



POLITÉCNICA

escuela técnica superior de
ingeniería
y diseño
industrial

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y DISEÑO INDUSTRIAL

Graduado/a en Ingeniería Electrónica Industrial y Automática

TRABAJO FIN DE GRADO

***Propuesta y desarrollo de contribuciones a la librería
de modelado fotovoltaico de código abierto
pvlib-python***

Autor: Echedey Luis Álvarez

Co-Tutor:

Rubén Núñez Judez
DEPARTAMENTO DE INGENIERÍA
ELÉCTRICA, ELECTRÓNICA, AUTOMÁTICA
Y FÍSICA APLICADA (D180)

Tutor:

César Domínguez Domínguez
DEPARTAMENTO DE INGENIERÍA
ELÉCTRICA, ELECTRÓNICA, AUTOMÁTICA
Y FÍSICA APLICADA (D180)

Madrid, Septiembre, 2024

Este Trabajo Fin de Grado se ha depositado en la Escuela Técnica Superior de Ingeniería y Diseño Industrial de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Ingeniería Electrónica Industrial y Automática

Título: Propuesta y desarrollo de contribuciones a la librería de modelado fotovoltaico de código abierto pvlib-python

Septiembre, 2024

Resumen

La finalidad de este Trabajo Fin de Grado es realizar contribuciones de modelos científicos aplicados a la fotovoltaica dentro de la iniciativa de código abierto *pplib-python*: un conjunto de herramientas escritas en el lenguaje de programación Python que permiten investigar y simular sistemas fotovoltaicos de forma completamente gratuita y libre. Adicionalmente, se mejora el proyecto transversalmente atendiendo especialmente a las necesidades de la comunidad.

Palabras Clave: fotovoltaica, código libre, *pplib-python*, simulación, modelado

Abstract

The purpose of this final-year thesis is the contribution of scientific models used in photovoltaic to the open-source project *pvlib-python*: a Python toolbox designed to simulate and research photovoltaic systems freely with zero cost. Additionally, the project has been improved from top to bottom giving attention to the pvlib community needs.

Keywords: photovoltaic, open source, pvlib-python, simulation, modelling

Agradecimientos

Me gustaría agradecer a mis tutores César Domínguez Domínguez y Rubén Núñez Judez por facilitarme la posibilidad de invertir mi capacidad y desarrollarme en un proyecto que se alinea con mis objetivos de autorrealización desde Junio de 2022, así como por ofrecerme medios suficientes para formarme en el campo de lo fotovoltaico. Además, por darme orientación y consejos exquisitos, como señalarme la reveladora existencia del sesgo que existe en la ciencia en torno a centrar los análisis en el hemisferio norte, que posteriormente ha tenido impacto en una contribución de este proyecto.

A la UPM por ofrecer máquinas virtuales de Linux al alumnado, tanto para el desarrollo de este proyecto como para hacer pruebas en terceros proyectos.

A Nuria Martín Chivelet por darme una muy buena acogida a pesar de no conocernos, explicarme detalladamente el funcionamiento de su modelo científico y ofrecerme continuar en esa misma línea de trabajo, con una copia física de su tesis doctoral incluida.

A Tomás García Aguado, responsable del departamento de pruebas de alta tensión del *Laboratorio Central Oficial de Electrotecnia, LCOE*, por facilitar información sobre los datos que incluyen sus informes de transformadores que sería necesaria para tomar decisiones en cuanto a aportaciones de la librería.

A todos los mantenedores de la librería *pvlib-python* por sus revisiones en profundidad. En especial a Kevin Anderson y a Adam Jensen por confiar en mí y guiarme para obtener una beca bajo el programa *Google Summer of Code*¹.

A todas las personas que públicamente han contribuido directa o indirectamente en crear ecosistemas de desarrollo de software libre y de código abierto, en especial a aquellos involucrados en la comunidad de Python y Visual Studio Code entre innumerables otros.

Finalmente, a Aurelio Acevedo Rodríguez, que además de cuidar de nuestra tierra, me mostró la importancia de la sección de agradecimientos. Que en paz descanse.

¹<https://summerofcode.withgoogle.com/programs/2024/projects/fxPFQqZc>.

Índice general

1. Introducción	1
1.1. Contexto del proyecto	1
1.2. Motivación del proyecto	2
1.3. Objetivos	2
1.4. Estructura del Documento	3
2. Fundamento Teórico y Estado del Arte	4
2.1. La energía solar fotovoltaica	4
2.1.1. Conceptos y efectos en los sistemas fotovoltaicos	5
2.1.1.1. Radiación y espectro solar	5
2.1.1.2. Medida de la radiación solar	6
2.1.1.3. Transposición e irradiancia en un colector plano	7
2.1.1.4. Células fotovoltaicas	8
2.1.1.5. Curva I-V: la característica de una célula solar	8
2.1.1.6. Respuesta espectral y eficiencia cuántica externa	9
2.1.1.7. Módulos	10
2.1.1.8. Efecto de la temperatura	11
2.1.1.9. Sombra y diodos de <i>bypass</i>	11
2.1.1.10. Sistemas fotovoltaicos	11
2.2. Simulación de sistemas fotovoltaicos	12
2.2.1. Herramientas de simulación	15
3. El código abierto y libre	16
3.1. ¿Qué es el código abierto y libre?	16
3.2. Colaboración en proyectos FOSS	17
3.2.1. Pasos para contribuir	19
3.3. La librería <i>pvlib-python</i>	20
3.3.1. Objetivos de la librería	21
3.3.2. Funcionalidades	22
3.3.3. Repositorio del proyecto	23
3.3.4. Frameworks de desarrollo del proyecto	24
3.4. Entorno de desarrollo y herramientas utilizadas	25
3.5. La documentación	27
4. Desarrollo	33
4.1. Contribuciones científicas	33
4.1.1. Modelado de ajuste espectral	33
4.1.1.1. Fundamento teórico	34

ÍNDICE GENERAL

4.1.1.2. Resultado	34
4.1.2. Proyección del cenit solar sobre las coordenadas de un colector	35
4.1.2.1. Fundamento teórico	35
4.1.2.2. Resultado	36
4.1.3. Cálculo de fracción de sombra unidimensional	36
4.1.3.1. Fundamento teórico	37
4.1.3.2. Resultado	38
4.1.4. Pérdidas por sombreado en módulos con diodos de bypass	38
4.1.4.1. Fundamento teórico	38
4.1.4.2. Resultado	40
4.1.5. Fracción diaria de radiación difusa fotosintetizable en función de la fracción difusa global	40
4.1.5.1. Fundamento teórico	41
4.1.5.2. Resultado	41
4.1.6. Modelo de pérdidas por heterogeneidad de irradiancia por célula	41
4.1.6.1. Fundamento teórico	42
4.1.6.2. Resultado	42
4.1.7. Transformación de respuesta espectral a eficiencia cuántica externa y viceversa	43
4.1.7.1. Fundamento teórico	43
4.1.7.2. Resultado	44
4.1.8. Adición de base de datos de respuesta espectral de algunas tecnologías	44
4.1.8.1. Fundamento teórico	44
4.1.8.2. Resultado	44
4.1.9. Adición de espectro estándar completo ASTM G173-03	45
4.1.9.1. Fundamento teórico	45
4.1.9.2. Resultado	45
4.1.10. Cálculo geométrico de sombras en 3D	46
4.1.10.1. Fundamento teórico	46
4.1.10.2. Resultado	47
4.2. Contribuciones técnicas	50
4.2.1. Arreglo a los tests de integración continua en Windows con Conda	50
4.2.1.1. Resultado	51
4.2.2. Arreglo a un parámetro ignorado en una función de transposición inversa	51
4.2.2.1. Resultado	51
4.2.3. Dar soporte a otra función para el cálculo del IAM en el flujo orientado a objetos	52
4.2.3.1. Resultado	52
4.2.4. Suprimir una advertencia al publicar la distribución en PyPI	52
4.2.4.1. Resultado	53
4.2.5. Exponer parámetros de tolerancia para resolver el modelo de un diodo	53
4.2.5.1. Resultado	53
4.2.6. Modificar tolerancias erróneas en varios tests unitarios	53
4.2.6.1. Resultado	54
4.2.7. Arreglo de un bug que ignoraba parámetros de una función de lectura de bases de datos	54

4.2.7.1. Resultado	54
4.2.8. Actualizar versiones de las dependencias de la documentación	54
4.2.8.1. Resultado	54
4.3. Contribuciones menores	55
4.3.1. Añadir una utilidad para obtener los ficheros de ejemplo internos de la librería	55
4.3.2. Corrección de erratas en la documentación	55
4.3.3. Corrección de erratas en ejemplos y en código	55
4.3.4. Modificación de escritura de los parámetros opcionales	56
4.3.5. Limpieza de advertencias al construir la documentación	57
4.3.6. Modificar documentación de parámetros para poder ejecutar procedimientos que verifican la integridad de la librería	57
4.3.7. Portar antiguos ejemplos de Jupyter a scripts integrados en la web	57
4.3.8. Integración continua para verificar los links externos de la documentación	57
4.3.9. Eliminar una función obsoleta y olvidada	58
5. Resultados y conclusiones	59
5.1. Resultados	59
5.2. Impacto	60
5.2.1. Impacto general	60
5.2.2. Impacto en otros proyectos	61
5.2.3. Objetivos de Desarrollo Sostenible	62
5.3. Conclusiones	62
5.4. Trabajo futuro	63
Bibliografía	64
Anexos	68
A. Anexo A: documentación de las funciones	68
A.1. Modelo de ajuste espectral	69
A.2. Proyección del cenit solar	71
A.3. Fracción de sombra unidimensional	73
A.4. Pérdidas por sombras en módulos con diodos de bypass	76
A.5. Fracción difusa de radiación fotosintetizable diaria	79
A.6. Modelo de pérdidas por irradiancia no uniforme	80
A.7. Conversión entre eficiencia cuántica y respuesta espectral	83
A.8. Espectro estándar ASTM G173-03	85
A.9. Cálculo geométrico de sombras en 3D	87
B. Anexo B: ejemplos de uso	89
B.1. Modelo de ajuste espectral	90
B.2. Fracción de sombra unidimensional	94
B.3. Pérdidas por sombras en módulos con diodos de bypass	97
B.4. Fracción difusa de radiación fotosintetizable diaria	101
B.5. Modelo de pérdidas por irradiancia no uniforme	104
B.6. Espectro estándar ASTM G173-03	106
B.7. Cálculo geométrico de sombras en 3D	107

Índice de Figuras

2.1. Espectro solar terrestre. Recuperado del ejemplo de este documento en B.6.	6
2.2. Componentes de la radiación solar y los instrumentos de medida. Recuperado de http://web.archive.org/web/20230624215046/https://yellowhaze.in/wp-content/uploads/2017/01/dni-dhi-ghi-1-768x437.png	7
2.3. Modelo simplificado de un diodo de una célula solar. Obtenido de Em3rgentOrdr - propio, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=149538519	8
2.4. Curva I-V de una célula solar. Recuperado de [14, Fig. 4.6].	9
2.5. Respuesta espectral de una célula solar. Recuperado de https://pvpmc.sandia.gov/modeling-guide/2-dc-module-iv/effective-irradiance/spectral-response/	10
2.6. Flujo de cálculo en la simulación de un sistema fotovoltaico. Recuperado de https://pvpmc.sandia.gov/	13
2.7. Ángulos solares. Recuperado de [16, Fig. 2].	14
3.1. Diagrama de flujo de trabajo de un desarrollador en un proyecto FOSS.	18
3.2. Pantalla principal de GitHub Desktop.	19
3.3. Captura de pantalla de <i>Visual Studio Code</i> con algunos ficheros usados en 4.1.4.	26
3.4. Un ejemplo de renderizado de la documentación de una función en <i>pvlib-python</i>	29
3.5. Un ejemplo de uso en la documentación de <i>pvlib-python</i>	31
4.1. Esquema dos colectores parametrizados, donde uno sombra al otro. La nomenclatura corresponde la ecuación 4.4. Fuente: Figura 3 en [31].	37
4.2. Esquema de un módulo con 3 diodos de bypass. Si un grupo cuenta con una célula sombreada, el exceso de corriente que no puede fluir a través de ella pasa por el diodo de bypass de su grupo. Fuente: Figura 5, a) en [32].	39
4.3. Diagrama UML de la propuesta de cálculo de sombras en 3D.	48
4.4. Ejemplo de sombreado para coordenadas solares instantáneas en 3D.	50

Índice de Tablas

4.1. Algunas de las expresiones regulares empleadas para modificar la escritura de los parámetros opcionales.	56
---	----

Índice de Listings

3.1. Ejemplo de documentación de una función en <i>pvlib-python</i>	28
3.2. Plantilla para elaborar un ejemplo en <i>pvlib-python</i>	29
3.3. Comandos para construir la documentación de <i>pvlib-python</i>	32
4.1. Caso de uso de ejemplo para la propuesta de cálculo de sombras en 3D.	48
4.2. Fragmento de código utilizado normalmente para obtener la ruta de los ficheros de la librería	55
4.3. Fragmento de código utilizando la función propuesta para obtener la ruta de los ficheros de la librería	55

Capítulo 1

Introducción

El cambio climático es un tema de actualidad que plantea un reto social, económico y tecnológico. Dentro de este marco, las energías renovables se presentan como una solución tecnológica a la dependencia de los combustibles fósiles, que son los principales responsables de la emisión de gases de efecto invernadero. Una de las tantas fuentes de energía renovables más prometedoras es la solar fotovoltaica [1], ya que es renovable y no contamina en su explotación directa.

Este Trabajo de Fin de Grado pretende potenciar la adquisición, investigación e implementación de la energía solar, mejorando herramientas de simulación y diseño de instalaciones fotovoltaicas. Para ello, se han realizado múltiples contribuciones a un proyecto de código abierto llamado *pplib-python* [2, 3, 4, 5, 6], que es una biblioteca de herramientas escrita en Python para el análisis de sistemas fotovoltaicos.

1.1. Contexto del proyecto

Con el incremento del interés por las energías renovables y las facilidades del desarrollo software como caldo de cultivo, se ha propuesto la realización de este Trabajo Fin de Grado, que consiste en apoyar en el desarrollo de la biblioteca *pplib-python*.

El proyecto *pplib-python* es un proyecto de código abierto que se encuentra en desarrollo por otros investigadores de centros de investigación y universidades públicas de múltiples países, y algunos miembros de empresas privadas del mismo campo.

El perfil de las personas usuarias de esta biblioteca se puede clasificar entre 3 grupos principales:

- Personal investigador que desean realizar simulaciones y estudios de sistemas fotovoltaicos.
- Personal técnico y de ingeniería que desean optimizar el diseño de instalaciones fotovoltaicas.
- Personal técnico y de ingeniería que quieren identificar fallos en este tipo de instalaciones, mediante la comparación de datos reales con simulaciones.

Por supuesto, tratándose de un proyecto de código abierto, cualquier persona puede utilizar la biblioteca, por lo que no se descarta la posibilidad de que otros

perfils de usuarios puedan beneficiarse de las mejoras realizadas en este Trabajo de Fin de Grado. Es por ello que esta iniciativa democratiza el acceso de personas más noveles en el campo de la energía solar fotovoltaica, que pudieran no tener acceso a herramientas comerciales.

1.2. Motivación del proyecto

La motivación para realizar este Trabajo de Fin de Grado se fundamenta en el interés por las energías renovables, el código libre y el querer poner a disposición global conocimiento para facilitar una transición energética.

Asimismo se espera generar o facilitar el desarrollo de nueva ciencia de forma accesible y contrastable, y que esta pueda ser utilizada por la comunidad científica y técnica de manera completamente transparente y democrática.

Con este TFG se pretende mejorar la fiabilidad de las simulaciones de sistemas fotovoltaicos.

1.3. Objetivos

La línea principal de este trabajo es la adición de nuevas funcionalidades a la biblioteca *pvlib-python*, que permitan mejorar la simulación, investigación y diseño de plantas fotovoltaicas. Para ello, se han establecido los siguientes objetivos:

1. Realizar contribuciones que mejoren las características que brinde el proyecto:
 - Contribuir con la aportación de modelos científicos variados en propósito y utilidad.
 - Añadir otras funcionalidades, que no siendo modelos, sean útiles para las personas usuarias.
 - Validar el funcionamiento mediante tests unitarios.
 - Hacer que cada contribución sea accesible aportando una documentación completa y concisa, y didáctica si fuera necesario.
 - Seguir los rigurosos estándares de calidad de un proyecto científico de código libre.
2. Participar y aportar positivamente a la comunidad de desarrolladores y potenciales contribuyentes:
 - Promover la discusión sobre temas relacionados con la mejora de la biblioteca.
 - Participar en la revisión del código de otros contribuyentes.
 - Ayudar a la comunidad usuaria de la biblioteca resolviendo dudas y problemas.
 - Animar e intentar involucrar a nuevas personas en el proyecto.
3. Reforzar la iniciativa y la comunidad que rodea a la biblioteca:

Introducción

- Ayudar a mantener la biblioteca actualizada para mejorar su ciclo de vida y arreglar errores de forma preventiva.
- Identificar posibles mejoras en los procedimientos marcados para el desarrollo, bien sean realizados manualmente o consistan en una secuencia automática de instrucciones.
- Trabajar en áreas que no sean prioritarias, pero puedan ser útiles de cara a mejorar la longevidad del proyecto.

Por otra parte, entre los objetivos secundarios destacan:

- Dar visibilidad a autores nacionales y de la Universidad Politécnica de Madrid.
- Potenciar proyectos accesibles desde la Universidad pública.

Si bien no es el propósito específico de este trabajo tratar estos últimos objetivos, se considera que son una consecuencia natural y deseable del presente proyecto.

1.4. Estructura del Documento

La estructura de este Trabajo de Fin de Grado pretende ser intuitiva y distribuida por bloques sobre temas similares, destacándose los siguientes:

- **Capítulo 2 Fundamento Teórico y Estado del Arte:** da a conocer el estado actual de la energía solar fotovoltaica y algunas de las herramientas de simulación utilizadas.
 - **Sección 3.3 La librería *pplib-python*:** expone las características de la biblioteca *pplib-python*.
- **Capítulo 4 Desarrollo:** detalla el desarrollo de las contribuciones propuestas a la biblioteca, con factores tanto técnicos como humanos sobre el resultado.
- **Sección 5.2 Impacto:** presenta el impacto de las contribuciones realizadas tanto en esta biblioteca como en otras, y en la comunidad del proyecto.
- **Capítulo 5 Resultados y conclusiones:** resume y hace una valoración global de los resultados obtenidos.

Capítulo 2

Fundamento Teórico y Estado del Arte

En este capítulo se cubre el estado del arte de la energía solar fotovoltaica y, particularmente, de la librería *pplib-python*. Se podrá encontrar en las siguientes secciones:

- **Sección 2.1 *La energía solar fotovoltaica*:** explica el estado actual de la energía solar fotovoltaica y su fundamento teórico base.
- **Sección 2.2 *Simulación de sistemas fotovoltaicos*:** detalla las herramientas de simulación utilizadas en el sector fotovoltaico y el marco general de comparación de la librería *pplib-python*.
- **Sección 3.3 *La librería pplib-python*:** presenta las características de la biblioteca *pplib-python*.

2.1. La energía solar fotovoltaica

En esta sección se presentará el estado del arte de las diferentes tecnologías, estudios y sistemas relacionados con la energía solar fotovoltaica.

La energía solar fotovoltaica es una tecnología que convierte la radiación solar en electricidad utilizando células solares mediante el efecto fotoeléctrico [7, pp. 701-706]. Este fenómeno consiste en la generación de una corriente eléctrica cuando la luz incide sobre un material semiconductor y excita los electrones de la banda de valencia a la banda de conducción. Esta excitación genera un par electrón-hueco que se separa por la acción de un campo eléctrico externo (en cuyo caso no se produce energía neta positiva) o mediante la distribución de cargas en un semiconductor p-n, que permite la extracción de energía. Este último caso es el de aplicación en células solares fotovoltaicas, pues la intención es obtener energía eléctrica.

Existen varias tecnologías de células solares, como las de silicio, las de película delgada y, las más experimentales, de concentración y de otros materiales orgánicos y multiunión, que se agrupan en generaciones [8]. Cada generación responde a una serie de características e implantación en el mercado, donde destacan:

- Primera generación: células de silicio monocristalino y policristalino. Se encuentran bien implantadas en el mercado y son las más utilizadas en aplicaciones fotovoltaicas. Según el límite teórico que calcularon Shockley y Queisser en 1961, el silicio es el material más apropiado para la fabricación de células solares, ya que su banda prohibida de 1.1eV es la que mejor se ajusta al máximo de la radiación solar [7, p. 1126]. Sin embargo, presentan un coste de producción moderado y un alto uso de material. El límite de eficiencia teórica obtenido por los anteriores autores es del 33.7% [9] para colectores planos y células solares de una sola unión, es decir, que no tratan ni con sistemas de concentración ni células multiunión o tandem.
- Segunda generación: células de película delgada, como las de telururo de cadmio (CdTe), las de di-seleniuro de cobre, indio y galio (*Copper indium gallium selenide*, CuInGaSe_2), las de arseniuro de galio (*Gallium arsenide*) (GaAs) y las de silicio amorfo (a-Si:H). Destacan por ser de capa delgada (*thin-film*) y, consecuentemente, más baratas de producir por el bajo uso de material, si bien pueden llegar a ser composiciones de materiales más caros. Nótese que el silicio amorfo es ampliamente utilizado, en especial en aplicaciones de baja potencia, como calculadoras, pero su eficiencia es inferior a las células de silicio cristalino.
- Tercera generación: células de concentración, células de múltiples uniones y células orgánicas. Se encuentran en fase de investigación y desarrollo, y se caracterizan por su alta eficiencia y coste elevado. Por un lado destacan la tecnología de concentración, que emplea lentes para concentrar la luz solar en células de alta eficiencia, normalmente de múltiples uniones, que pueden alcanzar eficiencias superiores al 40% [10, Tabla 5]. Se emplean sistemas ópticos para disminuir el uso de material semiconductor, que es el principal factor de coste en estas células.

Cada una tiene sus propias características y aplicaciones específicas. Las células de primera y segunda generación son las más comunes en aplicaciones fotovoltaicas, mientras que las de tercera generación se encuentran en fase de investigación y desarrollo. El lector puede remitirse a [8] para una revisión más detallada de cada grupo y las peculiaridades de cada material.

En la siguiente sección se hace una revisión de algunos conceptos y efectos que intervienen en la producción de energía fotovoltaica.

2.1.1. Conceptos y efectos en los sistemas fotovoltaicos

Una gran cantidad de factores influyen en la utilidad y el retorno energético y económico de un sistema de generación fotovoltaico. Entender estos factores es fundamental para el diseño y la operación de sistemas PV. En las siguientes líneas se presentan algunos que son de especial interés.

2.1.1.1. Radiación y espectro solar

La radiación es una onda electromagnética que se propaga a la velocidad de la luz y transmite energía. Como la misma luz visible, que cuenta con colores, la radiación que proviene del Sol también tiene diferentes longitudes de onda. Cuenta con regiones como la luz visible, el ultravioleta y el infrarrojo. La cantidad de energía que hay por cada “color” o longitud de onda se conoce como espectro solar.

Por un lado, la irradiancia (radiación incidente) tiene unidades de potencia por unidad de área [W/m^2], y el espectro solar es la distribución de este mismo por cada longitud de onda, por lo que se mide en [$W/m^2/nm$].

Dependiendo de las condiciones atmosféricas, la cantidad de masa de aire que debe atravesar y la presencia de unos gases u otros en diferentes concentraciones, la radiación extraterrestre sufre una reducción en ciertas longitudes de onda. Se debe a la absorción que presentan algunos gases. La forma que toma la radiación solar en la superficie terrestre es lo que se conoce como espectro solar terrestre. El estándar ASTM G173-03 [11] describe un espectro estándar para analizar la radiación solar en la superficie terrestre. En la siguiente figura se puede observar el espectro solar terrestre global, el directo que proviene del Sol sin contar con reflexiones, y el extraterrestre:

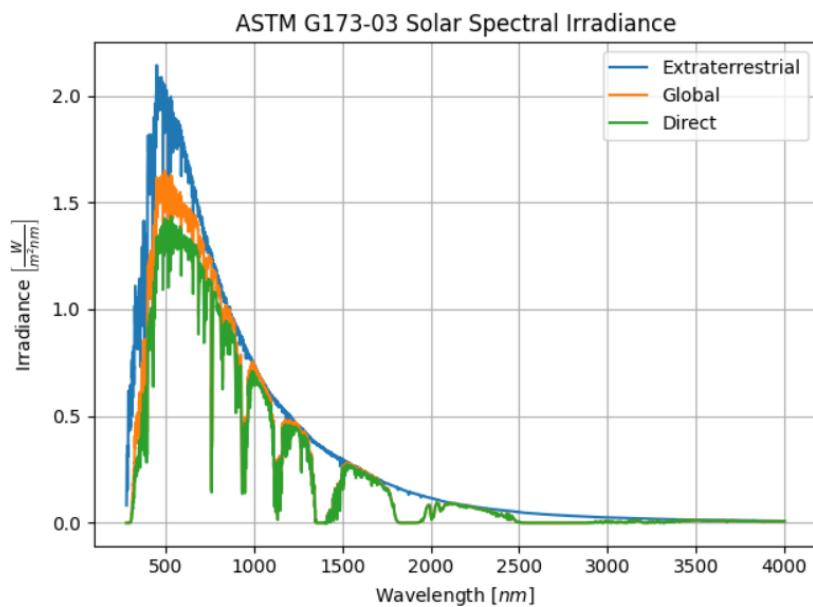


Figura 2.1: Espectro solar terrestre. Recuperado del ejemplo de este documento en B.6.

Para muchos análisis de radiación, se emplea más el valor global en potencia por unidad de área [W/m^2] que el espectro, en [$W/m^2/nm$], pues la forma del espectro no tiene grandes variaciones y el volumen de datos que se maneja es computacionalmente asequible.

2.1.1.2. Medida de la radiación solar

Hay tres medidas de la radiación solar que son interesantes para normalizar la forma en que se mide y posteriormente deducir la que pueda llegarle a un colector fotovoltaico. Estas son:

- Global en una superficie horizontal (*GHI, Global Horizontal Irradiance*): es la radiación que llega a una superficie horizontal.
- Directa normal al Sol (*DNI, Direct Normal Irradiance*): es la radiación que llega a una superficie normal al vector solar.

- Difusa en una superficie horizontal (*DHI*, *Diffuse Horizontal Irradiance*): es la radiación que llega a una superficie horizontal tras ser dispersada por la atmósfera.

A partir de estas medidas, se puede modelar cuál será la radiación solar en una superficie inclinada y orientada en función de los ángulos de inclinación y acimut de los módulos, mediante modelos científicos conocidos como de transposición. Estas tres componentes se relacionan mediante:

$$GHI = DNI \cdot \cos(\theta) + DHI \quad (2.1)$$

Donde θ es el ángulo cenital del Sol (o sea, el ángulo entre la línea vertical y su posición), que normalmente se obtiene con modelos de posición solar a partir del instante de tiempo y la localización geográfica.

La ecuación 2.1 es interesante porque facilita comprobar la calidad de las medidas si existen las tres, y si solo existen dos de ellas, permite deducir la tercera.

Las unidades de las componentes de la radiación solar son W m^{-2} .

Los instrumentos de medida en campo que permiten obtener estos parámetros son piranómetros para medir *GHI*; pirheliómetros, para *DNI*; y piranómetros con disco para *DHI*. En la figura 2.2 se puede observar la relación entre las componentes de la radiación solar y los instrumentos de medida:

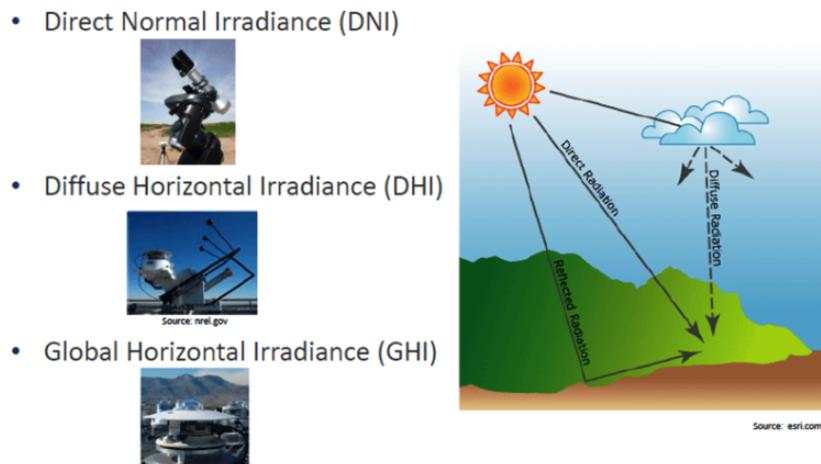


Figura 2.2: Componentes de la radiación solar y los instrumentos de medida. Recuperado de <http://web.archive.org/web/20230624215046/https://yellowhaze.in/wp-content/uploads/2017/01/dni-dhi-ghi-1-768x437.png>

2.1.1.3. Transposición e irradiancia en un colector plano

Al orientar un colector fotovoltaico, la radiación que llega a este dependerá de la inclinación y el acimut del colector. La radiación que llega a un colector plano se puede descomponer en tres componentes: la radiación directa, la radiación difusa -del cielo- y la radiación reflejada -del suelo-:

- Componente directa o circunsolar: es la radiación que llega directamente del Sol sin reflexiones. Es la mayoritaria en días despejados.

- Componente difusa: es la radiación que llega de la atmósfera tras ser dispersada por las partículas en suspensión y las nubes.
- Componente difusa del suelo o albedo: es la reflejada por el entorno. Se puede asumir que es nula para un módulo orientado horizontalmente.

Para calcular estas componentes en el plano del generador a partir de las medidas de radiación solar vistas en la sección anterior, se emplean modelos de transposición. Se tratan de relaciones geométricas entre la irradiancia en el plano horizontal y el inclinado. El modelo más común es el de Pérez [12], pero hay otros como el Pérez-Driesse que solventa un problema de continuidad [13].

2.1.1.4. Células fotovoltaicas

Anteriormente se detallaba sobre los materiales y algunas tecnologías de la fotovoltaica. Esta sección se centra en células planas convencionales, en especial las de silicio, que son las más extendidas.

Estos dispositivos semiconductores son similares en construcción a un diodo, donde la unión toma una gran superficie que es la que se usa para recolectar la radiación. Se considera que la radiación que llega es proporcional a la corriente que se genera: esta se denomina fotocorriente. Sin embargo, por tratarse de un diodo, este deja pasar corriente en el sentido inverso al de la generación de energía, lo que implica que hay pérdidas debidas a la propia construcción del diodo. El siguiente circuito eléctrico representa un modelo simplificado de una célula solar:

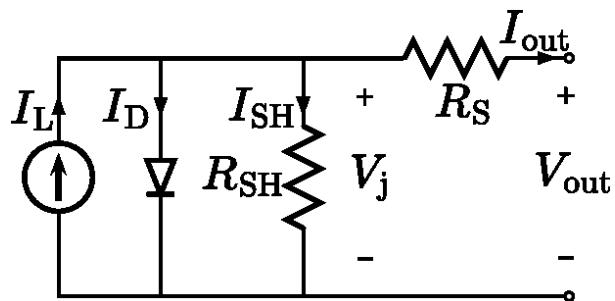


Figura 2.3: Modelo simplificado de un diodo de una célula solar. Obtenido de Em3rgent0rdr - propio, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=149538519>.

I_L representa la fotocorriente, I_D la corriente que circula por el diodo, I_{SH} la corriente de pérdidas en paralelo por la resistencia “shunt” R_{SH} , e I_{OUT} la corriente de salida que circula por R_S , la resistencia en serie. Estas resistencias de pérdidas se deben a la construcción de las células solares, las uniones entre el semiconductor y los contactos metálicos, y a la propia resistencia del material semiconductor.

