023  Added circular option.
%==========================================================================

surfaces.(surface_name).normal = surface_normal;
surfaces.(surface_name).type = surface_type ;
surfaces.(surface_name).surface_boundaries = surface_boundaries;
end
```

D.1.3 Retrieve

retrieveSettingsSample

```
function [radius_sample, ground_surface, number_of_blades, blade_geometry, ...
    grass_type,blade_width , blade_height, blade_tilt,height_tilted_top] = ...
    retrieveSettingsSample(settings_ray_tracer)

% Settings sample
radius_sample  = settings_ray_tracer.sample.radius_sample;
ground_surface = settings_ray_tracer.sample.ground_surface;
number_of_blades = settings_ray_tracer.sample.number_of_blades;
```

```
blade_geometry = settings_ray_tracer.sample.blade_geometry;
grass_type = settings_ray_tracer.sample.grass_type;

blade_width = settings_ray_tracer.sample.blade_width;%0.02; % measured in ...
    radius_sample
blade_height = settings_ray_tracer.sample.blade_height;%0.16; % measured in ...
    radius_sample (can be [0.1 1]) to vary height between 0.1r and 1r
blade_tilt = settings_ray_tracer.sample.blade_tilt; % measured in deg from ...
    xy-normal

height_tilted_top = settings_ray_tracer.sample.height_tilted_top;
end
```

retrieveSettingsIncidentLight

```
function [sun_azimuth_deg, sun_elevation_deg , wavelengths, ...
    number_of_unique_rays, ...
    number_of_repeated_rays,ratio_illuminated_over_sample_radius] = ...
    retrieveSettingsIncidentLight(settings_ray_tracer)

% Settings incoming light
sun_azimuth_deg = settings_ray_tracer.incident_light.sun_azimuth_deg;
sun_elevation_deg = settings_ray_tracer.incident_light.sun_elevation_deg;
wavelengths = settings_ray_tracer.incident_light.wavelengths;
number_of_unique_rays = ...
    settings_ray_tracer.incident_light.number_of_unique_rays;
number_of_repeated_rays = ...
    settings_ray_tracer.incident_light.number_of_repeated_rays;
ratio_illuminated_over_sample_radius = ...
    settings_ray_tracer.incident_light.ratio_illuminated_over_sample_radius;

end
```

retrieveSettingsRayDetection

```
function [accuracy_detection_deg, ratio_sphere_over_sample_radius, ...
    sphere_center] = retrieveSettingsRayDetection(settings_ray_tracer)

% Settings ray detection
accuracy_detection_deg = ...
    settings_ray_tracer.ray_detection.accuracy_detection_deg;  % in degrees
ratio_sphere_over_sample_radius = ...
    settings_ray_tracer.ray_detection.ratio_sphere_over_sample_radius;
sphere_center = settings_ray_tracer.ray_detection.sphere_center;
end
```

retrieveSettingsReturnedData

```
function [return_brdf,return_radiance,return_RMSE_lambertian, ...
    return_centers_detection_dome,return_albedo,return_settings] = ...
    retrieveSettingsReturnedData(settings_ray_tracer)

%Settings return
```

```
return_brdf = settings_ray_tracer.returned_data.return_brdf;
return_radiance = settings_ray_tracer.returned_data.return_radiance;
return_RMSE_lambertian = ...
    settings_ray_tracer.returned_data.return_RMSE_lambertian;

% Detection dome
return_centers_detection_dome = ...
    settings_ray_tracer.returned_data.return_centers_detection_dome;

% Albedo
return_albedo = settings_ray_tracer.returned_data.return_albedo;

% Settings
return_settings = settings_ray_tracer.returned_data.return_settings;

end
```

retrieveSettingsSaveResults

```
function [save_detection_dome, name_to_save_brdf, save_results] = ...
    retrieveSettingsSaveResults(settings_ray_tracer)



% Settings save results
save_detection_dome = settings_ray_tracer.save_results.save_detection_dome;
name_to_save_brdf = settings_ray_tracer.save_results.name_to_save_brdf;
save_results = settings_ray_tracer.save_results.save_results;

end
```

retrieveSettingsVisualisation

```
function [rays_visualised, ...
    show_diffuse_reflection,show_ray_until_intersection,wavelengths_visualised,...
show_incoming_light,radius_detection_dome] = ...
    retrieveSettingsVisualisation(settings_ray_tracer);

% Settings visualisation
rays_visualised =settings_ray_tracer.visualisation.rays_visualised;
show_diffuse_reflection ...
    =settings_ray_tracer.visualisation.show_diffuse_reflection;
show_ray_until_intersection = ...
    settings_ray_tracer.visualisation.show_ray_until_intersection;
wavelengths_visualised = ...
    settings_ray_tracer.visualisation.wavelengths_visualised;
show_incoming_light = settings_ray_tracer.visualisation.show_incoming_light;
radius_detection_dome = settings_ray_tracer.visualisation.radius_detection_dome;
end
```

retrieveRaySpecifications

```
function [ray_starting_point, ray_direction, ray_magnitude] = ...
    retrieveRaySpecifications(rays, specific_ray)
%=========================================================================
% Retrieves the starting point, direction and magnitude of a specific ray
% from the structure rays
%=========================================================================
% Input:  - rays                    structure       Contains information
%                                                   about the rays
%         - specific_ray            string          Ray ID
% -------------------------------------------------------------------------
% Output: - ray_starting_point      3x1 matrix      Starting point [x;y;z]
%         - ray_direction           3x1 matrix      Direction vector [x;y;z]
%         - ray_magnitude           1xn matrix      Magnitude ray for n
%                                                   wavelengths
%=========================================================================

ray_starting_point = rays.(specific_ray).start ;
ray_direction = rays.(specific_ray).direction ;
ray_magnitude = rays.(specific_ray).magnitude ;

end
```

retrieveSurfaceSpecifications

```
function [surface_boundaries, surface_normal] = ...
    retrieveSurfaceSpecifications(surfaces, surface_name)
%=========================================================================
% Retrieves the surface boundaries and normal of a specific plane (plane
% name) from the structure planes
%=========================================================================
% Input:  - surfaces           structure      Contains information on planes
%         - surface_name       string         Plane ID
% -------------------------------------------------------------------------
% Output: - surface_boundaries structure      Contains corners of the plane.
%         - surface_normal     3x1 matrix     Normal direction of the plane
%=========================================================================
% Update log:
%   04/05/2023  Added circular option.
%=========================================================================

surface_boundaries = surfaces.(surface_name).surface_boundaries;
surface_normal = surfaces.(surface_name).normal;
end
```

retrieveReflectanceSurface

```
function reflectance_surface  = ...
    retrieveReflectanceSurface(reflectance_surface, surface_type,wavelengths)
%=========================================================================
% Puts spectral reflectance (reflectance for given wavelengths) of a
% surface in a structure.
%=========================================================================
% Input:  - reflectance_surface struct      Structure with spectral
%                                           reflectance
%         - surface_type     string         Type of the surface
```

```
%         - wavelengths     1 x n matrix    Array with all wavelengths for
%                                           which a magnitude has to be
%                                           generated
% -------------------------------------------------------------------------
% Output: - reflectance_surface struct      Structure with spectral
%                                           reflectance of surface_type
%==========================================================================
% Internal functions
%         - checkInterpolationPossible      Checks if the desired
%                                           wavelengths are within the
%                                           bounds of the wavelengths
%                                           from the data
%==========================================================================
% Notes:  - Ensure used data units are correct in interpolation
%==========================================================================
% Debugging features
%         - checkInterpolationPossible to ensure data retrieved from tables
%           can be interpolated.
%==========================================================================
% Update log
%       23/05/2023 Added GrassRussellEtAl2017
%       10/05/2023 Separated from calculateNewRayMagnitude to improve
%                  computational costs.
%==========================================================================

if surface_type == "Lambertian"
            reflectance_surface.(surface_type) = 1;

elseif surface_type == "Blackbody"
            reflectance_surface.(surface_type) = 0;

elseif surface_type == "SoilAlfisolPaleustalf"
            %ray_magnitude = ray_magnitude;%* 0.8;
            % Retrieve and assign data
            table_reflectance = ...
                readtable('ReflectanceSoilAlfisolPaleustalf.xlsx',...
            'Sheet','Sheet1','Range','A1:B2844');
            wavelengths_table_nm = table2array(table_reflectance(1:end,1)) * ...
                100; % Unit: nm
            reflectance_table = table2array(table_reflectance(1:end,2))/100;

            % Error check
            [wavelengths_table_nm, reflectance_table] = ...
                checkInterpolationPossible(wavelengths_table_nm, wavelengths, ...
                surface_type)

            % Interpolate
            reflectance_surface.(surface_type) = ...
                interp1(wavelengths_table_nm,reflectance_table,wavelengths);
elseif surface_type == "GrassRussellEtAl2017"
            % compute correction spectral reflectance
            table_reflectance = ...
                readtable('ReflectanceGrassRussellEtAl2017.xlsx',...
            'Sheet','Sheet1','Range','A1:B261');
            wavelengths_table_nm = table2array(table_reflectance(1:end,1)); ...
                % Unit: nm
            reflectance_table = table2array(table_reflectance(1:end,2));

            % Error check
            [wavelengths_table_nm, reflectance_table] = ...
```

```
            checkInterpolationPossible(wavelengths_table_nm, wavelengths, ...
            surface_type, reflectance_table);


        % Interpolate
        reflectance_surface.(surface_type) = ...
            interp1(wavelengths_table_nm,reflectance_table,wavelengths);

elseif surface_type == "GrassKokalyEtAl2017"
        % compute correction spectral reflectance
        table_reflectance = ...
            readtable('ReflectanceGrassKokalyEtAl2017.xlsx',..
        'Sheet','Sheet1','Range','A1:B481');
        wavelengths_table_nm = table2array(table_reflectance(1:end,1)) * ...
            1000; % Unit: nm
        reflectance_table = table2array(table_reflectance(1:end,2));

        % Error check
        [wavelengths_table_nm, reflectance_table] = ...
            checkInterpolationPossible(wavelengths_table_nm, wavelengths, ...
            surface_type);

        % Interpolate
        reflectance_surface.(surface_type) = ...
            interp1(wavelengths_table_nm,reflectance_table,wavelengths);

elseif surface_type == "blade14"
        reflectance_surface.(surface_type) = 0.009;
end

%Debugging code
fprintf("When a ray of light falls upon %s, %.2f \% of the light is absorbed ...
    immediately (averaged over wavelength).\n", surface_type, 100*(1 - ...
    sum(reflectance_surface.(surface_type)/...
numel(reflectance_surface.(surface_type)))  ) )

%% Internal functions
    function [wavelengths_table_nm_new, reflectance_table_new] = ...
        checkInterpolationPossible(wavelengths_table_nm, wavelengths, ...
        surface_type,reflectance_table)
    % Error check: can interpolation of data in the table be used?

    % Checking boundaries of data
    if min(wavelengths) < min(wavelengths_table_nm)
        error(sprintf("The desired minimum wavelength cannot be retrieved ...
            from the current %s data.",surface_type))
    elseif  max(wavelengths) > max(wavelengths_table_nm)
        error(sprintf("The desired maximum wavelength cannot be retrieved ...
            from the current %s data.",surface_type))
    end

    % Ensuring unique wavelengths (otherwise integration does not work)
    [wavelengths_table_nm_new, indices_unique] = unique(wavelengths_table_nm);
    reflectance_table_new = reflectance_table(indices_unique);
end


end
```

D.1.4 Generate

generateSample2

```
function surfaces = generateSample(settings_ray_tracer)

%% Settings
plot_sample = false;

%% Initialization
reflectance_surface = struct();
surfaces = struct();
surfaces.individual_surfaces = struct();

% retrieve relevant variables out of settings
number_of_blades = settings_ray_tracer.sample.number_of_blades;
radius_sample = settings_ray_tracer.sample.radius_sample ;
ground_surface = settings_ray_tracer.sample.ground_surface ;
blade_geometry = settings_ray_tracer.sample.blade_geometry;
grass_type = settings_ray_tracer.sample.grass_type;

blade_width = settings_ray_tracer.sample.blade_width; % measured in ...
    radius_sample
blade_height = settings_ray_tracer.sample.blade_height; % measured in ...
    radius_sample (can be [0.1 1]) to vary height between 0.1r and 1r
blade_tilt = settings_ray_tracer.sample.blade_tilt; % measured in deg from ...
    xy-normal

wavelengths = settings_ray_tracer.incident_light.wavelengths ;


blades_list = repmat([grass_type], 1, number_of_blades);
surfaces_list = [ground_surface, blades_list];

radius_dome_for_visualisation = ...
    settings_ray_tracer.visualisation.radius_detection_dome;

% Retrieve reflectance of every surface
for k_surface = 1:length(unique(surfaces_list))
    reflectance_surface  = retrieveReflectanceSurface(reflectance_surface, ...
        surfaces_list(k_surface), wavelengths);
end

if plot_sample == true
    % Plot coordinate system
    plotCoordinateSystem('hemisphere',radius_dome_for_visualisation)
end

% Based on blade geometry, set some variables
additional_surface_specification.blade_width = blade_width;
additional_surface_specification.blade_height = blade_height;


switch blade_geometry
    case "VerticalSameHeight"
            additional_surface_specification.height_type ="VerticalSameHeight";
            tilted_top = false;
    case "VerticalVaryingHeight"
```

```
            additional_surface_specification.height_type = "VaryingHeight";
            tilted_top = false;
     case "VerticalAndTiltedTop"
            additional_surface_specification.height_type = "HalfLength";
            tilted_top = true;
    otherwise
        error("%s is not a valid blade geometry.", blade_geometry)
end

%% Generate surfaces
for k_surface = 1:length(surfaces_list)
    surface_type = surfaces_list(k_surface);
    if k_surface == 1 % is always the ground
        [surface_boundaries] = generateSurface2(radius_sample, "Ground");
    else
        [surface_boundaries] = generateSurface2(radius_sample, "Blade", ...
           additional_surface_specification);
%         surface_boundaries.corners =  [ 2 2 -2 -2 ; 5 -5 -5 5; 5 5 0 0];
%         surface_boundaries.center = ...
   [mean(surface_boundaries.corners(1,:)); ...
   mean(surface_boundaries.corners(2,:));mean(surface_boundaries.corners(3,:))];
%         surface_boundaries.shape = "Rectangle";
    end
    %surface_boundaries.corners = [ 2 -2 -2 2; 2 -2 -2 2 ; 2 2 0 0]; %[1 -3 ...
       -3 1; 4 -1 -1 4 ; 2 2 0 0];%[ 2 -2 -2 2; 2 -2 -2 2 ; 2 2 0 0];%  45 ...
       graden schuin           [ 2 -2 -2  2  ; 1 -4 -4 1; 2 2 0 0]; % schuin

    % Compute normal
    [surface_normal] = calculateNormal(surface_boundaries);

    if plot_sample == true
        % Plot surface and normal
        plotSurface(surface_boundaries)
        plotNormal(surface_normal, surface_boundaries)
    end

    % Save the surface specifications
    surface_name = strcat(surface_type,string(k_surface));
    surfaces.individual_surfaces = ...
       saveSurfaceSpecifications(surfaces.individual_surfaces, ...
       surface_boundaries, surface_name, surface_normal, surface_type);

    % Add tilted top
    if tilted_top == true  && k_surface > 1 % if surface is not the ground
        % Length blade
        length_blade = blade_height;
        % Shape blade
        surface_boundaries.shape = "Rectangle";

        % Corners blade: top corners bottom part become bottom corners top part
        surface_boundaries.corners(1:3,3) = surface_boundaries.corners(1:3,2);
        surface_boundaries.corners(1:3,4) = surface_boundaries.corners(1:3,1);

        % Determine tilt of the top
        tilt_top = blade_tilt(1) + (blade_tilt(2)-blade_tilt(1)) * rand(1); ...
           % max tilt of the top as seen from the vertical xy-normal. Degrees.

        % New height of the top
        z_new = cos(deg2rad(tilt_top)) * length_blade;
        r_new = sin(deg2rad(tilt_top)) * length_blade; % which is in ...
```

```
            direction of the normal of the bottom part - so decompose!

        % Angle between x axis and r_new
        u = [surface_normal(1); surface_normal(2)] .* r_new; % (x;y)

        angle_phi = atan(u(2)./u(1)); %acos(dot(u,v)./ norm(u) ./ norm(v) ); ...
            %atan2(norm(cross(surface_normal,[1; 0; 0])), ...
            dot(surface_normal,[1; 0; 0]));

        % Corners top part of the tilted blade
        surface_boundaries.corners(3,1:2) = ...
            surface_boundaries.corners(3,1:2) + z_new;  % z
        surface_boundaries.corners(1,1:2) = ...
            surface_boundaries.corners(1,1:2) + cos(angle_phi) * r_new ; % x
        surface_boundaries.corners(2,1:2) = ...
            surface_boundaries.corners(2,1:2) + sin(angle_phi) * r_new ; % y

        % Center of the blade
        surface_boundaries.center = [mean(surface_boundaries.corners(1,:)); ...
            mean(surface_boundaries.corners(2,:)); ...
            mean(surface_boundaries.corners(3,:))];

        % Calculate surface normal
        [surface_normal] = calculateNormal(surface_boundaries);

        % Plot surface and normal
        if plot_sample == true
            plotSurface(surface_boundaries)
            plotNormal(surface_normal, surface_boundaries)
        end

        % Save the surface specifications
        surface_name = strcat(surface_type,string(length(surfaces_list) -1 + ...
            k_surface));
        surfaces.individual_surfaces = ...
            saveSurfaceSpecifications(surfaces.individual_surfaces, ...
            surface_boundaries, surface_name, surface_normal, surface_type);

    end
end

if tilted_top == true
    surfaces_list = [surfaces_list surfaces_list(2:end)];
end
surfaces.surfaces_list = surfaces_list;
surfaces.reflectance_surface = reflectance_surface;

if plot_sample == true
    surfaces.sample_figure = gcf;
end

end
```

generateSurface2

```
function [surface_boundaries] = generateSurface(sample_size, surface_type, ...
    additional_surface_specification)
%=========================================================================
```

```
% Generates a surface of surface_type placed in the grid defined by
% grid_size_x and grid_size_y
%=========================================================================
% Input:  - sample_size       float         Radius of the sample or half
%                                           of the length of the sample
%         - surface_type      string        Surface type
%         - sample_geometry   string        Geometry sample size: either
%                                           square or circular
% ------------------------------------------------------------------------
% Output: - plane             3x4 matrix    Contains corners of the surface.
%=========================================================================
% External functions used:
%         - computeRotationMatrix   computes rotation matrix. used to
%                                   randomize orientation
%=========================================================================
% Notes: - The square surface is considered as follows:
%
%           1 ------2         Corner 1 : x1, y1, z1
%           |       |         Corner 2 : x2, y2, z2
%           |       |         Corner 3 : x3, y3, z3
%           |       |         Corner 4 : x4, y4, z4
%           4-------3
%
%=========================================================================
% Update log:
%   08/05/2023 Deleted square sample geometry because of computational
%              speed. For blades: added surface_boundaries.center as
%              output.
%   04/05/2023  Added circular option, where ground is a circle and the
%              grass blades are added on this circle
%=========================================================================