2.1.1.5. Curva I-V: la característica de una célula solar

Del circuito anterior se deduce por tanto que a igualdad de irradiancia incidente, y por ende, de fotocorriente, la tensión y la corriente que se extraerá dependerá del punto de trabajo de la célula. Cuanta más corriente se extraiga, más caerá la tensión

antes del diodo y este hará que se pierda menos corriente en la propia célula, pero también que se reduzca esta tensión.

La curva que se genera al cambiar la tensión de trabajo de la célula y medir la corriente que sale de ella se conoce como curva I-V. En la figura 2.4 se puede observar una curva I-V típica de una célula solar:

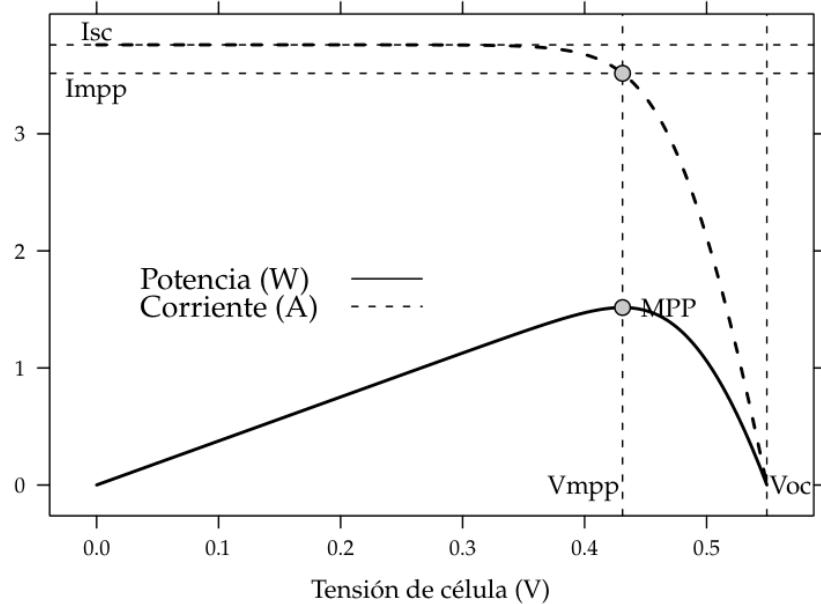


Figura 2.4: Curva I-V de una célula solar. Recuperado de [14, Fig. 4.6].

La curva sólida inferior es el producto de la tensión y la corriente en cada punto, la potencia de salida. Esta curva se conoce como curva P-V. Se puede observar que la máxima potencia de salida está en un punto que se conoce como punto de máxima potencia (MPP, *Maximum Power Point*). Este punto es el que interesa para la producción de energía eléctrica.

Una tensión de célula que no excede 0.7 V es típica de las hechas con silicio. Esta tensión es muy pequeña para ser extraída eficientemente, por ello que se deben crear módulos fotovoltaicos con muchas de ellas para solventar este problema.

2.1.1.6. Respuestapectral y eficiencia cuántica externa

En realidad la radiación solar no es homogénea en todas las longitudes de onda, y una célula solar tampoco es igual de sensible a ellas. La curva de respuestapectral se define como la capacidad de convertir una potencia lumínica a determinada longitud de onda en una corriente de cortocircuito. Esta métrica se cuantifica en $[A/W]$ y depende del material y otros aspectos constructivos, según se puede observar en la figura 2.5:

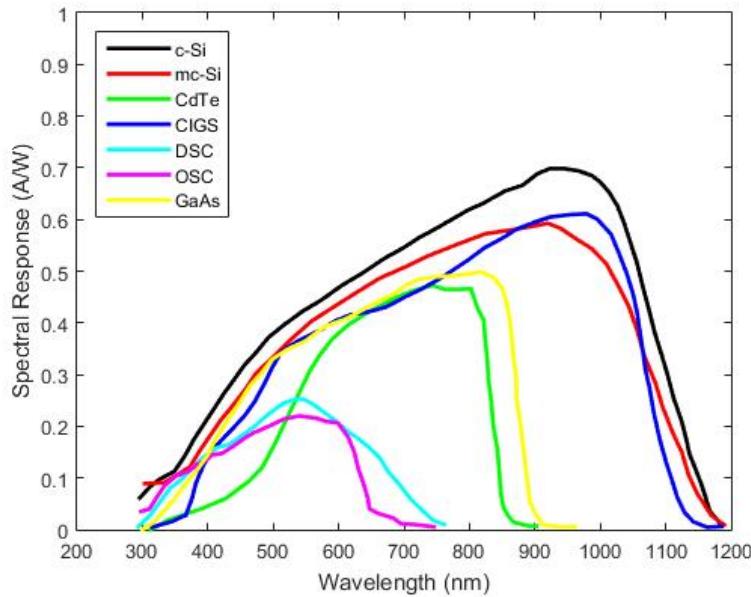


Figura 2.5: Respuesta espectral de una célula solar. Recuperado de <https://pvpmc.sandia.gov/modeling-guide/2-dc-module-iv/effective-irradiance/spectral-response/>.

La eficiencia cuántica externa es una medida similar, pero en este caso se mide cuál es la fracción de fotones incidentes a una longitud de onda determinada que se convierten en corriente. Es un número adimensional y se relaciona con la respuesta espectral mediante la ecuación:

$$SR(\lambda) = \frac{q \cdot \lambda}{h \cdot c} \cdot EQE(\lambda) \quad (2.2)$$

2.1.1.7. Módulos

Las células solares se conectan en serie para que la tensión de todas las células se sume y se pueda extraer una tensión mayor [15], con una corriente que es la misma que la de una sola célula. Al no aumentar la corriente, se evitan pérdidas mayores debidas al efecto Joule por la resistencia de las conexiones.

Un módulo también presenta una curva I-V, que idealmente será la misma que la de una célula pero escalada en la tensión en función del número de células. En realidad, por pequeñas variaciones en la construcción de las células y la conexión de estas, la curva I-V de un módulo no es exactamente la misma que la de una célula. Además, este problema se ve muy agravado cuando la irradiancia no es homogénea en todas las células del módulo y puede suponer un riesgo en el caso extremo de haber sombras. Por esta razón se emplean diodos de derivación o *bypass* para evitar que una célula sombreada disipe energía en forma de calor y se dañe ella o el propio módulo. Se detalla este efecto en la sección 2.1.1.9.

2.1.1.8. Efecto de la temperatura

La temperatura hace que el diodo en el equivalente circuital en 2.3 conduzca mucho más, aumentando las pérdidas que se disparan en este. Por ello, la eficiencia de las células disminuye con la temperatura [14]. La temperatura de las células se estima mediante modelos empíricos basados en la disipación de calor por conducción, convección y radiación.

Por esta razón se plantean sistemas fotovoltaicos algo más inusuales, como los sistemas flotantes sobre cuerpos de agua, que permiten refrigerar las células con el agua.

2.1.1.9. Sombra y diodos de bypass

Es necesario tener en cuenta factores de pérdidas como la sombra, la suciedad o la nieve en el diseño de sistemas fotovoltaicos. Cualquier obstrucción de la radiación sobre una célula genera una afección a la producción parcial o completa de un módulo. Las células se conectan en serie, por lo que la corriente que pasa por una es la misma que pasa por todas. Si una célula está sombreada, las demás células forzarán la corriente que pasa por ella, haciendo que esta se comporte como una carga y disipe energía en forma de calor [14].

Esta energía disipada en la célula sombreada se convierte en calor, y puede llegar a dañar el módulo.

Este efecto se conoce como *hot spot* y puede dañar la integridad del módulo. Se emplean diodos de derivación o *bypass* para mitigarlo. Estos permiten que la corriente mayoritaria de las células más iluminadas circule por el diodo y no se vea limitada por las células sombreadas. En las aplicaciones industriales típicas, sistemas de autoconsumo en la superficie terrestre, se emplea un diodo para proteger a un grupo de células. Normalmente se encuentran tres por módulo; se trata del compromiso entre el coste de poner diodos adicionales, la mitigación de los *hot spots* y las pérdidas por sombreado.

2.1.1.10. Sistemas fotovoltaicos

Se llama sistema fotovoltaico al acople de los módulos junto con componentes y dispositivos eléctricos para hacer que la energía producida por los paneles sea utilizable. Entre este equipamiento adicional destacan:

- Inversores: son los encargados de convertir la corriente continua producida por los módulos en corriente alterna, que es la que se emplea en la red eléctrica.
- Convertidores DC-DC: permiten adaptar la tensión y la corriente de los módulos a la que se necesita en el sistema.
- Seguidor del punto de máxima potencia: el MPP varía en función de las condiciones irradiancia y temperatura de los módulos, por lo que se emplean sistemas que maximizan la producción de energía mediante algoritmos que hayan este punto. Normalmente bien el convertidor DC-DC o el inversor incorpora esta función.
- Seguidores solares: son estructuras que orientan los módulos hacia el Sol a lo largo del día para maximizar la producción de energía.

2.2. Simulación de sistemas fotovoltaicos

- Sistemas de almacenamiento: permiten almacenar la energía producida por los módulos para su uso posterior mediante baterías.
- Sistemas de monitorización: para controlar el rendimiento de los módulos y detectar fallos en el sistema.

No todos los sistemas incorporan los equipos mencionados anteriormente. Se han desarrollado diferentes configuraciones, cada una con sus propias ventajas y dificultades, como sistemas conectados a la red, sistemas autónomos y sistemas de bombeo [14]:

- Sistemas conectados a la red: son los más comunes y se emplean para inyectar energía en la red eléctrica. Se necesitan inversores que conviertan la corriente continua en alterna.
- Sistemas autónomos: se emplean en lugares remotos donde no llega la red eléctrica o es inestable el acceso. Usan sistemas de almacenamiento y de gestión de la energía.
- Sistemas de bombeo: se emplean para bombear agua durante las horas diurnas a un depósito. Solo requieren un seguidor del punto de máxima potencia que trabaje adecuadamente con el motor de la bomba.

En todos estos casos, es fundamental contar con herramientas de simulación y análisis para evaluar el rendimiento de los sistemas, optimizar su diseño y diagnosticar fallos a partir de datos meteorológicos.

2.2. Simulación de sistemas fotovoltaicos

La simulación de sistemas es fundamental para diseñarlos y dimensionarlos adecuadamente.

Toda simulación de un sistema fotovoltaico habitual necesita calcular la radiación solar incidente en la superficie de los módulos. Las variables de partida de una simulación siempre involucran valores meteorológicos como la radiación solar, la temperatura ambiente y la velocidad del viento [14]. Adicionalmente, se emplean modelos empíricos para obtener valores como la radiación solar extraterrestre en la superficie de la atmósfera y deducir otros parámetros de entrada para el resto del flujo de cálculos, en función de los instantes de tiempo que se vayan a analizar.

La infografía 2.6 detalla los efectos que normalmente se tienen en cuenta para simular todo un sistema fotovoltaico:

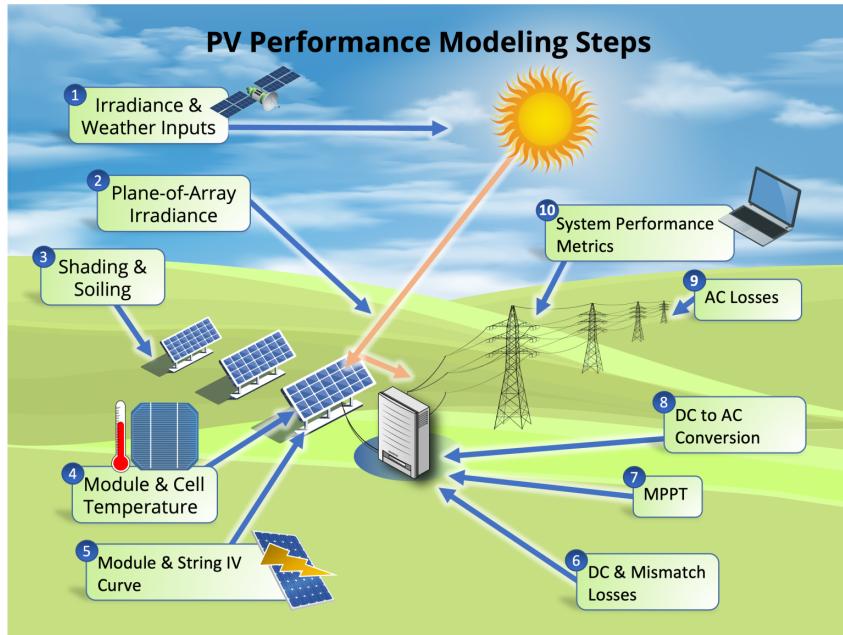


Figura 2.6: Flujo de cálculo en la simulación de un sistema fotovoltaico. Recuperado de <https://pvpmc.sandia.gov/>

El primer punto son datos meteorológicos y las componentes de la irradiancia.

Se comprueba la necesidad de establecer numéricamente las coordenadas del Sol en el cielo para hacer cálculos y comprobar las medidas. Se emplea un sistema de coordenadas esféricas con dos ángulos: el cenit y el acimut. El cenit es el ángulo entre una línea vertical y el Sol, y el acimut es el ángulo entre el norte y la proyección del Sol en el plano horizontal. Debe denotarse que en ocasiones se emplea el ángulo de elevación, que es el ángulo entre el Sol y el plano horizontal; es decir, el ángulo complementario al cenit. En la figura 2.7 se pueden observar los ángulos solares mencionados, donde β_s es la elevación y γ_s es el acimut:

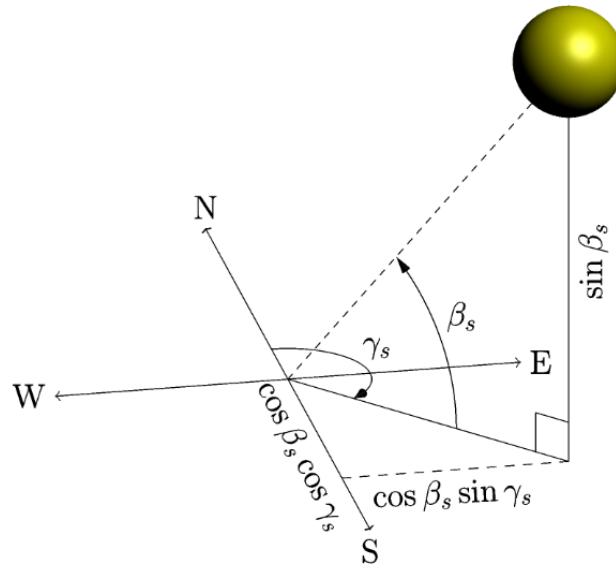


Figura 2.7: Ángulos solares. Recuperado de [16, Fig. 2].

Por definición de ángulos complementarios, el cenit θ_s se relaciona con la elevación β_s mediante la ecuación $\theta_s = 90^\circ - \beta_s$.

A partir de los valores de irradiancia solar *GHI*, *DNI* y *DHI*, se puede calcular la irradiancia por componentes que llega a la superficie de los módulos: la directa, la difusa y la reflejada. La componente directa es la que proviene de la dirección del sol y llega directamente a la superficie de los módulos, la irradiancia difusa es la que proviene del resto del cielo y se dispersa en todas direcciones, y la irradiancia reflejada o albedo es la que proviene de la reflexión y emisión de radiación sobre superficies cercanas. Estas componentes se suman para obtener la radiación total incidente en la superficie de los módulos, que es la que se emplea para calcular la producción de energía.

El siguiente paso es cuantificar el efecto de la temperatura: la eficiencia de las células disminuye conforme aumenta la temperatura debido a la recombinación interna y la reducción de la banda prohibida. La temperatura de las células se estima mediante modelos empíricos que, como mínimo, suelen depender de la estructura del módulo, la irradiancia, la temperatura ambiente y la velocidad del viento.

Algunos factores de pérdidas se toman en cuenta en este instante como el ajuste espectral o una corrección por no-uniformidad de la radiación, que suelen afectar a la irradiancia incidente.

Por otra parte, otros factores de pérdidas como la sombra, las pérdidas en los conductores, la suciedad o la nieve se tienen en cuenta en la potencia de salida de los módulos. A continuación se modelan la integración de los inversores, sistemas de almacenamiento o transformadores, si procede en cada caso.

Finalmente se obtiene la producción horaria de energía, que se puede integrar en el tiempo para obtener la producción, asignarle un retorno económico y estimar la viabilidad del proyecto de instalación fotovoltaica [17].

2.2.1. Herramientas de simulación

La importancia de simulaciones y análisis previo a la implantación de sistemas fotovoltaicos tanto para promotores, operaciones de financiación y diseñadores ha dado lugar a múltiples herramientas software y modelos como PVWatts, PVGIS, PV-Online, PV*SOL, PVsyst, System Advisor Model (SAM) y muchas más [18, 19]. Normalmente estas herramientas propietarias ponen el foco en calcular el rédito energético y económico de una instalación fotovoltaica, pero no en el desarrollo de la investigación y validación de modelos científicos.

La Sección 3.3 mostrará que la librería *pvlib-python*, y su predecesor en MatLab, se centra en la investigación y validación de modelos científicos para la simulación de sistemas fotovoltaicos, mientras que se obvia el cálculo de coste y rédito económico.

Capítulo 3

El código abierto y libre

Este capítulo trata sobre qué es el código abierto y libre, FOSS por sus siglas en inglés¹, y qué importancia tiene actualmente tanto en el campo de investigación como en la industria.

Para poder comprender este trabajo más adecuadamente, se introducen algunos conceptos como *issue* o *pull request*, que se utilizan más adelante en las aportaciones planteadas en el repositorio. En una línea más general, se tratará sobre la importancia de la documentación y los procesos para poder aportar en un proyecto de este tipo.

3.1. ¿Qué es el código abierto y libre?

Los proyectos FOSS son aquellos donde una comunidad de desarrolladores publican el código en un repositorio público y admiten que otras personas puedan aportar cambios y mejoras al mismo. Normalmente su uso es gratuito y se distribuye bajo una licencia que permite la modificación y redistribución del código.

Esta clase de iniciativas suelen estar respaldadas por comunidades de desarrolladores que comparten un interés en común, o incluso empresas que o bien se dedican a ello o facilitan que su personal aporte a estos proyectos.

El código abierto y libre tiene grandes éxitos y actualmente supone uno de los grandes pilares de la economía de software [20, 21]. No es de extrañar la elevada calidad con la que cuentan [22] gracias a la colaboración e interés en el código que muestran las comunidades que lo respaldan.

Es más, en el ámbito de la investigación también se puede encontrar una gran cantidad de proyectos - hasta el punto de ser tema de publicaciones con recomendaciones sobre la gestión y el desarrollo [23].

La elección de realizar este trabajo aportando a código abierto y libre se debe especialmente a este compendio de razones, así como a que el resultado sea visible y accesible a cualquier persona interesada en el tema.

¹De *Free and Open Source Software*.

3.2. Colaboración en proyectos FOSS

Se ha mencionado anteriormente que dichas iniciativas se nutren de la colaboración entre múltiples personas -no solo desarrolladoras, sino también usuarias, entre otras- para identificar y generar mejoras y mantener el proyecto. Es necesario por tanto establecer un marco de trabajo común sobre el que realizar estas aportaciones.

En general, cualquier proyecto de código con unos requisitos mínimos de calidad contará con un sistema de control de versiones. Esta herramienta facilitará ver un historial de cambios, hacerlos de forma independiente pero simultánea entre desarrolladores, revisarlos, trazar errores y arbitrar los cambios que conforman cada nueva versión del software. Existen varios sistemas de control de versiones, como *Mercurial* o *Subversion*, pero el más popular y extendido es *Git*; este último es el que se emplea en *pulib-python*.

Tradicionalmente estas propuestas se realizaban a través de listas de correo electrónico, pero actualmente se emplean plataformas como *GitHub*, *GitLab* o *Bitbucket* que facilitan ver el código fuente y las propuestas con una interfaz gráfica intuitiva. En el caso de *pulib-python*, se emplea *GitHub*, un plataforma muy conocida por las iniciativas de código abierto y libre que pertenece al grupo *Microsoft*.

Además, estas plataformas brindan una serie de características indispensables para la administración actual de proyectos:

- *Issues*: son temas de discusión que puedan tener un impacto concreto en el repositorio. Aquí se suelen informar sobre los errores, posibles mejoras o tareas relacionadas en el proyecto. Cualquier persona con una cuenta en *GitHub* puede abrir una *issue* y comentar en ella. Habitualmente se etiquetan para clasificarlos, priorizarlos y distinguir la fuente del error.
- *Pull Requests*: son las propuestas con cambios que plantean desarrolladores del proyecto. Se pueden revisar, comentar y aprobar o rechazar. En la administración de un proyecto serio y riguroso como *pulib-python*, es un paso imprescindible para aportar cualquier cambio al código. Las propuestas pasan un proceso de revisión y triage para garantizar la calidad y asignarlas a una versión apropiada a la tipología del cambio.
- *Discussions*: son foros para tratar cualquier tema, normalmente menos serios que una *issue*. Es uno de los lugares más apropiados para plantear dudas o sugerencias sin formalizar ni una aportación ni requerir la atención inmediata.

Todos los cambios que se proponen al código en este TFG se realizan a través de *pull requests*, y en el caso de las propuestas más grandes se acompaña con frecuencia de una *issue* para discutir sobre la propuesta antes de realizarla.

Desde un punto de las interacciones sociales y las responsabilidades, es interesante definir las acciones que toma un desarrollador cuando va a realizar una propuesta:

3.2. Colaboración en proyectos FOSS

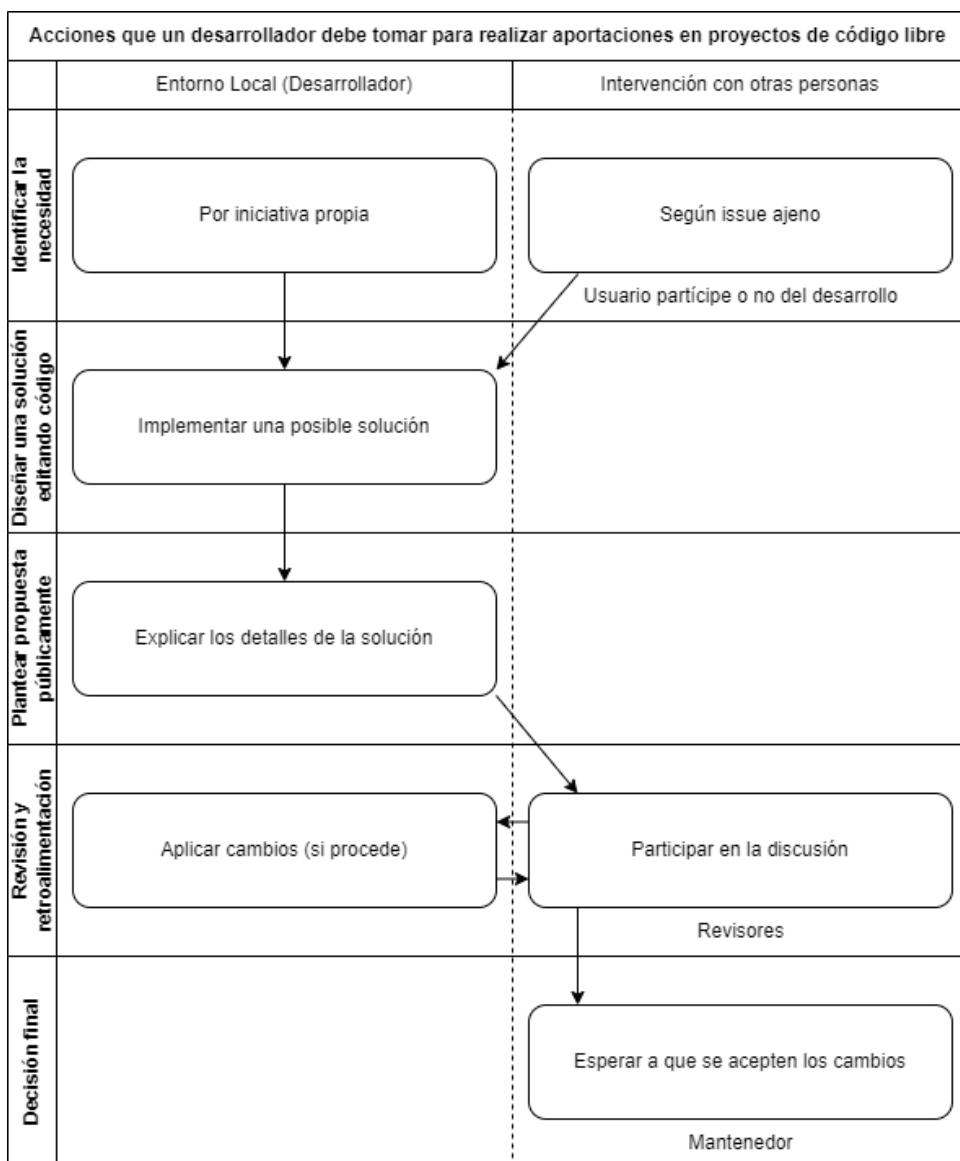


Figura 3.1: Diagrama de flujo de trabajo de un desarrollador en un proyecto FOSS.

De este flujo deben hacerse unas anotaciones importantes:

- El rol de la persona que interviene en cada caso puede corresponder a una misma persona en varias ocasiones; este es el caso de los administradores del proyecto.
 - A veces, en cada intervención hay más de una persona que toma el rol, en especial de quienes revisan.
 - A partir de “Plantear propuesta públicamente” todos los cambios del historial que se hayan hecho son visibles para todo público. Por esta razón es importante cuidar que no se incluya información sensible en ningún momento.

3.2.1. Pasos para contribuir

En este apartado se indican unas instrucciones concretas para iniciar en la edición del código del repositorio. Nótese que hay varias alternativas que se pueden emplear con este propósito, pero por referencia y simplificar se detalla el procedimiento más sencillo.

Por comodidad, *GitHub* proporciona un programa gráfico para ordenador que facilita la interacción con su web, *GitHub Desktop*.

1. Primero se debe tener una cuenta en *GitHub*.
2. Luego se clona el repositorio en el botón “Add” del menú izquierdo superior. Se introduce la dirección web del repositorio: <https://github.com/pvlib/pvlib-python> y se le da a continuar.

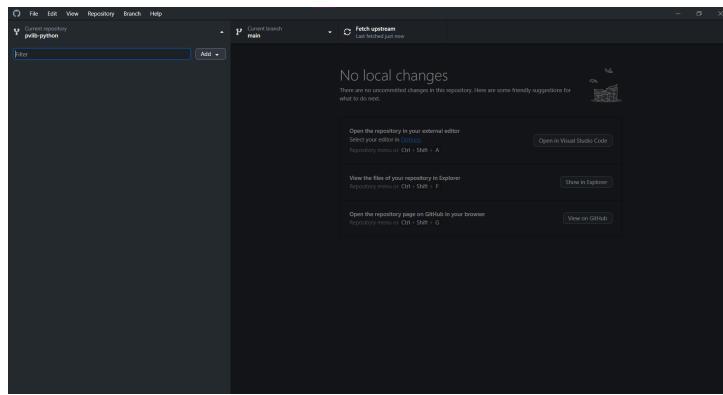


Figura 3.2: Pantalla principal de GitHub Desktop.

3. A continuación, se crea una rama de trabajo específica para los cambios que se vayan a realizar. Se crea pinchando sobre “Current Branch” → “New Branch”.
4. Ahora se puede abrir la carpeta en la que se ha clonado el repositorio y hacer las ediciones correspondientes.
5. Una vez hechos los cambios, se añade un mensaje descriptivo de qué se ha cambiado y se pincha en “Commit to ...”; así los cambios formarán parte de la rama creada.
6. El botón superior izquierdo indicará “Push” cuando hayan cambios en la rama local que no estén publicados en internet. Debe pulsarse.
7. En este momento se usa la combinación de teclas **Control+R** para abrir una *pull request*. Alternativamente se puede hacer esto mismo desde la web de GitHub.
8. A partir de aquí se detalla qué se propone y se interacciona con la comunidad de *pvlib-python* para que revisen la propuesta y decidan si se añaden o no estos cambios.

Otras opciones para contribuir van desde usar *Git* por línea de comandos, otros clientes como *SourceTree* o características integradas en los programas de edición de código.

3.3. La librería *pvlb-python*

Antes de proseguir con las contribuciones, es necesario tratar sobre qué es y qué aporta la librería *pvlb-python*.

pvlb-python es una biblioteca de código abierto que proporciona herramientas para la simulación, análisis e investigación de sistemas fotovoltaicos. Se encuentra desarrollada para el lenguaje de programación interpretado Python [24], que es ampliamente utilizado en las comunidades científica y de desarrollo de software por su gratuidad, sintaxis sencilla, facilidad de desarrollo y diseño orientado a objetos.

Surge como un proyecto independiente a partir de su predecesor escrito en MatLab, *MATLAB_PV_LIB*.

La librería *pvlb-python* aporta una serie de mejoras entre las que destacan la infraestructura de tests, la documentación y otros procedimientos de Integración Continua y Desarrollo Continuo (*CI/CD*) que facilitan el desarrollo. Además, está escrita en un lenguaje de programación libre y gratuito; mientras que MATLAB es propietario y de pago. Por el contrario, en *MATLAB_PV_LIB* no existen tests, el control de versiones resulta más confuso e inaccesible al estar en un documento de Microsoft Word y, en general, hay falta de mantenimiento².

Como todo proyecto de código abierto que se encuentra en constante desarrollo, *pvlb-python* cuenta con un grupo de desarrolladores y colaboradores que contribuyen a su mantenimiento y mejora continua. Estas personas partícipes del proyecto son mayoritariamente personal dedicado a la ciencia e investigación en diferentes instituciones y universidades públicas, aunque también existe personal de ingeniería e investigación del ámbito privado. El código y la colaboración se realiza a través del repositorio de código abierto en GitHub³.

El código se encuentra disponible al público bajo la licencia *BSD 3-Clause*⁴, que permite su uso, modificación, redistribución e integración en otros proyectos siempre que haga la atribución de autoría pertinente y no emplee la imagen de *pvlb-python* con fines publicitarios o promotores. La librería se distribuye a través del índice de paquetes de Python, PyPI⁵, y en el canal de *conda-forge*⁶ para la distribución *Conda*, que está orientada a facilitar conjuntos de librerías para ciencia de datos de forma estable y robusta. *Conda* se distribuye en dos *flavours* -sabores o conjuntos de librerías- conocidas como *Anaconda* y *Miniconda*, cada una con sus propias características y ventajas en tiempo de instalación, espacio en disco y paquetes preinstalados.

La documentación de la librería⁷ se encuentra disponible en la plataforma *ReadTheDocs*, donde se detallan las funcionalidades, los ejemplos de uso, la estructura del código -la API⁸-, las herramientas de desarrollo, las directrices para contribuir al proyecto y la historia de cambios y versiones. La documentación se encuentra

²Véase http://web.archive.org/web/20211207215130/https://pvlb-python.readthedocs.io/en/v0.9.0/comparison_pvlb_matlab.html.

³Acceso a través de <https://github.com/pvlb/pvlb-python>.

⁴Texto de la licencia disponible en <https://opensource.org/license/BSD-3-clause>.

⁵Véase <https://pypi.org/project/pvlb/>.

⁶Véase <https://anaconda.org/conda-forge/pvlb>.

⁷Accesible en <https://pvlb-python.readthedocs.io/en/stable/>

⁸API es la contracción de *Application Programming Interface* y se trata del conjunto de utilidades del que disponen los usuarios de la librería.

disponible exclusivamente en inglés.

3.3.1. Objetivos de la librería

Como se ha comentado anteriormente, la librería *pplib-python* tiene como objetivo principal proporcionar herramientas para la simulación, análisis e investigación de sistemas fotovoltaicos. Para ello, cuenta con una serie de funciones y clases que permiten realizar cálculos de radiación solar, generación de energía, sombreado, eficiencia de módulos y pérdidas de energía, entre otros. Estas herramientas se basan en modelos científicos y métodos de cálculo validados por la comunidad científica y se han implementado en Python para facilitar su uso y extensión. Es importante denotar que los autores pretenden establecer la implementación de la librería como una referencia fiable y certera de los modelos científicos que se emplean en la simulación de sistemas fotovoltaicos, poniendo un enfoque especial en la rigurosidad de las publicaciones y de los artículos científicos que se emplean como base.

Enumerando los objetivos de la librería, se pueden destacar los siguientes:

1. Proporcionar herramientas para la simulación y análisis de sistemas fotovoltaicos.
2. Implementar modelos científicos y métodos de cálculo validados por la comunidad científica.
3. Establecer un referente fiable de las implementaciones de las características citadas anteriormente.
4. Auditarse las contribuciones para garantizar la funcionalidad de las aportaciones.
5. Facilitar la integración con servicios de datos meteorológicos y bases de datos de radiación solar.
6. Fomentar la colaboración y la contribución de la comunidad científica y de desarrollo de software.

Cabe destacar que también hay fines que la comunidad de *pplib-python* excluye como objetivos, es decir, características que no se pretenden implementar o documentar. Entre estos destacan:

1. Proporcionar contenido didáctico sobre la energía solar fotovoltaica, más allá del estrictamente necesario para entender y aplicar las utilidades disponibles.
2. Hacer inferencias, sugerencias o recomendaciones que no estén publicadas formalmente sobre el uso de los modelos implementados.
3. Facilitar un sistema *ready-to-go* para usuarios finales.

Puede parecer que excluir estos objetivos son limitantes en cuanto a la divulgación y al alcance, pero en realidad se trata de una forma de garantizar la calidad y la fiabilidad de la librería. Así el esfuerzo limitado de los mantenedores y los contribuyentes se centra en la implementación rigurosa de publicaciones de la comunidad científica.

Una nota importante, en especial sobre el último punto, es que la definición de la

misma librería es que se trata de una “caja de herramientas”⁹, porque es la persona usuaria de la biblioteca es la responsable de generar un flujo de trabajo adecuado para sus fines. Esta nota es en contraposición a las herramientas de simulación y análisis de sistemas fotovoltaicos comerciales, que limitan y simplifican la experiencia del usuario, varias veces dejando de lado la documentación y la transparencia de los cálculos¹⁰.

3.3.2. Funcionalidades

El proyecto de código abierto *pvlib-python* cuenta múltiples características que abarcan un amplio rango de temas relacionados con la producción de energía solar fotovoltaica. Al momento de la redacción de este documento, la versión de la librería es la 0.11.0, publicada el 21 de junio de 2024. A continuación, se detallan algunas de las funcionalidades más destacadas:

- Cálculo de la posición del sol: para determinar la posición del sol en el cielo en función de la localización y la hora del día, en el submódulo `pvlib.solarposition`.
- Cálculo de valores estándares de la radiación solar: la librería proporciona herramientas para calcular la radiación solar en una superficie plana y horizontal en función de la localización, la posición del sol y parámetros atmosféricos. Véase `pvlib.solarposition` y `pvlib.clearsky`.
- Procedimientos de descomposición, transposición y transposición inversa de la radiación solar: se emplean para, a partir de la irradiance solar en una superficie plana y horizontal, obtener la incidente en un colector plano orientado arbitrariamente. Y una vez obtenidas las componentes directa, difusa y de albedo de esta irradiancia, aplicar correcciones convenientes para obtener la radiación efectiva colectada en la superficie. También se puede realizar el proceso inverso, de las componentes en el plano inclinado a la radiación en una superficie plana y horizontal. Estas utilidades se encuentran en el submódulo `pvlib.irradiance`.
- Obtención de parámetros atmosféricos como la columna o masa de aire, un número que representa cuánta atmósfera hay, relativa a la columna completamente vertical (AM=1), en el submódulo `pvlib.atmosphere`.
- Valores de albedo predefinidos, bien de materiales sólidos o de superficies naturales de agua, en `pvlib.albedo`.
- Corrección de la radiación incidente en función del ángulo de incidencia: debido a la reflexión que sufre la luz solar al incidir en una superficie de forma oblicua. Las utilidades se encuentran en el submódulo `pvlib.iam`.
- Producción fotovoltaica: mediante puntos característicos como el de máxima potencia (*MPP*) o calculando curvas I-V completas. Se emplean modelos de eficiencia de módulos y pérdidas de energía para estimar la producción de energía en un sistema fotovoltaico, mediante el modelo de un único diodo. Se encuentra en el submódulo `pvlib.singlediode`.

⁹Del inglés, “toolbox”. <https://github.com/pvlib/pvlib-python/blob/99619e8fc5aea5c5dc4dacabb75b617786cc4a1a/README.md?plain=1#L61>.

¹⁰Como hecho anecdótico, se puede observar alguna crítica sobre esto entre los participes del proyecto, por ejemplo en <https://github.com/pvlib/pvlib-python/issues/2057#issuecomment-2197279047>.

- Cálculo de sombreado: métodos analíticos para calcular la fracción sombreada de hileras de filas. Véase el submódulo `pvlib.shading`.
- Cálculo de ángulos de seguimiento: para seguidores de uno y dos ejes, que permitan colectar la máxima radiación solar directa, que normalmente se corresponderá con la máxima producción de energía. Véase el submódulo `pvlib.tracking`.
- Cálculo de temperatura de los módulos: normalmente empíricos, facilitan estimar la producción de energía ya que, habitualmente, la eficiencia de las células es bastante susceptible a la temperatura. Estas utilidades se encuentran en `pvlib.temperature`.
- Estimación de ganancias o pérdidas por efectos del espectro solar: pues en función de la composición del espectro solar incidente en los módulos, la eficiencia de los mismos puede variar. En `pvlib.spectrum`.
- Modelado de eficiencia de inversores: para estimar la eficiencia de los inversores DC-AC en función de la potencia de entrada y sus demás características. En `pvlib.inverter`.
- Modelado de pérdidas en transformadores: para estimar las pérdidas en los transformadores de conexión a red. En `pvlib.transformer`.
- Modelado de pérdidas en cables: para estimar las pérdidas resistivas en los cables. En `pvlib.pvsystem`.
- Integración de APIs externas para la obtención de datos meteorológicos relacionados con la fotovoltaica, en `pvlib.iotools`.
- Abstracciones de los componentes que conforman una instalación fotovoltaica, mediante las clases `Location`, `Array`, `PVSystem` y la clase que agrupa el flujo computacional `Modelchain`, en los submódulos `pvlib.location`, `pvlib.pvsystem` y `pvlib.modelchain`.
- Cálculos adicionales para sistemas bifaciales, que son aquellos colectores planos que permiten la captación de radiación solar por ambas caras del módulo. En `pvlib.bifacial`.

3.3.3. Repositorio del proyecto

Como se ha mencionado anteriormente, el código de la librería `pvlib-python` se encuentra alojado en un repositorio de código abierto. Un repositorio es una carpeta que contiene una serie de archivos, de los cuales los más importantes se gestionan mediante un sistema de control de versiones. Este sistema permite llevar un registro de los cambios realizados en los archivos, así como la posibilidad de volver a versiones anteriores en caso de que se produzca un error o un fallo en el código. El uso de esta herramienta es indispensable en proyectos de software, ya que facilita la colaboración entre los desarrolladores y la gestión de las versiones del código.

En el caso de `pvlib-python`, el repositorio se encuentra alojado en la plataforma GitHub, que es una de las más populares para el alojamiento de proyectos de código abierto.

El repositorio cuenta con múltiples archivos y sub-carpetas que configuran el proyecto. Algunos de los más destacables son:

- README.md: es el primer archivo en verse en la plataforma online y contiene la descripción del proyecto, su propósito y cómo instalarlo, entre otros.
- LICENSE: es el archivo que contiene la licencia del proyecto, en este caso la licencia *BSD 3-Clause*.
- pyproject.toml, setup.cfg, setup.py, MANIFEST.in: son los archivos que definen la configuración del proyecto, como el nombre, la versión, la descripción, las dependencias, los archivos que componen una distribución precompilada o de código fuente, etc. que las personas usuarias verán en la plataformas de distribución de paquetes.
- docs/: es la carpeta que contiene archivos de configuración y explicativos que permiten generar la documentación del proyecto. No obstante, el código que se expone públicamente se documenta automáticamente a partir de unos comentarios en el código.
- pvlib/: es la carpeta que contiene el código fuente de la librería, organizado en sub-carpetas y archivos según su funcionalidad. Los tests unitarios y de integración se encuentran en esta carpeta.
- AUTHORS.md, CODE_OF_CONDUCT.md, codecov.yml, readthedocs.yml, paper/, ci/, benchmarks/ y .github/: son archivos y carpetas que contienen información adicional sobre el proyecto y la configuración de todos los flujos de integración continua: destacan los tests, los benchmarks y la documentación.

3.3.4. Frameworks de desarrollo del proyecto

Como tal, el desarrollo del proyecto requiere el conocimiento de algunas herramientas que permiten la colaboración con el repositorio principal y cumplir con las directrices para contribuir al proyecto. Estas se suelen denominar *frameworks* de desarrollo, literalmente marcos de trabajo, que normalizan la forma de desarrollar y colaborar en el proyecto.

La elección de las herramientas empleadas corresponde a una preferencia en cuanto a uso y extensión en el desarrollo software, pero pueden variar en el futuro, como ya lo ha hecho en el pasado.

En el lado de las utilidades esenciales, se encuentran:

- *Git*: como sistema de control de versiones que se emplea para gestionar los cambios en el código.
- *GitHub*: es la plataforma de alojamiento de proyectos de código abierto que se emplea para colaborar en el desarrollo del proyecto. Permite crear *issues* o discusiones para reportar errores o sugerir mejoras; y proponer cambios y revisarlos públicamente en *pull requests*. Además proporciona máquinas virtuales para ejecutar los procedimientos de integración y desarrollo continuos.
- *ReadTheDocs* junto con *sphinx*: es la plataforma que se emplea para ejecutar *sphinx*, el entorno de trabajo que produce la documentación en formato HTML, y alojar el producto. La documentación se escribe en formato *reStructuredText* y se genera automáticamente a partir de los comentarios en el código, scripts con los ejemplos y archivos fuente con el cuerpo más general.

- *pytest*: es la librería de Python que se emplea para escribir y ejecutar tests unitarios y de integración en el código. Facilita la creación de *mocks* u observadores del estado de partes del mismo proyecto, la reutilización de datos de tests, la ejecución opcional de los tests en función de las dependencias disponibles y visualizar los resultados en un informe. Posiblemente la mejor característica es poder ejecutar los tests uno a uno, por grupo o todos a la vez.
- *Codecov*: es la plataforma que se emplea para medir la cobertura de los tests en el código. Permite visualizar la cobertura de los tests en un informe y comprobar si se están realizando pruebas suficientes en el código.
- *flake8*: es una utilidad que detecta errores de estilo en el código, como la longitud de las líneas, la indentación, la presencia de espacios en blanco, entre otros. Facilita la escritura de código limpio y legible, por ende, mantenible. Además, facilita la detección de errores estáticos en el código; esto es, condicionales que nunca se cumplen, variables que no se usan, importación innecesaria de módulos, identificadores duplicadas o que no siguen la convención de nombres, entre otros.

3.4. Entorno de desarrollo y herramientas utilizadas

Para el desarrollo de este trabajo se emplea el siguiente software para el desarrollo del proyecto consistentemente a lo largo de toda su ejecución:

- *Visual Studio Code*: es el editor de código que se emplea para escribir y editar el código fuente del proyecto. Es un editor de código semi-aberto, ligero y rápido que cuenta con una amplia gama de extensiones para facilitar el desarrollo de software. Es una alternativa multipropósito a otros editores de código como *PyCharm* o *Spyder* en el caso de Python. Es multipropósito en cuanto admite el desarrollo de muchísimos lenguajes de programación y sintaxis de ficheros ampliamente utilizados en el desarrollo software.

A continuación se muestra una captura de pantalla de *Visual Studio Code* con algunos ficheros usados en la implementación de la propuesta [Pull Request: #2070](#):

3.4. Entorno de desarrollo y herramientas utilizadas

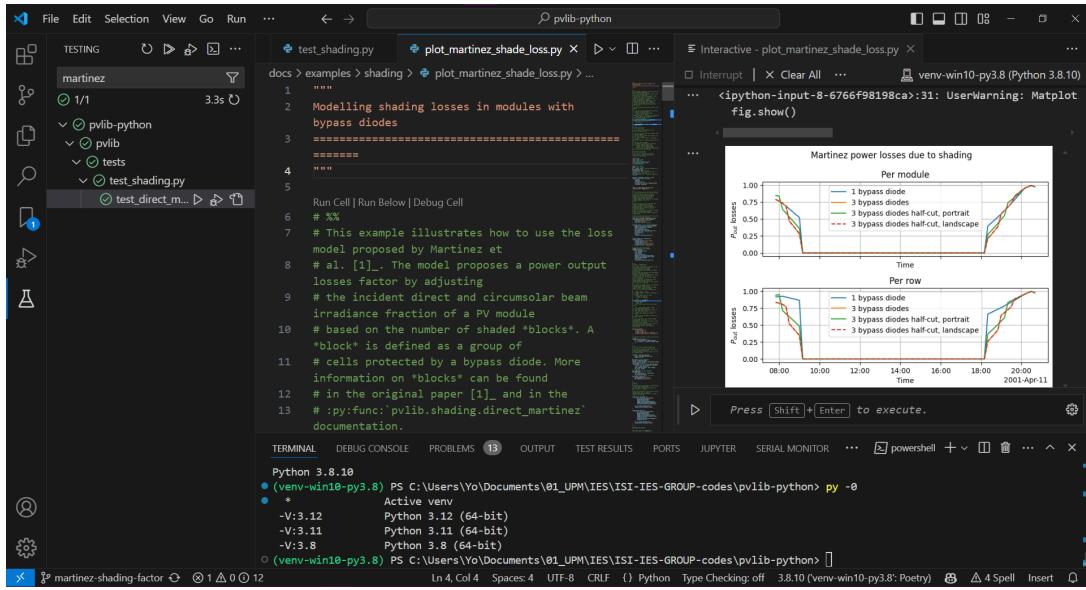


Figura 3.3: Captura de pantalla de *Visual Studio Code* con algunos ficheros usados en 4.1.4.

Es importante denotar que la inmensa utilidad que proporciona *VSCode* es gracias a las extensiones creadas por la comunidad de desarrolladores, que permiten desde la edición de archivos de texto plano hasta la depuración del código, pasando por la integración con servicios de control de versiones y la ejecución de tests desde la interfaz.

A continuación se detallan extensiones que se emplean en el desarrollo del proyecto:

- *Python*: es la extensión que se emplea para el desarrollo de código en Python. Proporciona funcionalidades como el autocompletado de código, la visualización de la documentación de las funciones, la ejecución de scripts y la depuración de código.
- *Ruff*: permite formatear el código según las directrices de estilo de *flake8*, la utilidad que da estilo uniforme al código de Python. Se recuerda que establece limitaciones en cuanto a la longitud de las líneas y a la forma de escribir todo el código.
- *Github Copilot*: un asistente de generación de código en línea integrado en el editor, que se emplea para sugerir fragmentos de código y documentación en función del contexto. Agiliza el desarrollo.
- *Code Spell Checker*: es un corrector ortográfico que se emplea para detectar errores de ortografía en el código y en la documentación.
- *Jupyter Notebooks*: es la extensión que se emplea para editar y ejecutar *notebooks* de Jupyter en el editor, un formato que permite visualizar las variables del contexto y facilita el debugging interactivo. Los ejemplos de *pvlib-python* se realizan con un formato similar a las celdas de texto o código de una *notebook*.

- Resaltado de sintaxis de varios formatos de archivos, como *reStructuredText*, *Markdown*, *YAML* y *TOML*: para facilitar la edición de la documentación y los archivos de configuración.
- *GitHub Desktop*: es la interfaz gráfica que se emplea para colaborar en el desarrollo del proyecto. Permite visualizar los cambios, crear *ramas*, hacer *commits*, *pull requests* y *merges*, entre otros. En resumen, administra nuestra copia del repositorio para que los desarrolladores del proyecto puedan revisar y aceptar las propuestas.
- *pip* y *venv*: como gestor de paquetes y entornos aislados de desarrollo de Python nativos. Se emplean para instalar las dependencias del proyecto y usar entornos con las versiones específicas requeridas aislado del resto del sistema.

De forma discreta, se ha hecho uso de otras herramientas como:

- *Git*: para clonar las ramas con los cambios planteados en este TFG y ejecutar partes de la integración continua, normalmente la documentación, en las máquinas virtuales de linux provistas por la UPM¹¹.
- *Miniconda*: una distribución de Python que se emplea para instalar y gestionar las dependencias del proyecto. Facilita la creación de entornos virtuales y la instalación de paquetes de Python. Se utilizó para diagnosticar un error de precisión por la compilación de algunas librerías y que hacía fallar un test.
- *LibreOffice Calc*: para generar datos de prueba y comprobar las implementaciones de las ecuaciones de los modelos.

El desarrollo de la librería se ha hecho habitualmente en un portátil con Windows 10 así que se ha aprovechado a usar el subsistema de Windows para Linux (WSL) para ejecutar tests y construir la documentación en un entorno similar al de integración continua ocasionalmente. No obstante, por los recursos limitados que ofrece un portátil, y por desear cambiar de ramas mientras los procedimientos de integración continua se efectúan, se ha hecho uso de las máquinas virtuales de la UPM para construir la documentación y analizar algún test.

3.5. La documentación

En esta sección se pondrá en valor lo que realmente es lo más importante de un proyecto de programación, en especial de código abierto: la documentación. Es más, se puede argumentar que el valor de todas las aportaciones es documentar lo más rigurosamente posible los modelos y métodos que se implementan, tanto para facilitar su correcto uso como para divulgar su existencia y problemática que resuelven.

Que una buena documentación respalde un proyecto de código abierto es crítica para el éxito del mismo [25]. Es la carta de presentación a público interesado, y es la guía de uso para las personas usuarias. El caso inverso es una documentación pobre, que puede llevar a la confusión y a la desconfianza en la calidad del software.

Como se comentaba anteriormente, la documentación de la librería *pvlib-python* se encuentra alojada en la plataforma *ReadTheDocs*, que expone los archivos HTML para que las personas usuarias puedan consultarla en línea. La documentación se

¹¹Accesible a través de <https://escritorio.upm.es/>.

genera automáticamente a partir de los comentarios escritos en el código fuente, que se escriben en formato *reStructuredText* y se construye con el framework *sphinx*.

Existen archivos específicos para indicar qué funciones o métodos son públicos, hacer páginas de inicio, de referencia, de ejemplos, de instalación, de contribución, etc. Además, se pueden incluir imágenes, tablas, gráficos, enlaces, referencias, entre otros elementos que facilitan la comprensión de los conceptos.

Sphinx emplea el estilo de documentación de *pydata-sphinx-theme*, que organiza las secciones de la documentación y da un estilo homogéneo a la web. Además, para la creación de los ejemplos se emplea la extensión *sphinx-gallery*, que ejecuta unos scripts de Python por secciones similares a una *notebook* de Jupyter y captura la salida de texto estándar y los gráficos para mostrarlos en la documentación.

La *docstring* de cada función, que es el comentario que se escribe en la primera línea de la definición de la función y se emplea para autogenerar la documentación, utiliza el estilo de *numpydoc*. Este estilo permite incluir información sobre los parámetros de entrada, los valores de retorno, las excepciones que se pueden lanzar, entre otros. Además, se pueden incluir ejemplos de uso de la función e informar avisos de precaución sobre aspectos más específicos.

A continuación se muestra como ejemplo una función mínima con su plantilla de documentación y de código de una función escrita con este fin:

Listing 3.1: Ejemplo de documentación de una función en *pvlib-python*.

```

1 def example_model(param1, param2):
2     """
3     Brief model description.
4
5     Long model description, also found at [1]_.
6
7     .. versionadded:: 0.1.0
8
9     .. warning::
10        This docstring is an example.
11
12    Parameters
13    -----
14    param1 : numeric
15        Description of the parameter.
16    param2 : numeric
17        Description of the parameter.
18
19    Returns
20    -----
21    float
22        Return type of the function.
23
24    Notes
25    -----
26    Additional notes about the function, detailed explanations if needed. Even an equation:
27
28    .. math::
29
30        f(x, y) = x + y
31
32    Examples
33    -----
34    >>> example_model(1, 5)
35    6.0
36
37    References

```

El código abierto y libre

```
38     -----
39     .. [1] Author, A. (2024). Title of the paper. Journal, 1(1), 1-10. :doi:`10.0001/populate`  
40  
41     return float(param1 + param2)
```

Esta función, una vez listada en el archivo correspondiente del índice que nos interese -aquí usamos el submódulo `pvlib.solarposition` como ejemplo-, creará una página como la que se muestra en la figura 3.4:

The screenshot shows the pvlib documentation website. The top navigation bar includes links for User Guide, Example Gallery, API reference, What's New, and Contributing. The main content area is titled "pvlib.example_model". It contains the function definition `pvlib.example_model(param1, param2)`, a brief model description, and a long model description. A green box indicates it is "New in version 0.1.0". A yellow warning box states that the docstring is an example. Below the function details are sections for Parameters, Returns, Notes, Examples, and References. The Examples section shows a code snippet: `>>> example_model(1, 5)` followed by the output `6.0`. The References section lists a citation: "[1] Author, A. (2024). Title of the paper. Journal, 1(1), 1-10. DOI: 10.0001/populate". At the bottom, there is a figure titled "Example title" showing a blue curve on a grid, and navigation links for "Previous" (pvlib.solarposition.hour_angle) and "Next" (Clear sky).

Figura 3.4: Un ejemplo de renderizado de la documentación de una función en *pvlib-python*.

Por otro lado, la siguiente estructura pertenece a la redacción de un ejemplo en la documentación. Se emplea para mostrar cómo se usa una función dentro de un contexto más elaborado en cuanto a variables de entrada y salida. Puede incluir todas las características de la documentación de una función:

Listing 3.2: Plantilla para elaborar un ejemplo en *pvlib-python*.

```
1 """  
2 Example title  
3 ======  
4  
5 Brief model description (shown in preview card).  
6 """
```

```
7 # %%
8 # Text paragraph, in reStructuredText format. Can use sections, subsections, etc., and math as
9 #     in LaTeX.
10 # More text.
11
12 # This is a comment (there is a newline above)
13 import matplotlib.pyplot as plt
14 from pvlib import example_model
15 print("Hello, world!")
16 plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
17 plt.show()
18
19 # %%
20 # Return to paragraph text.
21
22 sum_val = example_model(1, 5)
23 print(f"Sum of 1 and 5 is {sum_val}.")
```

El texto anterior constituye un archivo que, ubicado en la carpeta correcta (aquí el archivo es `docs/examples/solar-position/example_example.py`), hará que se cree automáticamente una página como la que se muestra en la Figura 3.5.

El código abierto y libre

The screenshot shows a documentation page for the `pvlib` library. At the top, there's a navigation bar with links to User Guide, Example Gallery, API reference, What's New, and Contributing. There are also social sharing icons and a "View on GitHub" link.

The main content area has a sidebar on the left with a search bar and a list of topics under "Solar Position": ADR Model for PV Module Efficiency, Agrivoltaic Systems Modeling, Bifacial Modeling, Irradiance Decomposition, Irradiance Transposition, I-V Modeling, Reflections, Shading, Soiling, and Solar Position. The "Solar Position" item is expanded, showing "Example title" and "Sun path diagram".

The main content area features a "Note" section with a link to download full example code. Below it is a section titled "Example title" with a brief model description. A code block shows a Python script:

```
# This is a comment (there is a newline above)
import matplotlib.pyplot as plt
from pvlib import example_model
print("Hello, world!")
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.show()
```

Below the code is a plot showing a parabolic curve from (1, 1) to (4, 16). The x-axis ranges from 1.0 to 4.0, and the y-axis ranges from 2 to 16. The plot is a simple line graph with a light blue line.

Under the plot, there's an "Out: Hello, world!" message in a yellow box. The text "Return to paragraph text." follows. Another code block shows:

```
sum_val = example_model(1, 5)
print(f"Sum of 1 and 5 is {sum_val}.")
```

With an "Out: Sum of 1 and 5 is 6.0." message in a yellow box. Below these, a note says "Total running time of the script: (0 minutes 0.060 seconds)".

At the bottom, there are two download buttons: "Download Jupyter notebook: example_example.ipynb" and "Download Python source code: example_example.py". A small note says "Gallery generated by Sphinx-Gallery". Navigation links "Previous Solar Position" and "Next Sun path diagram" are also present.

Figura 3.5: Un ejemplo de uso en la documentación de *pvlib-python*.

Por último queda construir la documentación. A continuación se muestran los comandos que deben ejecutarse para este fin:

- En un sistema basado en *Linux* o *WSL*.
- Clonando el repositorio original de la librería con *Git*.
- Empleando las mismas versiones que a día de la redacción de este documento se emplea en la integración continua (*pvlib-python=0.11.0*) - en especial hay que hacer la instalación de *Python3.8*.
- Con el entorno aislado de desarrollo (*venv*) activado.

- Y realizando una instalación local mediante *pip*.

Listing 3.3: Comandos para construir la documentación de *pvlib-python*.

```
1 # instalar Python 3.8 desde el repositorio de deadsnakes,
2 # en las librerías por defecto de Ubuntu no se encuentra disponible por antigüedad
3 sudo add-apt-repository ppa:deadsnakes/ppa -y
4 sudo apt-get update
5 sudo apt-get install python3.8 python3.8-venv -y
6
7 # clonar el repositorio de pvlib-python
8 git clone https://github.com/pvlib/pvlib-python
9 cd pvlib-python
10
11 # crear el entorno virtual, instalar pvlib y las dependencias de la documentación
12 python3.8 -m venv .venv
13 source .venv/bin/activate
14 python3.8 -m pip install .[doc]
15
16 # construir la documentación
17 cd docs/sphinx
18 make html
19
20 # abrir la documentación en el navegador
21 xdg-open build/html/index.html
```

Este conjunto de comandos ha sido ampliamente utilizado para la construcción de la documentación en entornos remotos y así aligerar la utilización de recursos en el portátil personal.

Capítulo 4

Desarrollo

En este capítulo se detallan las aportaciones realizadas a la librería *pvlib-python* dentro del marco de este Trabajo Fin de Grado.

Se ha separado en tres secciones, atendiendo principalmente a su tamaño y complejidad del desarrollo:

- En **Sección 4.1 Contribuciones científicas** se tratan aquellas que se espera que tengan mayor peso en los usuarios finales, pues incluyen modelos y utilidades alineadas con una mejor simulación de los sistemas fotovoltaicos.
- En **Sección 4.2 Contribuciones técnicas** se encuentran algunas contribuciones cuyo desarrollo ha permitido a mejorar la calidad a largo plazo del proyecto mediante la resolución de errores, la adición de soporte a otras funciones dentro de los procedimientos y arreglos a los tests cuando estos fallaban.
- En **Sección 4.3 Contribuciones menores** se agrupan aportaciones que solo afectan al renderizado de la documentación, erratas o el tamaño de los cambios es reducido.

4.1. Contribuciones científicas

Con estas se pretende mejorar las capacidades que ofrece la librería *pvlib-python* en forma de utilidades completamente novedosas como modelos científicos.

4.1.1. Modelado de ajuste espectral

- *Pull Request: #1658*

Esta propuesta plantea incluir otro modelo de ajuste espectral, un factor que toma un valor en torno a 1 y que se emplea para corregir la irradiancia incidente en un módulo fotovoltaico debido a la variación del espectro de la luz solar. El modelo se basa en la tesis doctoral de Nuria Martín Chivelet [26] y en un artículo de la misma autora [27].

Este modelo es de gran interés para la librería ya que depende de dos variables que se pueden obtener con extrema facilidad y normalmente se encuentran en el flujo de simulación: la masa de aire absoluta y el índice de claridad. Sin embargo, otros

modelos dependen de la cantidad de agua en la atmósfera, que no es tan sencillo de obtener, o son un polinomio en una sola variable, lo que disminuye la precisión del ajuste.

La propuesta finalmente se cierra sin incluirse los cambios en la librería al detectar que el modelo necesita otros modelos para funcionar, pero estos segundos no son fácilmente accesibles.

4.1.1.1. Fundamento teórico

El modelo [27] plantea una relación entre la efectividad bajo un espectro estándar y la efectividad bajo un espectro arbitrario caracterizado por la masa de aire y el índice de claridad. Originalmente esto se confunde al desarrollar esta aportación con un modelo de ajuste similar a otros en la literatura que corrigen la irradiancia incidente, para dar lugar a la efectiva; ejemplos de estos modelos se encuentran en la librería, como el modelo desarrollado por la empresa *First Solar* descrito en [28] o el de Caballero et al. en [29]. Los modelos en cuestión crean un factor de ajuste M que se define, según el estándar IEC 60904-7 [29, Eq. (2)]:

$$M = \frac{\int_{\lambda_1}^{\lambda_2} E(\lambda) SR(\lambda) d\lambda \int_{\lambda_3}^{\lambda_4} E^*(\lambda) d\lambda}{\int_{\lambda_1}^{\lambda_2} E^*(\lambda) SR(\lambda) d\lambda \int_{\lambda_3}^{\lambda_4} E(\lambda) d\lambda} \quad (4.1)$$

Sin embargo, el modelo en [27] no se ajustaba a esta descripción, sino que planteaba la siguiente ecuación:

$$\frac{S_{efE}(\lambda)}{S_{ef\bar{G}}(\lambda)} = c \exp^{a(K_t - 0.74) + b(AM - 1.5)} \quad (4.2)$$

Donde $S_{efE}(\lambda)$ es la efectividad bajo un espectro arbitrario, $S_{ef\bar{G}}(\lambda)$ es la efectividad bajo un espectro estándar, K_t es el índice de claridad, AM es la masa de aire, y a , b y c son constantes que dependen del material del módulo fotovoltaico y de cada componente de la irradiancia.

Esta constituye la primera fracción de las tres relaciones que se plantean en la tesis doctoral de Nuria Martín Chivelet [26], que es:

$$PS = 1 - \frac{S_{efE}(\lambda)}{S_{ef\bar{G}}(\lambda)} \frac{E_{\lambda < \lambda_0}}{\bar{G}_{\lambda < \lambda_0}} \frac{\bar{G}}{E} \quad (4.3)$$

Se puede observar que la ecuación 4.2 solo contempla una parte de la definición que aplica del ajuste espectral, por ende, invalida la implementación que se había hecho del modelo de dicho artículo.

4.1.1.2. Resultado

Para implementar este modelo se plantea una función que toma los valores de índice de claridad K_t , masa de aire AM y el material, y devuelve el factor de ajuste espectral M para cada componente de la irradiancia. Opcionalmente el usuario puede proveer sus propios coeficientes para cada componente.

Este patrón de código crea un poco de complejidad en la implementación, pero facilita la aplicación de los ajustes a cada componente de la irradiancia.

La documentación creada para la función es extensa y detallada; se puede consultar en [A.1](#).

Además, se crea un ejemplo de uso para mostrar su uso en un contexto más amplio, que se puede consultar en [B.1](#). Sobre este ejemplo se realiza una comparativa con los otros modelos de ajuste espectral de la librería y se detecta que no sigue las tendencias esperadas gracias a las dos últimas gráficas realizadas con este propósito.

Si bien el modelo del artículo [\[27\]](#) se demuestra que no funciona para el propósito de ajuste espectral, se desestima la posibilidad de incluir todo el desarrollo elaborado en la tesis [\[26\]](#) por las siguientes razones:

- La imposibilidad de acceder al documento de forma online.
- La ausencia de una versión en inglés.

Estos puntos son determinantes, pues es imprescindible la referencia de una implementación sirva a la comunidad internacional; recordemos que el proyecto *pplib-python* está en inglés y se espera que sea accesible a cualquier persona interesada.

4.1.2. Proyección del cenit solar sobre las coordenadas de un colector

- *Issue:* #1734
- *Pull Request:* #1904

Esta es la primera de una trilogía de contribuciones que se plantean para aplicar un modelo de pérdidas por sombreado en módulos con diodos bypass. El objetivo de esta primera contribución es calcular la proyección del cenit solar sobre las coordenadas de un colector, y se utiliza para calcular la fracción de sombra unidimensional en geometrías de paneles que comparten eje de rotación en común.