%"Circle" sample geometry
 switch surface_type
    case "Ground" % surface type
            % horizontal circle
            surface_boundaries.shape = "Circle";
            surface_boundaries.center = [0;0;0]; %[x;y;z]
            surface_boundaries.radius = sample_size;
            surface_boundaries.height = 0;
    case "Blade" % surface type
            surface_boundaries.shape = "Rectangle";
            r_blade = sample_size .* ...
               additional_surface_specification.blade_width;
            switch additional_surface_specification.height_type
                case "VerticalSameHeight"
                    z_blade = sample_size .* ...
                       additional_surface_specification.blade_height;
                case "VaryingHeight"
                    blade_height_difference = ...
                       additional_surface_specification.blade_height(2)- ...
                       additional_surface_specification.blade_height(1);
                    z_blade = sample_size ...
                       .*additional_surface_specification.blade_height(1) + ...
                       blade_height_difference .* rand(1); % between 1 and 2
                case "TiltedTop"
                    z_blade = sample_size .* ...
                       additional_surface_specification.blade_height;
            end
```

```
            % Determine point B, the point of the center of the grass
            % blade on the sample. Point B is determined by giving it a
            % radius from the centre and an azimuth from the centre of
            % the detection dome.
            % Radius: A radius between 0 and sample_size - 0.5*r_blade
            %         is assigned to ensure no blade exceeds the
            %         sample size.
            radius_from_center_b = (sample_size - 0.5*r_blade) * sqrt(rand(1));
            azimuth_b = rand(1) .*2 .* pi;
            [x_B , y_B , ~] = sph2cart(azimuth_b,0,radius_from_center_b);

            % vertical plane
            x1 = x_B - 0.5*r_blade;
            x2 = x1 + r_blade;
            x3 = x1 + r_blade ;
            x4 = x1;

            y1 = y_B;
            y2 = y_B;
            y3 = y_B;
            y4 = y_B;

            z1 = z_blade;
            z2 = z_blade;
            z3 = 0;
            z4 = 0;

            % rotation around z axis
            rotation_matrix = computeRotationMatrix ([0,0,1], 2*pi * rand(1));
            surface_boundaries.corners = rotation_matrix *  [[x1 x2 x3 x4]; ...
                [y1 y2 y3 y4]; [z1 z2 z3 z4]];
            surface_boundaries.center = ...
                [mean(surface_boundaries.corners(1,:)); ...
                mean(surface_boundaries.corners(2,:)); ...
                mean(surface_boundaries.corners(3,:))];

    case "blade14" % surface type
            surface_boundaries.shape = "Square";
            surface_boundaries.corners = [[6.0660    5.5368    5.5368    ...
                6.0660];[2.8890    3.3580    3.3580    2.8890];[     0 ...
                        0   10.0000   10.0000]];

        otherwise
        error(sprintf("%s is not a valid surface type.", surface_type))

 end
end
```

generateRayDirection2

```
function [ray_starting_point, ray_direction] = ...
   generateRayDirection(sphere_radius, sun_azimuth, sun_elevation, aim_height)
%=========================================================================
% Generates one ray coming from the sun by setting its direction and
% magnitude.
%=========================================================================
% Input:  - sphere_radius     float    Radius of the detection dome
%         - sun_azimuth       float    Azimuth angle of the sun. Degrees.
```

```
%         - sun_elevation     float    Elevation angle of the sun. Degrees.
%         - aim_height        float    Height that beams are aimed at. cm.
% -------------------------------------------------------------------------
% Output: - ray_starting_point      3x1 matrix     starting point x,y,z
%         - ray_direction           3x1 matrix     direction vector x,y,z
%=========================================================================
% Debugging features:
%         - reverse_ray    boolean     When true, the ray reverses
%                                      direction
%=========================================================================
% Update log:
%         -  17/04/2022     Removed magnitude assignment
%=========================================================================
%% Settings
reverse_ray = false; % Debugging option

%% Locate ray in space
% Creating a vector: p(t) = e + t (s-e) (advance from e along vector (s-e)
% a fractional distance t to find the point p)
% Starting point e (on the detection dome)
[xe, ye, ze] = sph2cart(deg2rad(sun_azimuth), deg2rad(sun_elevation), ...
   sphere_radius);
ray_starting_point= [xe ; ye ; ze];

% Destination point s (on the ground)
ray_destination_point = [0 ; 0 ; aim_height];

% Debugging code: reverse ray
if reverse_ray == true
    ray_starting_point = [xs ; ys ; zs];
    ray_destination_point =  [xe ; ye ; ze];
end

% Direction (s-e)
ray_direction = ray_destination_point - ray_starting_point;

end
```

generateParallelRayDirection2

```
function [ray_starting_point, ray_direction,t_intersect] = ...
   generateParallelRayDirection(spot_radius,ray_direction,sphere_radius,...
sphere_center, aim_height)
%=========================================================================
% Generates a ray parallel to ray_direction
%=========================================================================
% Input:   - spot_radius       float         radius of the spot size on the
%                                           reflector
%         - ray_direction     3x1 matrix    ray direction [x;y;z]
%         - sphere_radius     float         radius of the detection dome
%         - sphere_center     3x1 matrix    center of the detection dome
%         - aim_height        float         Height that beams are aimed at. cm.
% -------------------------------------------------------------------------
% Output: - ray_starting_point  3x1 matrix     Starting point of the ray
%         - ray_direction       3x1 matrix     Direction of the ray
%=========================================================================
% External functions used:
%         - calculateIntersectionTimeWithSphere   compute starting point of
```

```
%                                                    ray on the detection dome
%=========================================================================
% Update log:
%       -  04/05/2023     Option to choose destination point S chosen from
%                         a circle around (x,y) = (0,0) instead of a
%                         square grid
%       -  17/04/2022     Removed magnitude assignment
%
% -------------------------------------------------------------------------
% Update possibilities:
%         - [short desription of updates that could be useful in the future]
%=========================================================================



%% Locate ray in space
% Creating a vector: p(t) = e + t (s-e) (advance from e along vector (s-e)
% a fractional distance t to find the point p)

% Give rays destination point from circular grid
radius_from_center_s = spot_radius * sqrt(rand(1));
azimuth_s = rand(1) .*2 .* pi;
[xs , ys , zs] = sph2cart(azimuth_s,0,radius_from_center_s);
ray_ground_point= [xs ; ys ; zs + aim_height];


% Starting point e (on the detection dome)
t_intersect = ...
    calculateIntersectionTimeWithSphere(sphere_radius,sphere_center, ...
    ray_ground_point, ray_direction, "negative");
ray_starting_point =  ray_ground_point + t_intersect * ray_direction;


end
```

generateRayMagnitude

```
function [ray_magnitude] = generateRayMagnitude(wavelengths)
%=========================================================================
% Generates one ray coming from the sun by setting its direction and
% magnitude.
%=========================================================================
% Input:  - wavelengths     1 x n matrix    Array with all wavelengths for
%                                           which a magnitude has to be
%                                           generated
% -------------------------------------------------------------------------
% Output: - ray_magnitude   1 x n matrix    Array with magnitudes for each
%                                           wavelength
%=========================================================================
% Update log:
%         -  17/04/2022    Created from generateRayDirection
%=========================================================================

ray_magnitude = ones(1, length(wavelengths));


end
```

generateGridSky

```
function [azimuth_grid_dome_deg, elevation_grid_dome_deg] = generateGridSky ...
    (azimuth_range_deg,elevation_range_deg, accuracy_detection_deg)
%=========================================================================
% Generate grid on the sky
%=========================================================================
% Input:  - azimuth_range_deg        matrix      Range of azimuth angles
%                                                considered. Degrees.
%         - elevation_range_deg      matrix      Range of elvation angles
%                                                considered. Degrees.
%         - accuracy_detection_deg   float       Accuracy with which the
%                                                rays will be detected in
%                                                the sky. Degrees.
% ------------------------------------------------------------------------
% Output: - azimuth_grid_dome_deg    matrix      Range of azimuth angles
%                                                on the detection dome
%         - elevation_grid_dome_deg  matrix      Range of elevation angles
%                                                on the detection dome
%=========================================================================
% Implement a proper detection dome grid
% Problem:
%   E.g. detection dome grid with accuracy of 1:
%           azimuth_sky = [0, 1, 2, 3 ....]
%           elevation_sky = [0, 1, 2, 3 ....]
%           At (0,0) only points whose azimuth and elevation are <0.5 are taken
%           into account, which is twice as less as for (1,1) which takes
%           azimuths and elevations >0.5 and <1.5.
% Solution: Design the detection dome grid such that the points are in the
%        middle
%   E.g. detection dome grid with accuracy of 1:
%           azimuth_sky = [0.5, 1.5, 2.5, 3.5 ....]
%           elevation_sky = [0.5, 1.5, 2.5, 3.5 ....]
%           At (0.5,0.5) points whose azimuth and elevation are <1 are taken
%           into account, at (1.5,1.5) points whose azimuth and elevation are
%           >1 and <2, ect.
accuracy_detection_deg_half = accuracy_detection_deg / 2;
azimuth_grid_dome_deg = azimuth_range_deg(1:end-1) + accuracy_detection_deg ...
    / 2;
elevation_grid_dome_deg = elevation_range_deg(2:end) + ...
    accuracy_detection_deg / 2;
end
```

D.1.5 Check

checkBladesInInitiallyIlluminatedArea

```
function [list_initially_illuminated_surfaces] = ...
    checkBladesInInitiallyIlluminatedArea(grass_type,corners, surfaces)

list_initially_illuminated_surfaces = [];
for k_individual_surfaces = 2: length(surfaces.surfaces_list) % for all ...
    surfaces (except ground)
     name_blade = strcat(grass_type,string(k_individual_surfaces));
     center = ...
```

```
        surfaces.individual_surfaces.(name_blade).surface_boundaries.center;

    % If statement consists of 3 parts: checking if the
    % intersection point x, y and z is within the boundaries of the
    % surface
    %       1. Check x: if intersection_point(1,1) >= ...
        min(surface_boundaries.corners(1,:)) && intersection_point(1,1) <= ...
        max(surface_boundaries.corners(1,:))   % check x
    %       2. Check y: if intersection_point(2,1) >= ...
        min(surface_boundaries.corners(2,:)) && intersection_point(2,1)<= ...
        max(surface_boundaries.corners(2,:))    % check y
    %       3. Check z  if intersection_point(3,1) >= ...
        min(surface_boundaries.corners(3,:)) && intersection_point(3,1) <= ...
        max(surface_boundaries.corners(3,:))   % check z
    % For computational speed, these three if statements are
    % condensed into 1 if statement.
    if center(1,1) >= min(corners(1,:)) && center(1,1) <= max(corners(1,:)) ...
        && center(2,1) >= min(corners(2,:)) && center(2,1)<= max(corners(2,:))
            % hit = true!
            list_initially_illuminated_surfaces = ...
                [list_initially_illuminated_surfaces, name_blade];
    end
end



end
```

checkRayHitsSurface

```
function hit = checkRayHitsSurface(surface_boundaries, ...
    intersection_point,normal_surface)
%=========================================================================
% Evaluates if a ray hits a plane by checking if the intersection point of
% the ray and the plane is withing the edges of the surface.
%=========================================================================
% Input:  - surface_boundaries  struct      Contains at least the shape of the
%                                           surface.
%                                           - If the shape is a "Square",
%                                             contains the corners of the
%                                             surface.
%                                           - If the shape is a "Circle",
%                                             contains the radius, center
%                                             and height.
%         - intersection_point  3x1 matrix  Intersection point of ray and
%                                           plane ([x;y;z])
%         - normal_surface      3x1 matrix  Normal of the surface ([x;y;z])
% -------------------------------------------------------------------------
% Output: - hit                 Boolean     When true, the ray hits the
%                                           surface. When false: ray does
%                                           not hit the surface.
%=========================================================================
% Debugging features:
%         - Throws an error when the interssection point does not lie on
%           the plane
%=========================================================================
% Update log:
%   04/05/2023  Added circular option.
```

```
%=========================================================================

%% Initialization
hit = false;

%% Does intersection point  lie on the plane?
% first check for speed: does the intersection point intersect with the
% plane?

point = surface_boundaries.center;
intersection = dot(point - intersection_point , normal_surface);

 % Check if x, y and z coordinates of intersection_point are within or on ...
    the surface edges
if abs(intersection) >= 0 % Extra debugging check if intersection_point is ...
   on the plane
    switch surface_boundaries.shape
        case "Rectangle"
            % If statement consists of 3 parts: checking if the
            % intersection point x, y and z is within the boundaries of the
            % surface
            %       1. Check x: if intersection_point(1,1) >= ...
               min(surface_boundaries.corners(1,:)) && ...
               intersection_point(1,1) <= ...
               max(surface_boundaries.corners(1,:))   % check x
            %       2. Check y: if intersection_point(2,1) >= ...
               min(surface_boundaries.corners(2,:)) && ...
               intersection_point(2,1)<= ...
               max(surface_boundaries.corners(2,:))    % check y
            %       3. Check z  if intersection_point(3,1) >= ...
               min(surface_boundaries.corners(3,:)) && ...
               intersection_point(3,1) <= ...
               max(surface_boundaries.corners(3,:))   % check z
            % For computational speed, these three if statements are
            % condensed into 1 if statement.
            if intersection_point(1,1) >= ...
               min(surface_boundaries.corners(1,:)) && ...
               intersection_point(1,1) <= ...
               max(surface_boundaries.corners(1,:)) && ...
               intersection_point(2,1) >= ...
               min(surface_boundaries.corners(2,:)) && ...
               intersection_point(2,1)<= ...
               max(surface_boundaries.corners(2,:)) && ...
               intersection_point(3,1) >= ...
               min(surface_boundaries.corners(3,:)) && ...
               intersection_point(3,1) <= max(surface_boundaries.corners(3,:))
                    hit = true;
                    %plotRays(ray_starting_point, ray_direction,t_intersect)
            end
        case "Circle"
            [~, ~, intersection_point_radius] = ...
               cart2sph(intersection_point(1), intersection_point(2), ...
               intersection_point(3));
            % If statement consists of two parts:
            %      1. Check radius: if intersection_point_radius  <=  ...
               surface_boundaries.radius % check radius
            %      2. Check height: if intersection_point(3,1) == ...
               surface_boundaries.height     % check height
            % For computational speed, these three if statements are
            % condensed into 1 if statement.
```

```
            if intersection_point_radius  <=  surface_boundaries.radius && ...
                intersection_point(3,1) == surface_boundaries.height
               hit = true;
               %plotRays(ray_starting_point, ray_direction,t_intersect)
            end % end check radius
    end
else
    error("Intersection point does not lie on the plane.")
end


end
```

D.1.6 Calculate

calculateBRDFatShiftedAzimuth

```
function brdf_out = calculateBRDFatShiftedAzimuth(brdf_in, ...
    azimuth_desired,accuracy_detection_deg, azimuths_dome, elevations_dome)
% SHIFTED FROM AZIMUTH 0 DEG

%% Settings
make_3D_plots = false;

%% Initialization
% When using this function on its own:
% data = load('C:\Users\leoni\Documents\SET\7 8  Graduation ...
    Assignment\Code\Grass12cm_2\GrassVertical12cm_72.00_2_2.mat');
% accuracy_detection_deg = ...
    data.settings_ray_tracer.ray_detection.accuracy_detection_deg;
% brdf_in = data.brdf;
% elevations_dome = data.elevation_grid_dome_deg;
% azimuths_dome =  data.azimuth_grid_dome_deg;
% azimuth_desired = 39;
% azimuth = 0;

% 3D figure
if make_3D_plots == true
    rg = zeros(size(brdf_in,1) + 1,size(brdf_in,2) + 1);
    rg(:,1) = [0 ; elevations_dome ];
    rg(1,:) = [0 ,  azimuths_dome ];
    plotCoordinateSystem('hemisphere',1)
    rg(2:end,2:end) =  brdf_in(:,:,70)*4;
    plotReflectionLobe(rg,azimuths_dome ,elevations_dome)
end

%% Check if azimuth_desired is a valid input
% if isempty(find(azimuths_dome == azimuth_desired))
%     error('invalid input for azimuth_desired. Variable should be in ...
    data.azimuth_grid_dome_deg.')
% end

if rem( azimuth_desired,accuracy_detection_deg) ~= 0; %0.5 * ...
    accuracy_detection_deg
    error("Invalid input for azimuth_desired. Variable should be a multiple ...
       of accuracy_detection_deg.")
end
```

```
%% Shift to desired accuracy

to_shift = azimuth_desired / accuracy_detection_deg; %(azimuth_desired-  0.5 ...
    * accuracy_detection_deg) / accuracy_detection_deg ;
brdf_out = circshift(brdf_in,to_shift, 2);

% 3D figure
if make_3D_plots == true
    rg = zeros(size(brdf_in,1) + 1,size(brdf_in,2) + 1);
    rg(:,1) = [0 ; elevations_dome ];
    rg(1,:) = [0 ,  azimuths_dome ];
    plotCoordinateSystem('hemisphere',1)
    rg(2:end,2:end) =  brdf_out(:,:,70)*4;
    plotReflectionLobe(rg,azimuths_dome ,elevations_dome)
end

end
```

calculateBRDFgrass2

```
function returned_data = calculateBRDFgrass(settings_ray_tracer,surfaces)
%=========================================================================
% Computes the BRDF of grass by Monte Carlo ray tracing of a piece of grass
%=========================================================================
% Input:  - settings_ray_tracer     struct    Contains all settings needed
%                                             to run the program. See
%                                             "Settings" below which
%                                             settings are needed.
% -----------------------------------------------------------------------
% Output: - returned_data           struct    Contains results of the
%                                             program as specified in
%                                             settings_ray_tracer
%=========================================================================
% Debugging features:
%         - There are several checks in the external functions - see
%           "Debugging features" in the documentation string of these
%           functions.
%        - Possibility to plot the ray tracing software, see
%           settings_ray_tracer.visualisation
%=========================================================================
% Update log:
%         - 19/05/2023     Taking structure surfaces as an input instead of
%                          making this structure in the function. Allows
%                          for running multiple times with same sample.
%         - 05/05/2023     Changed into a function, taking in
%                          settings_ray_tracer and giving returned_data
%                          as output.
%         - 04/05/2023     Changed square of grass to a circular spot
%=========================================================================

% Start time measurement
 tic;
%% Settings - unpack the settings
% retrieveSettingsRayDetection
[accuracy_detection_deg, ratio_sphere_over_sample_radius, sphere_center] = ...
    retrieveSettingsRayDetection(settings_ray_tracer);
```

```
%retrieveSettingsIncidentLight
[sun_azimuth_deg, sun_elevation_deg , wavelengths, number_of_unique_rays, ...
    number_of_repeated_rays,ratio_illuminated_over_sample_radius] = ...
    retrieveSettingsIncidentLight(settings_ray_tracer);

%retrieveSettingsSample
[radius_sample, ~, ~,~,grass_type,~, blade_height, ~,~] = ...
    retrieveSettingsSample(settings_ray_tracer);

%retrieveSettingsVisualisation
[rays_visualised, ...
    show_diffuse_reflection,show_ray_until_intersection,wavelengths_visualised, ...
    show_incoming_light,radius_dome_for_visualisation] = ...
    retrieveSettingsVisualisation(settings_ray_tracer);

%retrieveSettingsSaveResults
[save_detection_dome, name_to_save_brdf,save_results] = ...
    retrieveSettingsSaveResults(settings_ray_tracer);

%retrieveSettingsReturnedData
[return_brdf,return_radiance,return_RMSE_lambertian, ...
    return_centers_detection_dome,return_albedo,return_settings] = ...
    retrieveSettingsReturnedData(settings_ray_tracer);

%% Initialization
% Initialization general
returned_data = struct();
rays = struct();

% Initialize radii
radius_illuminated  = radius_sample * ratio_illuminated_over_sample_radius; ...
    % spot radius on the sample
radius_sphere = radius_sample * ratio_sphere_over_sample_radius; % sample ...
    radius (compared to spot)

% Initialization detection dome
azimuth_range_deg = -180  : accuracy_detection_deg : 180;
elevation_range_deg = [90 :-accuracy_detection_deg : 0]';
azimuth_range_rad   = deg2rad(azimuth_range_deg);
elevation_range_rad = deg2rad(elevation_range_deg);
[azimuth_grid_dome_deg, elevation_grid_dome_deg] = ...
    generateGridSky(azimuth_range_deg,elevation_range_deg, ...
    accuracy_detection_deg);

% Initialize measuring radiance
radiance = ...
    zeros(length(elevation_grid_dome_deg),length(azimuth_grid_dome_deg), ...
    length(wavelengths));

% Initialize visualisation
if isempty(rays_visualised) == 0
    copyobj(surfaces.sample_figure.Children, figure); % plot surfaces
    plotGridDetectionDome(azimuth_grid_dome_deg,elevation_grid_dome_deg, ...
        radius_dome_for_visualisation)
end

%% Generate objects
% Generate rays
[ray_starting_point, ray_direction] = generateRayDirection2(radius_sphere, ...
    sun_azimuth_deg, sun_elevation_deg,max(blade_height).*radius_sample);
```

```
[ray_magnitude] = generateRayMagnitude(wavelengths);

% if show_incoming_light == true
%     plotRays(ray_starting_point, ray_direction,linspace(0,5,100))
%     plotRayStartPoint(ray_starting_point)
% end

for k_ray = 1:number_of_unique_rays
    [rays] = saveRaySpecifications(rays,ray_starting_point, ray_direction, ...
        ray_magnitude, strcat("ray", string(k_ray)) );
    if show_incoming_light == true %  Plot all rays - note that the N of ...
        linspace determines with which the rays are drawn.
         plotRays(ray_starting_point, ray_direction,linspace(0,5,100))
         plotPoint(ray_starting_point, 'green') % plotted on radius_sphere
    end
    if ray_direction(3)> 0 % a ray may not be shot at the sky.
         error(sprintf("The incident ray %d is aimed at the sky.",k_ray))
    end

    [ray_starting_point, ray_direction,~] = generateParallelRayDirection2( ...
        radius_illuminated,ray_direction,radius_sphere,sphere_center, ...
        max(blade_height).*radius_sample );
end

% Retrieve surfaces
surfaces_list = surfaces.surfaces_list;
reflectance_surface = surfaces.reflectance_surface;


% Check Surfaces initially hit
[corners] = computeCornersInitiallyIlluminatedArea(settings_ray_tracer);
[list_initially_illuminated_surfaces] = ...
   checkBladesInInitiallyIlluminatedArea(grass_type,corners, surfaces);
list_initially_illuminated_surfaces = [strcat(surfaces_list(1),string(1)) ...
   list_initially_illuminated_surfaces]; % add the ground surface

%% Main loop: ray tracer
% Intersection
for k_ray = 1: number_of_unique_rays

    %Debugging code: check how far program is
    if rem(k_ray,500) == 0
        fprintf("%.f unique rays have been computed out of %.f \n", k_ray, ...
            number_of_unique_rays)
    end


    for kk_ray = 1: number_of_repeated_rays
        % retrieve specifics of ray
        clear ray_starting_point ray_direction ray_magnitude
        [ray_starting_point, ray_direction, ray_magnitude] = ...
            retrieveRaySpecifications(rays, strcat("ray", string(k_ray)));
        initial_direction = true;

    flag_not_detected = true; % initially, the ray is not detected
 while logical(flag_not_detected) % when the ray is detected, we trace the ...
    next ray
    if ray_magnitude < 0.01
         break
    end
```

```
% compute time at which ray intersects with detection dome and all planes
sphere_intersection_time = ...
    calculateIntersectionTimeWithSphere(radius_sphere,sphere_center, ...
    ray_starting_point, ray_direction,"positive");
plane_intersection_time = [];

if initial_direction == true
    surfaces_to_check_list = list_initially_illuminated_surfaces;
else
    surfaces_to_check_list = fieldnames(surfaces.individual_surfaces); % ...
        all surfaces
end

for k_plane = 1: length(surfaces_to_check_list)
    surface_name = string(surfaces_to_check_list(k_plane));
    [surface_boundaries, surface_normal] = ...
        retrieveSurfaceSpecifications(surfaces.individual_surfaces, ...
        surface_name );  % plane that might be hit first.
    [plane_intersection_time] = [plane_intersection_time ...
        calculateIntersectionTimeWithPlane(surface_boundaries, ...
        ray_starting_point, ray_direction, surface_normal )];
end
% sort the planes on first intersections and add the intersection time ...
    with the dome
[sorted_intersection_time, sorted_planes_index] = ...
   sort([sphere_intersection_time ...
   plane_intersection_time],'MissingPlacement','last');

% ensure interection is not on same plane as the ray departed from
sorted_planes_index = sorted_planes_index(sorted_intersection_time>0.001);
sorted_intersection_time = ...
   sorted_intersection_time(sorted_intersection_time>0.001);

% initialize for while loop
k_plane = 1; % loop over the planes
while  k_plane <= length(sorted_intersection_time)%   every sorted plane ...
            %order the planes on intersection points, then go over this ...
   list (first hit surface first!)

    if sorted_planes_index(k_plane) == 1  % if ray hits the detection dome
        intersection_point = ...
            calculateIntersectionPoint(ray_starting_point, ...
            sphere_intersection_time, ray_direction);
        if intersection_point(3)>0 % if the ray hits the detection dome ...
            below zero, it should not be detected. If the 'if statement' ...
            is left out, the rays that hit the detection dome below z = 0 ...
             are detected at the edge (theta = 0 ) leading to skewed results.
             radiance = detectRay(radiance, ray_magnitude, ...
                intersection_point, radius_sphere, azimuth_grid_dome_deg, ...
                elevation_grid_dome_deg);
             if show_ray_until_intersection == true && ismember(k_ray, ...
                rays_visualised)
                plotPoint(intersection_point, 'blue')
            end
        end
        flag_not_detected = false; % stop computing directions for this ray
        if show_ray_until_intersection == true && ismember(k_ray, ...
            rays_visualised)
             plotRays(ray_starting_point, ray_direction, ...
```

```
                linspace(0,sorted_intersection_time(k_plane),2))
        end
        break
    end

    % retrieve plane details
    surface_name = ...
        string(surfaces_to_check_list(sorted_planes_index(k_plane)-1));
    [surface_boundaries, surface_normal] = ...
        retrieveSurfaceSpecifications(surfaces.individual_surfaces, ...
        surface_name );  % plane that might be hit first.

    % calculate intersection point with polygon
    intersection_point = calculateIntersectionPoint(ray_starting_point, ...
        sorted_intersection_time(k_plane), ray_direction);
    hit = checkRayHitsSurface(surface_boundaries, ...
        intersection_point,surface_normal);

    if hit == true % if ray hits surface
         if show_ray_until_intersection == true && ismember(k_ray, ...
             rays_visualised)
              plotPoint(ray_starting_point,"green")
              plotRays(ray_starting_point, ray_direction, ...
                  linspace(0,sorted_intersection_time(k_plane),2))
         end

        if show_diffuse_reflection == true && ismember(k_ray, ...
            rays_visualised)
             ray_magnitude = ...
                calculateNewRayMagnitude(surfaces.individual_surfaces.
             (surface_name).type, ray_magnitude, reflectance_surface);
             ray_direction = calculateNewRayDirection(intersection_point, ...
                ray_starting_point,surface_normal);
             count = 1;
             while count < 1000 % Debugging code (count)
                     ray_direction_check(:,count) = ...
                        calculateNewRayDirection(intersection_point, ...
                        ray_starting_point,surface_normal);
                     plotRays(intersection_point, ...
                        ray_direction_check(:,count), linspace(0,0.002,2))
                     count = count + 1;
             end %end while
             ray_starting_point = intersection_point;
        else
             ray_magnitude = ...
                calculateNewRayMagnitude(surfaces.individual_surfaces.
             (surface_name).type, ray_magnitude, reflectance_surface);
             ray_direction = calculateNewRayDirection(intersection_point,
             ray_starting_point,surface_normal );
             ray_starting_point = intersection_point;
        end % end if


        if ray_magnitude < 0.01
            break
        end
        initial_direction = false;
        break % we change direction, recompute intersection times

    end % end if hit
```

```
        k_plane = k_plane + 1; % go to next plane if we did not hit the ...
            first one we encountered
    end
    end
 end % end for ray direction
end


pbaspect([1,1,0.5])

%% Compute BRDF
ray_magnitude_initial = generateRayMagnitude(wavelengths);
number_of_rays = number_of_unique_rays * number_of_repeated_rays;
[brdf, albedo]  = computeBRDF(radiance, ray_magnitude_initial, ...
    number_of_rays, azimuth_grid_dome_deg, elevation_grid_dome_deg);

% Finish time measurement
run_time = toc;

%% Plot results
% initialize matrix to visualise 3D data
rg = zeros(size(radiance,1) + 1,size(radiance,2) + 1);
rg(:,1) = [0 ; elevation_grid_dome_deg ];
rg(1,:) = [0 , azimuth_grid_dome_deg ];
rg_wavelength = repmat(rg, 1,1,length(wavelengths));

for k_wavelength = wavelengths_visualised
    wavelength = wavelengths(k_wavelength);

    % 2D figure
    figure;
    imagesc(brdf(:,:,k_wavelength))
    colorbar;

    % 3D figure
    plotCoordinateSystem('hemisphere',1)
    rg(2:end,2:end) =  brdf(:,:,k_wavelength);
    rg_wavelength(2:end,2:end,k_wavelength) =  brdf(:,:,k_wavelength);

    plotReflectionLobe(rg, azimuth_grid_dome_deg, elevation_grid_dome_deg);
end


%% Save resulting BRDF
if save_results == true
    FileName = strcat(pwd,'\RayTracingResults'); mkdir(FileName)
    save(strcat(FileName,'\',name_to_save_brdf,'.mat'), 'brdf', 'radiance', ...
        'settings_ray_tracer' , 'azimuth_grid_dome_deg', ...
        'elevation_grid_dome_deg', 'wavelengths', 'surfaces', 'rays', ...
        'albedo','run_time');
end

%% Save grid detection dome
if save_detection_dome == true
    mark = sprintf("dome_grid_%.2f_accuracy", accuracy_detection_deg);
    FileName = strcat(pwd,'\RayTracingResults'); mkdir(FileName)
    save(strcat(FileName,'\',mark,'.mat'), 'azimuth_grid_dome_deg', ...
        'elevation_grid_dome_deg');
end
```

```
%% Error calculation
if return_RMSE_lambertian == true
    for k_wavelength = 1 : length(wavelengths)
        wavelength = wavelengths(k_wavelength);
        root_mean_squared_error(k_wavelength) = ...
           computeErrorLambertian(brdf(:,:,k_wavelength));
    end
end


%% Return values
if return_brdf == true
    returned_data.brdf = brdf;
end

if return_radiance == true
    returned_data.radiance = radiance;
end

if return_RMSE_lambertian == true
    returned_data.RMSELambertian = root_mean_squared_error;
end

if return_centers_detection_dome == true
    returned_data.detection_dome_azimuths = azimuth_grid_dome_deg;
    returned_data.detection_dome_elevations = elevation_grid_dome_deg;
end

if return_albedo == true
   returned_data.albedo = albedo;
end

if return_settings == true
    returned_data.settings_ray_tracer = settings_ray_tracer;
end

returned_data.run_time = run_time;

%% Notification when code is finished
% Play sound when finished (see xpsound)
gong(0.5,442,1)
pause(0.2)
gong(0.5,600,3)
```

calculateIntersectionPoint

```
function intersection_point = calculateIntersectionPoint(ray_starting_point, ...
   t_intersect, ray_direction)
%=========================================================================
% Computes the intersection point based on the vector equation of the ray
%=========================================================================
% Input:  - ray_starting_point     3x1 matrix    Starting point of the ray
%                                                [x;y;z]
%         - t_intersect            float         Time at which the ray
%                                                intersects
```

```
%         - ray_direction          3x1 matrix    Direction of the ray
%                                                [x;y;z]
% -------------------------------------------------------------------------
% Output: - intersection_point     3x1 matrix    Intersection point [x;y;z]
%==========================================================================


% The intersection point p can computed as:
%       p(t) = e + t*d
%       Where
%           e starting point
%           t intersection time
%           d direction

intersection_point = ray_starting_point + t_intersect.*ray_direction;

end
```

calculateIntersectionTimeWithPlane

```
function t_intersect = ...
    calculateIntersectionTimeWithPlane(surface_boundaries, ...
    ray_starting_point, ray_direction, surface_normal )
%==========================================================================
% Calculates the time at which a ray intersects with a plane
% (e.g. the ground, grass blades)
%==========================================================================
% Input:  - plane                  [type variable]     [explanation]
%         - ray_starting_point     [type variable]     [explanation]
%         - ray_direction          [type variable]     [explanation]
%         - normal_plane           [type variable]     [explanation]
% -------------------------------------------------------------------------
% Output: - t_intersect    [type variable]     [explanation]
%==========================================================================
% Update log:
%   08/05/2023  Changed efinition point to always be center - no switch
%               case needed depending on geometry of surface.
%   04/05/2023  Added circular option.
%==========================================================================

%% Define a point on the plane
point = surface_boundaries.center; % point on the plane (center)

%% Compute the intersection time
% The plane cointaining the polygon has the implicit equation:
%       (p-pl) *n = 0
%       where p is any point on the plane
%             pl a point on the plane
%             n the normal of the plane
%       and the multiplication is the dot product
% Consider a ray with direction d and starting point e
% We find point p at time t from e in direction d.
% The intersection time is:
%       t = (pl - e) * n / (d*n)
%       where the multiplication is the dot product

dot_product_d_n =  dot(ray_direction,surface_normal);
if abs(dot_product_d_n) < 1e-15 % if dot product practically zero, ray ...
```

```
    direction perpendicular to normal
     t_intersect = nan;
     fprintf("A ray was parallel to a plane. This plane is not seen by the ray.")
else
     t_intersect = dot((point - ray_starting_point), surface_normal) ./ ...
        dot(ray_direction,surface_normal);
end

end
```

calculateIntersectionTimeWithSphere

```
function t_intersect = ...
    calculateIntersectionTimeWithSphere(sphere_radius,sphere_center, ...
    ray_starting_point, ray_direction, time_sign )
%=========================================================================
% Calculates the time at which a ray intersects with a sphere
% (e.g. the detection dome)
%=========================================================================
% Input:  - sphere_radius       float        Radius of the sphere
%         - sphere_center       3x1 matrix   Center of the sphere
%         - ray_starting_point  3x1 matrix   Starting point of the
%                                            ray [x;y;z]
%         - ray_direction       3x1 matrix   Direction of the ray [x;y;z]
%         - time_sign           string       Specifies whether the
%                                            intersection is expected
%                                            forward (positive) or
%                                            backward (negative) in time
% -------------------------------------------------------------------------
% Output: - t_intersect         float     Time at which the intersection
%                                         takes place
%=========================================================================
% Debugging features:
%         - Gives error when
%               - discriminant < 0
%               - invalid time_sign input
%=========================================================================


%% Initialization
% Renaming variables to read the mathematical equations more easily
R = sphere_radius;
d = ray_direction;
e = ray_starting_point;
c = sphere_center;

%% Code
% A sphere can be written in vector form like:
%   (p-c) * (p-c) - R^2  = 0
%   where c is the center of the sphere and p is any point on the sphere
% Consider a ray with direction d and starting point e
% The intersection time can then be computed:
%   t = -d * (e-c) +- sqrt( (d*(e-c))^2 - (d*d)((e-c)*(e-c)-R^2) )/ d*d
%   where every product is a dot product


discriminant =  dot(d,(e-c)).^2 - dot( dot(d,d), (dot(e-c,e-c) -R^2 ));
```

```
if discriminant <0
    error('Ray never intersects with detection dome!')
elseif discriminant == 0
    t_intersect_nominator = - dot(d,(e-c)) + sqrt(discriminant);
    t_intersect = t_intersect_nominator / dot(d,d);
else % two intersection times
    t_intersect_nominator_1 = - dot(d,(e-c)) + sqrt(discriminant);
    t_intersect_nominator_2  =- dot(d,(e-c)) - sqrt(discriminant);

    t_intersect_1 = t_intersect_nominator_1 / dot(d,d);
    t_intersect_2 = t_intersect_nominator_2 / dot(d,d);

    switch time_sign
        case "positive"
            if t_intersect_1 >= 0
                t_intersect = t_intersect_1;
            elseif t_intersect_2 >= 0
                t_intersect = t_intersect_2;
            end
        case "negative"
            if t_intersect_1 <= 0
                t_intersect =  t_intersect_1;
            elseif t_intersect_2 <= 0
                t_intersect =  t_intersect_2;
            end
        otherwise
            error(sprintf('%s is not a valid input for the variable ...
               time_sign.', time_sign))
    end
end

end
```

calculateNewRayDirection

```
function ray_direction = ...
   calculateNewRayDirection(intersection_point,ray_starting_point,surface_normal)
%=========================================================================
% Computes a new direction and magnitude for the ray after it has hit a
% surface
%=========================================================================
% Input:  - intersection_point    3x1 matrix   Point where ray hits the
%                                               surface ([x;y;z])
%         - ray_starting_point    3x1 matrix   Starting point of the ray
%                                               ([x;y;z])
%         - surface_normal        3x1 matrix   Normal of the surface ([x;y;z])
% -------------------------------------------------------------------------
% Output: - ray_direction         3x1 matrix    Direction of the ray
%                                                   ([x;y;z])
%=========================================================================
% Debugging feautures
%         - plot_invalid_ray_directions    Boolean      If true plots
%                                                       invalid ray
%                                                       directions
%=========================================================================
% Update log:
%   23/05/2023  Added function to plot invalid ray directions for debugging
%               purposes
```

```
%   22/05/2023  Input variables w.r.t. azimuth and elevation angles
%               detection dome deleted. Method through which new direction
%               is obtained has changed: now sampling a sphere
%               uniformly (in trems of polar angles), only allowing the new
%               direction to be in the same half_sphere as the normal of
%               the surface ( which is in the same direction as the light
%               is coming from)
%==========================================================================


%% Debugging settings
plot_invalid_ray_directions = true;