Las otras dos contribuciones se detallan en [Subsección 4.1.3 Cálculo de fracción de sombra unidimensional](#) y en [Subsección 4.1.4 Pérdidas por sombreado en módulos con diodos de bypass](#).

Esta funcionalidad ya existía integrada en el código de una función de la librería, así que el aporte consistió en rehacerla de nuevo empleando referencias bibliográficas y contrastando las implementaciones. No se extrajo de la función anterior para duplicar la revisión, pero más tarde se modificó para que la función antigua llamara a la nueva.

4.1.2.1. Fundamento teórico

Dos cálculos de bastante interés en geometría solar son obtener los ángulos óptimos de seguimiento para un colector y calcular la fracción de sombra incidente. Ambos cálculos tienen en común un paso muy importante, que es saber con qué ángulo inciden los rayos directos del Sol sobre la superficie del colector, pero referenciado al plano de rotación del mismo. Este plano de rotación equivale a la sección transversal de los colectores, por ello que sea de utilidad en la resolución de problemas analíticos.

Para facilitar el entendimiento de este concepto, es preferible imaginar un sistema de rotación de un solo eje. Se puede considerar que un colector fijo, es decir, donde los módulos no rotan, es un caso particular de un seguidor uniaxial. Así será posible definir algunos conceptos relevantes para la geometría de una forma más sencilla.

Se encuentran estos dos casos de uso:

- Para el cálculo de los ángulos óptimos de seguimiento en seguidores de un solo eje, asumiendo que interesa seguir al Sol en su trayectoria diaria, se debe conocer el ángulo de incidencia de los rayos solares sobre la superficie del colector en el plano de rotación del mismo. Es decir, proyectar el cenit solar en el plano perpendicular al eje de rotación del colector, que es aquel que contiene todos los vectores normales al plano del colector.
- En el caso de la fracción unidimensional de sombra, interesa saber en donde impactan los límites del colector frontal sobre el trasero. Para ello, se proyecta el cenit solar en el plano perpendicular al eje de rotación del colector, que es aquel que contiene los dos vectores normales a los planos de los colectores.

Se puede encontrar en más detalle en el artículo de Lorenzo, Narvarte y Muñoz en [30] a partir del cual continúa el trabajo de [16], sección sobre *True-Tracking Angle*. Esta última referencia es ampliamente utilizada a lo largo del repositorio.

4.1.2.2. Resultado

Se crea una nueva función en el módulo `pvlib.shading` llamada `projected_solar zenith` que se encarga de calcular la proyección del cenit solar sobre las coordenadas de un colector.

Esta función cuenta con un reducido número de parámetros de entrada: el cenit y el azimut solar y la orientación y la inclinación del eje del colector.

Se puede consultar la documentación resultante en una captura de pantalla en A.2.

La propuesta es bien recibida y se incluyen los cambios como parte de la librería en la versión 0.10.4.

Accesible en `pvlib.shading.projected_solar zenith angle`.

4.1.3. Cálculo de fracción de sombra unidimensional

- *Issue:* #1689
- *Pull Request:* #1962

Esta propuesta es la segunda de la trilogía de propuestas para poder aplicar un modelo de pérdidas por sombreado. Continúa con la propuesta anterior, y se encarga de calcular la fracción de sombra unidimensional en paneles con determinadas geometrías.

Aquí se plantea la aplicación de un modelo para conocer la fracción de sombra unidimensional en paneles que comparten eje de rotación en común. La implementación se basa en [31].

Desarrollo

Este es un tema de interés sobre el que ya existían propuestas y discusiones en el repositorio, pero no se había llegado a implementar ninguna. Como contexto previo a la realización de esta propuesta, se puede mencionar la número #1725 fundada en un póster desarrollado por *First Solar* en la librería *pvlib-python*, que se puede consultar en <https://github.com/pvlib/pvlib-python/pull/1725>.

4.1.3.1. Fundamento teórico

El modelo propuesto en [31] parte de un diseño de dos colectores de un eje que comparten la misma dirección del eje, y uno se encuentra más cercano al Sol que el otro. Lo interesante de este diseño es que tiene en cuenta múltiples variables de diseño, como la pendiente, la separación entre el eje y el plano colector, e inclinaciones distintas del colector sombreado y el que sombra.

Se requiere conocer el ángulo proyectado del cenit. A partir de este ángulo, mediante intersección de rectas, se puede conocer la fracción de sombra unidimensional. Realmente se trata de una función muy compleja por el número de entradas que tiene, pues adicionalmente el cálculo de esta proyección se hace internamente en la función para simplificar la API - es decir, la interfaz programática que usarán los usuarios.

Se llama fracción de sombra **unidimensional** porque mide la fracción de sombra a lo largo de la linea del avance para una misma azimuth pero una elevación solar distinta. Normalmente este valor es el que dota de mayor información, pues los cambios de la sombra debido a la azimuth no cobran tanta importancia en la mayoría de ubicaciones terrestres. Son aquellas latitudes más cercanas a los polos las que más se ven afectados por la cambiante azimuth.

Para facilitar la comprensión de este apartado se muestra un esquema de la sección de dos colectores con distinta inclinación y el ángulo de incidencia solar proyectado, en la imagen 4.1. f_s representa la fracción sombreada del colector 2 por el colector 1:

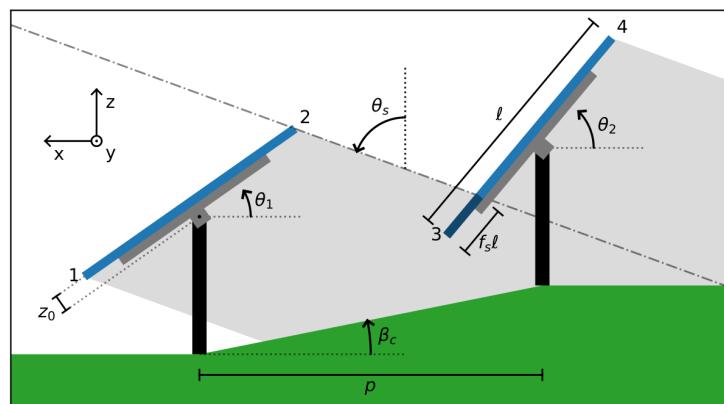


Figura 4.1: Esquema dos colectores parametrizados, donde uno sombra al otro. La nomenclatura corresponde la ecuación 4.4.

Fuente: Figura 3 en [31].

Las ecuaciones relevantes implementadas son (12) y (13) de [31], 4.4 y 4.5 respectivamente en este documento:

$$t^* = \frac{1}{2} \left(1 + \left| \frac{\cos(\theta_1 - \theta_s)}{\cos(\theta_2 - \theta_s)} \right| \right) \\ + sgn(\theta_s) \frac{z_0}{\ell} \left(\frac{\sin(\theta_2 - \theta_s) - \sin(\theta_1 - \theta_s)}{|\cos(\theta_2 - \theta_s)|} \right) \\ - \frac{p}{\ell} \left(\frac{\cos(\theta_s - \beta_c)}{|\cos(\theta_2 - \theta_s)| \cos(\beta_c)} \right) \quad (4.4)$$

$$f_s = \begin{cases} 0 & \text{si } t^* < 0 \\ t^* & \text{si } 0 \leq t^* \leq 1 \\ 1 & \text{si } t^* > 1 \end{cases} \quad (4.5)$$

4.1.3.2. Resultado

Después de 102 comentarios en la propuesta, finalmente se incluyen los cambios como parte de la librería en la versión 0.11.0.

Se ha creado un ejemplo de uso para facilitar su uso y aplicar un patrón que facilita cambiar entre los valores adecuados de fracción de sombra, pues esta función solo entiende de los ángulos del seguidor que sombreá y del que es sombreado, pero no de cómo cambian sus roles a lo largo del día en el caso de un sistema con eje orientado Norte-Sur.

La documentación de la función se puede previsualizar en [A.3](#), y la del caso de uso en [B.2](#).

Accesible en `pvlib.shading.shaded_fraction1d`.

Además, como anécdota se ha de mencionar que durante el desarrollo de esta propuesta se identifica la ausencia de ejemplos de módulos orientados al Norte (propio de las instalaciones en el hemisferio Sur) en todo el repositorio, por lo que se añade un sencillo ejemplo de este tipo en la documentación del cuerpo de la función.

4.1.4. Pérdidas por sombreado en módulos con diodos de bypass

- *Issue:* #2063
- *Pull Request:* #2070

Con esta propuesta finaliza la trilogía de contribuciones del modelo de pérdidas por sombreado en módulos con diodos de bypass.

La propuesta es del modelo planteado por Martínez-Moreno, F., Muñoz, J. y Lorenzo, E., en [\[32\]](#). Este se encarga de calcular las pérdidas por sombreado en módulos con diodos de bypass.

4.1.4.1. Fundamento teórico

Los módulos de paneles solares de silicio se conforman de múltiples células fotovoltaicas. Una célula, cuando es irradiada por la luz solar, genera una corriente eléctrica. Si conectamos todas las células en serie, la corriente generada por todas las células es la misma, pero la tensión generada por cada célula se suma. Si una

Desarrollo

célula se sombra, la corriente generada por ella disminuye, y hará que el resto de las células deban forzar la corriente por ella. Esta célula sombreada, polarizada inversamente, se comporta como una carga para el resto, y disipará energía en forma de calor.

Este efecto no es proporcional a la superficie de la sombra, sino que varía en función de la geometría de la sombra, de la geometría de las células y la conexión de los diodos de bypass.

Un riesgo que supone que una célula trabaje disipando energía, en vez de generando, es que puede calentarse excesivamente y dañar las capas materiales de su módulo.

La solución que se emplea industrialmente consiste en añadir *diodos de bypass*. Estos permiten que, cuando una serie de células tiene sombra, la corriente mayoritaria generada por el resto de células que están bien iluminadas fluya a través de este diodo, evitando así atravesar la célula sombreada; de esta forma, se protege frente al sobrecalentamiento y la degradación temprana.

Ha de hacerse énfasis en que cada diodo de bypass protege varias células, ya que no es rentable económicamente poner un diodo por cada célula. El planteamiento que nos encontramos en [32] consiste en que, a partir del número de grupos de células protegidos por un diodo que se encuentran sombreados, se puede asumir que el aporte de potencia de este grupo es casi nulo por tener un diodo en conducción y cancelar la potencia proporcional de este grupo frente al total.

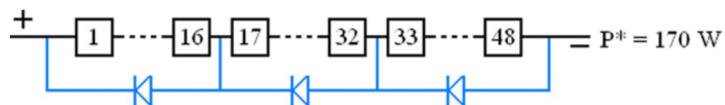


Figura 4.2: Esquema de un módulo con 3 diodos de bypass. Si un grupo cuenta con una célula sombreada, el exceso de corriente que no puede fluir a través de ella pasa por el diodo de bypass de su grupo.

Fuente: Figura 5, a) en [32].

En la imagen 4.2, suponiendo que la célula número 1 está sombreada, la corriente mayoritaria generada por los grupos 17 a 32 y 33 a 48 fluye a través del diodo que está en paralelo con las células 1 a 16. No importa en este caso la corriente del grupo 1 a 16, pues la tensión de este grupo es nula por tener un diodo en conducción.

Nótese que estos grupos se definen en [32] como *bloques*, y según los datos provistos en sus datos originales, un bloque está sombreado en cuanto una de sus células recibe una fracción infinitesimal de sombra.

Lo más complicado de esta contribución es explicar en detalle cómo identificar el número de bloques y su disposición en el módulo, pues existen varias posibilidades. Hay módulos lo suficientemente pequeños para que solo tengan un diodo, los hay con dos y con tres diodos, y los hay *half-cut* que también tienen 3 diodos, pero con una disposición que crea 6 bloques en vez de 3. Además, la progresión de los bloques que se va sombreando depende del sistema y la geometría de las sombras.

El resultado de este modelo establece la cantidad de potencia que se perdería

respecto de las mismas condiciones sin sombra, $SL = 1 - \frac{P_{\text{sombreado}}}{P_{\text{no sombreado}}}$. Además, anular la potencia de un bloque sombreado se hace sobre la componente directa de la irradiancia, que es la que normalmente genera las sombras, pues la componente difusa sigue impactando en las células sombreadas y creando un mínima parte de aporte energético.

La expresión de pérdidas de potencia es la ecuación 4.6, Eqs. [6] y [8] en [32].

$$(1 - F_{ES}) = (1 - F_{GS}) \left(1 - \frac{N_{SB}}{N_{TB} + 1} \right) \quad (6)$$

$$\left(1 - \frac{P_S}{P_{NS}} \right) = \left(1 - \frac{[(B + D^{CIR})(1 - F_{ES}) + D^{ISO} + R]}{G} \right) \quad (4.6) \quad (8)$$

4.1.4.2. Resultado

Se plantea una función que toma como entrada las irradiancias global y directa; los números de bloques total y sombreados, y la fracción de sombra. Se deduce internamente la irradiancia difusa tanto del cielo como de albedo, y se calcula el factor de pérdidas por sombreado.

La documentación de la función cuenta con un pequeño ejemplo de uso, y se puede consultar en A.4. Además, se incluye un ejemplo de uso en B.3 que compara distintas disposiciones de bloques por módulo a lo largo de un día, tanto en un solo módulo como en un sistema de módulos.

Tras unos 85 comentarios, finalmente se incluyen los cambios como parte de la librería en la versión 0.11.0.

Accesible en [pvlib.shading.direct_martinez](#).

4.1.5. Fracción diaria de radiación difusa fotosintetizable en función de la fracción difusa global

- *Issue:* #2047
- *Pull Request:* #2048

Esta contribución se trata de un pequeño modelo que abre un nuevo tema en la librería: la agrivoltaica. La agrivoltaica es una técnica en la que coexisten la producción de energía solar y la producción agrícola en un mismo terreno. La ventaja es que un exceso de irradiancia puede no aumentar la producción agrícola e incluso llegar a ser perjudicial para la plantación, y el retorno económico del terreno puede ser mayor que si solo se dedicase únicamente a la producción de energía solar o de alimentos. Además, disminuye el uso exclusivo de suelo para producción energética, que en ocasiones es un tema controversial¹.

El modelo en cuestión es la continuación de un trabajo realizado por Spitters C. J. T. et al. que cubre la separación de irradiación en directa y difusa en general [33], y que posteriormente él desglosa en dos expresiones en [34], que se pueden sustituir una en la otra. El objetivo de este modelado es calcular la fracción de irradiación

¹Por ejemplo, el siguiente artículo periodístico: <https://www.bbc.com/news/uk-politics-65926756>.

Desarrollo

difusa que es útil para las plantas -es decir, fotosintetizable- a partir de la fracción de irradiación difusa global.

4.1.5.1. Fundamento teórico

La irradiación difusa fotosintetizable (PAR, por sus siglas en inglés), es la radiación difusa que se encuentra en el rango de longitudes de onda que las plantas pueden absorber y utilizar para la fotosíntesis. Es interesante de cara a simular el crecimiento y producción de las plantas, y se emplea en modelos de cultivo. Esto último queda fuera del alcance de la librería, pero se plantea como un paso para motivar y facilitar el diseño de sistemas agrivoltaicos.

Se utiliza irradiación diaria [$J/m^2/dia$], en vez del valor instantáneo irradiancia propio de la simulación de sistemas fotovoltaicos [W/m^2].

Un análisis de distintos modelos y la validación de los mismos que contiene la expresión única se puede encontrar en [35], artículo del que origina esta propuesta de contribución inicialmente.

La fórmula que relaciona estas dos fracciones es la ecuación 4.7, que se puede encontrar en [34]:

$$k_{d_PAR} = \frac{PAR_{diffuse}}{PAR_{global}} = \frac{\left[1 + 0.3\left(1 - (k_d)^2\right)\right] k_d}{1 + \left(1 - (k_d)^2\right) \cos^2(90 - \beta) \cos^3 \beta} \quad (4.7)$$

Donde k_d es la fracción de irradiación difusa global, y β es la elevación solar media diaria.

4.1.5.2. Resultado

Este modelo se implementa en una función que toma como entrada la fracción de irradiación difusa global y el cenit solar medio diario. Este cambio de elevación a cenit se hace porque este último se emplea más habitualmente en el repositorio.

La documentación de la función es concisa y se puede consultar en A.5. Por otro lado, se incluye un ejemplo de uso mucho más elaborado en B.4 que muestra cómo varía la fracción de PAR difusa en función de la fracción de irradiación difusa global durante varios días de Septiembre.

Finalmente se incluyen los cambios como parte de la librería en la versión 0.11.0.

Accesible en `pvlib.irradiance.diffuse_par_spitters`.

4.1.6. Modelo de pérdidas por heterogeneidad de irradiancia por célula

■ *Issue:* #1541

■ *Pull Request:* #2046

Esta contribución implementa un modelo de pérdidas sobre la potencia de salida para módulos bifaciales, es decir, aquellos que pueden recibir irradiación solar

tanto por una cara delantera como por la trasera. El modelo se aplica para tener en cuenta irradiancia que no es homogénea en la superficie del módulo. Se trata del trabajo descrito en [36]. Presenta interés en sistemas bifaciales, donde la cara trasera normalmente se expone a la luz reflejada por el suelo y otras obstrucciones, y por tanto la irradiancia de esa cara no es homogénea.

En la cara frontal la irradiancia es mucho más homogénea, así que no se suele tener en cuenta este efecto. No obstante, este modelo se aplica para el valor global de las irradiancias a nivel de célula, o sea, de la suma de la frontal y de la trasera.

4.1.6.1. Fundamento teórico

Anteriormente se explicaba el mecanismo de interconexión de células solares fotovoltaicas, y cómo una célula sombreada puede evitar la producción de energía de las células circundantes. De forma similar ocurre a pequeña escala, cuando una o varias células reciben valores de irradiancia ligeramente distintos al resto. La célula que recibe menos irradiancia limita la corriente, y la potencia de salida del módulo se ve reducida. Sin embargo, por darse en una escala mucho menor, no se puede considerar que la célula sombreada anule la potencia de las demás ya que los diodos de bypass no entran en conducción, sino que simplemente la reduce ligeramente.

Desde un punto de vista computacional, resolver un sistema de múltiples células, cada una con su propia irradiancia, es realmente costoso en tiempo y en recursos. Cada célula tendría su propia curva I-V, que representa cuanta corriente y tensión genera dependiendo del punto de tensión de trabajo. El planteamiento que se hace en [36] es realizar un trabajo previo de caracterización de la heterogeneidad, cuantificarla y establecer un modelo de menor orden de complejidad.

Para caracterizar distribuciones de irradiancia, en el artículo de Deline et al. [36] se plantea utilizar la desviación estándar, muy común para distribuciones normales, o la *Diferencia Absoluta Media Relativa (RMAD)*, por sus siglas en inglés), que es una medida de dispersión que se argumenta ser más adecuada para distribuciones no normales [37].

La pérdida de potencia de salida M se calcula con un polinomio evaluado en RMAD:

$$M = 1 - \frac{P_{\text{array}}}{\sum P_{\text{cells}}} \quad (4.8)$$

donde P_{array} es la potencia de salida del módulo, y P_{cells} es la potencia máxima de salida de cada célula.

Se proponen dos modelos para el polinomio que define M , para dos perfiles de irradiancias globales distintos: uno para sistemas sujetos fijos y otro para seguidores de un eje. No obstante, solo se implementa el de sistemas fijos ya que la referencia indica que para valores anuales parece ser más preciso.

4.1.6.2. Resultado

La implementación del modelo es casi directa, pues el patrón para evaluar un polinomio en determinados valores es directo con el uso de la librería *NumPy*. Se

añade un parámetro opcional para que la persona usuaria final pueda especificar su propio polinomio o emplear alguno otro provisto en el artículo original; y se añaden dos parámetros que corrigen el factor de pérdidas en función de los factores de forma de las células.

Esta documentación es extensa porque es clave para su buen uso y entendimiento. Abarca desde la interfaz general hasta las ecuaciones que describen la entrada de *RMAD* y cómo la heterogeneidad de la irradiancia trasera se propaga a la global. Se recomienda revisar la documentación de la función en [A.6](#).

Asimismo se ha aportado un ejemplo de uso que parte de la Figura 1 (A) e implementa la función que calcula la *RMAD* y de la cual se deriva posteriormente la pérdida de potencia. Se acompaña de propiedades interesantes que pueden aligerar la ejecución. Se puede consultar en [B.5](#).

Tras involucrarse 7 personas y generar 86 comentarios en los que se plantean distintas formas de implementar el modelo, aclaraciones sobre las unidades de entrada y salida, y modificaciones al planteamiento original, finalmente se incluyen los cambios como parte de la librería en la versión 0.11.1 (sin publicar oficialmente a día de la redacción de este documento).

Accesible en [pvlib.bifacial.power_mismatch_deline](#).

Los autores originales del modelo científico piden contrastar la implementación con la suya, y se valora hacerlo en un futuro próximo antes de la siguiente versión de la librería. Se plantea como tarea futura.

4.1.7. Transformación de respuesta espectral a eficiencia cuántica externa y viceversa

- [Issue: #2040](#)
- [Pull Request: #2041](#)

Esta contribución podría calificarse de menor debido a la ausencia de dificultades en su implementación. No obstante, por dotar de una nueva funcionalidad científica a la librería, se incluye en este apartado.

La propuesta consiste en dos funciones, una que convierte la respuesta espectral a la eficiencia cuántica externa y otra que hace la operación inversa. Ambas son medidas de la eficiencia de una célula solar para determinadas longitudes de onda de la luz, la primera como corriente generada en función de la potencia recibida y la segunda como la razón de fotones incidentes que generan una corriente eléctrica.

4.1.7.1. Fundamento teórico

La eficiencia cuántica externa es una medida de la eficiencia de una célula solar, y se define como la razón de fotones incidentes que generan una corriente eléctrica para determinado color de la luz. La respuesta espectral es la corriente generada por una célula solar en función de la longitud de onda de la luz incidente.

La relación entre ambas es la siguiente [38, pp. 15-16, Eq. (7)]. Se puede encontrar en la ecuación [2.2](#):

4.1.7.2. Resultado

Ambas funciones se diseñan con un patrón que permite que la entrada sea única a través de una tabla tabulada e indexada por la longitud de onda en un pandas.Series o pandas.DataFrame, o bien mediante dos listas de irradiancias y longitudes de onda.

Además, se añade la posibilidad de normalizar los valores de salida, es decir, hacer que el máximo retornado sea 1. Esta utilidad es interesante porque las medidas de eficiencia cuántica externa y respuesta espectral es habitual que sean relativas al dispositivo de medida, y se necesite hacer un tratamiento posterior para deducir los valores absolutos.

Las dos funciones cuentan con ejemplos en el propio cuerpo de la documentación, y se puede consultar en [A.7](#).

Se realizan tests que comprueban cada función por separado y otro que confirma que son recíprocas.

Se incluyen los cambios sin mayores dificultades en la versión 0.11.0 de la librería.

Accesibles en:

- [pvlib.spectrum.qe_to_sr](#).
- [pvlib.spectrum sr_to_qe](#).

4.1.8. Adición de base de datos de respuesta espectral de algunas tecnologías

- [Issue: #2037](#)
- [Pull Request: #2038](#)

Con esta propuesta se pretendía añadir una serie de respuestas espetrales o de eficiencia cuántica externa de células solares de distintas tecnologías comunes, para facilitar la investigación.

4.1.8.1. Fundamento teórico

Una curva de respuesta espectral indica la capacidad que tiene un semiconducto fotovoltaico en convertir la luz incidente en corriente eléctrica en función de la longitud de onda. La eficiencia cuántica externa similarmente es una medida de la eficiencia de una célula solar para convertir fotones en pares electrón-hueco.

Dependiendo de la tecnología de cada material, la respuesta varía. Además, se puede argumentar que otros aspectos constructivos también afectan, como los espesores de las capas y la presencia de impurezas.

4.1.8.2. Resultado

La propuesta no se llegó a añadir en la librería ya que los datos provistos de un repositorio público (en Duramat²) no estaban respaldados por un procedimiento

²Véase <https://www.osti.gov/biblio/2204677>.

Desarrollo

que garantizase que los datos fueran representativos. Tras unas semanas sin mostrarse interés en la propuesta se solicita retroalimentación para tomar una decisión y procede a cerrarse.

4.1.9. Adición de espectro estándar completo ASTM G173-03

- *Issue:* #2039
- *Pull Request:* #1963

Esta adición plantea añadir las componentes de irradiancia directa y extraterrestre del estándar ASTM G173-03, que es un espectro de referencia para la radiación solar en la superficie terrestre. La componente global ya se encontraba en la librería. Se aprovecha para añadir flexibilidad y prevenir la adición de nuevos estándares, como el similar ASTM G173-23 en un futuro.

4.1.9.1. Fundamento teórico

Un espectro estándar es de gran utilidad para comparar módulos y establecer métodos idénticos para tomar las medidas en la industria. En el caso del espectro estándar ASTM G173-03, se establecen una serie de puntos entre los 280 nm y los 4000 nm que simulan una distribución espectral bastante plausible.

La irradiancia extraterrestre es la irradiancia que se recibe en el espacio, y la irradiancia directa es la irradiancia que se recibe en la superficie terrestre sin ser dispersada por la atmósfera, y la global es toda la que se recibe en la superficie terrestre.

El estándar ASTM G173-03 se encuentra en [11], pero los valores del espectro están disponibles abiertamente en <https://www.nrel.gov/grid/solar-resource/spectra-am1.5.html>. La nueva revisión ASTM G173-23 se encuentra en [39], pero no cuenta con datos abiertos en la red a día de la redacción de este documento.

Las componentes de este espectro se pueden consultar en la Figura B.6.

4.1.9.2. Resultado

Esta implementación permite obtener el estándar completo y todas sus componentes, lo que hace que la antigua función que solo devolvía la componente global quede obsoleta (`pvlib.spectrum.get_am15g`). Se aportan cambios y se añaden tests para asegurar que la función obsoleta se elimina adecuadamente en el futuro.

Se aplica un patrón que permitirá añadir otros estándares en el futuro sin necesidad de modificar la interfaz de los usuarios. Tanto los tests como la documentación requerirán modificaciones menores para añadir los nuevos estándares.

La documentación de esta utilidad se puede consultar en A.8. Además, se añade un ejemplo de uso, disponible para previsualizar en B.6.

Se incluyen los cambios en la versión 0.11.0 de la librería.

Accesible en `pvlib.spectrum.get_reference_spectra`.

4.1.10. Cálculo geométrico de sombras en 3D

- *Issue:* #2069
- *Pull Request:* #2106

Se ha podido comprobar que el cálculo de sombras en sistemas fotovoltaicos dentro de *pvlib-python* es un tema de interés que gracias a este TFG se ha mejorado. Adicionalmente, se ha podido comprobar que la librería no cuenta con una función que permita calcular sombras a partir de objetos en 3D, pero realmente parece ser un tema sobre el que hay literatura. Además, *PVsyst*, un software de simulación de sistemas fotovoltaicos ampliamente conocido cuenta con esta funcionalidad.

Todo esto se puede explorar en un artículo sobre el sombreado de campos de cultivo en sistemas agrivoltaicos [40].

La propuesta que aquí se hace realiza una generalización del cálculo de sombras a superficies limitadas y libres en el espacio 3D.

Es importante denotar que el valor de esta propuesta está pendiente de contar con el interés positivo del grupo, en especial por la extensión del código y la adición de algunas modificaciones al procedimiento del artículo original. Actualmente cuenta con opiniones divididas al respecto.

4.1.10.1. Fundamento teórico

Para esta contribución es imprescindible aplicar conceptos de geometría y cálculo vectorial:

- Definición de una recta a partir de un punto y un vector.
- Intersección de una recta con un plano.
- Traslación de puntos.
- Rotación de puntos respecto del origen, mediante matrices de rotación o representación de ángulos de Euler.
- Limitar superficies a determinadas coordenadas de otro plano.

No se ahonda en estos detalles que generalmente nos brindan algunas librerías de cálculo científico en Python, como *NumPy*, *SciPy* y *Shapely*:

- *NumPy* para operaciones con vectores y matrices de forma muy veloz.
- *SciPy* para cálculos matemáticos más complejos, particularmente en este caso para hacer rotaciones según que convención se desee, y aplicarla rápidamente mediante el uso de cuaterniones, una manera de representar de rotaciones más compacta y veloz computacionalmente.
- *Shapely* para operaciones geoespaciales (según un modelo topológico que permite operaciones geométricas con polígonos). En este caso, se emplea para limitar las sombras a una superficie de interés y representar los objetos de forma uniforme.

No obstante, la intersección plano-recta más eficiente y la traslación de puntos no se proporcionan de forma nativa por estas librerías, pero son operaciones que se

pueden realizar muy fácilmente con los operadores de Python.

El procedimiento puramente matemático se propone en [40], pero ante todo es necesario priorizar la vectorización del cálculo mediante librerías más rápidas para que sea útil frente al volumen de datos que se suele analizar en simulaciones anuales fotovoltaicas, cuya resolución suele ser horaria como mínimo.

Lo primero que necesita un flujo de trabajo de este tipo es definir coordenadas para los objetos de la escena y sus límites. Para ello, habrá de establecerse un sistema de referencia. En el caso de la propuesta realizada, se opta por cambiar el sistema de coordenadas propuesto en [40] por el de [16], que es más conocido y utilizado en la librería.

Posteriormente, una vez creadas las superficies, tanto las sombreadas como las que generan sombras, se debe calcular el vector de posición solar. Este vector se calcula a partir de la posición del Sol en el cielo, que se puede obtener con la función `pvlib.solarposition.get_solarposition` y con relaciones trigonométricas.

A continuación, se proyectan los vértices de las superficies que sombrean sobre el plano sombreado, y estos puntos definirán una sombra en la escena 3D. Nótese por tanto que esta sombra puede y debe tener tres coordenadas.

Para obtener la sombra 3D final, debe limitarse los límites de esta a la superficie de interés. Para esto se emplea *Shapely*, que permite realizar operaciones geométricas con polígonos.

Si se desease obtener la sombra en un plano 2D, para realizar otros cálculos y facilitar la visualización, se debe realizar una traslación que ubique la figura en un plano que pase por el origen y unas rotaciones contrarias a las que definen el plano sobre la que se proyectó. Este paso se hace trasladando los puntos del objeto en 3D tomando una referencia cualquiera en su mismo plano.

4.1.10.2. Resultado

Ahondando en los detalles, para la resolución de este problema se decide aplicar un paradigma orientado a objetos para facilitar el uso de esta funcionalidad. Se plantean dos objetos:

- `FlatSurface`: una superficie poligonal en el espacio 3D, genérica y que implementa las operaciones que no dependen de la geometría de la superficie, como el cálculo de la sombra que le llega.
- `RectangularSurface`: una especialización de `FlatSurface` que permite definir superficies rectangulares, que son las más comunes en sistemas fotovoltaicos. En especial, este objeto facilita la creación de superficies a partir de los parámetros más habituales en una instalación fotovoltaica.

una base para cualquier superficie poligonal (`FlatSurface`) y una especialización para superficies rectangulares, que son las más comunes en sistemas fotovoltaicos (`RectangularSurface`).

El diagrama UML resultante sería:

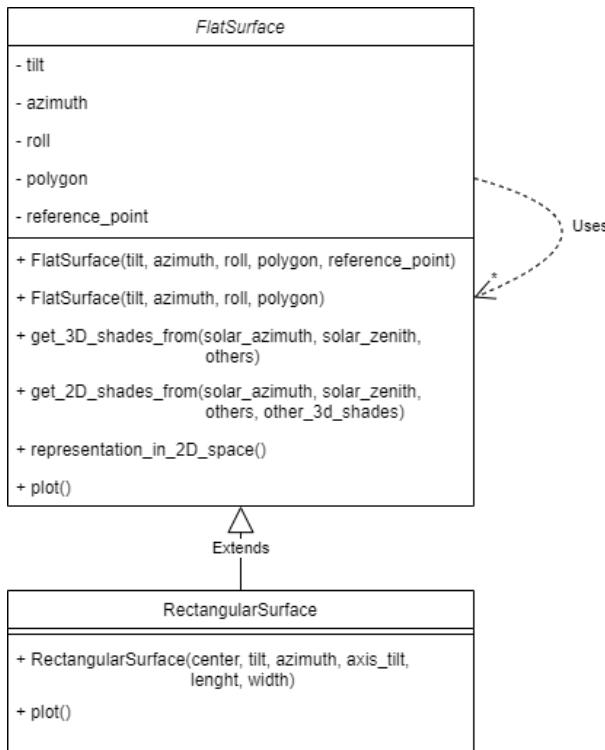


Figura 4.3: Diagrama UML de la propuesta de cálculo de sombras en 3D.