%% Choosing a new direction
% Model the plane as in the Cartesian equation:
%   a * x + b * y + c* z = d
%   where x, y and z are unit vectors in those respective directions.
% Then, the following relations are true for any point v and the surface
% with surface normal n:
%   n * v > d       -> v is in the same half space as n
%   n * v = d       -> v is in the plane
%   n * v < d       -> v is in the opposite half space
%       See: ...
   https://www.quora.com/Given-a-point-and-a-plane-how-would-you-determine-which-
side-of-the-plane-the-point-lies
% We can determine d knowing a point on the plane and the surface normal.
%   d = n_x * P_x + n_y * P_y + n_z * P_z
%       See: https://tutorial.math.lamar.edu/classes/calcIII/EqnsOfPlanes.aspx

d = intersection_point(1)* surface_normal(1) + ...
    intersection_point(2)*surface_normal(2) + intersection_point(3)* ...
    surface_normal(3);

%% Check from which side of the plane the ray is coming.
% Is the ray coming in the direction of the normal?
% Check by using the relations descirbed above
%       Take v = direction of the ray = start point - intersection point
%       If n * v < d the surface normal is in the opposite direction of the
%       ray and should be turned towards the ray

if dot(surface_normal, ray_starting_point - intersection_point) < d
    surface_normal = -surface_normal; % change normal direction
end

% The surface normal is now always in the same direction as the ray is coming
% from. The relations above are used to check if the new direction of the
% ray is viable (i.e. is in the same half-space as surface normal)

check_correct_side_of_plane = false;

count_while = 0;
while check_correct_side_of_plane == false % check if point in correct ...
    half-space
    if count_while > 1000
        error("Cannot find a suitable direction.")
    end

    % Choose a new direction in polar coordinates (azimuth phi, elevation
    % th) from the entire sphere of possibilities
    sphi = (-0.5*pi + pi*rand(1)); %(-0.5*pi  - angle_from_x_axis + ...
```

```
       pi*rand(1)); % between 0 and 2pi
    stheta = -pi + 2* pi * rand(1);  % angle_from_z_axis + pi * ...
       rand(1);%acos(rand(1)); % TEst deg2rad(20);

    % Rewrite to Cartesian coordinates ([x;y;z])
    [direction_x, direction_y, direction_z] =  sph2cart(sphi,stheta,100);
    ray_direction = [direction_x;direction_y;direction_z];

    % Reject ray direction if it is not in the half-space of the normal.
    % Else, continue
    if dot(surface_normal, ray_direction) > d
        check_correct_side_of_plane = true;
        break
    elseif plot_invalid_ray_directions == true
        ray = intersection_point + linspace(0,0.02,1).*ray_direction;
        plot3(ray(1,:), ray(2,:),ray(3,:),'r', 'LineWidth',1)
        hold on
    end
    count_while = count_while + 1;
end
end
```

calculateNewRayMagnitude

```
function ray_magnitude = calculateNewRayMagnitude(surface_type, ...
   ray_magnitude, reflectance_surface)
%=========================================================================
% Computes a new magnitude for the ray after it has hit a surface
%=========================================================================
% Input:  - surface_type     string          Type of the surface
%         - ray_magnitude   1 x n matrix    Array with magnitudes for each
%                                           wavelength
%         - - reflectance_surface struct    Structure with spectral
%                                           reflectance
% -------------------------------------------------------------------------
% Output: - ray_magnitude   1 x n matrix    Array with magnitudes for each
%                                           wavelength
%=========================================================================
% Update log
%       10/05/2023 Simplified to only one equation, moved definition of
%                  spectral reflectance surface to another function.
%       26/04/2023 Deleted "blade" angle dependence.
%=========================================================================

%% Determine the new magnitude of the ray

ray_magnitude = ray_magnitude .* reflectance_surface.(surface_type);



end
```

calculateNormal

```
function [normal_surface] = calculateNormal(surface_boundaries)
```

```
%=========================================================================
% Finds the normal of a surface
%=========================================================================
% Input:  - surface_boundaries  struct  Contains at least the shape of the
%                                       surface.
%                                       - If the shape is a "Square",
%                                         contains the corners of the
%                                         surface.
%                                       - If the shape is a "Circle",
%                                         contains the radius, center
%                                         and height.
% -------------------------------------------------------------------------
% Output: - normal_plane        1x3 matrix    Normal direction of the surface
%                                             [x;y;z]
%=========================================================================
% Notes: - The plane is considered as follows:
%
%           1 ------2         Corner 1 : x1, y1, z1
%           |       |         Corner 2 : x2, y2, z2
%           |       |         Corner 3 : x3, y3, z3
%           |       |         Corner 4 : x4, y4, z4
%           3-------4
%
%=========================================================================
% Update log:
%   04/05/2023  Added circular option.
%=========================================================================

switch surface_boundaries.shape
    case "Circle"
    % vector from center to east of the circle
    point_east = surface_boundaries.center;
    point_east(1) = point_east(1) + surface_boundaries.radius;
    vector_top_left_right = point_east - surface_boundaries.center;

    % vector from center to south of the circle
    point_south = surface_boundaries.center;
    point_south(2)  = point_south(2) + surface_boundaries.radius;
    vector_left_top_bottom = surface_boundaries.center - point_south;

    case "Rectangle"
    % finding the normal of the plane
    vector_top_left_right = surface_boundaries.corners(:,2) - ...
       surface_boundaries.corners(:,1);
    vector_left_top_bottom = surface_boundaries.corners(:,3) - ...
       surface_boundaries.corners(:,1);

end

normal_surface = cross(vector_left_top_bottom,vector_top_left_right );

% normalization
normal_surface = normal_surface / sqrt(normal_surface(1)^2 + ...
   normal_surface(2)^2 + normal_surface(3)^2  );
end
```

computeBRDF

```
function [brdf, albedo]  = ...
    computeBRDF(radiance,initial_ray_magnitude,number_of_rays,...
     azimuth_grid_dome_deg, elevation_grid_dome_deg)
%=========================================================================
% Computes the BRDF based on the matrix radiance.
%=========================================================================
% Input:  - radiance                 a x b matrix magnitude of the rays
%                                                detected in a specific
%                                                detection dome pixel
%         - initial_ray_magnitude       float      magnitude of the rays
%                                                when entering the
%                                                system
%         - number_of_rays              integer    total number of rays
%                                                entering the system
%         - azimuth_grid_dome_deg       1 x b matrix Azimuth angles of the
%                                                detection dome grid.
%                                                Degrees.
%         - elevation_grid_dome_deg     a x 1 matrix Elevation angles of
%                                                the detection dome
%                                                grid. Degrees.
% -------------------------------------------------------------------------
% Output: - brdf                        a x b matrix tabulated BRDF
%=========================================================================
% Debugging features:
%         - Explicitely checks if energy is not increasing
%         - plot_track_shape_brdf boolean to check the shape of
%         radiance, radiance and brdf
%=========================================================================


%% Debugging settings
plot_track_shape_brdf = false;

%% Initialization
radians = pi/180;


%% Determine incoming irradiance
incoming_irradiance = initial_ray_magnitude .* number_of_rays;



%% Determine brdf
for k_wavelength = 1: size(radiance, 3) % for all wavelengths
    % Plot radiance
    % radiance = radiant flux per solid angle per projected source area
    % radiance is the flux per pixel over the area in the detection dome
    if plot_track_shape_brdf == true
        figure;
        rg = zeros(size(radiance(:,:,k_wavelength),1) + ...
           1,size(radiance(:,:,k_wavelength),2) + 1);
        rg(:,1) = [0 ; elevation_grid_dome_deg ];
        rg(1,:) = [0 , azimuth_grid_dome_deg  ];
        rg(2:end,2:end) =  radiance(:,:,k_wavelength);
        plotReflectionLobe(rg, azimuth_grid_dome_deg, elevation_grid_dome_deg);
        title('Radiance')
        pbaspect([1 1 0.5])
    end

    % Normalisation factor for radiance
```

```
    if numel(elevation_grid_dome_deg)==1
        normalization = 1 ./ abs(trapz( (azimuth_grid_dome_deg.*radians),
        abs(radiance(:,:,k_wavelength).*cosd(90-elevation_grid_dome_deg).*
        sind(90-elevation_grid_dome_deg))) ) ;
    else
        normalization = 1 ./ abs(trapz( (azimuth_grid_dome_deg.*radians), ...
            abs(trapz((elevation_grid_dome_deg).*radians, ...
            abs(radiance(:,:,k_wavelength).*cosd(90-elevation_grid_dome_deg).*
        sind(90-elevation_grid_dome_deg))) ))) ;
    end


    % What percentage of energy entering is leaving?
    % Albedo = radiosity / irradiance
    %     Radiosity =  radiant flux leaving a surface per unit area
    %     Irradiance = radiant flux received by a surface per unit area
    albedo(k_wavelength) = sum(radiance(:,:,k_wavelength), 'all') ./ ...
        incoming_irradiance(k_wavelength);

    % Compute the brdf (taking into account absorption)
    brdf(:,:,k_wavelength) = normalization .* radiance(:,:,k_wavelength) .* ...
        albedo(k_wavelength);

    % Plot BRDF
    if plot_track_shape_brdf == true
        plotCoordinateSystem('hemisphere',1)
        rg = zeros(size(radiance,1) + 1,size(radiance,2) + 1);
        rg(:,1) = [0 ; elevation_grid_dome_deg ];
        rg(1,:) = [0 , azimuth_grid_dome_deg  ];
        rg(2:end,2:end) =  brdf(:,:,k_wavelength);
        plotReflectionLobe(rg, azimuth_grid_dome_deg, elevation_grid_dome_deg);
        title('BRDF')
        pbaspect([1 1 0.5])
    end

    % Check conservation of energy
    if numel(elevation_grid_dome_deg)==1
        check_conservation = abs(trapz( (azimuth_grid_dome_deg.*radians),
        abs(brdf(:,:,k_wavelength).* cosd(90-elevation_grid_dome_deg).*
        sind(90-elevation_grid_dome_deg)) )) ;
    else
        check_conservation = abs(trapz( (azimuth_grid_dome_deg.*radians), ...
            abs(trapz((elevation_grid_dome_deg).*radians, ...
            abs(brdf(:,:,k_wavelength).*cosd(90-elevation_grid_dome_deg).*
        sind(90-elevation_grid_dome_deg)) )))) ;
    end
    conservation_accuracy = 1E-10;
    if check_conservation > 1 + conservation_accuracy
        error("The computed BRDF does not obey energy conservation.")
    end
end



end
```

computeCornersInitiallyIlluminatedArea

```
function [corners] = computeCornersInitiallyIlluminatedArea(settings_ray_tracer)

%% Settings
plot_illuminated_area = true; % in figure that is already made
plot_sketch_illuminated_area = false;

%% Initialization
% Angle definitions
phi_1 = deg2rad(settings_ray_tracer.incident_light.sun_azimuth_deg);
phi_2 = pi/2 - phi_1;

% Distances
r_sample = settings_ray_tracer.sample.radius_sample;
r_sample_adjusted = r_sample * 10; % if corners 3 and 4 have a straight line ...
    between them, we there is a small part of the sampl which is not in the ...
    illuminated area. To avoid this, set corners 3 and 4 very far away.
r_illuminated = r_sample .* ...
    settings_ray_tracer.incident_light.ratio_illuminated_over_sample_radius;

% Blade width
blade_width = r_sample .* settings_ray_tracer.sample.blade_width;

% Define distance a:
a = r_illuminated + blade_width/2;

% Origin
origin = [0; 0];

%% Determine support points

support_point_1 = [origin(1) +  a * cos(phi_2); origin(2) - a * sin(phi_2)];
support_point_2 = [origin(1) - a * cos(phi_1); origin(2) - a * sin(phi_1)];

d_origin_support_point_1 =  norm(support_point_1);
d_support_point_1_to_edge_sample = sqrt(r_sample_adjusted^2 - ...
    d_origin_support_point_1.^2);

%% Find corners

corner_4 = [support_point_1(1)  - d_support_point_1_to_edge_sample * ...
    cos(phi_1) ; support_point_1(2) - d_support_point_1_to_edge_sample * ...
    sin(phi_1)];
corner_3 = [support_point_1(1) + d_support_point_1_to_edge_sample * ...
    cos(phi_1) ; support_point_1(2) + d_support_point_1_to_edge_sample * ...
    sin(phi_1)];

support_point_3 = [origin(1) -  a * cos(phi_2); origin(2) + a * sin(phi_2)];

corner_2 = [support_point_3(1) + d_support_point_1_to_edge_sample * ...
    cos(phi_1) ; support_point_3(2) + d_support_point_1_to_edge_sample * ...
    sin(phi_1)];
corner_1 = [support_point_3(1) - d_support_point_1_to_edge_sample * ...
    cos(phi_1) ; support_point_3(2) - d_support_point_1_to_edge_sample * ...
    sin(phi_1)];

%% Return corners matrix
corners = [[corner_1 , corner_2  , corner_3 , corner_4]; [0,0,0,0]];

%% Plot
if or(plot_sketch_illuminated_area==true, plot_illuminated_area == true)
```

```
    % Initialize
    surface_boundaries.shape = "Rectangle";
    surface_boundaries.corners = corners;

    if plot_sketch_illuminated_area == true
    %       support points      in blue
    %       corners             in green
    %       center of sample    in black


    % Plot origin
    figure;
    scatter(origin(1),origin(2), 'k')

    % Plot support points
    hold on
    scatter(support_point_1(1),support_point_1(2), 'b')
    hold on
    scatter(support_point_2(1),support_point_2(2), 'b')
    hold on
    scatter(support_point_3(1),support_point_3(2), 'b')

    % Plot corners
    hold on
    scatter(corner_4(1),corner_4(2), 'g')
    hold on
    scatter(corner_1(1),corner_1(2), 'g')
    hold on
    scatter(corner_3(1),corner_3(2), 'g')
    hold on
    scatter(corner_2(1),corner_2(2), 'g')
    hold on

    % Plot initial illuminated area
    plotSurface(surface_boundaries)
    hold on

    % Plot support circles
    plotCircle(0,0,r_illuminated,'y')
    hold on
    plotCircle(0,0,a, 'c')
    hold on
    plotCircle(0,0,r_sample,'g')
    hold on
    plotCircle(0,0,r_sample_adjusted,'c')
    hold on

    pbaspect([1,1,1])
    elseif plot_illuminated_area == true
        plotSurface(surface_boundaries)
    end % end if plot

 end % end if or()


%% Internal functions
function h = plotCircle(x,y,r,color)
    th = 0:pi/50:2*pi;
    xunit = r * cos(th) + x;
    yunit = r * sin(th) + y;
```

```
    h = plot(xunit, yunit,color);
    end % end internal function

end % end function
```

computeErrorLambertian

```
function root_mean_squared_error = computeErrorLambertian(brdf)
%=========================================================================
% Computes the mean square error of brdf compared to an ideal Lambertian
% (with the same grid size)
%=========================================================================
% Input:  - azimuth_grid_dome_deg       1xb matrix     Azimuth angles of
%                                                      centers of detection
%                                                      dome pixels
%         - elevation_grid_dome_deg     ax1 matrix     Elevation angles of
%                                                      centers of detection
%                                                      dome pixels
%         - brdf                        axb matrix     BRDF with elevations
%                                                      on rows and azimuths
%                                                      on columns
% ------------------------------------------------------------------------
% Output: - mean_square_error     float     mean square error of brdf
%                                           compared to ideal lambertian
%=========================================================================
% Debugging features:
%         - Error can be printed
%=========================================================================

%% Settings
make_Lambertian_plot = false;
make_individual_error_plot = false;

%% Error calculation
% Squared error for every pixel
analytical_lambertian= 1/pi;
squared_error_individual_pixel = abs(analytical_lambertian - brdf).^2;

% error total shape
mean_squared_error = sum(squared_error_individual_pixel, 'all')/numel(brdf);
root_mean_squared_error = sqrt(mean_squared_error);

% Debugging code: Print error
% sprintf("Error with my BRDF: %.10f ", root_mean_squared_error)

%% Plot error
if make_individual_error_plot == true
figure;
imagesc(squared_error_individual_pixel)
title("Error with my brdf")
colorbar;
end


end
```

computeRotationMatrix

```
function rotation_matrix = computeRotationMatrix (rotation_axis, rotation_angle)
%==========================================================================
% Returns the rotation matrix given the axis of rotation and rotation angle
%==========================================================================
% Input:  - rotation_axis      3x1 matrix          axis to rotate around
%         - rotation_angle     float               angle of rotation
% -------------------------------------------------------------------------
% Output: - rotation_matrix    3x3 matrix          rotation matrix
%==========================================================================


%% Initialize
% Variables are renamed such that rotation_matrix can be checked more
% easily

% Separate rotation axis
ux = rotation_axis(1);
uy = rotation_axis(2);
uz = rotation_axis(3);

% Rotation angle
th = rotation_angle;

%% Compute the rotation matrix
% Source: wikipedia
rotation_matrix = [[ cos(th)+ux^2 * (1-cos(th))          , ux*uy*(1-cos(th))- ...
    uz*sin(th) , ux*uz*(1-cos(th))+uy*sin(th)];
                    [ uy*ux*(1-cos(th)) + uz* sin(th )   , cos(th)+ uy^2 * ...
                       (1-cos(th))     , uy*uz*(1-cos(th))-ux*sin(th)];
                    [uz*ux*(1-cos(th))-uy*sin(th)        , ...
                       uz*uy*(1-cos(th))+ux*sin(th)   , cos(th) + uz^2 * ...
                       (1-cos(th))]];
end
```

D.1.7 Analyze

analyzeBRDFgrass

```
%% Analyze BRDF grass
% This script should be run per section.

%% Set settings
settings_ray_tracer = struct();
surfaces = struct();

% retrieveSettingsRayDetection
settings_ray_tracer = setSettingsRayDetection(settings_ray_tracer);

%retrieveSettingsIncidentLight
settings_ray_tracer = setSettingsIncidentLight(settings_ray_tracer);

%retrieveSettingsSample
settings_ray_tracer = setSettingsSample(settings_ray_tracer);
```

```
%retrieveSettingsVisualisation
settings_ray_tracer = setSettingsVisualisation(settings_ray_tracer);

%retrieveSettingsSaveResults
settings_ray_tracer = setSettingsSaveResults(settings_ray_tracer);

%retrieveSettingsReturnedData
settings_ray_tracer= setSettingsReturnedData(settings_ray_tracer);

%make sample
surfaces = generateSample2(settings_ray_tracer);

%run
returned_data = calculateBRDFgrass2(settings_ray_tracer, surfaces);


%% Run for several theta
elevation_list = 70;%[0 30 70 90]; %0:10:90;
illumination_spot_list = 1;

surfaces = generateSample(settings_ray_tracer);
for k_elevation = 1 : length(elevation_list)
    elevation = elevation_list(k_elevation);

    for k_illumination = 1 : length(illumination_spot_list)
     spot_size = illumination_spot_list(k_illumination);

     fprintf("Elevation %.3f \t Spot size %.1f \n", elevation, spot_size)

     % to save results
     save_name = ...
        sprintf("OneBladedomesmall_%.1f_percentage_illuminated_%d_15May",
     elevation, spot_size * 100 );

     %save_name = sprintf("BCgrass_%.1f_percentage_illuminated_%d_7", ...
        elevation, spot_size * 100 );
     settings_ray_tracer = setSettingsSaveResults(settings_ray_tracer,
     "name_to_save_brdf",save_name );

     % to change the elevation
     settings_ray_tracer = setSettingsIncidentLight(settings_ray_tracer, ...
        "ratio_illuminated_over_sample_radius", ...
        spot_size,"sun_elevation_deg", elevation);

     % compute BRDF
     tic;
     returned_data = calculateBRDFgrass(settings_ray_tracer, surfaces);
     timed_run(k_elevation,k_illumination) = toc;

    end
end

%% Run for several number of blades
blades_list = [0 50 100 250 350 500];%[0 30 70 90]; %0:10:90;

for k_blades = 1 : length(blades_list) %6% [ 3, length(elevation_list) - 2]
    num_of_blades = blades_list(k_blades);

     fprintf("Blades %.1f \n", num_of_blades)
```

```
    % to change the num of blades
    settings_ray_tracer = ...
        setSettingsSample(settings_ray_tracer,"number_of_blades", ...
        num_of_blades);
    surfaces = struct();
    tic;
    surfaces = generateSample2(settings_ray_tracer);
    time_generate_sample(k_blades) = toc;

    % compute BRDF
    tic;
    returned_data = calculateBRDFgrass2(settings_ray_tracer, surfaces);
    timed_run(k_blades) = toc;
end


%% Run for several elevation angles and blade lengths
elevation_list = [0:10:90];
grass_length_list = [0.16 0.32];

for k_grass_length = 1 : length(grass_length_list)
    grass_length = grass_length_list(k_grass_length);
    grass_blade_geometry = strcat("VerticalSameHeight");

    % to change the num of blades
    settings_ray_tracer = ...
        setSettingsSample(settings_ray_tracer,"blade_geometry", ...
        grass_blade_geometry,"blade_height",grass_length_list);

    % make sample
    surfaces = struct();
    surfaces = generateSample2(settings_ray_tracer);

   for k_elevation = 1 : length(elevation_list)
            elevation = elevation_list(k_elevation);
            settings_ray_tracer = setSettingsIncidentLight(settings_ray_tracer,
            "sun_elevation_deg", elevation);

            fprintf("Grass length %.1f Elevation %.1f \n", grass_length, ...
                elevation)

            % to save results
            save_name = sprintf("VerticalGrass_%.1f_Elevation_%1.f_25May", ...
                grass_length, elevation);
            settings_ray_tracer = ...
                setSettingsSaveResults(settings_ray_tracer, ...
                "name_to_save_brdf",save_name );

            % compute BRDF
            tic;
            calculateBRDFgrass2(settings_ray_tracer, surfaces);
            timed_run(k_elevation,k_grass_length) = toc;
   end
end


 fprintf("Mixed heights \n")
  % to change the num of blades
 settings_ray_tracer = ...
    setSettingsSample(settings_ray_tracer,"blade_geometry", ...
```

```
    "VerticalVaryingHeight","blade_height",grass_length_list);

 % make sample
 surfaces = struct();
 surfaces = generateSample2(settings_ray_tracer);
  for k_elevation = 1 : length(elevation_list)
     elevation = elevation_list(k_elevation);
     settings_ray_tracer = ...
         setSettingsIncidentLight(settings_ray_tracer,"sun_elevation_deg", ...
         elevation);

    save_name = sprintf("MixedHeightsGrass_Elevation_%1.f_25May", elevation);
    settings_ray_tracer = ...
        setSettingsSaveResults(settings_ray_tracer,"name_to_save_brdf", ...
        save_name);

    fprintf("Mixed Heights Elevation %.1f \n", elevation)

     % compute BRDF
     tic;
     calculateBRDFgrass2(settings_ray_tracer, surfaces);
     timed_run(k_elevation,3) = toc;
  end

%% Run for several tilted tops and elevation angles
tilted_tops_list = [10 45];

for k_tilted_top = 1 : length(tilted_tops_list) %6% [ 3, ...
    length(elevation_list) - 2]
     tilted_top = tilted_tops_list(k_tilted_top);


     % to change the num of blades
     settings_ray_tracer = ...
         setSettingsSample(settings_ray_tracer,"blade_geometry", ...
         "VerticalAndTiltedTop", "blade_tilt",[-tilted_top, tilted_top]);

     % make sample
     surfaces = struct();
     surfaces = generateSample2(settings_ray_tracer);

      for k_elevation = 1 : length(elevation_list)
             elevation = elevation_list(k_elevation);
             settings_ray_tracer = ...
                 setSettingsIncidentLight(settings_ray_tracer,"sun_elevation_deg", ...
                 elevation);

             % to save results
             fprintf("Tilted top %.1f Elevation %.1f  \n", tilted_top, ...
                 elevation)

            % to save results
            save_name = sprintf("MixedHeightsGrass_Elevation_%1.f_25May", ...
                elevation);
            settings_ray_tracer = ...
                setSettingsSaveResults(settings_ray_tracer,"name_to_save_brdf", ...
                save_name);


             % compute BRDF
```

```
            tic;
            calculateBRDFgrass2(settings_ray_tracer, surfaces);
            timed_run(k_elevation, k_tilted_top+3) = toc;
      end
end


%% a_illuminated change to see change in BRDF - keep r_illuminated constant ...
   and increase r_sample
elevation_list = [0:10:90];
a_illuminated_list = [1 0.8 0.5 0.2 0.1];
r_illuminated = 2.5; % cm
width_blade_set = 0.5; % cm
height_blade_set = 4; % cm
num_blades_set = 45; % #
width_blade_set_ratio = width_blade_set/ r_illuminated;
height_blade_set_ratio = height_blade_set/ r_illuminated;

for k_a_illuminated = 1: length(a_illuminated_list)
    a_illuminated = a_illuminated_list(k_a_illuminated);
    r_sample = r_illuminated / a_illuminated;
    width_blade = width_blade_set_ratio * a_illuminated;
    height_blade = height_blade_set_ratio * a_illuminated;
    num_blades = round(r_sample.^2/ r_illuminated.^2 *num_blades_set);

    settings_ray_tracer = setSettingsIncidentLight(settings_ray_tracer, ...
       "ratio_illuminated_over_sample_radius", a_illuminated);

    % to change the num of blades
    settings_ray_tracer = ...
       setSettingsSample(settings_ray_tracer,"blade_width", width_blade, ...
       "blade_height",height_blade,"number_of_blades",num_blades, ...
       "radius_sample", r_sample);

    % make sample
    surfaces = struct();
    surfaces = generateSample2(settings_ray_tracer);

     for k_elevation = 1 : length(elevation_list)
            elevation = elevation_list(k_elevation);
            settings_ray_tracer = ...
               setSettingsIncidentLight(settings_ray_tracer, ...
               "sun_elevation_deg", elevation);

            % to save results
            fprintf("a_illumination %.1f Elevation %.1f  \n", ...
               a_illuminated, elevation)

           % to save results
           save_name = ...
              sprintf("Convergence_aillumination_%.1f_El_%1.f_26May", ...
              a_illuminated, elevation);
           settings_ray_tracer = ...
              setSettingsSaveResults(settings_ray_tracer, ...
              "name_to_save_brdf", save_name);


            % compute BRDF
            calculateBRDFgrass2(settings_ray_tracer, surfaces);
      end
```

```
end
```

analyzeMakeLibrary

```
%% Set settings
settings_ray_tracer = struct();
surfaces = struct();

% retrieveSettingsRayDetection
settings_ray_tracer = setSettingsRayDetection(settings_ray_tracer);

%retrieveSettingsIncidentLight
settings_ray_tracer = setSettingsIncidentLight(settings_ray_tracer);

%retrieveSettingsSample
settings_ray_tracer = setSettingsSample(settings_ray_tracer);

%retrieveSettingsVisualisation
settings_ray_tracer = setSettingsVisualisation(settings_ray_tracer);

%retrieveSettingsSaveResults
settings_ray_tracer = setSettingsSaveResults(settings_ray_tracer);

%retrieveSettingsReturnedData
settings_ray_tracer= setSettingsReturnedData(settings_ray_tracer);


%% Make library BRDF
elevation_list = [1e-6 6:6:72]; %0:10:90;
number_of_runs = 10;

for i = 1:10
    surfaces = generateSample2(settings_ray_tracer);
    for k_elevation = 1 : 2% length(elevation_list)
        elevation = elevation_list(k_elevation);

         fprintf("Run: %d, Elevation %.3f \t \n", i, elevation)

         % to save results
         save_name = ...
            sprintf("VerticalTiltedTop40_4cm4cm_theta_%.2f_%d",elevation,i);
         settings_ray_tracer = setSettingsSaveResults(settings_ray_tracer, ...
            "name_to_save_brdf",save_name );

         % to change the elevation
         settings_ray_tracer = setSettingsIncidentLight(settings_ray_tracer, ...
            "sun_elevation_deg", elevation);

         % compute BRDF
         returned_data = calculateBRDFgrass2(settings_ray_tracer, surfaces);
    end
end
```

analyzeCalibrationGrassReflector

```
%% analyzeCalibrationGrassReflector
%% Set settings
settings_ray_tracer = struct();
surfaces = struct();

% retrieveSettingsRayDetection
settings_ray_tracer = setSettingsRayDetection(settings_ray_tracer);

%retrieveSettingsIncidentLight
settings_ray_tracer = setSettingsIncidentLight(settings_ray_tracer);

%retrieveSettingsSample
settings_ray_tracer = setSettingsSample(settings_ray_tracer);

%retrieveSettingsVisualisation
settings_ray_tracer = setSettingsVisualisation(settings_ray_tracer);

%retrieveSettingsSaveResults
settings_ray_tracer = setSettingsSaveResults(settings_ray_tracer);

%retrieveSettingsReturnedData
settings_ray_tracer= setSettingsReturnedData(settings_ray_tracer);

%% Settings for various runs
elevation_list = [1e-6 10:10:90];
a_illuminated_list = [1 0.75 0.5 0.25];
r_illuminated = 2.5; % cm
width_blade_set = 0.5; % cm
height_blade_set = 4; % cm
num_blades_set = 45; % #
number_of_runs = 1;%10;


%% Initialization
width_blade_set_ratio = width_blade_set/ r_illuminated;
height_blade_set_ratio = height_blade_set/ r_illuminated;


%% a_illuminated change to see change in BRDF
for k_n_runs = 1 : number_of_runs
for k_a_illuminated = 1%1: length(a_illuminated_list)
    a_illuminated = a_illuminated_list(k_a_illuminated);
    r_sample = r_illuminated / a_illuminated;
    width_blade = width_blade_set_ratio * a_illuminated;
    height_blade = height_blade_set_ratio * a_illuminated;
    num_blades = round(r_sample.^2/ r_illuminated.^2 *num_blades_set);

    settings_ray_tracer = setSettingsIncidentLight(settings_ray_tracer, ...
        "ratio_illuminated_over_sample_radius",a_illuminated);

    % to change the num of blades
    settings_ray_tracer = ...
        setSettingsSample(settings_ray_tracer,"blade_width", width_blade, ...
        "blade_height",height_blade, ...
        "number_of_blades",num_blades,"radius_sample", r_sample);

    % make sample
    surfaces = struct();
    surfaces = generateSample2(settings_ray_tracer);
```

```
    for k_elevation = 1%1 : length(elevation_list)
            elevation = elevation_list(k_elevation);
            settings_ray_tracer = ...
               setSettingsIncidentLight(settings_ray_tracer, ...
               "sun_elevation_deg", elevation);

            % to save results
            fprintf("a_illumination %.1f Elevation %.1f  \n", ...
               a_illuminated, elevation)

           % to save results
           save_name = ...
              sprintf("Convergence_aillumination_%.2f_El_%.2f_6Jun_%d", ...
              a_illuminated, elevation,k_n_runs);
           settings_ray_tracer = ...
              setSettingsSaveResults(settings_ray_tracer, ...
              "name_to_save_brdf", save_name);


            % compute BRDF
            calculateBRDFgrass2(settings_ray_tracer, surfaces);
     end

end
end
```

analyzeLambertianReflection

```
%% Settings
accuracy_detection_deg = 3;
azimuth_range_deg = [-180  : accuracy_detection_deg : 180];
elevation_range_deg = [90 :-accuracy_detection_deg : 0]';
number_of_rays_list = [1e3 1e4 1e5];% 7.5e4 1e5 2.5e5 5e5 7.5e5 1e6 2.5e6 ...
   5e6 7.5e6 1e7 2.5e7 5e7 7.5e7 1e8];
a_dome_inverse_list = [1e-8 5e-7 1e-7 5e-5]; % for center, set to 0
number_of_repeating_runs = 2;
save_data = true;
investigate_direction = "vertical"; % "vertical" or "horizontal"


%% Initialization
sphere_radius = 1;
azimuth_range_rad = deg2rad(azimuth_range_deg);
elevation_range_rad = deg2rad(elevation_range_deg);
[azimuth_grid_dome_deg, elevation_grid_dome_deg] = generateGridSky ...
   (azimuth_range_deg,elevation_range_deg, accuracy_detection_deg);
ray_starting_point =[3;2;1]; % does not matter for Lambertian

%% save detection dome
mark= sprintf("dome_grid_%.2f_accuracy",accuracy_detection_deg);
FileName = strcat('C:\Users\leoni\Documents\SET\7 8  Graduation ...
   Assignment\Code\RayTracingResultsLambertianAdome');  mkdir(FileName);
save(strcat(FileName,'\',mark,'.mat'), 'azimuth_grid_dome_deg', ...
   'elevation_grid_dome_deg');


%% Main loop
for k_repeat = 1 : number_of_repeating_runs
```

```
for k_radius_aim_from_center = 1: length(a_dome_inverse_list)
    switch investigate_direction
        case "horizontal"
            intersection_point_ground = ...
               [a_dome_inverse_list(k_radius_aim_from_center);0;0];
        case "vertical"
            intersection_point_ground = ...
               [0;0;a_dome_inverse_list(k_radius_aim_from_center)];
        otherwise
            error(sprintf("Not a valid value for %s", investigate_direction))
    end

    for k_number_of_rays = 1 : length(number_of_rays_list)
        tic; % measure time

        % For computational speed, update number of the upcoming ray that
        % we are running.
        %       E.g. number_of_rays_list =  [100,1000];
        %            First run : from ray 1 to ray 100,
        %            Second run: rays 101 to 1000 are added instead of
        %                        running from 1 to 1000 again.

        % Initialize starting ray
        if k_number_of_rays ==1
            radiance = zeros(length(elevation_grid_dome_deg), ...
               length(azimuth_grid_dome_deg) );
            number_of_rays_start = 1;
        else
            number_of_rays_start = number_of_rays_list(k_number_of_rays-1)+1;
        end

        % Initialize max number of rays
        max_number_of_rays = number_of_rays_list (k_number_of_rays);

        for k_ray_number = number_of_rays_start : max_number_of_rays %for ...
           every ray

            %Debugging code: check how far program is
            if rem(k_ray_number,1e4) == 0
                fprintf("%.f unique rays have been computed out of %.f \n", ...
                   k_ray_number, max_number_of_rays)
            end

            % New direction from ground
            new_direction = ...
               calculateNewRayDirection(intersection_point_ground, ...
               ray_starting_point,[0;0;1] );

            % Intersection with detection dome
            sphere_intersection_time = ...
               calculateIntersectionTimeWithSphere(sphere_radius,[0;0;0], ...
               intersection_point_ground, new_direction,"positive");
            intersection_point = ...
               calculateIntersectionPoint(intersection_point_ground, ...
               sphere_intersection_time, new_direction);

            % Detect
            if intersection_point(3) >= 0
                radiance = detectRay(radiance, 1, intersection_point, ...
```

```
                    sphere_radius, azimuth_grid_dome_deg, ...
                    elevation_grid_dome_deg);
            end
        end

        % MakeBRDF
        [brdf, albedo(k_number_of_rays, k_radius_aim_from_center,:) ] = ...
            computeBRDF(radiance,1,max_number_of_rays, azimuth_grid_dome_deg, ...
            elevation_grid_dome_deg);

        root_mean_squared_error(k_number_of_rays, k_radius_aim_from_center) ...
            = computeErrorLambertian(brdf);

        % Save data
        if save_data == true
            run_time = toc; % measure time
            mark = ...
                sprintf("Lambertian_rays_%.f_beta_%.2f_%d",max_number_of_rays, ...
                k_radius_aim_from_center , k_repeat);
            FileName = strcat('C:\Users\leoni\Documents\SET\7 8  Graduation ...
                Assignment\Code\RayTracingResultsLambertian'); mkdir(FileName)
            save(strcat(FileName,'\',mark,'.mat'), 'radiance', 'brdf', ...
                'albedo', 'root_mean_squared_error','run_time' );
        end
    end
end
end
```

convergenceAnalysisLambertianReflector

```
% comparison error file
clear all
%% Settings
number_of_rays_list = [1e4 5e4 1e5 5e5 1e6 5e6];% % 5e6 1e7 5e7 1e8 5e8 1e9];%
accuracy_detection_deg = 3;
number_of_repeating_runs = 10;
make_comparison_plots_n = false;
make_comparison_plots_a_dome = true;
make_comparison_plots_h= true;

% Only when make_comparison_plots_a_dome/ make_comparison_plots_h == true
a_dome_inverse_list = [1e-8 1e-7 1e-6 1e-5 1e-4 1e-3 1e-2 0.02 0.04 0.066667 ...
    0.1 0.25 0.5 0.6 0.7 0.8 0.9 1];
number_of_rays_specific = 1e6;


%% Initialization
% Retrieve detection dome grid
mark = sprintf("dome_grid_%.2f_accuracy",accuracy_detection_deg);
FileName = strcat('C:\Users\leoni\Documents\SET\7 8  Graduation ...
    Assignment\Code\RayTracingResultsLambertianAdome'); mkdir(FileName)
loaded_data_grid = load(strcat(FileName,'\',mark,'.mat'));
azimuth_grid_dome_deg = loaded_data_grid.azimuth_grid_dome_deg;
elevation_grid_dome_deg = loaded_data_grid.elevation_grid_dome_deg;

% Set accuracy list, consisting of angles. Note that not all angles are
% allowed as both the azimuths and the elevation angles should be divided
% into this angle.
```

```
accuracy_list = accuracy_detection_deg .* [1:90]';
accuracy_list = accuracy_list(mod(90,accuracy_list)==0);
accuracy_list = accuracy_list(mod(360,accuracy_list)==0);


% Colours
colour_choice_accuracy = parula(length(accuracy_list));
colour_choice_number_rays = parula(length(number_of_rays_list));

%% Main code: compute mean square error
if make_comparison_plots_n == true
for k_rays =  1: length(number_of_rays_list)
    number_of_rays = number_of_rays_list(k_rays);