La elección de un diagrama tan sencillo no es arbitraria: aquí prima la simplicidad y la facilidad de revisar el código para determinar si tiene valor o no dentro de la librería *pvlib-python*. Y realmente se logra sintetizar la API con un buen compromiso entre lo automatizado y la intervención de una persona usuaria. Véase el ejemplo elaborado a continuación:

Listing 4.1: Caso de uso de ejemplo para la propuesta de cálculo de sombras en 3D.

```

1 from pvlib.spatial import RectangularSurface
2 import matplotlib.pyplot as plt
3 from mpl_toolkits.mplot3d.art3d import Poly3DCollection
4 import shapely
5
6 solar_azimuth = 165 # degrees
7 solar_zenith = 75 # degrees
8
9 # Define two rows of panels
10 row1 = RectangularSurface( # south-most row
11     center=[0, 0, 3], azimuth=165, tilt=20, axis_tilt=10, width=2, length=20
12 )
13
14 row2 = RectangularSurface( # north-most row
15     center=[0, 3, 3], azimuth=165, tilt=30, axis_tilt=10, width=2, length=20
16 )
17
18 # Calculate shadows
19 shades_3d = row2.get_3D_shades_from(solar_zenith, solar_azimuth, row1)
20 shades_2d = row2.get_2D_shades_from(
21     solar_zenith, solar_azimuth, shades_3d=shades_3d
22 )
23
24 # Plot
25 row_style = {"color": "darkblue", "alpha": 0.5}
26 shade_style = {"color": "dimgrey", "alpha": 0.8}
  
```

Desarrollo

```
27 row_style_2d = {**row_style, "add_points": False}
28 shade_style_2d = {**shade_style, "add_points": False}
29
30 fig = plt.figure(figsize=(10, 10))
31
32 # Split the figure in two axes
33 gs = fig.add_gridspec(10, 1)
34 ax1 = fig.add_subplot(gs[0:7, 0], projection="3d")
35 ax2 = fig.add_subplot(gs[8:, 0])
36
37 # 3D plot
38 ax1.view_init(
39     elev=60,
40     azim=-30, # matplotlib's azimuth is right-handed to Z+, measured from X+
41 )
42 row1.plot(ax=ax1, **row_style)
43 row2.plot(ax=ax1, **row_style)
44 for shade in shades_3d.geoms:
45     if shade.is_empty:
46         continue # skip empty shades; else an exception will be raised
47     # use Matplotlib's Poly3DCollection natively since experimental
48     # shapely.plotting.plot_polygon does not support 3D
49     vertexes = shade.exterior.coords[:-1]
50     ax1.add_collection3d(Poly3DCollection([vertexes], **shade_style))
51
52 ax1.axis("equal")
53 ax1.set_zlim(0)
54 ax1.set_xlabel("West(-) <X> East(+) [m]")
55 ax1.set_ylabel("South(-) <Y> North(+) [m]")
56
57 # 2D plot
58 row2_2d = row2.representation_in_2D_space()
59 shapely.plotting.plot_polygon(row2_2d, ax=ax2, **row_style_2d)
60 for shade in shades_2d.geoms:
61     shapely.plotting.plot_polygon(shade, ax=ax2, **shade_style_2d)
62
63 # Calculate the shaded fraction
64 shaded_fraction = sum(shade.area for shade in shades_2d.geoms) / row2_2d.area
65 print(f"The shaded fraction is {shaded_fraction:.2f}")
```

Debe denotarse que la mayor parte del código supone imprimir la escena y las sombras por pantalla. La parte más interesante, que es el cálculo de las sombras y la fracción sombreada, se reduce a unas pocas 14 líneas de puro código, mientras que se necesitan 27 para mostrar el resultado en 3D y 2D. La escena con las sombras es la que se muestra en la figura 4.4:

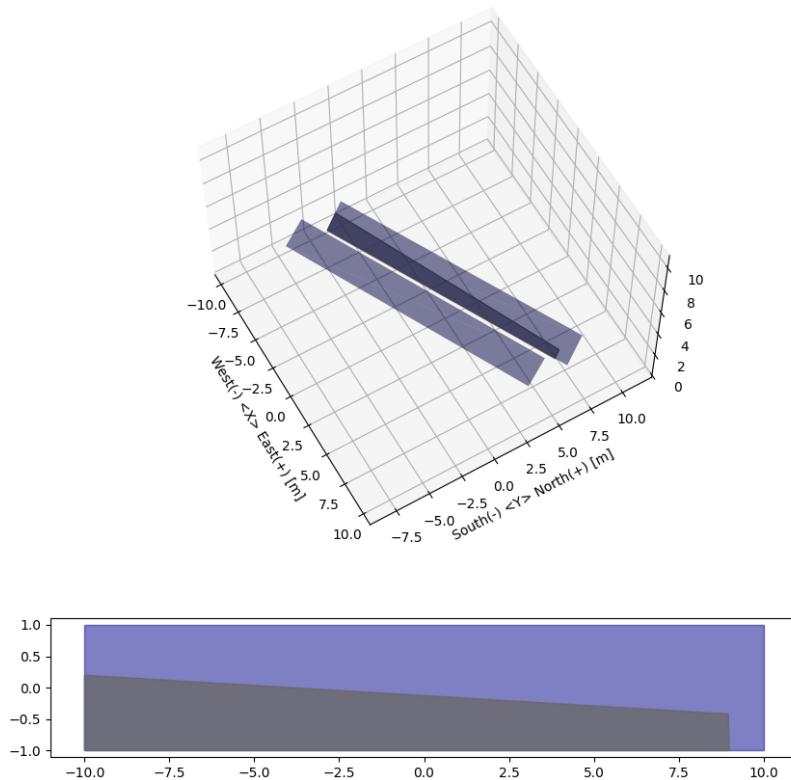


Figura 4.4: Ejemplo de sombreado para coordenadas solares instantáneas en 3D.

Puede consultarse la documentación de los dos objetos planteados en A.9 y de los dos ejemplos de uso en B.7. Uno de los ejemplos es una simulación temporal en la que se puede observar cómo varía la sombra a lo largo de los equinoccios y los solsticios, mientras que la otra es la misma escena planteada aquí arriba, pero con el texto explicativo propio de un ejemplo.

Se deja abierto el aporte a la librería, con opiniones diversas sobre si incluir o no la propuesta.

4.2. Contribuciones técnicas

Son aquellas que requieren trabajar con el código base y hacer modificaciones pertinentes para mejorar las prestaciones de la librería. Por ejemplo, aquí se encontrarán algunos errores que se han arreglado o pequeñas funcionalidades que se han expuesto a la interfaz.

4.2.1. Arreglo a los tests de integración continua en Windows con Conda

- *Issue:* #2000

Desarrollo

- *Pull Request: #2007*

Un test empezó a fallar en la librería, se desconoce las causas, pero solo afectaba a las versiones de la librería cuando se instalaba en Windows con Conda. Este test comprobaba que una curva I-V de una célula solar - esto es, la tensión que aparece en sus terminales en función de la corriente que deja pasar la carga - resultaba válida, y específicamente esperaba que para determinado valor de entrada, la función devolviese un cero prácticamente exacto. Empezó a fallar y retornar un valor igualmente cercano a cero, pero fuera de la tolerancia que se suponía debía de estar.

Algunos de los mantenedores determinaron que no era necesario replantear ni el test ni conocer el motivo la causa, sino que se trataba de un exceso en la precisión que se le pedía al test. Se modificó para que aceptase un margen de error mayor.

Se alega que esto se debió al entorno de pre-compilación de Conda.

4.2.1.1. Resultado

El cambio se incluyó con éxito con bastante rapidez y fue agradecido por los mantenedores del proyecto.

4.2.2. Arreglo a un parámetro ignorado en una función de transposición inversa

- *Issue: #1970*
- *Pull Request: #1971*

Los procedimientos de integración continua supusieron un gran cambio de cara a garantizar la calidad del código en versiones posteriores a la implantación de los mismos. Previo a esto, algunas erratas permanecieron vigentes hasta día de la elaboración de este TFG.

Hoy se identifican fallos de análisis estático en el código nada más hacer una propuesta de cambios. Estos fallos son aquellos que se pueden identificar sin necesidad de ejecutar el código, como el de este caso, que se trataba de un parámetro que no se estaba utilizando en una función.

Este parámetro fue encontrado gracias al resultado de sintaxis del entorno de desarrollo y se procedió a modificar la función para que cumplir con su propósito: establecer la tolerancia absoluta de convergencia de un método numérico.

El parámetro es `xtol` de la función [`pvlib.irradiance.ghi_from_poa_driesse_2023`](#). Esta función hace una transposición inversa, que es deducir la irradiancia global horizontal a partir de la irradiancia en un plano inclinado. El parámetro `xtol` es la tolerancia absoluta en la convergencia del método numérico y es un factor determinante en el tiempo de ejecución de la función.

4.2.2.1. Resultado

El cambio se incluyó con éxito, aportando tests de integridad para la función y con el agradecimiento del autor original de la función.

4.2.3. Dar soporte a otra función para el cálculo del IAM en el flujo orientado a objetos

- *Issue:* #1742
- *Pull Request:* #1832

El modificador del ángulo de incidencia tiene en cuenta la reflexión de la luz incidente cuando impacta oblicuamente en una superficie transparente. Cuando la radiación es perpendicular a una superficie, se suele considerar que no hay pérdidas ($IAM = 1$), pero conforme el ángulo de incidencia aumenta, debido a la diferencia de índices de refracción una parte de la radiación se refleja, y las pérdidas aumentan ($IAM < 1$).

La librería *pvlb-python* cuenta con muchos modelos que calculan este modificador, pero no todos están disponibles en la Modelchain: un paradigma orientado a objetos que facilita la simulación de sistemas fotovoltaicos.

El modelo al que le faltaba dar soporte era un interpolador de datos ángulo-modificador, `pvlb.iam.interp`. Este toma valores experimentales para, mediante una estimación del valor que se pide, retornar el modificador que cabría esperar. Este interés fue reportado por un usuario donde explicaba que hay estándares para que se provean datos experimentales de este modificador, en vez de los coeficientes que otros modelos necesitan que se aporten.

4.2.3.1. Resultado

El cambio se incluyó con éxito y ahora se encuentra disponible su interfaz en la Modelchain, en `pvlb.modelchain.ModelChain.interp_aoi_loss`.

Además, en un plano igual o más importante, se detectó un error en la implementación que ignoraba un parámetro opcional en otro de los modelos que se pueden emplear para simular este efecto.

4.2.4. Suprimir una advertencia al publicar la distribución en PyPI

- *Pull Request:* #1778

Es buena práctica revisar los procedimientos que se realizan automáticamente, ya que con cierta frecuencia dan lugar a advertencias que se ignoran por no ser una cuestión crítica.

Siguiendo este criterio, se revisa manualmente el proceso:

1. Ejecución de los tests unitarios: pasan sin nuevos problemas detectados.
2. Creación de la documentación: no genera advertencias desconocidas anteriormente.
3. Construcción de la distribución: también pasa sin crear avisos.
4. Publicación de prueba en TestPyPI: se detecta una advertencia sobre el formato de la descripción que se obtiene de `README.md`, el archivo de presentación del proyecto.

No se trata de nada crítico: tan sólo es la plataforma PyPI que solicita explícitamente el formato del texto de la descripción del proyecto. Puede aportarse en texto plano, MarkDown o reStructuredText.

4.2.4.1. Resultado

Se suprime adecuadamente la advertencia especificando que el formato es reStructuredText y se añade el cambio propuesto sin mayor discusión.

4.2.5. Exponer parámetros de tolerancia para resolver el modelo de un diodo

- *Issue:* #1249
- *Pull Request:* #1764

Esta contribución da inicio con la solicitud de un usuario que desea modificar los parámetros de tolerancia de unas funciones que resuelven las curvas I-V de las células o módulos fotovoltaicos.

Estas curvas son de especial interés para saber cómo se va a comportar eléctricamente el circuito cuando se conecte una carga y hallar el punto de máxima potencia, pero la complejidad matemática que suponen hacen que se resuelvan mediante métodos numéricos. Estos métodos son iterativos y requieren de una precisión para converger.

Por defecto esta tolerancia es de 10^{-6} [voltios, amperios o vatios], lo que a este usuario le parecía excesivo y quería agilizar el proceso modificando dicho valor. Este caso no es extensible a todos los contextos, pero el planteamiento es completamente válido sobre las hipótesis que plantea el usuario.

4.2.5.1. Resultado

Por un lado, se trabajó directamente en esta propuesta y se planteó. Fue un cambio bienvenido en la librería.

Asimismo, se planteó que de cara a exponer parte del sistema de resolución por métodos numéricos, debía devolverse opcionalmente un objeto del método con información como su convergencia, entre otros.

Este último apartado supuso la mayor dificultad en implementarse, pero se consiguió exitosamente. Se añadieron tests profundos para asegurar que la nueva funcionalidad cumplía con la especificación.

4.2.6. Modificar tolerancias erróneas en varios tests unitarios

- *Pull Request:* #2082

Originalmente la librería empleó un framework de testing que especificaba las tolerancias como número de decimales después de la coma. Al cambiar de framework al nuevo y más moderno *pytest*, no se modificaron estos números para reflejar el nuevo significado de las tolerancias, como un margen absoluto en torno al que deben estar los números.

Supuso que números enteros que representaban número de ceros hasta el primer uno se empezasen a interpretar como una tolerancia absoluta. Por ejemplo, donde antes había un 6 ahora debía escribirse un $1e - 6 = 10^{-6}$.

Existe el riesgo de que estos fallos hubiesen pasado por alto problemas de implementación.

4.2.6.1. Resultado

El cambio se incluyó con éxito, sin diagnosticar ningún fallo derivado de esta errata, garantizando una vez más la integridad y calidad de la librería.

4.2.7. Arreglo de un bug que ignoraba parámetros de una función de lectura de bases de datos

- *Issue:* #2018
- *Pull Request:* #2020

En *pvlib-python* existe la función `pvlib.pvsystem.retrieve_sam` que permite leer de bases de datos ya inclusas en la librería o remotas mediante una URL, mediante dos parámetros distintos. Resulta que si se especificaban ambos parámetros, solo se leía la base de datos de la distribución, y no la remota, sin emitir ningún aviso.

4.2.7.1. Resultado

Se aplicó un patrón de diseño que excluía la posibilidad de que ambos parámetros se especificasen a la vez, y se incluyó sin dificultades.

4.2.8. Actualizar versiones de las dependencias de la documentación

- *Pull Request:* #2112

Esta contribución se trata de una iniciativa propia dentro de este TFG. Se trata de actualizar las versiones de las dependencias de la documentación, que se construye automáticamente con *Sphinx* y algunas extensiones extras, para dotar de un nuevo *look-and-feel* (o, estilo) a la documentación.

Se prioriza que el renovado entorno pueda pasar el mayor tiempo posible sin generar problemas ni requerir atención.

4.2.8.1. Resultado

Entre las características más destacables de la nueva documentación está el atajo de teclado para realizar búsquedas; tema gráfico para cambiar entre unos colores claros y otros oscuros según la preferencia del lector, y una ubicación un poco más estética de algunos elementos.

Se han aceptado los cambios, que serán visibles en la rama estable de la documentación a partir de la próxima versión 0.11.1.

4.3. Contribuciones menores

Aquí se tratan aquellas que no suponen un desafío ni técnico ni científico, si bien suponen y parten de la interacción en el repositorio y la identificación particular de necesidades. En especial, son cambios que únicamente modifican la documentación o no requieren de un análisis profundo.

4.3.1. Añadir una utilidad para obtener los ficheros de ejemplo internos de la librería

- *Issue:* #924
- *Pull Request:* #1763

El objetivo consistió en añadir una función que ahorrarse escribir la introspección de la ubicación de la librería para formar las rutas manualmente.

Listing 4.2: Fragmento de código utilizado normalmente para obtener la ruta de los ficheros de la librería

```
1 import os
2 import pvlib
3
4 PVLIB_DIR = pvlib.__path__[0]
5 DATA_FILE = os.path.join(PVLIB_DIR, 'data', '723170TYA.CSV')
```

Concretamente, la propuesta crea una función que se utilizaría como sigue:

Listing 4.3: Fragmento de código utilizando la función propuesta para obtener la ruta de los ficheros de la librería

```
1 import pvlib
2
3 DATA_FILE = pvlib.tools.get_example_dataset_path('723170TYA.CSV')
```

Se desestima la propuesta porque se considera que primero deben reorganizarse todos los ficheros de ejemplo, si bien esto último nunca se ha puesto en marcha debido a la inmensidad de la tarea.

4.3.2. Corrección de erratas en la documentación

- *Pull Request:* #1599
- *Pull Request:* #1860
- *Pull Request:* #1996

Se han solventado erratas en la documentación, encontradas mayoritariamente de casualidad al navegar entre el código con otros fines.

4.3.3. Corrección de erratas en ejemplos y en código

- *Pull Request:* #1776
- *Pull Request:* #1833

Son pequeñas modificaciones que no requieren mayor explicación: actualizar la salida desfasada de un ejemplo y comprobar que un único script de ejemplo que no se ejecuta automáticamente fuera capaz de hacerlo localmente.

4.3.4. Modificación de escritura de los parámetros opcionales

- *Issue:* #1574
- *Pull Request:* #1828
- *Pull Request:* #2084

En Python existe un patrón de diseño que consiste en darle el valor `None` a los parámetrosopcionales, y luego comprobar si efectivamente son `None` con el fin de asignarles un valor por defecto o ignorarlos si fuese el caso.

Normalmente, a la hora de documentarlo lo recomendable es indicar que el parámetro es opcional, pero lo habitual hasta antes de esta contribución era indicar que el valor por defecto es `None`, lo cual es redundante y obfusca las intenciones del código.

Se cambiaron todas las ocurrencias de `default None` a `optional`, aunque hubo que iterar múltiples veces para dar con todos los falsos negativos y no obtener falsos positivos. Para esta tarea se empleó el buscador en archivos integrado en *Visual Studio Code*, que está basado en *ripgrep* y hace uso de expresiones regulares, una forma de búsqueda y sustitución muy potente: las expresiones regulares son un lenguaje formal que permite buscar patrones en texto, por ejemplo sustituyendo un carácter por grupos de caracteres, o buscando un patrón que se repite un número determinado de veces.

Se emplearon múltiples expresiones regulares, escritas en el sabor PCRE2, que es el que soporta *ripgrep*. Según la utilidad, se pueden encontrar pequeñas variaciones en la forma de escribir estas expresiones. En los *commits* de la propuesta se indican todas las expresiones empleadas, pero aquí se adjuntan algunas de las usadas:

Expresión regular	Patrón de sustitución
(? {8} {4}) (?\\w*) ?: (?.*), default: ? None\\.?	\$1\$2 : \$3, optional
(? {8} {4}) (?\\w*) ?: (?.*), default: ? None?	
(? {8} {4}) (?\\w*) ?: (?.*) or [nN]one, optional	
(? {8} {4}) (?\\w*) ?: [Nn]one, (?.*), default: ? (?.*)\\.?	\$1\$2 : \$3, default \$4
(? {8} {4}) (?\\w*) ?: (?.*) or [Nn]one(?.*)	\$1\$2 : \$3\$4
If None	If not specified,

Tabla 4.1: Algunas de las expresiones regulares empleadas para modificar la escritura de los parámetros opcionales.

Adicionalmente se arreglaron algunos links a los *DOI*, que son identificadores únicos de documentos científicos, gracias también a la búsqueda y sustitución con expresiones regulares:

- Expresión de búsqueda: `(?:doi|DOI):(?!`)\s?(.*?)(\.\\n|\\n)`
- Expresión de sustitución: `:doi:`$1`$2`

4.3.5. Limpieza de advertencias al construir la documentación

- [Pull Request: #2030](#)
- [Pull Request: #2128](#)

De nuevo, en la línea de eliminar advertencias en los flujos de integración continua, se eliminaron algunas que se emitían al construir la documentación. Estas advertencias no eran críticas, pero al formar parte de unos registros que se consultan con cierta periodicidad cada vez que alguien hace contribuciones, es deseable evitar ruido visual. De esta forma, el proceso de revisión es más eficiente y no confunde ni a los revisores ni a los contribuyentes.

Además, se mejoró así el renderizado de la documentación en bastantes casos.

4.3.6. Modificar documentación de parámetros para poder ejecutar procedimientos que verifican la integridad de la librería

- [Pull Request: #1790](#)

Se identifica la falta de un sistema que compruebe que la documentación realmente está bien formateada, así que se plantea una propuesta para automatizar esta comprobación en forma de un procedimiento.

Sin embargo, por el hecho de introducir nuevas herramientas que no eran del todo compatibles y requerir muchas modificaciones a la base de código actual, se desestimó la propuesta relativamente temprano.

4.3.7. Portar antiguos ejemplos de Jupyter a scripts integrados en la web

- [Pull Request: #1818](#)

En los inicios del repositorio, los ejemplos se redactaban en *Jupyter Notebooks*, que son documentos interactivos para ejecutar código. Sin embargo, no se facilita su acceso desde la página web, y por la complejidad que suponen, no se ejecutan cada vez que se hacen comprobaciones al repositorio. Por ende, quedaron obsoletos pronto.

En esta propuesta se extrajeron todos los comentarios de texto y los fragmentos de código y se estilizan para que se puedan ejecutar y ver en la página web. Se desestima por tratarse de ejemplos obsoletos, con poco valor didáctico y utilizable. A día de hoy, se pueden encontrar en la carpeta `docs/tutorials`.

4.3.8. Integración continua para verificar los links externos de la documentación

- [Pull Request: #1794](#)

Otra de las mejoras que se identifica como necesaria a lo largo de este proyecto es comprobar que los links externos de la documentación funcionan correctamente. Resulta que el entorno de desarrollo de la documentación *sphinx* permite realizar esto periódicamente de forma sencilla, pero debe configurarse los comandos para crear el procedimiento de integración continua.

Las opiniones de los mantenedores quedan divididas respecto a este tema: unos muestran interés en incluirlo, otros prefieren añadir pasos manuales para hacer la verificación aunque suponga un costo en tiempo y asigne la responsabilidad a una sola persona.

Adicionalmente, se ayuda a un mantenedor a ejecutar los tests y así actualizar los links que se encuentran rotos.

A día de la redacción de este documento, no se ha tomado una decisión definitiva, pero se mantiene abierta la propuesta.

4.3.9. Eliminar una función obsoleta y olvidada

- *Pull Request:* #2131

La ubicación de la función `pvlib.atmosphere.first_solar_spectral_correction`³ se encontraba obsoleta tras plantear una reorganización de aquellos modelos relativos a correcciones espectrales. Se hubo avisado de su obsoleta ubicación en la versión 0.10.0, pero no se eliminó en la siguiente versión menor, 0.11.0, que son aquellas que admiten cambios que pueden romper el código de los usuarios.

Se plantea eliminar la presencia de dicha interfaz que no tiene fecha para su desaparición, pero lo más probable es que hasta una versión menor posterior no se elimine ya que supone cambios que pueden romper el código de usuarios. La siguiente es la versión menor 0.12.0, aunque la versión de arreglos 0.11.1 se encuentra en desarrollo y se espera que se publique en Septiembre.

La propuesta se mantiene abierta a día de esta redacción.

³Este link podría no funcionar si finalmente se elimina.

Capítulo 5

Resultados y conclusiones

En este capítulo se resumen los resultados obtenidos en el desarrollo del Trabajo Fin de Grado, aquellos tanto satisfactorios como los que no, y el impacto que ha tenido la participación en *pplib-python*. Además, se proponen posibles líneas de trabajo futuro, algunas que seguirán desarrollándose y otras que se podrían seguir a partir de este TFG.

5.1. Resultados

A pesar de que los ciclos de revisiones y correcciones toman un elevado tiempo de desarrollo, se han conseguido implementar varios modelos y funcionalidades nuevos en *pplib-python*. Además, se ha mejorado la documentación de la librería y se han corregido errores en la misma. Cabe destacar:

1. Exponer el cálculo de la proyección del cenit solar sobre las coordenadas de un colector.
2. Implementar un cálculo de la fracción de sombra unidimensional.
3. La implementación de un modelo de pérdidas de potencia por sombreado, según número de diodos de bypass.
4. Un modelo de fracción de irradiancia difusa fotosintetizable, en función de la irradiancia difusa global.
5. Un modelo de pérdidas de potencia por heterogeneidad en la irradiancia incidente, de interés para módulos bifaciales.
6. Dos funciones de conversión recíprocas de responsividad espectral y eficiencia cuántica externa, con posibilidad de normalizar la salida.
7. Adición de una función general para obtener espectros estándares, que cuenta con el estándar ASTM G173-03, pero que será muy fácilmente extendible.
8. Solventar 5 bugs y 2 características nuevas de las que han informado usuarios varios.
9. Mejorar la documentación ya existente de forma amplia.

10. Participar en detectar posibles contribuciones para primeros contribuyentes en el futuro.
11. Aportar ideas y planteamientos que faciliten contribuir a la librería.

Algunas cifras significativas, para la librería *pplib-python* son:

- Se han añadido 4653 líneas de código y documentación, y otras 2004 de datos. Solo se han sustituido 771 líneas de código y documentación, y 2003 de un archivo de datos.
- Se ha abierto 34 *pull requests*¹.
 - De las cuales 23 se han aceptado.
 - De las cuales 5 se han desestimado.
 - De las cuales 6 se mantienen abiertas.
- Se ha revisado 18 *pull requests* de otros usuarios².
- Se ha abierto 19 *issues*³.
 - De las cuales se han cerrado 11.
- Se ha participado en 28 *issues* de otros usuarios⁴.
 - De las cuales se han cerrado 13.
- Se han creado 5 ejemplos completamente nuevos.

Asimismo este proyecto ha tenido impacto en otros proyectos, como se detalla en la sección 5.2.2.

5.2. Impacto

En esta sección se analizará el valor que aporta el trabajo realizado en este TFG.

El impacto se divide en dos partes: la primera sección se centrará en el impacto general del trabajo, mientras que la segunda sección analizará el impacto específico en relación con los Objetivos de Desarrollo Sostenible (ODS).

5.2.1. Impacto general

Este Trabajo Fin de Grado permite visibilizar y extender el uso de algunos modelos científicos relacionados con la fotovoltaica. Además, se ha facilitado el mantenimiento de la librería *pplib-python* y se ha mejorado la documentación de la misma, contribuyendo así a la persistencia de este proyecto. Incluso sin haberse añadido

¹Consultar última información en <https://github.com/pplib/pplib-python/pulls?q=is%3Apr+author%3Aechedey-ls+>.

²Consultar última información en <https://github.com/pplib/pplib-python/pulls?q=is%3Apr+reviewed-by%3Aechedey-ls+-author%3Aechedey-ls+>.

³Consultar última información en <https://github.com/pplib/pplib-python/issues?q=is%3Aissue+author%3Aechedey-ls+>.

⁴Consultar última información en <https://github.com/pplib/pplib-python/issues?q=is%3Aissue+-author%3Aechedey-ls+involves%3Aechedey-ls+>.

Resultados y conclusiones

algunas de las contribuciones propuestas, el trabajo realizado ha permitido reforzar líneas de trabajo que antes ni se consideraban.

Se espera que algunos de los modelos sean bastante útiles para la comunidad científica y técnica, en especial aquellos que consisten en modelos de pérdidas. Estos modelos permiten optimizar el diseño de plantas y mejorar el rendimiento económico de las instalaciones. Además, la mejora de la documentación de la librería *pplib-python* facilita su uso y extensión, lo que puede llevar a un aumento de la comunidad de usuarios y contribuidores.

Dentro del programa subvencionado *Google Summer of Code*, se ha trabajado estrechamente con otros compañeros desarrolladores y se les ha ayudado a que se familiaricen con las herramientas de desarrollo y la forma de colaborar en el proyecto. Esto ha permitido que se hayan añadido nuevas funcionalidades a la librería y se haya mejorado la calidad del código.

Asimismo, este trabajo se alinea con los objetivos de la Universidad pública de promover la investigación, la transferencia y democratización libre del conocimiento.

5.2.2. Impacto en otros proyectos

▪ En *pplib/solarfactors*:

Este repositorio se trata de un clon de *pvfactors*, que se emplea para calcular factores de vista de sistemas fotovoltaicos bifaciales. Es un clon porque *pvfactors* inició dentro de una empresa, que contaba con los permisos para administrar sus cambios, pero dejó de mantenerla. La comunidad de *pplib*, ya que la usaba en el repositorio *pplib-python*, decidió clonarla y mantenerla.

- *Issue: #16*[<https://github.com/pplib/solarfactors>]

Se detecta y se informa sobre un problema de las dependencias de *solarfactors* en *Python 3.12*, y se ha propuesto eliminar la dependencia de *Shapely* en favor de código nativo de Python. Se ha iniciado el trabajo, pero no se ha completado por dificultad técnica y falta de vinculación con los objetivos de este trabajo.

▪ En *openpvtools/pv-foss-engagement*:

Este pequeño proyecto es un compendio de librerías de código abierto de fotovoltaica, con sus datos de contribuciones, contribuyentes y otras métricas relativas al impacto e interés que generan.

Se puede consultar en <https://openpvtools.github.io/pv-foss-engagement/>.

- *Issue: #8*[<https://github.com/openpvtools/pv-foss-engagement>]
- *Pull Request: #9*[<https://github.com/openpvtools/pv-foss-engagement>]

Se detecta y se informa sobre la ausencia del clon citado anteriormente, *pplib-solarfactors*, en este repositorio. Un mantenedor añade las estadísticas para que posteriormente se hagan las modificaciones pertinentes para que se muestren en la web.

Dentro de este TFG se realiza el archivo y las modificaciones necesarias para poder ver las estadísticas de *solarfactors* en la web. Se incluyen los datos a fecha de esta redacción.

5.2.3. Objetivos de Desarrollo Sostenible

Con este trabajo se ha contribuido a la consecución de los Objetivos de Desarrollo Sostenible (ODS):

- **ODS 3: Salud y bienestar.** Al potenciar el uso de energías renovables, se contribuye a la reducción de la contaminación y, por tanto, a la mejora de la salubridad del medio ambiente.
- **ODS 7: Energía asequible y no contaminante.** Debido a que se facilitan herramientas de diseño y análisis de instalaciones fotovoltaicas, se contribuye a la mejora de la eficiencia de las mismas y, por tanto, a la reducción del impacto de instalar paneles solares.
- **ODS 9: Industria, innovación e infraestructura.** Se ha trabajado en la mejora de la infraestructura de la librería *pvlib-python*, lo que facilita la innovación y desarrollo en el sector de la energía fotovoltaica.
- **ODS 11: Ciudades y comunidades sostenibles.** Las mejoras facilitadas permiten implementar instalaciones fotovoltaicas de forma más fiable, lo que contribuye a su implantación.
- **ODS 13: Acción por el clima.** Este ODS sigue en la misma línea que los demás: como fuente de energía renovable, y en especial por ser la fotovoltaica, se reduce la emisión de gases de efecto invernadero y se contribuye a la lucha contra el cambio climático.

5.3. Conclusiones

Con la realización de este Trabajo Fin de Grado, se promueven y mejoran proyectos de código abierto ya establecidos, lo que garantiza la usabilidad de las aportaciones aceptadas.