    % load data
    for k_number_of_runs = 1: number_of_repeating_runs
        mark = sprintf("Lambertian_rays_%.f_%d",number_of_rays, ...
            k_number_of_runs);
        FileName = strcat('C:\Users\leoni\Documents\SET\7 8  Graduation ...
            Assignment\Code\RayTracingResultsLambertian'); mkdir(FileName)
        loaded_data = load(strcat(FileName,'\',mark,'.mat'));
        radiance_per_run(:,:,k_number_of_runs) = loaded_data.radiance;
        run_time_per_run(:,k_number_of_runs) = loaded_data.run_time;
    end

    radiance = mean(radiance_per_run,3);
    mean_run_time(k_rays) = mean(run_time_per_run,2);

    for k_accuracy = 1: length(accuracy_list)  %accuracy detection dome
    accuracy = accuracy_list(k_accuracy);
        % compute error
        if accuracy == accuracy_detection_deg
            [brdf, albedo(k_accuracy,:)]= computeBRDF(radiance, 1, ...
                number_of_rays, azimuth_grid_dome_deg, elevation_grid_dome_deg);
            root_mean_squared_error(k_accuracy, k_rays) = ...
                computeErrorLambertian(brdf);
        else
            clear brdf_reduced azimuth_reduced elevation_reduced ...
                radiance_reduced
            [radiance_reduced, azimuth_reduced, elevation_reduced] = ...
                reduceResolutionData(radiance, azimuth_grid_dome_deg, ...
                elevation_grid_dome_deg,accuracy_detection_deg, accuracy);
            sprintf("Number of rays: %.f \t \t Grid accuracy: ...
                %.2f",number_of_rays,  accuracy)
            brdf_reduced = computeBRDF(radiance_reduced, 1, number_of_rays, ...
                azimuth_reduced, elevation_reduced);
            root_mean_squared_error(k_accuracy, k_rays) = ...
                computeErrorLambertian(brdf_reduced);
        end
    end
end

%% Plot

% y axis RMSE, x axis accuracy grid
figure;
for k_rays = 1: length(number_of_rays_list)
    number_of_rays = number_of_rays_list(k_rays);
    s1 = scatter(accuracy_list, root_mean_squared_error(:,k_rays),'filled');
    s1.MarkerFaceColor = colour_choice_number_rays(k_rays,:);
```

```
    %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
    set(gca,'yscale','log')
    set(gca,'xscale','log')
    hold on
end
lgd = legend(append("n = ", string(number_of_rays_list)));
xlabel("\beta")
ylabel("RMSE")
title("Accuracy test of the ray tracer")
grid on
ylim([1e-3 1])
set(gca,'FontSize',20)
lgd.Location = "northeastoutside";
box on


% trial
figure
t = tiledlayout(1,2,'TileSpacing','compact');
bgAx = axes(t,'XTick',[],'YTick',[],'Box','off');
bgAx.Layout.TileSpan = [1 2];
ax1 = axes(t);
for k_rays = 1: length(number_of_rays_list)
    number_of_rays = number_of_rays_list(k_rays);
    s1 = scatter(ax1, accuracy_list, ...
        root_mean_squared_error(:,k_rays),'filled');
    s1.MarkerFaceColor = colour_choice_number_rays(k_rays,:); ...
        %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
    set(gca,'yscale','log')
    set(gca,'xscale','linear')
    hold on
end
xline(ax1,10,':');
ax1.Box = 'off';
set(gca,'FontSize',20)
xlim(ax1,[0 10])
ylim(ax1,[1e-3 1])
xticks(ax1, [0:1:10])
grid on

ax2 = axes(t);
ax2.Layout.Tile = 2;
for k_rays = 1: length(number_of_rays_list)
    number_of_rays = number_of_rays_list(k_rays);
    s1 = scatter(ax2, accuracy_list, ...
        root_mean_squared_error(:,k_rays),'filled');
    s1.MarkerFaceColor = colour_choice_number_rays(k_rays,:); ...
        %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
    set(gca,'yscale','log')
    set(gca,'xscale','linear')
    hold on
end
xline(ax2,10,':');
ax2.YAxis.Visible = 'off';
ax2.Box = 'off';
set(gca,'FontSize',20)
xlim(ax2,[10 45])
ylim(ax2,[1e-3 1])
xticks(ax2, [10:5:45])
grid on
```

```
xlabel(t,"\beta")
ylabel(t,"RMSE")
title(t,"Accuracy test of the ray tracer")

% y axis RMSE, x axis number of rays, extra variable accuracy
figure;
for k_accuracy = 1:length(accuracy_list)
    accuracy = accuracy_list(k_accuracy);
    s2 = scatter(number_of_rays_list, ...
        root_mean_squared_error(k_accuracy,:),'filled');
    s2.MarkerFaceColor = colour_choice_accuracy(k_accuracy,:); ...
        %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
    set(gca,'xscale','log')
    set(gca,'yscale','log')
    hold on
end
if length(accuracy_list)>1
    legend(string(accuracy_list))
end
xlabel("n")
ylabel("RMSE")
grid on
set(gca,'FontSize',20)
box on
title("Influence number of rays on RMSE (\beta = 0.5)")



% y-axis RMSE  x-axis number of rays for beta = 0.5
figure;
for  k_rays =  1: length(number_of_rays_list)
    s3 = scatter(number_of_rays_list(k_rays),  root_mean_squared_error(1, ...
        k_rays), 'filled');
    s3.MarkerFaceColor = colour_choice_number_rays(k_rays,:); ...
        %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
    set(gca,'xscale','log')
    set(gca,'yscale','log')
    hold on
end
xlabel("n")
ylabel("RMSE")
ylim(gca,[1e-2 1])
title("Influence number of rays on RMSE (\beta = 0.5)")
grid on
set(gca,'FontSize',20)
box on

% y-axis RMSE  x-axis a_dome  for n = 1e6 and beta = 3 graden
figure;
for  k_rays =  1: length(number_of_rays_list)
    s3 = scatter(number_of_rays_list(k_rays),  root_mean_squared_error(1, ...
        k_rays), 'filled');
    s3.MarkerFaceColor = colour_choice_number_rays(k_rays,:); ...
        %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
    set(gca,'xscale','log')
    set(gca,'yscale','log')
    hold on
end
xlabel("n")
ylabel("RMSE")
```

```
title("Accuracy test of the ray tracer")
grid on
set(ax2,'FontSize',20)
box on

% y-axis run time, x-axis number of rays
figure;
for  k_rays =  1: length(number_of_rays_list)
    s3 = scatter(number_of_rays_list(k_rays),  mean_run_time(k_rays), 'filled');
    s3.MarkerFaceColor = colour_choice_number_rays(k_rays,:); ...
       %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
    set(gca,'xscale','log')
    set(gca,'yscale','log')
    hold on
end
xlabel("n")
ylabel("t (s)")
title("Run time per number of rays")
grid on
set(gca,'FontSize',20)
box on



end



if make_comparison_plots_a_dome == true
    for k_a_dome =  1: length(a_dome_inverse_list)
    a_dome = 1./a_dome_inverse_list(k_a_dome);

    % load data
    for k_number_of_runs = 1: number_of_repeating_runs
        if a_dome > 1.9
             mark = sprintf("Lambertian_rays_%.f_a_dome_%.f_%d", ...
                number_of_rays_specific,a_dome, k_number_of_runs);
        else
             mark = sprintf("Lambertian_rays_%.f_a_dome_%.2f_%d", ...
                number_of_rays_specific,a_dome, k_number_of_runs);
        end

        FileName = strcat('C:\Users\leoni\Documents\SET\7 8  Graduation ...
           Assignment\Code\RayTracingResultsLambertianAdome'); mkdir(FileName)
        loaded_data = load(strcat(FileName,'\',mark,'.mat'));
        radiance_per_run(:,:,k_number_of_runs) = loaded_data.radiance;
        run_time_per_run(:,k_number_of_runs) = loaded_data.run_time;
    end
    radiance = mean(radiance_per_run,3);
    mean_run_time(k_a_dome) = mean(run_time_per_run,2);

    % compute error

    [brdf, albedo(k_a_dome,:)]= computeBRDF(radiance, 1, ...
       number_of_rays_specific, azimuth_grid_dome_deg, elevation_grid_dome_deg);
    root_mean_squared_error(k_a_dome) = computeErrorLambertian(brdf);

    end
```

```
% y-axis RMSE  x-axis a_dome  for n = 1e6 and beta = 3 graden
figure;
a_dome_list = 1./a_dome_inverse_list;
for   k_a_dome =  1: length(a_dome_inverse_list)
    s3 = scatter(a_dome_list(k_a_dome),  root_mean_squared_error(k_a_dome), ...
        'filled');
    s3.MarkerFaceColor = colour_choice_number_rays(5,:); ...
        %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
    set(gca,'xscale','log')
    set(gca,'yscale','log')
    hold on
end
xlabel("a_{dome}")
ylabel("RMSE")
title("Relation RMSE and a_{dome} for n = 1e^6 and \beta = 3")
grid on
set(gca,'FontSize',20)
yticks([1e-3 1e-2 1e-1 1 1e1 1e2])
ylim([1e-3 1e2])
box on


% y-axis run time, x-axis a_dome
figure;
for  k_a_dome =  1: length(a_dome_inverse_list)
    s3 = scatter(a_dome_list(k_a_dome),  mean_run_time(k_a_dome), 'filled');
    s3.MarkerFaceColor = colour_choice_number_rays(5,:); ...
        %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
    set(gca,'xscale','log')
    set(gca,'yscale','log')
    hold on
end
xlabel("n")
ylabel("t (s)")
title("Run time per number of rays")
grid on
set(gca,'FontSize',20)
box on
end

if make_comparison_plots_h == true

   for k_a_dome =  1: length(a_dome_inverse_list)
        a_dome = 1./a_dome_inverse_list(k_a_dome);

        % load data
        for k_number_of_runs = 1: number_of_repeating_runs
            mark = sprintf("Lambertian_rays_%.f_a_dome_vert_%.2f_%d", ...
                number_of_rays_specific,a_dome, k_number_of_runs);

            FileName = strcat('C:\Users\leoni\Documents\SET\
            7 8  Graduation Assignment\Code\LibraryLambertian\
            RayTracingResultsLambertianH'); mkdir(FileName)
            loaded_data = load(strcat(FileName,'\',mark,'.mat'));
            radiance_per_run(:,:,k_number_of_runs) = loaded_data.radiance;
            run_time_per_run(:,k_number_of_runs) = loaded_data.run_time;
        end
        radiance = mean(radiance_per_run,3);
        mean_run_time(k_a_dome) = mean(run_time_per_run,2);
```

```
        % compute error

        [brdf, albedo(k_a_dome,:)]= computeBRDF(radiance, 1, ...
            number_of_rays_specific, azimuth_grid_dome_deg, ...
            elevation_grid_dome_deg);
        root_mean_squared_error(k_a_dome) = computeErrorLambertian(brdf);

    end

    % y-axis RMSE  x-axis a_dome  for n = 1e6 and beta = 3 graden
    figure;
    a_dome_list = 1./a_dome_inverse_list;
    for   k_a_dome =  1: length(a_dome_inverse_list)
        s3 = scatter(a_dome_list(k_a_dome),  ...
            root_mean_squared_error(k_a_dome), 'filled');
        s3.MarkerFaceColor = colour_choice_number_rays(5,:); ...
            %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
        set(gca,'xscale','log')
        set(gca,'yscale','log')
        hold on
    end
    xlabel("a_h")
    ylabel("RMSE")
    title("Relation RMSE and a_h for n = 1e^6 and \beta = 3")
    ylim([1e-3 1])
    yticks([1e-3 1e-2 1e-1 1])
    xlim([1 1e8])
    grid on
    set(gca,'FontSize',20)
    box on




    % y-axis run time, x-axis a_dome
    figure;
    for  k_a_dome =  1: length(a_dome_inverse_list)
        s3 = scatter(a_dome_list(k_a_dome),  mean_run_time(k_a_dome), 'filled');
        s3.MarkerFaceColor = colour_choice_number_rays(5,:); ...
            %repmat(colour_choice(k_accuracy,:),length(number_of_rays_list),1);
        set(gca,'xscale','log')
        set(gca,'yscale','log')
        hold on
    end
    xlabel("n")
    ylabel("t (s)")
    title("Run time per number of rays")
    grid on
    set(gca,'FontSize',20)
    box on



end
```

D.1.8 Plot

plotCoordinateSystem

```
function plotCoordinateSystem(sphere_type,r)
%==========================================================================
% Plots the polar and cartesian coordinate system for a sphere or a
% hemisphere.
%==========================================================================
% Input:  - sphere_type     string     Specifies the type of coordinate
%                                      system that has to be plotted.
%         - r               float      Radius of the coordinate system
%                                      plotted
%==========================================================================
% Debugging features
%         - The Boolean use_coordinate_system_rg can be used to label the
%           phi's and theta's according to the coordinate system of rg.
%==========================================================================
% Update log:
%         -  08/05/2023 Added viewpoint
%         -  06/04/2023 Added radius as input parameter
%         -  31/03/2023 Created out of plotReflectionLobe
% -------------------------------------------------------------------------
% Update possibilities:
%         - Changing coordinate system (azimuth, zenith etc.)
%==========================================================================


%% Settings
%r = 5; % radius of the coordinate system plotted
resolution_dome = 30; % resolution of dome
display_size_dome = false;


% Colour azimuth related
colour_azimuth = [0.0000 0.9800 0.0000];
colour_zenith = [0.2000 0.2000 0.8000];


% axes
use_coordinate_system_rg = false;

%% Initialization
number_of_points_axes = 2*r+1;

% Debugging Code
if use_coordinate_system_rg == true % coordinate system of rg
    azimuth_display = ["-90", "0", "", "90"];
else
    % our coordinate system
    azimuth_display = [0 90 180 -90]; % anti-clockwise, starting from x>0
end

%% Initialize based on sphere_type
switch sphere_type
    case 'sphere'
        r_st = -r;
        number_of_points_axes_z = number_of_points_axes;
        full_sphere = true;
    case 'hemisphere'
        r_st = 0; % radius depending on sphere type
        number_of_points_axes_z = r+1;
        full_sphere = false;
    otherwise
```

```
        error(sprintf ('%s is not a valid sphere_type. Choose "sphere" or ...
           "hemisphere".'), sphere_type)
end



%% Plot and label axes (x,y,z)
figure();

%plotting axes
plot_origin_line_z = ...
   plot3(zeros(number_of_points_axes_z,1),zeros(number_of_points_axes_z,1), ...
   [r_st:r],'Color',"#A9A9A9",'LineWidth',2 );
hold on
plot_origin_line_x = plot3([-r:r],zeros(number_of_points_axes,1), ...
   zeros(number_of_points_axes,1),'Color', "#A9A9A9",'LineWidth',2 );
hold on
plot_origin_line_x = plot3(zeros(number_of_points_axes,1),[-r:r], ...
   zeros(number_of_points_axes,1),'Color', "#A9A9A9",'LineWidth',2 );
hold on

%limits axes
xlim([-r,r])
ylim([-r,r])
zlim([r_st,r])

%label axes
% text(r/2,-0.2,0,'x','HorizontalAlignment','left','FontSize',12,'Color','k');
% text(-0.2,r/2,0,'y','HorizontalAlignment','left','FontSize',12,'Color','k');
% text(0,-0.1,r/2,'z','HorizontalAlignment','left','FontSize',12,'Color','k');


% Setting ticks
xticks([-r,0,r])
yticks([-r,0,r])
if full_sphere == false
zticks([0,r])
else
zticks([-r,0,r])
end

% Setting tick labels
if display_size_dome == true
    xticklabels({sprintf('x = %.f', r) , 'x = 0', sprintf('x = %.f', -r)});
    yticklabels({sprintf('y = %.f', r) , 'y = 0', sprintf('y = %.f', -r)});
    if full_sphere == false
        zticklabels({'z = 0',sprintf('z = %.f', r) });
    else
        zticklabels({sprintf('z = %.f', r) , 'z = 0', sprintf('z = %.f', -r)});
    end
else
    xticklabels({'','y = 0',''});
    yticklabels({'','x = 0',''});
    if full_sphere == false
        zticklabels({'z = 0',''});
    else
        zticklabels({'', 'z = 0',''});
    end
end
```

```
%% Plot angles (theta, phi)
% Displaying the angles in the plot (anti-clockwise, starting from x>0!)
if r <= 1
    add_space = 0.1;
else
    add_space = r*0.1;
end

% Display azimuth
text(r + 1.5*add_space,     0,              0,  strcat('\phi', sprintf('= ...
    %.f', azimuth_display(1))),'HorizontalAlignment','left','FontSize',12, ...
    'Color',colour_azimuth);
text(0,                 r+add_space*3,  0,  strcat('\phi', sprintf('= %.f', ...
    azimuth_display(2))),'HorizontalAlignment','left','FontSize',12, ...
    'Color',colour_azimuth);
text(-r - 3* add_space,    -add_space,    0.5* add_space,  strcat('\phi', ...
    sprintf('= %.f', ...
    azimuth_display(3))),'HorizontalAlignment','left','FontSize',12, ...
    'Color',colour_azimuth);
text(add_space,         -r+add_space*3, 0,  strcat('\phi', sprintf('= %.f', ...
    azimuth_display(4))),'HorizontalAlignment','left','FontSize',12, ...
    'Color',colour_azimuth);

%Display elevation angle
text(0,0,r+1.5*add_space,'\theta = ...
    90','HorizontalAlignment','left','FontSize',12,'Color',colour_zenith);
text(r+1.5*add_space,0,0+add_space,'\theta = ...
    0','HorizontalAlignment','left','FontSize',12,'Color',colour_zenith);

%% Plot dome and circles
% generate half a dome (half a sphere)
[x_to_plot,y_to_plot,z_to_plot] = sphere(resolution_dome);
if full_sphere == false
    x_to_plot = x_to_plot(round(resolution_dome/2)+1:end,:);
    y_to_plot = y_to_plot(round(resolution_dome/2)+1:end,:);
    z_to_plot = z_to_plot(round(resolution_dome/2)+1:end,:);
end

% plot dome
surf(r*x_to_plot,r*y_to_plot,r*z_to_plot,'FaceColor',[0.3010 0.7450 ...
    0.9330],'FaceAlpha',0.2, 'EdgeAlpha',0);
hold on

% indicate theta
th = linspace(0,90,resolution_dome);
y = zeros(5,1);
x = r*cosd(th);
z = r*sind(th);
x = repmat([x 0 x(1)],5,1);
z = repmat([z 0 z(1)],5,1);
surf(x, zeros(size(x)),  z ,'FaceAlpha',0.8, 'EdgeAlpha',0.5, 'EdgeColor', ...
    colour_zenith,'LineWidth',3)


% indicate phi
phi = linspace(0,360,resolution_dome);
z = zeros(5,1);
x = r*cosd(phi);
```

```
y = r*sind(phi);
x = repmat([x 0 x(1)],5,1);
y = repmat([y 0 y(1)],5,1);
surf(x, y, zeros(size(x)),'FaceAlpha',0.8, 'EdgeAlpha',0.5, 'EdgeColor', ...
    colour_azimuth,'LineWidth',3)


%% Other settings used
box off
if full_sphere == false
    pbaspect([1 1 0.5])
else
    pbaspect([1 1 1])
end

set(gca,'fontSize',12)
ax = gca;
ax.Visible = "off";
view([-30 -10])




end
```

plotGridDetectionDome

```
function ...
    plotGridDetectionDome(azimuth_grid_dome_deg,elevation_grid_dome_deg, ...
    radius, accuracy_detection_dome)


% Note: number of arguments in (nargin == 4): highlight a surface area on ...
    the sphere


%% Generate all longitudes and latitudes
if nargin == 4 % highlight a patch: find the elevation and azimuths of the ...
    quadrangle
    elevations = [ elevation_grid_dome_deg - (accuracy_detection_dome) / 2, ...
        elevation_grid_dome_deg + (accuracy_detection_dome) / 2 ];
    azimuths = [azimuth_grid_dome_deg - (accuracy_detection_dome) / 2, ...
        azimuth_grid_dome_deg + (accuracy_detection_dome) / 2];

    elevation_around  = linspace( ...
        deg2rad(elevations(1)),deg2rad(elevations(2)) ,50);
    azimuth_around =    linspace( deg2rad(azimuths(1))  ...
        ,deg2rad(azimuths(2))   ,100);

else
    % generate the lines of the pixels over the entire dome
    elevations = [ elevation_grid_dome_deg(2:end) + ...
        (elevation_grid_dome_deg(1:end-1) - elevation_grid_dome_deg(2:end)) / ...
        2 ; 0];
    azimuths = [-180, azimuth_grid_dome_deg(2:end) + ...
        (azimuth_grid_dome_deg(1:end-1) - azimuth_grid_dome_deg(2:end)) / 2];
```

```
    azimuth_around = linspace(0, 2 * pi,100);
    elevation_around = linspace(0,pi, 50);
end

%% plot elevation line(s)
for k_elevations = 1:length(elevations)
    elevation = deg2rad(elevations(k_elevations));
    [x,y,z] = sph2cart(azimuth_around, repmat(elevation, 1, 100), ...
       repmat(radius, 1, 100));

    if nargin == 4
        p = plot3(x,y,z,"Color","#00BFFF", "LineWidth", 3); % highlight
    else
        p = plot3(x,y,z,"Color", "#00BFFF"); % display over the whole dome
    end
    p.Color(4) = 0.5; % set transparency
end

%% Plot azimuth line(s)
for k_azimuths= 1:round(length(azimuths)/2) +1
    azimuth = deg2rad(azimuths(k_azimuths));
    [x,y,z] = sph2cart(repmat(azimuth, 1, 50), elevation_around, ...
       repmat(radius, 1, 50));

    if nargin == 4
        p = plot3(x,y,z,"Color","#00BFFF", "LineWidth", 3); % highlight
    else
        p = plot3(x,y,z,"Color","#00BFFF"); % display over the whole dome
    end
    p.Color(4) = 0.5; % set transparency
end

%% Plot centers of every pixel
[xc, yc, zc]  = sph2cart(deg2rad(azimuth_grid_dome_deg), ...
   deg2rad(elevation_grid_dome_deg), radius);
scatter3(xc,yc,zc,3,'filled',"MarkerFaceColor","#00BFFF","MarkerFaceAlpha",0.8);
end
```

plotNormal

```
function plotNormal(normal_plane, surface_boundaries)
%=========================================================================
% Plots the normal of a plane
%=========================================================================
% Input:  - normal_plane        3x1 matrix   Normal of the plane ([x;y;z])
%         - surface_boundaries  struct  Contains at least the shape of the
%                                       surface.
%                                       - If the shape is a "Square",
%                                         contains the corners of the
%                                         surface.
%                                       - If the shape is a "Circle",
%                                         contains the radius, center
%                                         and height.
% ------------------------------------------------------------------------
% Output: plot of the normal
%=========================================================================
% Update log:
```

```
%   04/05/2023  Added circular option.
%=========================================================================

% Find starting point of the normal
switch surface_boundaries.shape
    case "Circle"
        starting_point = surface_boundaries.center;
    case "Rectangle"
        % find middle of the plane
        starting_point = [mean(surface_boundaries.corners(1,:)); ...
           mean(surface_boundaries.corners(2,:)); ...
           mean(surface_boundaries.corners(3,:))];
end

% Make a vector using the normal as direction
t = linspace(0,0.5,1.5);
normal_to_plot = starting_point + t.* normal_plane ;

% Plot the normal
plot3(normal_to_plot(1,:), normal_to_plot(2,:),normal_to_plot(3,:))


end
```

plotPoint

```
function plotPoint(ray_starting_point, color)
    scatter3(ray_starting_point(1), ...
       ray_starting_point(2),ray_starting_point(3), 15,color,'filled');

end
```

plotRays

```
function plotRays(ray_starting_point, ray_direction,t_plot)

t = t_plot;
ray = ray_starting_point + t.*ray_direction;
plot3(ray(1,:), ray(2,:),ray(3,:),'k', 'LineWidth',1)
hold on
end
```

plotReflectionLobe

```
function [reflectance_to_plot_x,reflectance_to_plot_y,reflectance_to_plot_z] ...
    = plotReflectionLobe(rg, azimuth_angle, elevation_angle)
%=========================================================================
% Produces a plot of the reflection lobe and the specularly reflected line.
%-------------------------------------------------------------------------
% Input:- rg              matrix          Values of glossy reflection
%       - azimuth_angle   matrix          azimuth angles
%       - elevation_angle matrix          elevation angles
%-------------------------------------------------------------------------
```

```
% Output:    - figure displaying the reflection lobe
%=========================================================================
% Update information
%   24/04/2023  Added extra datapoint to form a full lobe
%   06/04/2023  Deleted multiplication of rg(2:end,2:end) by
%               cosd(elevation_angle)
%   31/03/2023  Function simplified by taking out plotting coordinate
%               system and taking out plotting one specularly reflected beam
%   20/03/2023  Function made out of scraps of plot_tryout
%=========================================================================
% Possible Future Updates
%       - inputs azimuth_angle and elevation_angle not needed bescause in rg
%
%=========================================================================

[reflectance_to_plot_x,reflectance_to_plot_y,reflectance_to_plot_z] = ...
   sph2cart(deg2rad(azimuth_angle),deg2rad(elevation_angle),rg(2:end,2:end));
reflectance_to_plot_x = [reflectance_to_plot_x, reflectance_to_plot_x(:,1)];
reflectance_to_plot_y = [reflectance_to_plot_y, reflectance_to_plot_y(:,1)];
reflectance_to_plot_z = [reflectance_to_plot_z, reflectance_to_plot_z(:,1)];
surf_reflectance = surf(reflectance_to_plot_x ...
   ,reflectance_to_plot_y,reflectance_to_plot_z, 'EdgeAlpha',0);

end
```

plotSurface

```
function plotSurface(surface_boundaries)
%=========================================================================
% Plots a surface
%=========================================================================
% Input:  - surface_boundaries  struct  Contains at least the shape of the
%                                       surface.
%                                       - If the shape is a "Square",
%                                         contains the corners of the
%                                         surface.
%                                       - If the shape is a "Circle",
%                                         contains the radius, center
%                                         and height.
% ------------------------------------------------------------------------
% Output: - plot of a surface
%=========================================================================
% Notes: - The square surface is considered as follows:
%
%           1 ------2         Corner 1 : x1, y1, z1
%           |       |         Corner 2 : x2, y2, z2
%           |       |         Corner 3 : x3, y3, z3
%           |       |         Corner 4 : x4, y4, z4
%           4-------3
%
%=========================================================================
% Update log:
%   04/05/2023  Added circular option.
%=========================================================================

switch surface_boundaries.shape
    case "Circle"
        azimuth = linspace(0,2*pi,100);
```

```
        x = surface_boundaries.center(1) + surface_boundaries.radius * ...
            cos(azimuth);
        y = surface_boundaries.center(2) + surface_boundaries.radius * ...
            sin(azimuth);
        z = repmat(surface_boundaries.center(3) + ...
            surface_boundaries.height,1,100);
        plot3([x], [y], [z])

    case "Rectangle"
        plot3([surface_boundaries.corners(1,:) ...
            surface_boundaries.corners(1,1)], ...
            [surface_boundaries.corners(2,:) ...
            surface_boundaries.corners(2,1)], ...
            [surface_boundaries.corners(3,:) surface_boundaries.corners(3,1)])
    end
end
```

D.1.9 Other

addToLibrary

```
% add to library
clear data_library data

% library_location = "C:\Users\leoni\Documents\SET\7 8  Graduation ...
    Assignment\AnnesCode\";
% library_name = "GrassTilt40";

%% Settings for BRDF grass library
% Retrieve files
folder_to_add  = "C:\Users\leoni\Documents\SET\7 8  Graduation ...
    Assignment\Code\Library\TiltedTop40_Bottom4cm_Top8cm\";
name_individual_files = ...
    "VerticalTiltedTop40_Bottom4cm_Top8cm_Elevation_%.2f_%d";

% Where to save mean BRDFs
library_location = "C:\Users\leoni\Documents\SET\7 8  Graduation ...
    Assignment\Code\Library\";
library_name = "TiltedTop40_Bottom4cm_Top8cm";

% Settings file names to loop over
elevation_angles = [0:6:72];
number_of_runs = 10;

%% Settings for convergence grass library
% Retrieve files
folder_to_add  = "C:\Users\leoni\Documents\SET\7 8  Graduation ...
    Assignment\Code\LibraryConvergenceAnalysisNew\";
name_individual_files = "Convergence_aillumination_1.00_El_%.2f_%d";

% Where to save mean BRDFs
library_location = "C:\Users\leoni\Documents\SET\7 8  Graduation ...
    Assignment\Code\LibraryConvergenceAnalysisNew\";
library_name = "Convergence_aillumination_1.00.mat";

% Settings file names to loop over
elevation_angles = [0:10:90];
```

```
number_of_runs = 10;



%% Generate mean BRDF and albedo
for k_angle = 1: length(elevation_angles) % for every angle
    angle = elevation_angles(k_angle);
    angle_name = sprintf("Angle%d", angle);

    for k_run = 1 : number_of_runs % for every run
        run_name = sprintf("Run%d", k_run);

        % retrieve data
        data = load(strcat(folder_to_add, sprintf(name_individual_files, ...
           angle,  k_run  )) );

        % temporarily save data
        brdf(:,:,:,k_run) = data.brdf;
        albedo(k_run,:) = data.albedo;
    end

    % compute mean albedo and BRDF
    data_library.(angle_name).mean_brdf = mean(brdf, 4);
    data_library.(angle_name).mean_albedo= mean(albedo, 1);

    % also save azimuths and elevations detection dome
    if k_angle == length(elevation_angles)
         data_library.azimuths_detection_dome = data.azimuth_grid_dome_deg;
         data_library.elevations_detection_dome = data.elevation_grid_dome_deg;
    end

end

%% Save mean BRDFs and albedo
save(strcat(library_location,library_name), 'data_library')
```

detectRay

```
function  radiance = detectRay(radiance, ...
     ray_magnitude , intersection_point, radius_sphere, ...
    azimuth_grid_dome_deg, elevation_grid_dome_deg)      % angles related
%=========================================================================
% Detects the ray on the detection dome
%=========================================================================
% Input:  - radiance             matrix         Contains the detected
%                                                  reflectance.
%                                                  Rows: elevation angle
%                                                  Columns: azimuth angle
%                                                  Sheets: wavelengths
%         - intersection_point      3x1 matrix     Intersection point of
%                                                  the ray with the
%                                                  detection dome [x;y;z]
%         - ray_magnitude           1xn matrix     Magnitude ray for n
%                                                  wavelengths
%         - radius_sphere           float          Radius of the detection
%                                                  dome
%         - azimuth_grid_dome_deg   matrix     Range of all azimuth
```

```
%                                                   angles of dome (degrees)
%         - elevation_grid_dome_deg matrix     Range of all elevation
%                                                   angles of dome (degrees)
% -------------------------------------------------------------------------
% Output: - radiance             matrix         Contains the detected
%                                                   reflectance, including
%                                                   the magnitude of the
%                                                   ray that intersects
%                                                   with the dome
%=========================================================================
% Internal functions used:
%         - checkRadius       Checks if intersection point is on the
%                              detection dome
%-------------------------------------------------------------------------
% External functions used:
%         - findIndexQueryPoint   Finds index of a query point in range.
%         - plotGridDetectionDome Highlights the pixel that has been hit
%=========================================================================
% Debugging features:
%         - The function checkRadius checks if intersection point is on the
%           dectection dome. If not, an error is shown.
%         - The function plotGridDetectionDome can be used to check if the
%           intersection point is counted towards the correct pixel.
%=========================================================================
% Update log
% 11/05/2023    Added pixel highlight option.
% 14/04/2023    Adapted the grid of the detection dome. Pulled this definition
%               outside of this funtion. Redefined the way the index of the ...
   detection
%               matrix is determined.
%=========================================================================

%% Intersection point ray-detection dome in spherical coordinates
% Find the spherical coordinates of the intersection point
[azimuth_sky,elevation_sky,sphere_radius_check] = ...
    cart2sph(intersection_point(1),intersection_point(2),intersection_point(3));
% check if radius is as expected
checkRadius (radius_sphere, sphere_radius_check);

%% Find the azimuth and elevation indexes in the detect_ray matrix
azimuth_sky = rad2deg(azimuth_sky);
elevation_sky = rad2deg(elevation_sky);

% Find the right index based on the azimuth and elevation.
[azimuth_i, elevation_i] = findIndexQueryPoint ...
    (azimuth_grid_dome_deg,elevation_grid_dome_deg, azimuth_sky, elevation_sky);

% Debugging code: highlight the pixel that is hit
%plotGridDetectionDome(azimuth_grid_dome_deg(azimuth_i), ...
    elevation_grid_dome_deg(elevation_i), radius_sphere, ...
    azimuth_grid_dome_deg(2)-azimuth_grid_dome_deg(1))

%% Add the magnitude of the deteced ray to the radiance matrix
to_add = reshape(ray_magnitude,[1,1, length(ray_magnitude)]);
radiance(elevation_i, azimuth_i,:) = radiance( elevation_i, azimuth_i,: ) + ...
    to_add;


%% Internal functions
function checkRadius (sphere_radius, sphere_radius_check)
```

```
    % sphere_radius = actual radius as entered
    % sphere_radius_check =  radius as computed from cart2sph
    allowed_radius_error = 0.005;
    if abs(sphere_radius_check) > abs(sphere_radius + allowed_radius_error)
        error("The radius of the dome is incorrect.")
    end
end

end
```

findIndexQueryPoint

```
function [azimuth_index, elevation_index] = findIndexQueryPoint ...
    (azimuth_range,elevation_range, azimuth_query, elevation_query)
%=========================================================================
% Finds the index of the value closest to azimuth_query and elevation_query
% in azimuth_range and elevation_range.
%=========================================================================
% Input:  - azimuth_range       1 x n matrix
%         - elevation_range     n x 1 matrix
%         - azimuth_query       float
%         - elevation_query     float
% -------------------------------------------------------------------------
% Output: - azimuth_index        integer     [explanation]
%         - elevation_index     integer      [explanation]
%=========================================================================


azimuth_index = find( abs(azimuth_query - azimuth_range) ==  ...
    min(abs(azimuth_query - azimuth_range)),1);
elevation_index = find( abs(elevation_query - elevation_range) ==  ...
    min(abs(elevation_query - elevation_range)),1);

% Because of this definition, a query point P on the boundary of the grid,
% exactly between two centers C given by azimuth_index and elevation_index,
% will be assigned to the left C.
% Exception: if azimuth_query = 180 or -180, then the point P is assigned
% to the last and first C, respectively

end
```

gong

```
function gong(vol,frq,dur)
% gong: sounds gong
% by John Gooden - The Netherlands
% 2007
%
% call gong
% call gong(vol)
% call gong(vol,frq)
% call gong(vol,frq,dur)
%
% input arguments (optional, if 0 then default taken)
% vol = volume (default = 1)
```

```
% frq = base frequency (default = 440 Hz)
% dur = duration (default = 1 s)
fb  = 440;
td  = 1;
vl  = 1;
if nargin>=1
    if vol>0 vl = vol; end
end
if nargin>=2
    if frq>0 fb = frq; end
end
if nargin>=3
    if dur>0 td = dur; end
end

t   =[0:8192*td]'/8192;
env = exp(-5*t/td);
f   = fb;
vol = 0.3*vl;
tpft = 2*pi*f*t;
sl  = sin(tpft)+0.1*sin(2*tpft)+0.3*sin(3*tpft);
sl  = vol*sl;
sr  = [sl(100:end);sl(1:99)];
vl  = cos(20*t);
vr  = 1-vl;
y(:,1) = vl.*env.*sl;
y(:,2) = vr.*env.*sr;
sound(y)
```

reduceResolutionData

```
function [data_reduced, column_axis_reduced, row_axis_reduced] = ...
   reduceResolutionData(data, column_axis,row_axis,accuracy, accuracy_desired)
%=========================================================================
% Reduces the resolution of the data with a certain accuracy to obtain
% data_reduced, which has a reduced resolution, with a accuracy_desired.
%=========================================================================
% Input:  - data                    a x b matrix    Data to be reduced
%         - column_axis             b array         Columns axis of data
%         - row_axis                a array         Rows axis of data
%         - accuracy                float           Current accuracy of
%                                                   data
%         - accuracy_desired        float           Desired accuracy of
%                                                   data
% -------------------------------------------------------------------------
% Output: - data_reduced            matrix          Data with a reduced
%                                                   resolution
%         - column_axis_reduced     b array         Columns axis of reduced
%                                                   data
%         - row_axis_reduced        a array         Rows axis of reduced
%                                                   data
%=========================================================================
% Notes:  - The data matrix should be such that the steps in the rows and
%           the column are the same. Example: (accuracy = 5)
%                         data
%           X     | 5      10      15      20    <- column_axis
%           ------------------------------------
%           2.5   | 1       2       3       4
```

```
%           7.5   | 1       2       3       4
%           12.5  | 5       6       7       8
%           17.5  | 5       6       7       8
%            ^row_axis
%=========================================================================
% Debugging features:
%         - Desired accuracy should be smaller than current accuracy.
%         - Desired accuracy should fit an integer times in current
%           accuracy.
%=========================================================================

    %% Debugging checks
    % Check if the desired accuracy is possible
    if accuracy_desired < accuracy
        error("The desired accuracy has to be larger than the current ...
           accuracy.")
    end

    if rem(accuracy_desired, accuracy) > 0
        error("The desired accuracy has to be an integer times larger than ...
           the current accuracy.")
    end


    %% Initialization
    accuracy_factor = accuracy_desired ./ accuracy;

    if rem(360, accuracy_desired) ~= 0
        error("The desired accuracy has to be able to generate a uniform ...
           grid, also at the edges.")
    elseif rem(90, accuracy_desired) ~= 0
        error("The desired accuracy has to be able to generate a uniform ...
           grid, also at the edges.")
    end

    % New matrix columns
    column_start =  column_axis(1) - accuracy/2;
    column_axis_reduced(1) = column_start + accuracy_desired/2;
    k_column = 1;
    while column_axis_reduced(k_column) + accuracy_desired < 180
        k_column = k_column + 1;
        column_axis_reduced(k_column) = column_axis_reduced(k_column-1) + ...
           accuracy_desired;
    end