Se han mejorado múltiples aspectos de la librería *pvlib-python*, desde errores menores en el código a la implementación de nuevos modelos y funcionalidades. Se ha mejorado la documentación, dotando al proyecto de una mayor usabilidad. El aporte que realiza este TFG es enriquece transversalmente a todos los efectos de este repositorio y se espera que sea de utilidad para la comunidad científica y técnica. Se espera que además se facilite el acceso de nuevos contribuyentes a la librería.

Desde un punto de vista de los logros, es destacable el impacto que este proyecto tiene iniciativas de código abierto. Por otro lado, hay dos dificultades principales que cabría destacar:

- La exploración de artículos con modelos candidatos, a pesar de partir de una tanda inicial, encontrar otras posibles contribuciones adicionales supone un elevado esfuerzo y un resultado reducido.
- La planificación del trabajo, que responde a ciclos de trabajo extensos por el aprendizaje paralelo al desarrollo y, en especial, al lanzamiento de versiones, que es el factor principal que motiva la revisión de las propuestas.

Asimismo cabe destacar que se han hecho pequeños gestos que han hecho del proyecto un lugar más inclusivo, como identificar el sesgo hacia los ejemplos de fotovoltaica

Resultados y conclusiones

exclusivamente orientada al sur, el recibimiento a nuevos contribuyentes o la resolución de algunas dudas.

5.4. Trabajo futuro

Hay varias líneas de desarrollo que sería deseable continuar en el futuro, algunas de las cuales se han planteado en este TFG:

1. Inicialmente se trabajará en contrastar la implementación del modelo descrito en [4.1.6](#), cuyos autores solicitan y facilitan información para contrastar la implementación con la suya.
2. Se continuarán las propuestas ya planteadas, tanto dentro como fuera del programa subvencionado *Google Summer of Code*. Y se priorizará dotar a la web de una mejor estética y usabilidad empleando las últimas versiones posibles.
3. Además, posiblemente se intente continuar en la línea de solventar el problema de la dependencia `shapely<2` en `solarfactors`, en función de si el equipo decide trabajar en ello.
4. Por último, se ayudará a mejorar la documentación sobre cómo contribuir, y plantear una hoja de ruta para futuros contribuyentes que no hayan visto cómo programar en esta librería antes, pues se identifica que no todos los contribuyentes reciben formación sobre el uso de control de versiones en sus titulaciones.

Se espera que este trabajo contribuya y motive el desarrollo de algunas áreas de estudio, así como que extienda y facilite la implantación de esta energía renovable en la sociedad.

Bibliografía

- [1] Christian Breyer et al. «On the role of solar photovoltaics in global energy transition scenarios». En: *Progress in Photovoltaics: Research and Applications* 25.8 (ago. de 2017), págs. 727-745. ISSN: 1062-7995. DOI: [10.1002/pip.2885](https://doi.org/10.1002/pip.2885).
- [2] Kevin S. Anderson et al. «pvlib-python: 2023 project update». En: *Journal of Open Source Software* 8.92 (2023), pág. 5994. DOI: [10.21105/joss.05994](https://doi.org/10.21105/joss.05994). URL: <https://doi.org/10.21105/joss.05994>.
- [3] Joshua S. Stein. «The photovoltaic Performance Modeling Collaborative (PVPMC)». En: *2012 38th IEEE Photovoltaic Specialists Conference*. Jun. de 2012, págs. 003048-003052. DOI: [10.1109/PVSC.2012.6318225](https://doi.org/10.1109/PVSC.2012.6318225). URL: <https://ieeexplore.ieee.org/document/6318225>.
- [4] Robert W. Andrews et al. «Introduction to the open source PV LIB for python Photovoltaic system modelling package». En: *2014 IEEE 40th Photovoltaic Specialist Conference (PVSC)*. Jun. de 2014, págs. 0170-0174. DOI: [10.1109/PVSC.2014.6925501](https://doi.org/10.1109/PVSC.2014.6925501). URL: <https://ieeexplore.ieee.org/document/6925501>.
- [5] William F. Holmgren et al. «pvlib-python 2015». En: *2015 IEEE 42nd Photovoltaic Specialist Conference (PVSC)*. Jun. de 2015, págs. 1-5. DOI: [10.1109/PVSC.2015.7356005](https://doi.org/10.1109/PVSC.2015.7356005). URL: <https://ieeexplore.ieee.org/document/7356005>.
- [6] William F. Holmgren y Derek G. Groenendyk. «An open source solar power forecasting tool using PVLIB-Python». En: *2016 IEEE 43rd Photovoltaic Specialists Conference (PVSC)*. Jun. de 2016, págs. 0972-0975. DOI: [10.1109/PVSC.2016.7749755](https://doi.org/10.1109/PVSC.2016.7749755). URL: <https://ieeexplore.ieee.org/document/7749755>.
- [7] K.W. Böer y D. Bimberg. *Survey of Semiconductor Physics, Survey of Semiconductor Physics*. Survey of Semiconductor Physics. Wiley, 2002. ISBN: 9780471355724. URL: https://books.google.es/books?id=_pdvAQAAQAAJ.
- [8] Mahmood H. Shubbak. «Advances in solar photovoltaics: Technology review and patent trends». En: *Renewable and Sustainable Energy Reviews* 115 (nov. de 2019), pág. 109383. ISSN: 1364-0321. DOI: [10.1016/j.rser.2019.109383](https://doi.org/10.1016/j.rser.2019.109383).
- [9] William Shockley y Hans J. Queisser. «Detailed Balance Limit of Efficiency of p-n Junction Solar Cells». En: *Journal of Applied Physics* 32.3 (mar. de 1961), págs. 510-519. ISSN: 0021-8979. DOI: [10.1063/1.1736034](https://doi.org/10.1063/1.1736034).
- [10] Martin A. Green et al. «Solar cell efficiency tables (Version 63)». en. En: *Progress in Photovoltaics: Research and Applications* 32.1 (2024), págs. 3-13. ISSN: 1099-159X. DOI: [10.1002/pip.3750](https://doi.org/10.1002/pip.3750).
- [11] Standard Tables for Reference Solar Spectral Irradiances: Direct Normal and Hemispherical on 37° Tilted Surface — astm.org. <https://www.astm.org/g0173-03.html>. 2003.

BIBLIOGRAFÍA

- [12] Richard Perez et al. «Modeling daylight availability and irradiance components from direct and global irradiance». En: *Solar Energy* 44.5 (1990), págs. 271-289. ISSN: 0038-092X. DOI: [https://doi.org/10.1016/0038-092X\(90\)90055-H](https://doi.org/10.1016/0038-092X(90)90055-H).
- [13] Anton Driesse, Adam R. Jensen y Richard Perez. «A continuous form of the Perez diffuse sky model for forward and reverse transposition». En: *Solar Energy* 267 (2024), pág. 112093. ISSN: 0038-092X. DOI: <https://doi.org/10.1016/j.solener.2023.112093>.
- [14] O. Perpiñán. *Energía Solar Fotovoltaica*. 2020. URL: <http://oscarperpinan.github.io/esf/>.
- [15] M. Victoria. *Fundamentals of Solar Cells and Photovoltaic Systems Engineering*. Elsevier Science, 2024. ISBN: 9780323996761. URL: <https://books.google.dk/books?id=OCfqEAAAQBAJ>.
- [16] Kevin Anderson y Mark Mikofski. *Slope-Aware Backtracking for Single-Axis Trackers*. English. NREL/TP-5K00-76626. Jul. de 2020. DOI: <10.2172/1660126>. URL: <https://www.osti.gov/biblio/1660126>.
- [17] Marco Rosa-Clot y Giuseppe Marco Tina. «Chapter 10 - Levelized Cost of Energy (LCOE) Analysis». En: *Floating PV Plants*. Ed. por Marco Rosa-Clot y Giuseppe Marco Tina. Academic Press, ene. de 2020, págs. 119-127. ISBN: 978-0-12-817061-8. DOI: <10.1016/B978-0-12-817061-8.00010-5>. URL: <https://www.sciencedirect.com/science/article/pii/B9780128170618000105>.
- [18] Joshua S Stein y Geoffrey T Klise. «Models used to assess the performance of photovoltaic systems.» En: (dic. de 2009). DOI: <10.2172/974415>. URL: <https://www.osti.gov/biblio/974415>.
- [19] Nallapaneni Manoj Kumar. «Simulation Tools for Technical Sizing and Analysis of Solar PV Systems». En: *Proceedings of the 6th World Conference on Applied Sciences, Engineering and Technology (WCSET-2017), 26-27 August 2017, UMP-PO, Indonesia, ISBN 13: 978-81-930222-3-8, pp 218-222, At Universitas Muhammadiyah Ponorogo, Indonesia*. (ene. de 2017). URL: https://www.academia.edu/35141273/Simulation_Tools_for_Technical_Sizing_and_Analysis_of_Solar_PV_Systems.
- [20] C. DiBona y S. Ockman. *Open Sources: Voices from the Open Source Revolution*. O'Reilly Media, 1999. ISBN: 9780596553906. URL: <https://books.google.es/books?id=bjMsCKvV9I4C>.
- [21] Brian Fitzgerald. «The Transformation of Open Source Software». En: *MIS Quarterly* 30.3 (2006), págs. 587-598. ISSN: 02767783. URL: <http://www.jstor.org/stable/25148740> (visitado 22-07-2024).
- [22] Mamdouh Alenezi e Ibrahim Abunadi. «Quality of Open Source Systems from Product Metrics Perspective». En: arXiv:1511.03194 (nov. de 2015). arXiv:1511.03194 [cs]. DOI: <10.48550/arXiv.1511.03194>. URL: <http://arxiv.org/abs/1511.03194>.
- [23] Philip Guo. «Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia». En: *The 34th Annual ACM Symposium on User Interface Software and Technology*. UIST '21. Virtual Event, USA: Association for Computing Machinery, 2021, págs. 1235-1251. ISBN: 9781450386357. DOI: <10.1145/3472749.3474819>. URL: <https://doi.org/10.1145/3472749.3474819>.
- [24] G. van Rossum. *Python tutorial*. Inf. téc. CS-R9526. Amsterdam: Centrum voor Wiskunde en Informatica (CWI), mayo de 1995.

- [25] Aaron Imani et al. «Does Documentation Matter? An Empirical Study of Practitioners' Perspective on Open-Source Software Adoption». En: arXiv:2403.03819 (mar. de 2024). arXiv:2403.03819 [cs]. DOI: [10.48550/arXiv.2403.03819](https://doi.org/10.48550/arXiv.2403.03819). URL: <http://arxiv.org/abs/2403.03819>.
- [26] Nuria Martín Chivelet. «Estudio de la influencia de la reflexión, el ángulo de incidencia y la distribución espectral de la radiación solar en los generadores fotovoltaicos». PhD Thesis. 1999.
- [27] N. Martín y J. M. Ruiz. «A new method for the spectral characterisation of PV modules». En: *Progress in Photovoltaics: Research and Applications* 7.4 (1999), págs. 299-310. ISSN: 1099-159X. DOI: [10.1002/\(SICI\)1099-159X\(199907/08\)7:4<299::AID-PIP260>3.0.CO;2-0](https://doi.org/10.1002/(SICI)1099-159X(199907/08)7:4<299::AID-PIP260>3.0.CO;2-0).
- [28] Mitchell Lee y Alex Panchula. «Spectral correction for photovoltaic module performance based on air mass and precipitable water». En: *2016 IEEE 43rd Photovoltaic Specialists Conference (PVSC)*. Jun. de 2016, págs. 1351-1356. DOI: [10.1109/PVSC.2016.7749836](https://doi.org/10.1109/PVSC.2016.7749836). URL: <https://ieeexplore.ieee.org/abstract/document/7749836>.
- [29] J. A. Caballero et al. «Spectral Corrections Based on Air Mass, Aerosol Optical Depth, and Precipitable Water for PV Performance Modeling». En: *IEEE Journal of Photovoltaics* 8.2 (mar. de 2018), págs. 552-558. ISSN: 2156-3403. DOI: [10.1109/JPHOTOV.2017.2787019](https://doi.org/10.1109/JPHOTOV.2017.2787019).
- [30] E. Lorenzo, L. Narvarte y J. Muñoz. «Tracking and back-tracking». En: *Progress in Photovoltaics: Research and Applications* 19.6 (2011), págs. 747-753. ISSN: 1099-159X. DOI: [10.1002/pip.1085](https://doi.org/10.1002/pip.1085).
- [31] Kevin S. Anderson y Adam R. Jensen. «Shaded fraction and backtracking in single-axis trackers on rolling terrain». En: *Journal of Renewable and Sustainable Energy* 16.2 (abr. de 2024), pág. 023504. ISSN: 1941-7012. DOI: [10.1063/5.0202220](https://doi.org/10.1063/5.0202220).
- [32] F. Martínez-Moreno, J. Muñoz y E. Lorenzo. «Experimental model to estimate shading losses on PV arrays». En: *Solar Energy Materials and Solar Cells* 94.12 (dic. de 2010), págs. 2298-2303. ISSN: 0927-0248. DOI: [10.1016/j.solmat.2010.07.029](https://doi.org/10.1016/j.solmat.2010.07.029).
- [33] C. J. T. Spitters, H. A. J. M. Toussaint y J. Goudriaan. «Separating the diffuse and direct component of global radiation and its implications for modeling canopy photosynthesis Part I. Components of incoming radiation». En: *Agricultural and Forest Meteorology* 38.1 (oct. de 1986), págs. 217-229. ISSN: 0168-1923. DOI: [10.1016/0168-1923\(86\)90060-2](https://doi.org/10.1016/0168-1923(86)90060-2).
- [34] C. J. T. Spitters. «Separating the diffuse and direct component of global radiation and its implications for modeling canopy photosynthesis Part II. Calculation of canopy photosynthesis». En: *Agricultural and Forest Meteorology* 38.1 (oct. de 1986), págs. 231-242. ISSN: 0168-1923. DOI: [10.1016/0168-1923\(86\)90061-4](https://doi.org/10.1016/0168-1923(86)90061-4).
- [35] S. Ma Lu et al. «Photosynthetically active radiation decomposition models for agrivoltaic systems applications». En: *Solar Energy* 244 (sep. de 2022), págs. 536-549. ISSN: 0038-092X. DOI: [10.1016/j.solener.2022.05.046](https://doi.org/10.1016/j.solener.2022.05.046).
- [36] Chris Deline et al. «Estimating and parameterizing mismatch power loss in bifacial photovoltaic systems». En: *Progress in Photovoltaics: Research and Applications* 28.7 (2020), págs. 691-703. ISSN: 1099-159X. DOI: [10.1002/pip.3259](https://doi.org/10.1002/pip.3259).
- [37] URL: <https://www.semanticscholar.org/paper/Gini%E2%80%99s-Mean-difference%3A-a-superior-measure-of-for-Yitzhaki/e4d00851cbbadfc386ed051397c>

BIBLIOGRAFÍA

- [38] Tom Markvart y Luis Castañer. «Principles of solar cell operation». En: *Practical Handbook of Photovoltaics*. Elsevier, 2012, págs. 7-31.
- [39] *Standard Tables for Reference Solar Spectral Irradiances: Direct Normal and Hemispherical on 37° Tilted Surface — astm.org*. <https://www.astm.org/standards/g173>. 2023.
- [40] Sebastian Zainali et al. «Direct and diffuse shading factors modelling for the most representative agrivoltaic system layouts». En: *Applied Energy* 339 (jun. de 2023), pág. 120981. ISSN: 0306-2619. DOI: [10.1016/j.apenergy.2023.120981](https://doi.org/10.1016/j.apenergy.2023.120981).

Apéndice A

Anexo A: documentación de las funciones

En este anexo se puede encontrar la documentación generada para las nuevas funciones añadidas a la librería *pvlib-python*. Se incluye en forma de imágenes, solo el cuerpo con el contenido relevante, para poder apreciar el resultado que vería un usuario final.

Si se desea ver el código fuente de la documentación, en el caso de las propuestas aceptadas basta con acceder al link de la función y pinchar en el botón *[source]*. Para el caso de las no aceptadas, habría que acceder a la rama de trabajo de la propuesta mediante la *pull request* correspondiente.

A.1. Modelo de ajuste espectral

Propuesta en *Pull Request*: #1658, no aceptada.

pvlib.spectrum.martin_ruiz

`pvlib.spectrum.martin_ruiz(clearness_index, airmass_absolute,
module_type=None, model_parameters=None)` [\[source\]](#)

Calculate spectral mismatch modifiers for POA direct, sky diffuse and ground diffuse irradiances using the clearness index and the absolute airmass.

⚠ Warning

Included model parameters for `monosi`, `polysi` and `asi` were estimated using the airmass model `kasten1966` [1]. The same airmass model *must* be used to calculate the airmass input values to this function in order to not introduce errors. See [get_relative_airmass\(\)](#).

Parameters:

- `clearness_index (numeric)` – Clearness index of the sky.
- `airmass_absolute (numeric)` – Absolute airmass. `kasten1966` airmass algorithm must be used for default parameters of `monosi`, `polysi` and `asi`, see [1].
- `module_type (string, optional)` – Specifies material of the cell in order to infer model parameters. Allowed types are `monosi`, `polysi` and `asi`, either lower or upper case. If not specified, `model_parameters` has to be provided.
- `model_parameters (dict-like, optional)` –

Provide either a dict or a `pd.DataFrame` as follows:

```
# Using a dict  
# Return keys are the same as specifying 'module_type'  
model_parameters = {  
    'poa_direct': {'c': c1, 'a': a1, 'b': b1},  
    'poa_sky_diffuse': {'c': c2, 'a': a2, 'b': b2},  
    'poa_ground_diffuse': {'c': c3, 'a': a3, 'b': b3}  
}  
# Using a pd.DataFrame  
model_parameters = pd.DataFrame({  
    'poa_direct': [c1, a1, b1],  
    'poa_sky_diffuse': [c2, a2, b2],  
    'poa_ground_diffuse': [c3, a3, b3]},  
    index=('c', 'a', 'b'))
```

`c`, `a` and `b` must be scalar.

Unspecified parameters for an irradiance component ('`poa_direct`', '`poa_sky_diffuse`', or '`poa_ground_diffuse`') will cause `np.nan` to be returned in the corresponding result.

Returns:

Modifiers (`pd.DataFrame` (iterable input) or `dict` (scalar input) of numeric) – Mismatch modifiers for direct, sky diffuse and ground diffuse irradiances, with indexes '`poa_direct`', '`poa_sky_diffuse`', '`poa_ground_diffuse`'. Each mismatch modifier should be multiplied by its corresponding POA component.

Raises:

- `ValueError` – If `model_parameters` is not suitable. See examples given above.
- `ValueError` – If neither `module_type` nor `model_parameters` are given.
- `ValueError` – If both `module_type` and `model_parameters` are provided.
- `NotImplementedError` – If `module_type` is not found in internal table of parameters.

Notes

The mismatch modifier is defined as

$$M = c \cdot \exp(a \cdot (K_t - 0.74) + b \cdot (AM - 1.5))$$

where c , a and b are the model parameters, different for each irradiance component.

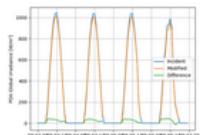
References

[1]([1.2](#)) Martín, N. and Ruiz, J.M. (1999), A new method for the spectral characterisation of PV modules. Prog. Photovolt: Res. Appl., 7: 299-310. [DOI: 10.1002/\(SICI\)1099-159X\(199907/08\)7:4<299::AID-PIP260>3.0.CO;2-0](#)

See also

`pvlib.irradiance.clearness_index`, `pvlib.atmosphere.get_relative_irmass`,
`pvlib.atmosphere.get_absolute_irmass`, `pvlib.atmosphere.first_solar`

Examples using `pvlib.spectrum.martin_ruiz`



N. Martin & J. M.
Ruiz Spectral
Mismatch Modifier

A.2. Proyección del cenit solar

Pull Request: #1904, accesible en `pvlib.shading.projected_solar zenith_angle`.

`pvlib.shading.projected_solar zenith_ang`

```
pvlib.shading.projected_solar zenith_angle(solar zenith, solar azimuth,
axis tilt, axis azimuth)
```

[\[source\]](#)

Calculate projected solar zenith angle in degrees.

This solar zenith angle is projected onto the plane whose normal vector is defined by `axis_tilt` and `axis_azimuth`. The normal vector is in the direction of `axis_azimuth` (clockwise from north) and tilted from horizontal by `axis_tilt`. See Figure 5 in [1]:

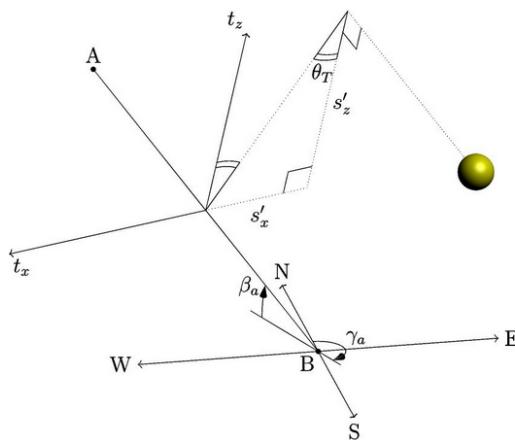


Fig. 5, [1]: Solar coordinates projection onto tracker rotation plane.

Parameters:

- `solar zenith` (*numeric*) – Sun's apparent zenith in degrees.
- `solar azimuth` (*numeric*) – Sun's azimuth in degrees.
- `axis_tilt` (*numeric*) – Axis tilt angle in degrees. From horizontal plane to array plane.
- `axis_azimuth` (*numeric*) – Axis azimuth angle in degrees. North = 0°; East = 90°; South = 180°; West = 270°

Returns:

`Projected_solar zenith` (*numeric*) – In degrees.

Notes

This projection has a variety of applications in PV. For example:

- Projecting the sun's position onto the plane perpendicular to the axis of a single-axis tracker (i.e. the plane whose normal vector coincides with the tracker torque tube) yields the tracker rotation angle that maximizes direct irradiance capture. This tracking strategy is called *true-tracking*. Learn more about tracking in [Single-axis tracking](#).
- Self-shading in large PV arrays is often modeled by assuming a simplified 2-D array geometry where the sun's position is projected onto the plane perpendicular to the PV rows. The projected zenith angle is then used for calculations regarding row-to-row shading.

Examples

Calculate the ideal true-tracking angle for a horizontal north-south single-axis tracker:

```
>>> rotation = projected_solar zenith_angle(solar zenith, solar azimuth,  
>>>                                axis_tilt=0, axis_azimuth=180)      >>>
```

Calculate the projected zenith angle in a south-facing fixed tilt array (note: the `axis_azimuth` of a fixed-tilt row points along the length of the row):

```
>>> psza = projected_solar zenith_angle(solar zenith, solar azimuth,  
>>>                                axis_tilt=0, axis_azimuth=90)      >>>
```

References

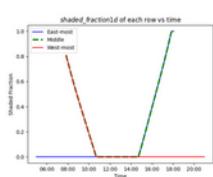
- [1]([1,2](#)) K. Anderson and M. Mikofski, 'Slope-Aware Backtracking for Single-Axis Trackers', National Renewable Energy Lab. (NREL), Golden, CO (United States); NREL/TP-5K00-76626, Jul. 2020. [DOI: 10.2172/1660126](#).

See also

[pvlib.solarposition.get_solarposition](#)

Examples using

[pvlib.shading.projected_solar zenith_angle](#)



Shaded fraction of a horizontal single-axis tracker

A.3. Fracción de sombra unidimensional

Pull Request: #1962, accesible en `pvlib.shading.shaded_fraction1d`.

pvlib.shading.shaded_fraction1d

```
pvlib.shading.shaded_fraction1d(solar zenith, solar azimuth, axis azimuth,  
shaded_row_rotation, *, collector_width, pitch, axis_tilt=0,  
surface_to_axis_offset=0, cross_axis_slope=0,  
shading_row_rotation=None)
```

[\[source\]](#)

Shaded fraction in the vertical dimension of tilted rows, or perpendicular to the axis of horizontal rows.

If `shading_row_rotation` isn't provided, it is assumed that both the shaded row and the shading row (the one blocking the direct beam) have the same rotation and azimuth values.

⚠ Warning

The function assumes that the roles of the shaded and shading rows remain the same during the day. In the case where the shading and shaded rows change throughout the day, e.g. a N-S single-axis tracker, the inputs must be switched depending on the sign of the projected solar zenith angle. See the Examples section below.

ⓘ Added in version 0.11.0.

Parameters:

- **solar_zenith** (*numeric*) – Solar zenith angle, in degrees.
- **solar_azimuth** (*numeric*) – Solar azimuth angle, in degrees.
- **axis_azimuth** (*numeric*) – Axis azimuth of the rotation axis of a tracker, in degrees. Fixed-tilt arrays can be considered as a particular case of a tracker. North=0°, South=180°, East=90°, West=270°.
- **shaded_row_rotation** (*numeric*) – Right-handed rotation of the row receiving the shade, with respect to `axis_azimuth`. In degrees °.
- **collector_width** (*numeric*) – Vertical length of a tilted row. The returned `shaded_fraction` is the ratio of the shadow over this value.
- **pitch** (*numeric*) – Axis-to-axis horizontal spacing of the row.
- **axis_tilt** (*numeric, default 0*) – Tilt of the rows axis from horizontal. In degrees °.
- **surface_to_axis_offset** (*numeric, default 0*) – Distance between the rotating axis and the collector surface. May be used to account for a torque tube offset.
- **cross_axis_slope** (*numeric, default 0*) – Angle of the plane containing the rows' axes from horizontal. Right-handed rotation with respect to the rows axes. In degrees °.
- **shading_row_rotation** (*numeric, optional*) – Right-handed rotation of the row casting the shadow, with respect to the row axis. In degrees °.

Returns:

shaded_fraction (*numeric*) – The fraction of the collector width shaded by an adjacent row. A value of 1 is completely shaded and 0 is no shade.

Notes

All length parameters must have the same units.

Parameters are defined as follow:

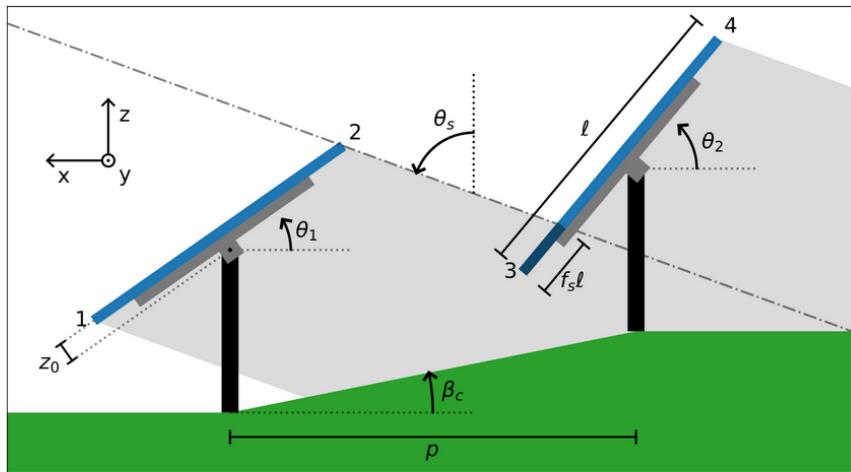


Figure 3 of [1]. See correspondence between this nomenclature and the function parameters in the table below.

Symbol	Parameter	Units
θ_1	shading_row_rotation	Degrees °
θ_2	shaded_row_rotation	
β_c	cross_axis_slope	
p	pitch	Any consistent length unit across all these parameters, e.g. m.
ℓ	collector_width	
z_0	surface_to_axis_offset	
f_s	Return value	Dimensionless

Examples

Fixed-tilt south-facing array on flat terrain

Tilted row with a pitch of 3 m, a collector width of 2 m, and row rotations of 30°. In the morning.

```
>>> shaded_fraction1d(solar zenith=80, solar azimuth=104.5,
...     axis azimuth=90, shaded row rotation=30, shading row rotation=30,
...     collector width=2, pitch=3, axis tilt=0,
...     surface to axis offset=0.05, cross axis slope=0)
0.6827437712114521
```

Anexo A: documentación de las funciones

Fixed-tilt north-facing array on sloped terrain

Tilted row with a pitch of 4 m, a collector width of 2.5 m, and row rotations of 50° for the shaded row and 30° for the shading row. The rows are on a 10° slope, where their axis is on the most inclined direction (zero cross-axis slope). Shaded in the morning.

```
>>> shaded_fraction1d(solar_zenith=65, solar_azimuth=75.5,
...     axis_azimuth=270, shaded_row_rotation=50, shading_row_rotation=30,
...     collector_width=2.5, pitch=4, axis_tilt=10,
...     surface_to_axis_offset=0.05, cross_axis_slope=0)
0.6975923460352351
```

>>>

N-S single-axis tracker on sloped terrain

Horizontal trackers with a pitch of 3 m, a collector width of 1.4 m, and tracker rotations of 30° pointing east, in the morning. Terrain slope is 7° west-east (east-most tracker is higher than the west-most tracker).

```
>>> shaded_fraction1d(solar_zenith=50, solar_azimuth=90, axis_azimuth=180,
...     shaded_row_rotation=-30, collector_width=1.4, pitch=3, axis_tilt=0,
...     surface_to_axis_offset=0.10, cross_axis_slope=7)
0.5828961460616938
```

>>>

Note the previous example only is valid for the shaded fraction of the west-most tracker in the morning, and assuming it is the shaded tracker during all the day is incorrect. During the afternoon, it is the one casting the shadow onto the east-most tracker.

To calculate the shaded fraction for the east-most tracker, you must input the corresponding `shaded_row_rotation` in the afternoon.

```
>>> shaded_fraction1d(solar_zenith=50, solar_azimuth=270, axis_azimuth=180,
...     shaded_row_rotation=30, collector_width=1.4, pitch=3, axis_tilt=0,
...     surface_to_axis_offset=0.10, cross_axis_slope=7)
0.4399034444363955
```

>>>

You must switch the input/output depending on the sign of the projected solar zenith angle. See [projected_solar_zenith_angle\(\)](#) and the example [Shaded fraction of a horizontal single-axis tracker](#).

See also

[pvlib.shading.projected_solar_zenith_angle](#)

References

- [1] Kevin S. Anderson, Adam R. Jensen; Shaded fraction and backtracking in single-axis trackers on rolling terrain. J. Renewable Sustainable Energy 1 March 2024; 16 (2): 023504. DOI: [10.1063/5.0202220](https://doi.org/10.1063/5.0202220)

A.4. Pérdidas por sombras en módulos con diodos de bypass

Pull Request: #2070, accesible en `pvlib.shading.direct_martinez`.

pvlib.shading.direct_martinez

```
pvlib.shading.direct_martinez(poа_global, poа_direct, shaded_fraction,  
shaded_blocks, total_blocks) \[source\]
```

A shading loss power factor for non-monolithic silicon modules and arrays with an arbitrary number of bypass diodes.

This experimental model reduces the direct and circumsolar irradiance reaching the module's cells based on the number of *blocks* affected by the shadow. More on blocks in the *Notes* section and in [1].

 *Added in version 0.11.0.*

Parameters:

- **poа_global** (*numeric*) – Plane of array global irradiance. [W/m²].
- **poа_direct** (*numeric*) – Plane of array direct and circumsolar irradiance. [W/m²].
- **shaded_fraction** (*numeric*) – Fraction of module surface area that is shaded. [Unitless].
- **shaded_blocks** (*numeric*) – Number of blocks affected by the shadow. [Unitless]. If a floating point number is provided, it will be rounded up.
- **total_blocks** (*int*) – Number of total blocks. Unitless.

Returns:

shading_losses (*numeric*) – Fraction of DC power lost due to shading. [Unitless]

Notes

The implemented equations are (6) and (8) from [1]:

$$(1 - F_{ES}) = (1 - F_{GS}) \left(1 - \frac{N_{SB}}{N_{TB} + 1} \right) \quad (6)$$

$$\left(1 - \frac{P_S}{P_{NS}} \right) = \left(1 - \frac{[(B + D^{CIR})(1 - F_{ES}) + D^{ISO} + R]}{G} \right) \quad (8)$$

In (6), $(1 - F_{ES})$ is the correction factor to be multiplied by the direct and circumsolar irradiance, F_{GS} is the shaded fraction of the collector, N_{SB} is the number of shaded blocks and N_{TB} is the number of total blocks.