    % New matrix rows
    row_start =  row_axis(1) + accuracy/2;
    row_axis_reduced(1) = row_start - accuracy_desired/2;
    k_row = 1;
    while row_axis_reduced(k_row,1) - accuracy_desired > 0
        k_row = k_row + 1;
        row_axis_reduced(k_row,1) = row_axis_reduced(k_row-1,1) - ...
           accuracy_desired;
    end

    if 90 - row_axis_reduced(1) ~= row_axis_reduced(end) - 0
        error("The desired accuracy has to be able to generate a uniform ...
           grid, also at the edges.")
    elseif -180 - column_axis_reduced(1) ~= column_axis_reduced(end) - 180
        error("The desired accuracy has to be able to generate a uniform ...
```

```
            grid, also at the edges.")
    end

    %% Construct the reduced data
    % k_col_red index columns of reduced data
    % k_row_red index rows of reduced data
    % k_col index columns of data
    % k_row index rows of data

    for k_col_red = 1 : length(column_axis_reduced) % loop over the new ...
       matrix columns
       if k_col_red == 1
           k_col = 1;
       end
       for k_row_red = 1 : length(row_axis_reduced) % loop over the new ...
          matrix rows
           if k_row_red == 1
                k_row = 1;
           end
           data_reduced(k_row_red, k_col_red) = sum(data(k_row : k_row + ...
              accuracy_factor - 1, k_col : k_col + accuracy_factor - 1),'all');
           k_row = k_row + accuracy_factor; % next row
       end
       k_col = k_col + accuracy_factor; % next column
    end

end
```

D.2 Analyze Library

plotFromLibrary

```
% Retieve desired data
settings_plotter.folder = "C:\Users\leoni\Documents\SET\7 8  Graduation ...
   Assignment\Code\Library\";
settings_plotter.name_to_run = "MeanVaryingHeight4cm12cm.mat"; ...
   %"Convergence_aillumination_0.25.mat"; ...
   %"VerticalVaryingHeight4cm8cm_%.2f_%d";%"Convergence_aillumination_%.1f_El_%d_28May_%d"

%When compare plot Comparison
settings_plotter.for_comparison = ...
   ["Convergence_aillumination_1.00.mat","Convergence_aillumination_0.75.mat", ...
   "Convergence_aillumination_0.50.mat","Convergence_aillumination_0.25.mat"];

% Set data characteristics
settings_plotter.angles_to_run = [0:6:72];%[0:10:90];
settings_plotter.a_ill_list = 0.5;
settings_plotter.wavelengths = [300:10:1100];

% Saving plot
settings_plotter.name_to_save = "Test.jpg";


%%  plots BRDF
plotFromLibraryBRDFperElevationAngle(settings_plotter)
```

```
%% plots Albedo
plotFromLibrarySpectralAlbedoPerElevationAngle(settings_plotter)

%% comparison albedo between samples
plotFromLibraryAlbedoComparisonSamples(settings_plotter)

%% plot reflectance 2D polar plot
plotFromLibrary2DPolarPlot(settings_plotter)
```

plotFromLibrary2DPolarPlot

```
function plotFromLibrary2DPolarPlot(settings_plotter)
%% Settings
add_experimental_data_Jelle = false;
add_experimental_data_Jelle_separately = true;

%% Initialization
folder = settings_plotter.folder; %= 'C:\Users\leoni\Documents\SET\7 8  ...
    Graduation Assignment\Code\RayTracingResults28May\';
name_to_run = settings_plotter.name_to_run; %= ...
    'Convergence_aillumination_0.2_El_'; %'BCgrass_';%'BCsmall_elevation_';
angles_to_run = settings_plotter.angles_to_run;
wavelengths = settings_plotter.wavelengths;
name_to_save = settings_plotter.name_to_save;


% retrieve data simulations
data = load(strcat(folder,name_to_run));
elevations_detection_dome =  data.data_library.elevations_detection_dome;
azimuths_detection_dome =  data.data_library.azimuths_detection_dome;


% retrieve data Jelle
if or(add_experimental_data_Jelle == true, ...
    add_experimental_data_Jelle_separately == true)
    if length(angles_to_run) > 3
        error("Max 3 angles allowed!")
    end

    for k_angle = 1:3
    figdata_550nm_jelle = openfig("grass_550nm_alldata.fig",'invisible'); % ...
        angles 30 50 70
    theta_jelle(k_angle) = ...
        rad2deg(figdata_550nm_jelle.Children.Children(k_angle+1).
    Children(1).ThetaData(1));
    data_jelle_th(:,k_angle) = ...
        figdata_550nm_jelle.Children.Children(k_angle+1).Children(2).ThetaData';
    data_jelle_r(k_angle,:) = ...
        figdata_550nm_jelle.Children.Children(k_angle+1).Children(2).RData;
    experimental_angles = [30,50,70];
    end
end

desired_wavelength = 550; %nm
figure;
if add_experimental_data_Jelle_separately == true
    t = tiledlayout(2,3);
else
```

```
    t = tiledlayout(1,3);
end

for k_angle = 1: length(angles_to_run)
    nexttile;
    angle_name = angles_to_run (k_angle);
    brdf = data.data_library.(strcat("Angle", ...
        string(angle_name))).mean_brdf(:,:, find(desired_wavelength == ...
        wavelengths));
    brdf_at_phi0 = mean(data.data_library.(strcat("Angle", ...
        string(angle_name))).mean_brdf(:,[30, 31], find(desired_wavelength == ...
        wavelengths)),2);
    brdf_at_phi180 =  mean(data.data_library.(strcat("Angle", ...
        string(angle_name))).mean_brdf(:,[1, 60], find(desired_wavelength == ...
        wavelengths)),2);

    polarscatter(deg2rad(elevations_detection_dome),brdf_at_phi0,'filled', ...
        'b ');
    hold on
    polarscatter(pi - ...
        deg2rad(elevations_detection_dome),brdf_at_phi180,'filled', 'b ')
    title(strcat("\theta_{source} = ", string(angle_name),"  "))

    hold on
    polarplot(repmat(deg2rad(angle_name),10), ...
        linspace(0,1,10),'LineWidth',3, 'Color', 'y')

    % setting axes
    ax = gca;
    ax.RLim = [0 0.05];
    ax.RTick= [0:0.01:0.05];
    ax.RTickLabel = [0:0.01:0.05];
    ax.ThetaLim = [0 180];
    ax.ThetaTick = [0:15:180];
    ax.ThetaTickLabel = strcat(string([0:15:90 75:-15:0]), '  ');
    ax.RAxis.Label.String = "BRDF";
    set(gca,'FontSize',20)

    % adding label with angle of incidence
    text(deg2rad(angle_name),0.07,"$ \theta_{source}$" ,...
    'HorizontalAlignment','center',...
    'VerticalAlignment','bottom', 'Interpreter', 'latex', 'Color', 'y', ...
        'FontSize',20)

    if add_experimental_data_Jelle == true
        hold on
        polarscatter(data_jelle_th(:,k_angle) ,data_jelle_r(k_angle,:)','x','r')

        hold on
        polarplot(repmat(deg2rad(theta_jelle(k_angle)),10), ...
            linspace(0,1,10),'LineWidth',3, 'Color', 'r')

    end


end
    if add_experimental_data_Jelle_separately == true
    for k_angle = 1: length(angles_to_run)
        nexttile;
        angle_name = angles_to_run (k_angle);
```

```
        polarscatter(data_jelle_th(:,k_angle) ,data_jelle_r(k_angle,:)','x','r')

        hold on
        polarplot(repmat(deg2rad(theta_jelle(k_angle)),10), ...
            linspace(0,1,10),'LineWidth',3, 'Color', 'r')
        title(strcat("\theta_{source} = ", ...
            string(experimental_angles(k_angle))," "))
             text(deg2rad(angle_name),0.3,"$ \theta_{source}$" ,...
    'HorizontalAlignment','center',...
    'VerticalAlignment','bottom', 'Interpreter', 'latex', 'Color', 'r', ...
       'FontSize',20)
        ax = gca;
        ax.RLim = [0 0.25];
        ax.RTick= [0:0.05:0.25];
        ax.RTickLabel = [0:0.05:0.25];
        ax.ThetaLim = [0 180];
        ax.ThetaTick = [0:15:180];
        ax.ThetaTickLabel = strcat(string([0:15:90 75:-15:0]), '  ');
         ax.RAxis.Label.String = "Reflection (%)";
         set(gca,'FontSize',20)
    end
    end
    title(t, {"Comparison shape reflection to experimental data","",""}, ...
       'FontSize',20);
    set(gca,'FontSize',20)
end
```

plotFromLibraryAlbedoComparisonSamples

```
%compareFromLibrary
function plotFromLibraryAlbedoComparisonSamples(settings_plotter)
%% Initialization
% % Set data characteristics
% settings_plotter.angles_to_run = [0:6:72];  %[0.1 10:10:90];
% settings_plotter.a_ill_list = 0.5;%[1 0.75 0.5 0.25 ];
% settings_plotter.wavelengths = [300:10:1100];
%
% libraries = ["GrassVertical4cm","GrassVertical12cm","GrassVertical20cm", ...
   "GrassVaryingHeight"];%, "GrassTilt20", "GrassTilt40"];
% library_location = "C:\Users\leoni\Documents\SET\7 8  Graduation ...
   Assignment\Code\Library\";


%% Settings
desired_wavelength = 900; % nm

%% Initialization
%retrieve settings
folder = settings_plotter.folder;
name_to_run = settings_plotter.name_to_run;
angles_to_run = settings_plotter.angles_to_run;
wavelengths = settings_plotter.wavelengths;
a_ill_list = settings_plotter.a_ill_list;
name_to_save = settings_plotter.name_to_save;

libraries = settings_plotter.for_comparison;
```

```
colourlist = parula(length(angles_to_run));
wavelength_i_900nm = find(wavelengths == desired_wavelength);

%% Find mean albedo

for k_library =  1 : length(libraries)
    for k_angle = 1: length(angles_to_run)
        angle = angles_to_run(k_angle);
        load(strcat(folder, libraries(k_library)), 'data_library'); % 900 nm
        albedo_bar_plot(k_library, k_angle) =  ...
           data_library.(sprintf("Angle%d",angle)).mean_albedo(1,61);
    end

end
%%
figure;
b = bar(albedo_bar_plot, 'FaceColor','flat');
for k_angle = 1: length(angles_to_run)
    b(k_angle).CData(:,:,:) = repmat(colourlist(k_angle,:), ...
       length(libraries),1);
end

ylabel('$$\rho(\lambda = 900 nm)$$', 'Interpreter', 'latex')
xticklabels(["Vertical  4 cm", "Vertical 8 cm", "Vertical 12 cm", "Varying ...
   height 4-12 cm", "Tilted top 20   bottom 4cm top 4cm", "Tilted top 40   ...
   bottom 4cm top 4cm", "Tilted top 20   bottom 8cm top 4cm", "Tilted top ...
   40   bottom 8cm top 4cm","Tilted top 20   bottom 4cm top 8cm","Tilted top ...
   40   bottom 4cm top 8cm"])
ylim([0, 0.3])
yticks(0:0.01:0.3)
y_ticks = string(0:0.01:0.3);
y_ticks(2:2:end-1) = "";
yticklabels(y_ticks)
l = legend(strcat('$$\theta_{source} = $$', ...
   {'$$10^{-5}$$','6','12','18','24', '30','36','42', '48', '54', '60', ...
   '66', '72'}));
l.Title.String = 'Elevation angle (degree)';
l.Interpreter = 'latex';
l.Location = 'northeastoutside';
l.FontSize = 14;
title("Comparison of albedo at $$\lambda = 900$$ nm for various samples", ...
   'Interpreter', 'latex')
set(gca,'FontSize',14)

grid on
```

plotFromLibraryBRDFperElevationAngle

```
function [] = plotFromLibraryBRDFperElevationAngle(settings_plotter)

%% Settings
plot_3D = false;
desired_wavelength = 900;% nm

%% Initialization

folder = settings_plotter.folder; %= 'C:\Users\leoni\Documents\SET\7 8  ...
   Graduation Assignment\Code\RayTracingResults28May\';
```

```
name_to_run = settings_plotter.name_to_run; %= ...
    'Convergence_aillumination_0.2_El_'; %'BCgrass_';%'BCsmall_elevation_';
angles_to_run = settings_plotter.angles_to_run;
wavelengths = settings_plotter.wavelengths;
a_ill_list = settings_plotter.a_ill_list;
name_to_save = settings_plotter.name_to_save;


% retrieve data
data = load(strcat(folder,name_to_run));
elevations_detection_dome =  data.data_library.elevations_detection_dome;
azimuths_detection_dome =  data.data_library.azimuths_detection_dome;

%% Make 2D figure
figure;
tiles = tiledlayout(1,length(angles_to_run));
tiles.TileSpacing = 'tight';
title_figure = title(tiles,{'BRDF for wavelength \lambda = 900 nm', ''});
for k_options = 1:length(angles_to_run)
    nexttile;
    angle_name = angles_to_run(k_options);
    i_wavelength = find(wavelengths== desired_wavelength,1);
    brdf_to_plot = data.data_library.(strcat("Angle", ...
        string(angle_name))).mean_brdf(:,:,i_wavelength);
    brdf_3Dplot(k_options,:,:) = brdf_to_plot;

    imagesc( elevations_detection_dome,azimuths_detection_dome,   ...
        brdf_to_plot', [0 0.2])


    accuracy_dome = abs(azimuths_detection_dome(2)- azimuths_detection_dome(1));
    xlim([0,90])
    ylim([-180,180])
    xticks([0:accuracy_dome:90])
    yticks([-180:accuracy_dome:180])
    pbaspect([1 4 1])
    if k_options >1
        set(gca,'YTickLabel',[]);
    end
    ax = gca;
    labels_x = string(ax.XAxis.TickLabels);
    labels_y = string(ax.YAxis.TickLabels);
    labels_x([2 3 5 6 8 9 11 12 14 15]) = ' ';
    labels_y(2:3:length(labels_y)-1) = ' ';
    labels_y(3:3:length(labels_y)-1) = ' ';
    ax.XAxis.TickLabels = labels_x;
    ax.YAxis.TickLabels = labels_y;
    grid on
    set(gca, 'FontSize',14)
    title({sprintf('%s = %.1f',  '$ \theta_{source} $' ...
        ,angle_name)},'Interpreter','latex','FontSize',12)
end

title_figure.FontSize = 20;
c = colorbar;
c.Layout.Tile = 'east';
c.Label.String = 'BRDF';
c.Label.FontSize = 18;
c.Ticks = [0:0.05:0.5];
```

```
tiles.XLabel.String = {'','','Elevation angle $$\ \theta_{dome} (degree)$$'};
tiles.XLabel.Interpreter = 'latex';
tiles.XLabel.FontSize = 18;

tiles.YLabel.String = 'Azimuth angle $ \phi_{dome} (degree) $';
tiles.YLabel.Interpreter = 'latex';
tiles.YLabel.FontSize = 18;

hold off

%% Make 3D figure
for k_options = 1:length(angles_to_run) % elevation angles incoming light
    if plot_3D == true
        rg = zeros(size(brdf_to_plot,1) + 1,size(brdf_to_plot,2) + 1);
        rg(:,1) = [0 ; elevations_detection_dome ];
        rg(1,:) = [0 ,  azimuths_detection_dome ];

        % 3D figure
        plotCoordinateSystem('hemisphere',1)
        rg(2:end,2:end) =  brdf_3Dplot(k_options,:,:).*4;
        plotReflectionLobe(rg,azimuths_detection_dome,  ...
            elevations_detection_dome);

        % plot incident ray
        if angles_to_run(k_options) == 0
            angles_to_run(k_options) = 1;
        end

        [ray_starting_point(1),ray_starting_point(2),ray_starting_point(3) ] ...
            = sph2cart(0,deg2rad(angles_to_run(k_options)),1);
        ray_end_points = [0;0;0];

        ray = ray_starting_point' - ...
            linspace(0,800,100).*(ray_starting_point' - ray_end_points);
        plot3(ray(1,:), ray(2,:),ray(3,:),'y', 'LineWidth',3)

        name_fig = sprintf(name_to_save,  angles_to_run(k_options));
        exportgraphics(gcf,name_fig,'Resolution',600)

    end
end



end
```

plotFromLibrarySpectralAlbedoPerElevationAngle

```
function plotFromLibrarySpectralAlbedoPerElevationAngle(settings_plotter)


%% Initialization
folder = settings_plotter.folder; %= 'C:\Users\leoni\Documents\SET\7 8  ...
    Graduation Assignment\Code\RayTracingResults28May\';
name_to_run = settings_plotter.name_to_run; %= ...
    'Convergence_aillumination_0.2_El_'; %'BCgrass_';%'BCsmall_elevation_';
angles_to_run = settings_plotter.angles_to_run;
wavelengths = settings_plotter.wavelengths;
```

```
name_to_save = settings_plotter.name_to_save;


% retrieve data
data = load(strcat(folder,name_to_run));
elevations_detection_dome =  data.data_library.elevations_detection_dome;
azimuths_detection_dome =  data.data_library.azimuths_detection_dome;

colourlist = parula(length(angles_to_run));


%% Make figure
figure;
for k_angle = 1: length(angles_to_run)
    angle_name = angles_to_run (k_angle);
    albedo =  data.data_library.(strcat("Angle", ...
        string(angle_name))).mean_albedo;
    scatter(wavelengths,albedo,'filled','MarkerFaceColor',colourlist(k_angle,:))
    hold on
end
ylabel("\rho")
ylim([0,0.25])
yticks([0:0.01:0.25])
title("Relation between \rho and \theta_{source}")

hold on
xlim([300 1100])
xlabel("\lambda")

ax = gca;
xticks(wavelengths(1):50:wavelengths(end))
labels_x = string(ax.XAxis.TickLabels);
labels_y = string(ax.YAxis.TickLabels);
ax.XAxis.TickLabels = labels_x;
ax.YAxis.TickLabels = labels_y;

grid on
l = legend(strcat('$$\theta_{source} = $$', {'$$10^{-5}$$','12','24','36', ...
    '48', '60',  '72'}));%legend(strcat('$$\theta_{source} = $$', ...
    {'$$10^{-5}$$','6','12','18','24', '30','36','42', '48', '54', '60', ...
    '66', '72'}));
l.Title.String = 'Elevation angle (degree)';
l.Interpreter = 'latex';
l.Location = 'northeastoutside';
l.FontSize = 20;
set(gca, 'FontSize',14)

title("Spectral albedo for several elevation angles of the source")

hold off
end
```

plotSample

```
%% Settings
data_to_plot =load('C:\Users\leoni\Documents\SET\7 8  Graduation ...
    Assignment\Code\Results grass 10 ...
    runs\GrassTilt20\GrassTiltedTop20_0.00_1.mat');
```

```
%% Innitialization
a_dome = 5;
radius_sample = 1;
surfaces = data_to_plot.surfaces;
number_of_individual_surfaces = ...
    length(fieldnames(surfaces.individual_surfaces));
all_fields = fieldnames(surfaces.individual_surfaces);


%% Plot
plotCoordinateSystem("hemisphere", a_dome* radius_sample)
for n_surfaces = 1: number_of_individual_surfaces
    plotSurface(surfaces.individual_surfaces.
    (all_fields{n_surfaces}).surface_boundaries)
end
```