In (8), $\frac{P_S}{P_{NS}}$ is the fraction of DC power lost due to shading, P_S is the power output of the shaded module, P_{NS} is the power output of the non-shaded module, $B + D^{CIR}$ is the beam and circumsolar irradiance, $D^{ISO} + R$ is the sum of diffuse and albedo irradiances and G is the global irradiance.

Anexo A: documentación de las funciones

Blocks terminology:

A *block* is defined in [1] as a group of solar cells protected by a bypass diode. Also, a *block* is shaded when at least one of its cells is partially shaded.

The total number of blocks and their layout depend on the module(s) used. Many manufacturers don't specify this information explicitly. However, these values can be inferred from:

- the number of bypass diodes
- where and how many junction boxes are present on the back of the module
- whether or not the module is comprised of *half-cut cells*

The latter two are heavily correlated.

For example:

1. A module with 1 bypass diode behaves as 1 block.
2. A module with 3 bypass diodes and 1 junction box is likely to have 3 blocks.
3. A half-cut module with 3 junction boxes (split junction boxes) is likely to have 3x2 blocks.
The number of blocks along the longest side of the module is 2 and along the shortest side is 3.
4. A module without bypass diodes doesn't constitute a block, but may be part of one.

Examples

Minimal example. For a complete example, see [Modelling shading losses in modules with bypass diodes](#).

```
>>> import numpy as np
>>> from pvlib import shading
>>> total_blocks = 3 # blocks along the vertical of the module
>>> POA_direct_and_circumsolar, POA_diffuse = 600, 80 # W/m2
>>> POA_global = POA_direct_and_circumsolar + POA_diffuse
>>> P_out_unshaded = 3000 # W
>>> # calculation of the shaded fraction for the collector
>>> shaded_fraction = shading.shaded_fraction1d(
    >>>     solar_zenith=80, solar_azimuth=180,
    >>>     axis_azimuth=90, shaded_row_rotation=25,
    >>>     collector_width=0.5, pitch=1, surface_to_axis_offset=0,
    >>>     cross_axis_slope=5.711, shading_row_rotation=50)
>>> # calculation of the number of shaded blocks
>>> shaded_blocks = np.ceil(total_blocks*shaded_fraction)
>>> # apply the Martinez power losses to the calculated shading
>>> loss_fraction = shading.direct_martinez(
    >>>     POA_global, POA_direct_and_circumsolar,
    >>>     shaded_fraction, shaded_blocks, total_blocks)
>>> P_out_corrected = P_out_unshaded * (1 - loss_fraction)
```

A.4. Pérdidas por sombras en módulos con diodos de bypass

See also

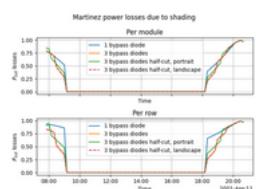
[shaded_fraction1d](#)

to calculate 1-dimensional shaded fraction

References

- [1]([1,2,3](#)) F. Martínez-Moreno, J. Muñoz, and E. Lorenzo, 'Experimental model to estimate shading losses on PV arrays', Solar Energy Materials and Solar Cells, vol. 94, no. 12, pp. 2298-2303, Dec. 2010, DOI: [10.1016/j.solmat.2010.07.029](https://doi.org/10.1016/j.solmat.2010.07.029).

Examples using `pvlib.shading.direct_martinez`



Modelling shading
losses in modules
with bypass diodes

Anexo A: documentación de las funciones

A.5. Fracción difusa de radiación fotosintetizable diaria

Pull Request: #2048, accesible en `pvlib.irradiance.diffuse_par_spitters`.

`pvlib.irradiance.diffuse_par_spitters`

```
pvlib.irradiance.diffuse_par_spitters(daily_solar zenith,  
global_diffuse_fraction)
```

[\[source\]](#)

Derive daily diffuse fraction of Photosynthetically Active Radiation (PAR) from daily average solar zenith and diffuse fraction of daily insolation.

The relationship is based on the work of Spitters et al. (1986) [1]. The resulting value is the fraction of daily PAR that is diffuse.

Note

The diffuse fraction is defined as the ratio of diffuse to global daily insolation, in J m^{-2} day $^{-1}$ or equivalent.

Parameters:

- `daily_solar zenith` (numeric) – Average daily solar zenith angle. In degrees [$^{\circ}$].
- `global_diffuse_fraction` (numeric) – Fraction of daily global broadband insolation that is diffuse. Unitless [0, 1].

Returns:

- `par_diffuse_fraction` (numeric) – Fraction of daily photosynthetically active radiation (PAR) that is diffuse. Unitless [0, 1].

Notes

The relationship is given by equations (9) & (10) in [1] and (1) in [2]:

$$k_{\text{diffuse_PAR}}^{\text{model}} = \frac{PAR_{\text{diffuse}}}{PAR_{\text{total}}} = \frac{\left[1 + 0.3(1 - (k_d)^2)\right]k_d}{1 + (1 - (k_d)^2)\cos^2(90 - \beta)\cos^3\beta}$$

where k_d is the diffuse fraction of the global insolation, and β is the daily average of the solar elevation angle in degrees.

A comparison using different models for the diffuse fraction of the global insolation can be found in [2] in the context of Sweden.

References

- [1](1,2) C. J. T. Spitters, H. A. J. M. Toussaint, and J. Goudriaan, 'Separating the diffuse and direct component of global radiation and its implications for modeling canopy photosynthesis Part I. Components of incoming radiation', Agricultural and Forest Meteorology, vol. 38, no. 1, pp. 217-229, Oct. 1986, [DOI: 10.1016/0168-1923\(86\)90060-2](https://doi.org/10.1016/0168-1923(86)90060-2).
- [2](1,2) S. Ma Lu et al., 'Photosynthetically active radiation decomposition models for agrivoltaic systems applications', Solar Energy, vol. 244, pp. 536-549, Sep. 2022, [DOI: 10.1016/j.solener.2022.05.046](https://doi.org/10.1016/j.solener.2022.05.046).

Examples using

`pvlib.irradiance.diffuse_par_spitters`



Calculating daily
diffuse PAR using
Spitter's relationship

A.6. Modelo de pérdidas por irradiancia no uniforme

Pull Request: #2046, accesible en `pvlib.bifacial.power_mismatch_deline`.

`pvlib.bifacial.power_mismatch_deline`

`pvlib.bifacial.power_mismatch_deline(rmad, coefficients=(0, 0.142, 3.2), fill_factor: float = None, fill_factor_reference: float = 0.79)` [\[source\]](#)

Estimate DC power loss due to irradiance non-uniformity.

This model is described for bifacial modules in [1], where the backside irradiance is less uniform due to mounting and site conditions.

The power loss is estimated by a polynomial model of the Relative Mean Absolute Difference (RMAD) of the cell-by-cell total irradiance.

Use `fill_factor` to account for different fill factors between the data used to fit the model and the module of interest. Specify the model's fill factor with `fill_factor_reference`.

 **Added in version 0.11.1.**

Parameters:

- **rmad** (*numeric*) –

The Relative Mean Absolute Difference of the cell-by-cell total irradiance. [Unitless]

See the Notes section for the equation to calculate `rmad` from the bifaciality and the front and back irradiances.

- **coefficients** (float collection or `numpy.polynomial.polynomial.Polynomial`, default `(0, 0.142, 0.032 * 100)`) –

The polynomial coefficients to use.

If a `numpy.polynomial.polynomial.Polynomial`, it is evaluated as is. If not a `Polynomial`, it must be the coefficients of a polynomial in `rmad`, where the first element is the constant term and the last element is the highest order term. A `Polynomial` will be created internally.

- **fill_factor** (`float`, *optional*) – Fill factor at standard test condition (STC) of the module.

Accounts for different fill factors between the trained model and the module under non-uniform irradiance. If not provided, the default `fill_factor_reference` of 0.79 is used.

- **fill_factor_reference** (`float`, *default 0.79*) – Fill factor at STC of the module used to train the model.

Returns:

`loss` (*numeric*) – The fractional power loss. [Unitless]

Output will be a `pandas.Series` if `rmad` is a `pandas.Series`.

Anexo A: documentación de las funciones

Notes

The default model implemented is equation (11) [1]:

$$\begin{aligned} M[\%] &= 0.142\Delta[\%] + 0.032\Delta^2[\%] \quad (11) \\ M[-] &= 0.142\Delta[-] + 0.032 \times 100\Delta^2[-] \end{aligned}$$

where the upper equation is in percentage (same as paper) and the lower one is unitless. The implementation uses the unitless version, where $M[-]$ is the mismatch power loss [unitless] and $\Delta[-]$ is the Relative Mean Absolute Difference [unitless] of the global irradiance, Eq. (4) of [1] and [2]. Note that the n-th power coefficient is multiplied by 100^{n-1} to convert the percentage to unitless.

The losses definition is Eq. (1) of [1], and it's defined as a loss of the output power:

$$M = 1 - \frac{P_{Array}}{\sum P_{Cells}} \quad (1)$$

To account for a module with a fill factor distinct from the one used to train the model (0.79 by default), the output of the model can be modified with Eq. (7):

$$M_{FF_1} = M_{FF_0} \frac{FF_1}{FF_0} \quad (7)$$

where parameter `fill_factor` is FF_1 and `fill_factor_reference` is FF_0 .

In the section *See Also*, you will find two packages that can be used to calculate the irradiance at different points of the module.

Note

The global irradiance RMAD is different from the backside irradiance RMAD.

In case the RMAD of the backside irradiance is known, the global RMAD can be calculated as follows, assuming the front irradiance RMAD is negligible [2]:

$$RMAD(k \cdot X + c) = RMAD(X) \cdot k \frac{\bar{X}}{k\bar{X} + c} = RMAD(X) \cdot \frac{k}{1 + \frac{c}{k\bar{X}}}$$

by similarity with equation (2) of [1]:

$$G_{total\ i} = G_{front\ i} + \phi_{Bifi}G_{rear\ i} \quad (2)$$

which yields:

$$RMAD_{total} = RMAD_{rear} \frac{\phi_{Bifi}}{1 + \frac{G_{front}}{\phi_{Bifi}G_{rear}}}$$

See also

[solarfactors](#)

Calculate the irradiance at different points of the module.

[bifacial_radiance](#)

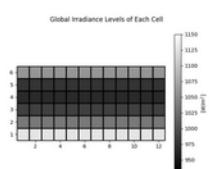
Calculate the irradiance at different points of the module.

References

- [1]([1](#),[2](#),[3](#),[4](#),[5](#)) C. Deline, S. Ayala Pelaez, S. MacAlpine, and C. Olalla, 'Estimating and parameterizing mismatch power loss in bifacial photovoltaic systems', Progress in Photovoltaics: Research and Applications, vol. 28, no. 7, pp. 691-703, 2020, [DOI: 10.1002/pip.3259](#).
- [2]([1](#),[2](#)) "Mean absolute difference," Wikipedia, Sep. 05, 2023. https://en.wikipedia.org/wiki/Mean_absolute_difference#Relative_mean_absolute_difference (accessed 2024-04-14).

Examples using

`pvlib.bifacial.power_mismatch_deline`



Plot Irradiance Non-uniformity Loss

A.7. Conversión entre eficiencia cuántica y respuesta espectral

Pull Request: #2041, accesibles en `pvlib.spectrum.qe_to_sr` y `pvlib.spectrum(sr_to_qe)`.

`pvlib.spectrum.qe_to_sr`

`pvlib.spectrum.qe_to_sr(qe, wavelength=None, normalize=False)` [\[source\]](#)

Convert quantum efficiencies to spectral responsivities. If `wavelength` is not provided, the quantum efficiency `qe` must be a `pandas.Series` or `pandas.DataFrame`, with the wavelengths in the index.

Provide wavelengths in nanometers, [nm].

Conversion is described in [\[1\]](#).

❶ Added in version 0.11.0.

Parameters:

- `qe` (numeric, `pandas.Series` or `pandas.DataFrame`) – Quantum efficiency. If pandas subtype, index must be the wavelength in nanometers, [nm].
- `wavelength` (numeric, optional) – Points where quantum efficiency is measured, in nanometers, [nm].
- `normalize` (`bool`, default `False`) – If True, the spectral response is normalized so that the maximum value is 1. For `pandas.DataFrame`, normalization is done for each column. For 2D arrays, normalization is done for each sub-array.

Returns:

`spectral_response` (numeric, same type as `qe`) – Spectral response, [A/W].

Notes

- If `qe` is of type `pandas.Series` or `pandas.DataFrame`, column names will remain unchanged in the returned object.
- If `wavelength` is provided it will be used independently of the datatype of `qe`.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from pvlib import spectrum
>>> wavelengths = np.array([350, 550, 750])
>>> quantum_efficiency = np.array([0.80, 0.90, 0.94])
>>> spectral_response = spectrum.qe_to_sr(quantum_efficiency, wavelengths)
>>> print(spectral_response)
array([0.24277287, 0.39924442, 0.56862085])
```

```
>>> quantum_efficiency_series = pd.Series(quantum_efficiency, index=wavelengths, name=""
>>> sr = spectrum.qe_to_sr(quantum_efficiency_series)
>>> print(sr)
350    0.242773
550    0.399244
750    0.568621
Name: dataset, dtype: float64
```



```
>>> sr = spectrum.qe_to_sr(quantum_efficiency_series, normalize=True)
>>> print(sr)
350    0.426950
550    0.702128
750    1.000000
Name: dataset, dtype: float64
```

pvlib.spectrum sr_to_qe

```
pvlib.spectrum.sr_to_qe(sr, wavelength=None, normalize=False) [source]
```

Convert spectral responsivities to quantum efficiencies. If `wavelength` is not provided, the spectral responsivity `sr` must be a `pandas.Series` or `pandas.DataFrame`, with the wavelengths in the index.

Provide wavelengths in nanometers, [nm].

Conversion is described in [1].

❶ Added in version 0.11.0.

Parameters:

- `sr` (numeric, `pandas.Series` or `pandas.DataFrame`) – Spectral response, [A/W]. Index must be the wavelength in nanometers, [nm].
- `wavelength` (numeric, optional) – Points where spectral response is measured, in nanometers, [nm].
- `normalize` (bool, default `False`) – If True, the quantum efficiency is normalized so that the maximum value is 1. For `pandas.DataFrame`, normalization is done for each column. For 2D arrays, normalization is done for each sub-array.

Returns:

`quantum_efficiency` (numeric, same type as `sr`) – Quantum efficiency, in the interval [0, 1].

Notes

- If `sr` is of type `pandas.Series` or `pandas.DataFrame`, column names will remain unchanged in the returned object.
- If `wavelength` is provided it will be used independently of the datatype of `sr`.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from pvlib import spectrum
>>> wavelengths = np.array([350, 550, 750])
>>> spectral_response = np.array([0.25, 0.40, 0.57])
>>> quantum_efficiency = spectrum.sr_to_qe(spectral_response, wavelengths)
>>> print(quantum_efficiency)
array([0.88560142, 0.90170326, 0.94227991])
```

```
>>> spectral_response_series = pd.Series(spectral_response, index=wavelengths, name="da"
>>> qe = spectrum.sr_to_qe(spectral_response_series)
>>> print(qe)
350    0.885601
550    0.901703
750    0.942280
Name: dataset, dtype: float64
```

```
>>> qe = spectrum.sr_to_qe(spectral_response_series, normalize=True)
>>> print(qe)
350    0.939850
550    0.956938
750    1.000000
Name: dataset, dtype: float64
```

A.8. Espectro estándar ASTM G173-03

Pull Request: #1963, accesible en `pvlib.spectrum.get_reference_spectra`.

`pvlib.spectrum.get_reference_spectra`

`pvlib.spectrum.get_reference_spectra(wavelengths=None, standard='ASTM G173-03')` [\[source\]](#)

Read a standard spectrum specified by `standard`, optionally interpolated to the specified wavelength(s).

Defaults to `ASTM G173-03` AM1.5 standard [1], which returns `extraterrestrial`, `global` and `direct` spectrum on a 37-degree tilted surface, optionally interpolated to the specified wavelength(s).

Parameters:

- **wavelengths** (*numeric, optional*) – Wavelengths at which the spectrum is interpolated. [nm]. If not provided, the original wavelengths from the specified standard are used. Values outside that range are filled with zeros.
- **standard** (*str, default "ASTM G173-03"*) – The reference standard to be read. Only the reference `"ASTM G173-03"` is available at the moment.

Returns:

`standard_spectra` (*pandas.DataFrame*) – The standard spectrum by `wavelength [nm]`. [W/(m²nm)]. Column names are `extraterrestrial`, `direct` and `global`.

Notes

If `wavelength` is specified, linear interpolation is used.

If the values in `wavelength` are too widely spaced, the integral of each spectrum may deviate from its standard value. For global spectra, it is about 1000.37 W/m².

The values of the ASTM G173-03 provided with pvlib-python are copied from an Excel file distributed by NREL, which is found here [2]: <https://www.nrel.gov/grid/solar-resource/assets/data/astmg173.xls>

Examples

```
>>> from pvlib import spectrum
>>> am15 = spectrum.get_reference_spectra()
>>> am15_extraterrestrial, am15_global, am15_direct = \
>>>     am15['extraterrestrial'], am15['global'], am15['direct']
>>> print(am15.head())
      extraterrestrial    global    direct
wavelength
280.0          0.082  4.730900e-23  2.536100e-26
280.5          0.099  1.230700e-21  1.091700e-24
281.0          0.150  5.689500e-21  6.125300e-24
281.5          0.212  1.566200e-19  2.747900e-22
282.0          0.267  1.194600e-18  2.834600e-21
```

```
>>> am15 = spectrum.get_reference_spectra([300, 500, 800, 1100])
>>> print(am15)
      extraterrestrial    global    direct
wavelength
300          0.45794  0.00102  0.000456
500          1.91600  1.54510  1.339100
800          1.12480  1.07250  0.988590
1100         0.60000  0.48577  0.461130
```

References

- [1] ASTM "G173-03 Standard Tables for Reference Solar Spectral Irradiances: Direct Normal and Hemispherical on 37° Tilted Surface."
- [2] "Reference Air Mass 1.5 Spectra," www.nrel.gov. <https://www.nrel.gov/grid/solar-resource/spectra-am1.5.html>

Examples using

`pvlib.spectrum.get_reference_spectra`



ASTM G173-03
Standard Spectrum

Anexo A: documentación de las funciones

A.9. Cálculo geométrico de sombras en 3D

Pull Request: #2106, propuesta no aceptada, pero documentación disponible temporalmente en `pvlb.spatial.FlatSurface` y `pvlb.spatial.RectangularSurface`.

`pvlb.spatial.FlatSurface`

```
class pvlb.spatial.FlatSurface(azimuth, tilt, polygon_boundaries, roll=0.0,  
reference_point=None) [source]
```

Represents a flat surface in 3D space with a given azimuth and tilt and boundaries defined by a shapely Polygon. Allows to calculate the shading on this surface from other objects, both in 2D and 3D. In addition, it can

⚠ Warning

This constructor does **not** check the `azimuth` and `tilt` match the `polygon` orientation nor the `polygon` vertices are coplanar. It is the user's responsibility to ensure the surface is correctly defined.

Parameters:

- `tilt (float)` – Surface tilt, angle it is inclined with respect to the horizontal plane. Tilted downwards `azimuth`. 0°=Horizontal, 90°=Vertical. In degrees [°].
- `azimuth (float)` – Surface azimuth, angle at which it points downwards. 0°=North, 90°=East, 180°=South, 270°=West. In degrees [°].
- `polygon (shapely.Polygon or array[N, 3])` – Shapely Polygon or boundaries to build it. Holes are ignored for now.
- `roll (float, default 0.0)` – Right-handed rotation around the surface normal vector defined by `tilt` and `azimuth`. In degrees [°].
- `reference_point (array-like, shape (3,), optional)` – Point to use as reference for 2D projections. If not provided, the first vertex of the polygon is used.

References

1. S. Zainali et al., 'Direct and diffuse shading factors modelling for the most representative agrivoltaic system layouts', Applied Energy, vol. 339, p. 120981, Jun. 2023, DOI: [10.1016/j.apenergy.2023.120981](https://doi.org/10.1016/j.apenergy.2023.120981).
2. Y. Cascone, V. Corrado, and V. Serra, 'Calculation procedure of the shading factor under complex boundary conditions', Solar Energy, vol. 85, no. 10, pp. 2524–2539, Oct. 2011, DOI: [10.1016/j.solener.2011.07.011](https://doi.org/10.1016/j.solener.2011.07.011).
3. Kevin S. Anderson, Adam R. Jensen; Shaded fraction and backtracking in single-axis trackers on rolling terrain. J. Renewable Sustainable Energy 1 March 2024; 16 (2): 023504. DOI: [10.1063/5.0202220](https://doi.org/10.1063/5.0202220).

Methods

<code>__init__(azimuth, tilt, polygon_boundaries)</code>	
<code>combine_2D_shades(*shades)</code>	Combine overlapping shades into a single one, but keep non-overlapping shades separated.
<code>get_2D_shades_from(solar Zenith, ...[, ...])</code>	Calculate 2D shades on this surface from obstacles <code>others*</code> .
<code>get_3D_shades_from(solar Zenith, ...)</code>	Calculate 3D shades on this surface from other objects.
<code>plot([ax])</code>	
<code>representation_in_2D_space()</code>	Get the 2D representation of the surface in its local plane, with respect to the reference point.

Attributes

<code>azimuth</code>	
<code>polygon</code>	
<code>reference_point</code>	
<code>roll</code>	
<code>tilt</code>	

Examples using `pvlb.spatial.FlatSurface`



pvlb.spatial.RectangularSurface

```
class pvlb.spatial.RectangularSurface(center, azimuth, tilt, axis_tilt, width, length,
reference_point=None)
```

[\[source\]](#)

Represents a rectangular surface in 3D space with a given `azimuth`, `tilt`, `axis_tilt` and a center point. This is a subclass of `FlatSurface` handy for rectangular surfaces like PV arrays.

See `pvlb.spatial.FlatSurface` for information on methods and properties.

Parameters:

- `center` (*array-like, shape (3,)*) – Center of the surface
- `azimuth` (*float*) – Azimuth of the surface. Positive is clockwise from the North in the horizontal plane. North=0°, East=90°, South=180°, West=270°. In degrees [°].
- `tilt` (*float*) – Tilt of the surface, angle it is inclined with respect to the horizontal plane. Positive is downwards `azimuth`. In degrees [°].
- `axis_tilt` (*float*) – Rotation around the normal vector of the surface. Right-handed. In degrees [°].
- `width` (*width of the surface*) – For a horizontal surface, the width is parallel to the azimuth
- `length` (*length of the surface*) – Perpendicular to the surface azimuth
- `reference_point` (*array-like, shape (3,), optional*) – Point to use as reference for 2D projections. If not provided, the central point is used.

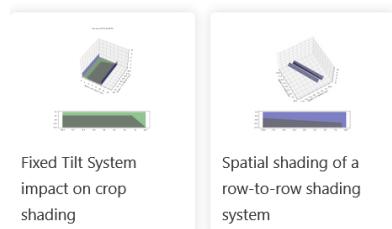
Methods

<code>__init__(center, azimuth, tilt, axis_tilt, ...)</code>	
<code>combine_2D_shades(*shades)</code>	Combine overlapping shades into a single one, but keep non-overlapping shades separated.
<code>get_2D_shades_from(solar Zenith, ..., ...)</code>	Calculate 2D shades on this surface from obstacles <code>others</code> *
<code>get_3D_shades_from(solar zenith, ...)</code>	Calculate 3D shades on this surface from other objects.
<code>plot([ax])</code>	Plot the rectangular surface.
<code>representation_in_2D_space()</code>	Get the 2D representation of the surface in its local plane, with respect to the reference point.

Attributes

<code>azimuth</code>
<code>polygon</code>
<code>reference_point</code>
<code>roll</code>
<code>tilt</code>

Examples using `pvlb.spatial.RectangularSurface`



Fixed Tilt System
impact on crop
shading

Spatial shading of a
row-to-row shading
system

Apéndice B

Anexo B: ejemplos de uso

En este anexo se encuentran ejemplos elaborados para demostrar el uso de las propuestas en un contexto más cercano a la realidad. Se muestran en forma de imágenes, pues es lo que una persona usuaria final vería.

En las propuestas aceptadas se podrá descargar el código fuente de la misma página. En el caso de las no aceptadas, habría que acceder a la rama de trabajo de la propuesta mediante la *pull request* correspondiente o la página de la documentación que se aloja temporalmente en la web, si es que existe.

B.1. Modelo de ajuste espectral

Propuesta en *Pull Request*: #1658, no aceptada.

N. Martin & J. M. Ruiz Spectral Mismatch Modifier

How to use this correction factor to adjust the POA global irradiance.

Effectiveness of a material to convert incident sunlight to current depends on the incident light wavelength. During the day, the spectral distribution of the incident irradiance varies from the standard testing spectra, introducing a small difference between the expected and the real output. In [1], N. Martin and J. M. Ruiz propose 3 mismatch factors, one for each irradiance component. These mismatch modifiers are calculated with the help of the airmass, the clearness index and three experimental fitting parameters. In the same paper, these parameters have been obtained for m-Si, p-Si and a-Si modules. With `pvlb.spectrum.martin_ruiz()` we are able to make use of these already computed values or provide ours.

References

- [1] Martin, N. and Ruiz, J.M. (1999), A new method for the spectral characterisation of PV modules. *Prog. Photovolt: Res. Appl.*, 7: 299-310. DOI: 10.1002/(SICI)1099-159X(199907/08)7:4<299::AID-PIP260>3.0.CO;2-0

Calculating the incident and modified global irradiance

Mismatch modifiers are applied to the irradiance components, so first step is to get them. We define an hypothetical POA surface and use a TMY to compute sky diffuse, ground reflected and direct irradiances.

```
import os
import matplotlib.pyplot as plt
import pvlb
from scipy import stats
import pandas as pd

surface_tilt = 40
surface_azimuth = 180 # Pointing South

# Get TMY data & create location
datapath = os.path.join(
    pvlb.__path__[0], "data", "tmy_45.000_8.000_2005_2016.csv"
)
pvgs_data, _, metadata, _ = pvlb.iotools.read_pvgs_tmy(
    datapath, map_variables=True
)
site = pvlb.location.Location(
    metadata["latitude"], metadata["longitude"], altitude=metadata["elevation"]
)

# Coerce a year: function above returns typical months of different years
pvgs_data.index = [ts.replace(year=2022) for ts in pvgs_data.index]
# Select days to show
weather_data = pvgs_data["2022-09-03":"2022-09-06"]

# Then calculate all we need to get the irradiance components
solar_pos = site.get_solarposition(weather_data.index)

extra_rad = pvlb.irradiance.get_extra_radiation(weather_data.index)

poa_sky_diffuse = pvlb.irradiance.haydavies(
    surface_tilt,
    surface_azimuth,
    weather_data["dni"],
    weather_data["dni"],
    extra_rad,
    solar_pos["apparent zenith"],
    solar_pos["azimuth"],
)

poa_ground_diffuse = pvlb.irradiance.get_ground_diffuse(
    surface_tilt, weather_data["ghi"]
)

aoi = pvlb.irradiance.aoi(
    surface_tilt,
    surface_azimuth,
    solar_pos["apparent zenith"],
    solar_pos["azimuth"],
)

# Get dataframe with all components and global (includes 'poa_direct')
poa_irrad = pvlb.irradiance.poa_components(
    aoi, weather_data["dni"], poa_sky_diffuse, poa_ground_diffuse
)

# Apply Martin & Ruiz IAM modifiers
iam_direct = pvlb.iam.martin_ruiz(aoi)
iam_sky_diffuse, iam_ground_diffuse = pvlb.iam.martin_ruiz_diffuse(
    surface_tilt
)

poa_irrad["poa_direct"] = poa_irrad["poa_direct"] * iam_direct
poa_irrad["poa_sky_diffuse"] = poa_irrad["poa_sky_diffuse"] * iam_sky_diffuse
poa_irrad["poa_ground_diffuse"] = (
    poa_irrad["poa_ground_diffuse"] * iam_ground_diffuse
)
```

Anexo B: ejemplos de uso

Here come the modifiers. Let's calculate them with the airmass and clearness index. First, let's find the airmass and the clearness index. Little caution: default values for this model were fitted obtaining the airmass through the 'kasten1966' method, which is not used by default.

```
airmass = site.get_airmass(solar_position=solar_pos, model="kasten1966")
airmass_absolute = airmass["airmass_absolute"] # We only use absolute airmass
clearness_index = pvlib.irradiance.clearness_index(
    weather_data["ghi"], solar_pos["zenith"], extra_rad
)
# Get the spectral mismatch modifiers
spectral_modifiers = pvlib.spectrum.martin_ruiz(
    clearness_index, airmass_absolute, module_type="monosi"
)
```

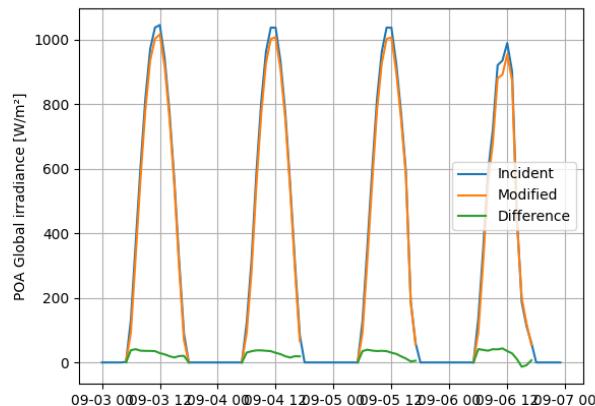
And then we can find the 3 modified components of the POA irradiance by means of a simple multiplication. Note, however, that this does not modify `poa_global` nor `poa_diffuse`, so we should update the dataframe afterwards.

```
poa_irrad_modified = poa_irrad * spectral_modifiers
# Line above is equivalent to:
# poa_irrad_modified = pd.DataFrame()
# for component in ('poa_direct', 'poa_sky_diffuse', 'poa_ground_diffuse'):
#     poa_irrad_modified[component] = (poa_irrad[component]
#                                         * spectral_modifiers[component])

# We want global modified irradiance
poa_irrad_modified["poa_global"] = (
    poa_irrad_modified["poa_direct"]
    + poa_irrad_modified["poa_sky_diffuse"]
    + poa_irrad_modified["poa_ground_diffuse"]
)
# Don't forget to update ``poa_diffuse`` if you want to use it
# poa_irrad_modified["poa_diffuse"] = \
#     (poa_irrad_modified['poa_sky_diffuse']
#      + poa_irrad_modified['poa_ground_diffuse'])
```

Finally, let's plot the incident vs modified global irradiance, and their difference.

```
poa_irrad_global_diff = (
    poa_irrad["poa_global"] - poa_irrad_modified["poa_global"]
)
plt.figure()
datetimes = poa_irrad.index # common to poa_irrad *
plt.plot(datetimes, poa_irrad["poa_global"].to_numpy())
plt.plot(datetimes, poa_irrad_modified["poa_global"].to_numpy())
plt.plot(datetimes, poa_irrad_global_diff.to_numpy())
plt.legend(["Incident", "Modified", "Difference"])
plt.ylabel("POA Global irradiance [W/m2]")
plt.grid()
plt.show()
```



Comparison against other models

During the addition of this model, a question arose about its trustworthiness so, in order to check the integrity of the implementation, we will compare it against

`pvlib.spectrum.spectral_factor_sapm()` and
`pvlib.spectrum.spectral_factor_firstrsolar()`. Former model needs the parameters that characterise a module, but which one? We will take the mean of Sandia parameters 'A0', 'A1', 'A2', 'A3', 'A4' for the same material type. On the other hand, `firstrsolar()` needs the precipitable water. We assume the standard spectrum, 1.42 cm.

```
# Retrieve modules and select the subset we want to work with the SAPM model
module_type = "c-Si" # Equivalent to monosi
sandia_modules = pvlib.pvsystem.retrieve_sam(name="SandiaMod")
modules_subset = sandia_modules.loc[
    :, sandia_modules.loc["Material"] == module_type
]

# Define typical module and get the means of the A0 to A4 parameters
modules_aggregated = pd.DataFrame(index=["mean", "std"])
for param in ("A0", "A1", "A2", "A3", "A4"):
    result, _ = stats.mvdist(modules_subset.loc[param])
    modules_aggregated[param] = result.mean(), result.std()

# Check if 'mean' is a representative value with help of 'std' just in case
print(modules_aggregated)

# Then apply the SAPM model and calculate introduced difference
modifier_sapm_f1 = pvlib.spectrum.spectral_factor_sapm(
    airmass_absolute, modules_aggregated.loc["mean"])
poa_irrad_sapm_modified = poa_irrad["poa_global"] * modifier_sapm_f1
poa_irrad_sapm_difference = poa_irrad["poa_global"] - poa_irrad_sapm_modified

# spectrum.spectral_factor_firstrsolar model
firstrsolar_pw = 1.42 # Default for AM1.5 spectrum
modifier_firstrsolar = pvlib.spectrum.spectral_factor_firstrsolar(
    firstrsolar_pw, airmass_absolute, module_type="monosi")
poa_irrad_firstrsolar_mod = poa_irrad["poa_global"] * modifier_firstrsolar
poa_irrad_firstrsolar_diff = (
    poa_irrad["poa_global"] - poa_irrad_firstrsolar_mod
)
```

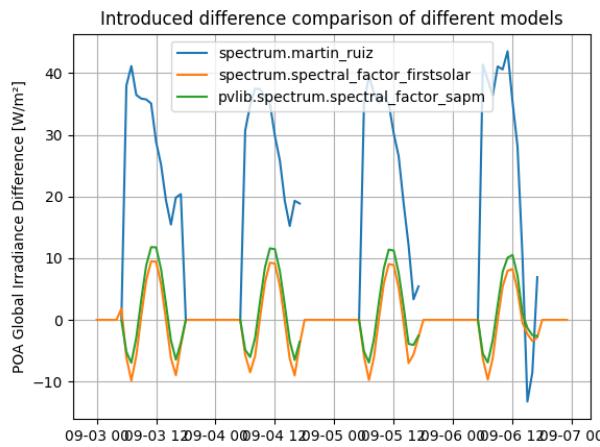
Out:

	A0	A1	A2	A3	A4
mean	0.933789	0.058730	-0.011163	0.000956	-0.000034
std	0.000777	0.000764	0.000303	0.000066	0.000005

Plot global irradiance difference over time

```
datetimes = poa_irrad_global_diff.index # common to poa_irrad_*_diff*
plt.figure()
plt.plot(
    datetimes, poa_irrad_global_diff.to_numpy(), label="spectrum.martin_ruiz")
plt.plot(
    datetimes,
    poa_irrad_sapm_difference.to_numpy(),
    label="spectrum.spectral_factor_firstrsolar",
)
plt.plot(
    datetimes,
    poa_irrad_firstrsolar_diff.to_numpy(),
    label="pvlib.spectrum.spectral_factor_sapm",
)
plt.legend()
plt.title("Introduced difference comparison of different models")
plt.ylabel("POA Global Irradiance Difference [W/m²]")
plt.grid()
plt.show()
```

Anexo B: ejemplos de uso

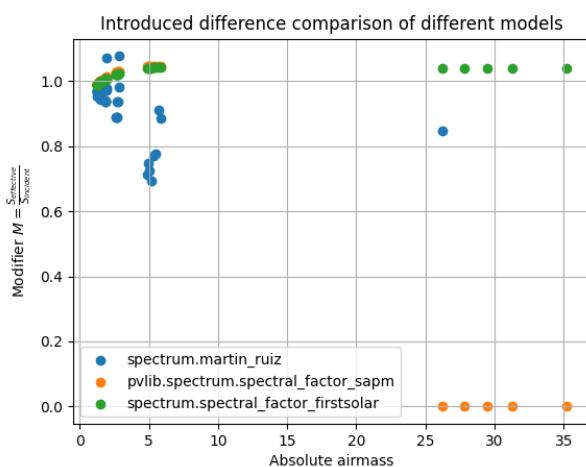


Plot modifier vs absolute airmass

```

ama = airmass_absolute_to_numpy()
# spectrum.martin_ruiz has 3 modifiers, so we only calculate one as
# M = S_eff / S_incident that takes into account the global effect
martin_ruiz_agg_modifier = (
    poa_irrad_modified["poa_global"] / poa_irrad["poa_global"]
)
plt.figure()
plt.scatter(
    ama,
    martin_ruiz_agg_modifier.to_numpy(),
    label="spectrum.martin_ruiz"
)
plt.scatter(
    ama,
    modifier_sapm_f1.to_numpy(),
    label="pvlib.spectrum.spectral_factor_sapm",
)
plt.scatter(
    ama,
    modifier_first_solar.to_numpy(),
    label="spectrum.spectral_factor_firstsolar",
)
plt.legend()
plt.title("Introduced difference comparison of different models")
plt.xlabel("Absolute airmass")
plt.ylabel(r"Modifier $M = \frac{S_{\text{effective}}}{S_{\text{incident}}}$")
plt.grid()
plt.show()

```



B.2. Fracción de sombra unidimensional

Pull Request: #1962, acceso a ejemplo en este link.

Shaded fraction of a horizontal single-axis tracker

This example illustrates how to calculate the 1D shaded fraction of three rows in a North-South horizontal single axis tracker (HSAT) configuration.

`pvlib.shading.shaded_fraction1d()` exposes a useful method for the calculation of the shaded fraction of the width of a solar collector. Here, the width is defined as the dimension perpendicular to the axis of rotation. This method for calculating the shaded fraction only requires minor modifications to be applicable for fixed-tilt systems.

It is highly recommended to read `pvlib.shading.shaded_fraction1d()` documentation to understand the input parameters and the method's capabilities. For more in-depth information, please see the journal paper [10.1063/5.0202220](#) describing the methodology.

Let's start by obtaining the true-tracking angles for each of the rows and limiting the angles to the range of -50 to 50 degrees. This decision is done to create an example scenario with significant shading.

Key functions used in this example are:

1. `pvlib.tracking.singleaxis()` to calculate the tracking angles.
2. `pvlib.shading.projected_solar zenith angle()` to calculate the projected solar zenith angle.
3. `pvlib.shading.shaded_fraction1d()` to calculate the shaded fractions.

```
import pvlib

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter

# Define the solar system parameters
latitude, longitude = 28.51, -13.89
altitude = pvlib.location.lookup_altitude(latitude, longitude)

axis_tilt = 3 # degrees, positive is upwards in the axis_azimuth direction
axis_azimuth = 180 # degrees, N-S tracking axis
collector_width = 3.2 # m
pitch = 4.15 # m
gcr = collector_width / pitch
cross_axis_slope = -5 # degrees
surface_to_axis_offset = 0.07 # m

# Generate a time range for the simulation
times = pd.date_range(
    start="2024-01-01T05",
    end="2024-01-01T21",
    freq="5min",
    tz="Atlantic/Canary",
)

# Calculate the solar position
solar_position = pvlib.solarposition.get_solarposition(
    times, latitude, longitude, altitude
)
solar_azimuth = solar_position["azimuth"]
solar_zenith = solar_position["apparent_zenith"]

# Calculate the tracking angles
rotation_angle = pvlib.tracking.singleaxis(
    solar_zenith,
    solar_azimuth,
    axis_tilt,
    axis_azimuth,
    max_angle=(-50, 50), # (min, max) degrees
    backtrack=False,
    gcr=gcr,
    cross_axis_tilt=cross_axis_slope,
    )["tracker_theta"]
```

Anexo B: ejemplos de uso

Once the tracker angles have been obtained, the next step is to calculate the shaded fraction. Special care must be taken to ensure that the shaded or shading tracker roles are correctly assigned depending on the solar position. This means we will have a result for each row,

`eastmost_shaded_fraction`, `middle_shaded_fraction`, and `westmost_shaded_fraction`. Switching the parameters will be based on the sign of `pvlib.shading.projected_solar zenith angle()`.

The following code is verbose to make it easier to understand the process, but with some effort you may be able to simplify it. This verbosity also allows to change the premises easily per case, e.g., in case of a tracker failure or with a different system configuration.

```
psza = pvlib.shading.projected_solar zenith angle(
    solar zenith, solar azimuth, axis tilt, axis azimuth
)

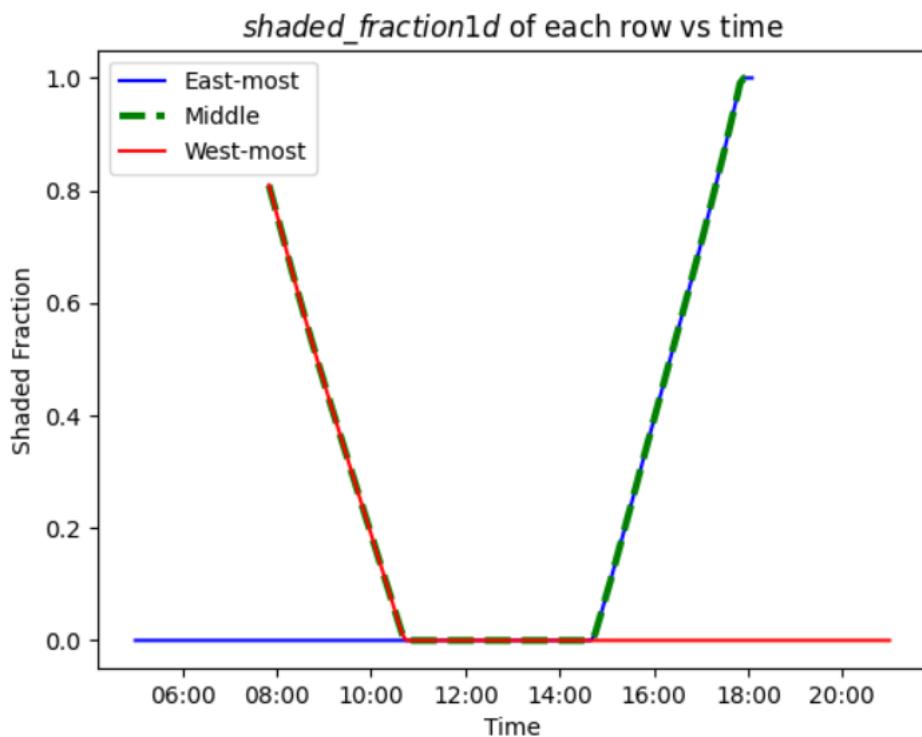
# Calculate the shaded fraction for the eastmost row
eastmost_shaded_fraction = np.where(
    psza < 0,
    0, # no shaded fraction in the morning
    # shaded fraction in the evening
    pvlib.shading.shaded_fraction1d(
        solar zenith,
        solar azimuth,
        axis azimuth,
        shaded row rotation=rotation angle,
        axis tilt=axis tilt,
        collector width=collector width,
        pitch=pitch,
        surface to axis offset=surface to axis offset,
        cross axis slope=cross axis slope,
        shading row rotation=rotation angle,
    ),
)

# Calculate the shaded fraction for the middle row
middle_shaded_fraction = np.where(
    psza < 0,
    # shaded fraction in the morning
    pvlib.shading.shaded_fraction1d(
        solar zenith,
        solar azimuth,
        axis azimuth,
        shaded row rotation=rotation angle,
        axis tilt=axis tilt,
        collector width=collector width,
        pitch=pitch,
        surface to axis offset=surface to axis offset,
        cross axis slope=cross axis slope,
        shading row rotation=rotation angle,
    ),
    # shaded fraction in the evening
    pvlib.shading.shaded_fraction1d(
        solar zenith,
        solar azimuth,
        axis azimuth,
        shaded row rotation=rotation angle,
        axis tilt=axis tilt,
        collector width=collector width,
        pitch=pitch,
        surface to axis offset=surface to axis offset,
        cross axis slope=cross axis slope,
        shading row rotation=rotation angle,
    ),
)

# Calculate the shaded fraction for the westmost row
westmost_shaded_fraction = np.where(
    psza < 0,
    # shaded fraction in the morning
    pvlib.shading.shaded_fraction1d(
        solar zenith,
        solar azimuth,
        axis azimuth,
        shaded row rotation=rotation angle,
        axis tilt=axis tilt,
        collector width=collector width,
        pitch=pitch,
        surface to axis offset=surface to axis offset,
        cross axis slope=cross axis slope,
        shading row rotation=rotation angle,
    ),
    0, # no shaded fraction in the evening
)
```

Plot the shaded fraction result for each row:

```
plt.plot(times, eastmost_shaded_fraction, label="East-most", color="blue")
plt.plot(times, middle_shaded_fraction, label="Middle", color="green",
          linewidth=3, linestyle="--") # fmt: skip
plt.plot(times, westmost_shaded_fraction, label="West-most", color="red")
plt.title(r"$shaded\_fraction1d$ of each row vs time")
plt.xlabel("Time")
plt.gca().xaxis.set_major_formatter(DateFormatter("%H:%M"))
plt.ylabel("Shaded Fraction")
plt.legend()
plt.show()
```



Total running time of the script: (0 minutes 0.139 seconds)

[Download Jupyter notebook: `plot_shaded_fraction1d_ns_hsat_example.ipynb`](#)

[Download Python source code: `plot_shaded_fraction1d_ns_hsat_example.py`](#)

Anexo B: ejemplos de uso

B.3. Pérdidas por sombras en módulos con diodos de bypass

Pull Request: #2070, acceso a ejemplo en este link.

Modelling shading losses in modules with bypass diodes

This example illustrates how to use the loss model proposed by Martinez et al. [1]. The model proposes a power output losses factor by adjusting the incident direct and circumsolar beam irradiance fraction of a PV module based on the number of shaded *blocks*. A *block* is defined as a group of cells protected by a bypass diode. More information on *blocks* can be found in the original paper [1] and in the [pvlib.shading.direct_martinez\(\)](#) documentation.

The following key functions are used in this example:

1. [pvlib.shading.direct_martinez\(\)](#) to calculate the power output losses fraction due to shading.
2. [pvlib.shading.shaded_fraction1d\(\)](#) to calculate the fraction of shaded surface and consequently the number of shaded *blocks* due to row-to-row shading.
3. [pvlib.tracking.singleaxis\(\)](#) to calculate the rotation angle of the trackers.

Problem description

Let's consider a PV system with the following characteristics:

- Two north-south single-axis trackers, each one having 6 modules.
- The rows have the same true-tracking tilt angles. True tracking is chosen in this example, so shading is significant.
- Terrain slope is 7 degrees downward to the east.
- Row axes are horizontal.
- **The modules are comprised of multiple cells. We will compare these cases:**
 - modules with one bypass diode
 - modules with three bypass diodes
 - half-cut cell modules with three bypass diodes in portrait and landscape

Setting up the system

Let's start by defining the system characteristics, location and the time range for the analysis.

```
import pvlib
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.dates import ConciseDateFormatter

pitch = 4 # meters
width = 1.5 # meters
gc = width / pitch # ground coverage ratio
N_modules_per_row = 6
axis_azimuth = 180 # N-S axis
axis_tilt = 0 # flat because the axis is perpendicular to the slope
cross_axis_tilt = -7 # 7 degrees downward to the east

latitude, longitude = 40.2712, -3.7277
locus = pvlib.location.Location(
    latitude,
    longitude,
    tz="Europe/Madrid",
    altitude=pvlib.location.lookup_altitude(latitude, longitude),
)
times = pd.date_range("2001-04-11T04", "2001-04-11T20", freq="10min")
```

True-tracking algorithm and shaded fraction

Since this model is about row-to-row shading, we will use the true-tracking algorithm to calculate the trackers rotation. Back-tracking eliminates shading between rows, and since this example is about shading, we will not use it.

Then, the next step is to calculate the fraction of shaded surface. This is done using [pvlib.shading.shaded_fraction1d\(\)](#). Using this function is straightforward with the variables we already have defined. Then, we can calculate the number of shaded blocks by rounding up the shaded fraction by the number of blocks along the shaded length.

B.3. Pérdidas por sombras en módulos con diodos de bypass

```

# Calculate solar position to get single-axis tracker rotation and irradiance
solar_pos = locus.get_solarposition(times)
solar_apparent Zenith, solar_azimuth = (
    solar_pos["apparent_zenith"],
    solar_pos["azimuth"],
) # unpack for better readability

tracking_result = pvlib.tracking.singleaxis(
    apparent_zenith=solar_apparent_zenith,
    apparent_azimuth=solar_azimuth,
    axis_tilt=axis_tilt,
    axis_azimuth=axis_azimuth,
    max_angle=(-90 + cross_axis_tilt, 90 + cross_axis_tilt), # (min, max)
    backtrack=False,
    GCR=GCR,
    cross_axis_tilt=cross_axis_tilt,
)

tracker_theta, aoi, surface_tilt, surface_azimuth = (
    tracking_result["tracker_theta"],
    tracking_result["aoi"],
    tracking_result["surface_tilt"],
    tracking_result["surface_azimuth"],
) # unpack for better readability

# Calculate the shade fraction
shaded_fraction = pvlib.shading.shaded_fraction1d(
    solar_apparent_zenith,
    solar_azimuth,
    axis_azimuth,
    axis_tilt=axis_tilt,
    shaded_row_rotation=tracker_theta,
    shading_row_rotation=tracker_theta,
    collector_width=width,
    pitch=pitch,
    cross_axis_slope=cross_axis_tilt,
)

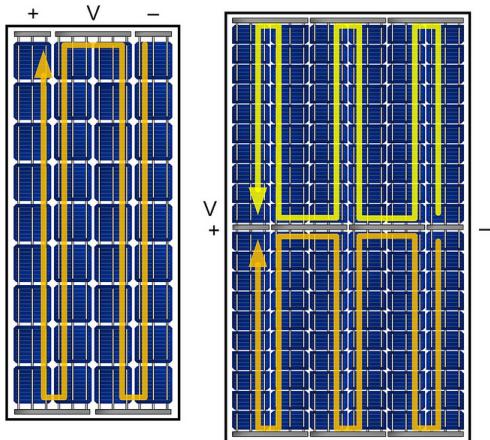
```

Number of shaded blocks

The number of shaded blocks depends on the module configuration and number of bypass diodes. For example, modules with one bypass diode will behave like one block. On the other hand, modules with three bypass diodes will have three blocks, except for the half-cut cell modules, which will have six blocks; 2x3 blocks where the two rows are along the longest side of the module. We can argue that the dimensions of the system change when you switch from portrait to landscape, but for this example, we will consider it the same.

The number of shaded blocks is calculated by rounding up the shaded fraction by the number of blocks along the shaded length. So let's define the number of blocks for each module configuration:

- 1 bypass diode: 1 block
- 3 bypass diodes: 3 blocks in landscape; 1 in portrait
- 3 bypass diodes half-cut cells: - 2 blocks in portrait - 3 blocks in landscape



Left: common module layout. Right: half-cut cells module layout. Each module has three bypass diodes. On the left, they connect cell columns 1-2, 2-3 & 3-4. On the right, they connect cell columns 1-2, 3-4 & 5-6. Source: César Domínguez. CC BY-SA 4.0, Wikimedia Commons

Anexo B: ejemplos de uso

In the image above, each orange U-shaped string section is a block. By symmetry, the yellow inverted-U's of the subcircuit are also blocks. For this reason, the half-cut cell modules have 6 blocks in total: two along the longest side and three along the shortest side.

```
blocks_per_module = {
    "1 bypass diode": 1,
    "3 bypass diodes": 3,
    "3 bypass diodes half-cut, portrait": 2,
    "3 bypass diodes half-cut, landscape": 3,
}

# Calculate the number of shaded blocks during the day
shaded_blocks_per_module = {
    k: np.ceil(blocks_N * shaded_fraction)
    for k, blocks_N in blocks_per_module.items()
}
```

Plane of array irradiance example data

To calculate the power output losses due to shading, we need the plane of array irradiance. For this example, we will use synthetic data:

```
clearsky = locus.get_clearsky(
    times, solar_position=solar_pos, model="ineichen"
)
dni_extra = pvlib.irradiance.get_extra_radiation(times)
airmass = pvlib.atmosphere.get_relative_airmass(solar_apparent_zenith)
sky_diffuse = pvlib.irradiance.perez_driesse(
    surface_tilt, surface_azimuth, clearsky["dhi"], clearsky["dni"],
    solar_apparent_zenith, solar_azimuth, dni_extra, airmass,
) # fmt: skip
poa_components = pvlib.irradiance.poa_components(
    aoi, clearsky["dni"], sky_diffuse, poa_ground_diffuse=0
) # ignore ground diffuse for brevity
poa_global, poa_direct = (
    poa_components["poa_global"],
    poa_components["poa_direct"],
)
```

Results

Now that we have the number of shaded blocks for each module configuration, we can apply the model and estimate the power loss due to shading.

Note that this model is not linear with the shaded blocks ratio, so there is a difference between applying it to just a module or a whole row.

```
shade_losses_per_module = {
    k: pvlib.shading.direct_martinez(
        poa_global=poa_global,
        poa_direct=poa_direct,
        shaded_fraction=shaded_fraction,
        shaded_blocks=module_shaded_blocks,
        total_blocks=blocks_per_module[k],
    )
    for k, module_shaded_blocks in shaded_blocks_per_module.items()
}

shade_losses_per_row = {
    k: pvlib.shading.direct_martinez(
        poa_global=poa_global,
        poa_direct=poa_direct,
        shaded_fraction=shaded_fraction,
        shaded_blocks=module_shaded_blocks * N_modules_per_row,
        total_blocks=blocks_per_module[k] * N_modules_per_row,
    )
    for k, module_shaded_blocks in shaded_blocks_per_module.items()
}
```

B.3. Pérdidas por sombras en módulos con diodos de bypass

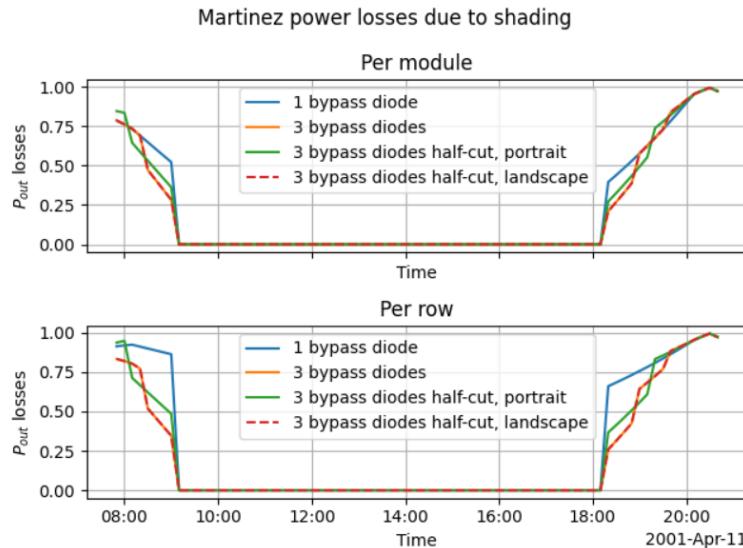
Plotting the results

```

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
fig.suptitle("Martinez power losses due to shading")
for k, shade_losses in shade_losses_per_module.items():
    linestyle = "--" if k == "3 bypass diodes half-cut, landscape" else "-"
    ax1.plot(times, shade_losses, label=k, linestyle=linestyle)
ax1.legend(loc="upper center")
ax1.grid()
ax1.set_xlabel("Time")
ax1.xaxis.set_major_formatter(
    ConciseDateFormatter("%H:%M", tz="Europe/Madrid"))
)
ax1.set_ylabel(r"$P_{out}$ losses")
ax1.set_title("Per module")

for k, shade_losses in shade_losses_per_row.items():
    linestyle = "--" if k == "3 bypass diodes half-cut, landscape" else "-"
    ax2.plot(times, shade_losses, label=k, linestyle=linestyle)
ax2.legend(loc="upper center")
ax2.grid()
ax2.set_xlabel("Time")
ax2.xaxis.set_major_formatter(
    ConciseDateFormatter("%H:%M", tz="Europe/Madrid"))
)
ax2.set_ylabel(r"$P_{out}$ losses")
ax2.set_title("Per row")
fig.tight_layout()
fig.show()

```



Note how the half-cut cell module in portrait performs better than the normal module with three bypass diodes. This is because the number of shaded blocks is less along the shaded length is higher in the half-cut module. This is the reason why half-cut cell modules are preferred in portrait orientation.

References

- [1](1,2) F. Martínez-Moreno, J. Muñoz, and E. Lorenzo, 'Experimental model to estimate shading losses on PV arrays', Solar Energy Materials and Solar Cells, vol. 94, no. 12, pp. 2298-2303, Dec. 2010, DOI: [10.1016/j.solmat.2010.07.029](https://doi.org/10.1016/j.solmat.2010.07.029).

Anexo B: ejemplos de uso

B.4. Fracción difusa de radiación fotosintetizable diaria

Pull Request: #2048, acceso a ejemplo en este link.

Calculating daily diffuse PAR using Spitter's relationship

This example demonstrates how to calculate the diffuse photosynthetically active radiation (PAR) from diffuse fraction of broadband insolation.

The photosynthetically active radiation (PAR) is a key metric in quantifying the photosynthesis process of plants. As with broadband irradiance, PAR can be divided into direct and diffuse components. The diffuse fraction of PAR with respect to the total PAR is important in agrivoltaic systems, where crops are grown under solar panels. The diffuse fraction of PAR can be calculated using the Spitter's relationship [1] implemented in `diffuse_par_spitters()`. This model requires the average daily solar zenith angle and the daily fraction of the broadband insolation that is diffuse as inputs.

Note

Understanding the distinction between the broadband insolation and the PAR is a key concept. Broadband insolation is the total amount of solar energy that gets to a surface, often used in PV applications, while PAR is a measurement of a narrower spectrum of wavelengths that are involved in photosynthesis. See section on *Photosynthetically Active insolation* in pp. 222-223 of [1].

References

[1]([1](#),[2](#),[3](#)) C. J. T. Spitters, H. A. J. M. Toussaint, and J. Goudriaan, 'Separating the diffuse and direct component of global radiation and its implications for modeling canopy photosynthesis Part I. Components of incoming radiation', Agricultural and Forest Meteorology, vol. 38, no. 1, pp. 217-229, Oct. 1986, DOI: [10.1016/0168-1923\(86\)90060-2](https://doi.org/10.1016/0168-1923(86)90060-2).

Read some example data

Let's read some weather data from a TMY3 file and calculate the solar position.

```
import pvlib
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.dates import AutoDateLocator, ConciseDateFormatter
from pathlib import Path

# Datafile found in the pvlib distribution
DATA_FILE = Path(pvlib.__path__[0]).joinpath("data", "723170TYA.CSV")

tmy, metadata = pvlib.iotools.read_tmy3(
    DATA_FILE, coerce_year=2002, map_variables=True
)
tmy = tmy.filter(
    ["ghi", "dni", "dni", "pressure", "temp_air"]
) # remaining columns are not needed
tmy = tmy["2002-09-06":"2002-09-21"] # select some days

solar_position = pvlib.solarposition.get_solarposition(
    # TMY timestamp is at end of hour, so shift to center of interval
    tmy.index.shift(freq="-30T"),
    latitude=metadata["latitude"],
    longitude=metadata["longitude"],
    altitude=metadata["altitude"],
    pressure=tmy["pressure"] * 100, # convert from millibar to Pa
    temperature=tmy["temp_air"],
)
solar_position.index = tmy.index # reset index to end of the hour
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pvlib-python/checkouts/latest/docs/examples
tmy.index.shift(freq="-30T"),
```

```
< >
```

B.4. Fracción difusa de radiación fotosintetizable diaria

Calculate daily values

The daily average solar zenith angle and the daily diffuse fraction of broadband insolation are calculated as follows:

```
daily_solar_zenith = solar_position["zenith"].resample("D").mean()
# integration over the day with a time step of 1 hour
daily_tmy = tmy[["ghi", "dhi"]].resample("D").sum() * 1
daily_tmy["diffuse_fraction"] = daily_tmy["dhi"] / daily_tmy["ghi"]
```

Calculate Photosynthetically Active Radiation

The total PAR can be approximated as 0.50 times the broadband horizontal insolation (integral of GHI) for an average solar elevation higher than 10°. See section on *Photosynthetically Active Radiation* in pp. 222-223 of [1].

```
par = pd.DataFrame({"total": 0.50 * daily_tmy["ghi"]}, index=daily_tmy.index)
if daily_solar_zenith.min() < 10:
    raise ValueError(
        "The total PAR can't be assumed to be half the broadband insolation "
        + "for average zenith angles lower than 10°."
    )

# Calculate broadband insolation diffuse fraction, input of the Spitter's model
daily_tmy["diffuse_fraction"] = daily_tmy["dhi"] / daily_tmy["ghi"]

# Calculate diffuse PAR fraction using Spitter's relationship
par["diffuse_fraction"] = pvlib.irradiance.diffuse_par_spitters(
    solar_position["zenith"], daily_tmy["diffuse_fraction"]
)

# Finally, calculate the diffuse PAR
par["diffuse"] = par["total"] * par["diffuse_fraction"]
```

Plot the results

Insolation on left axis, diffuse fraction on right axis

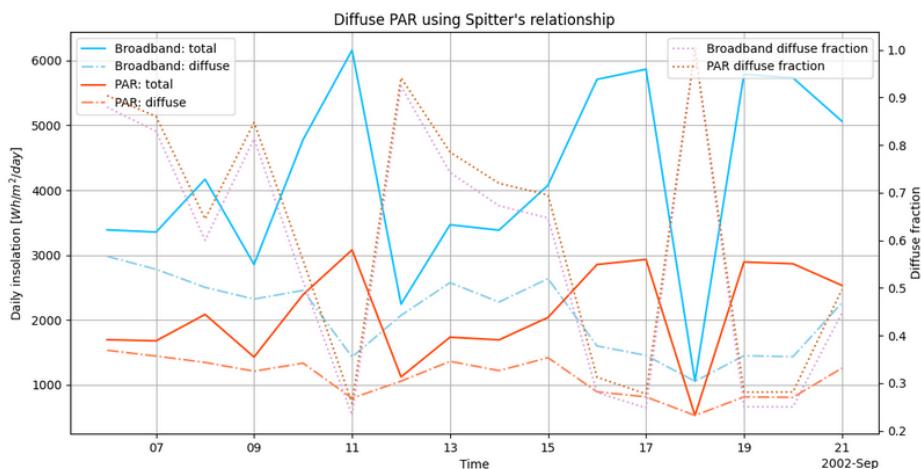
```
fig, ax_l = plt.subplots(figsize=(12, 6))
ax_l.set(
    xlabel="Time",
    ylabel="Daily insolation $[Wh/m^2/day]$",
    title="Diffuse PAR using Spitter's relationship",
)
ax_l.xaxis.set_major_formatter(
    ConciseDateFormatter(AutoDateLocator(), tz=daily_tmy.index.tz)
)
ax_l.plot(
    daily_tmy.index,
    daily_tmy["ghi"],
    label="Broadband: total",
    color="deepskyblue",
)
ax_l.plot(
    daily_tmy.index,
    daily_tmy["dhi"],
    label="Broadband: diffuse",
    color="skyblue",
    linestyle="--",
)
```

Anexo B: ejemplos de uso

```
ax_l.plot(daily_tmy.index, par["total"], label="PAR: total", color="orangered")
ax_l.plot(
    daily_tmy.index,
    par["diffuse"],
    label="PAR: diffuse",
    color="coral",
    linestyle="--",
)
ax_l.grid()
ax_l.legend(loc="upper left")

ax_r = ax_l.twinx()
ax_r.set(ylabel="Diffuse fraction")
ax_r.plot(
    daily_tmy.index,
    daily_tmy["diffuse_fraction"],
    label="Broadband diffuse fraction",
    color="plum",
    linestyle=":",
)
ax_r.plot(
    daily_tmy.index,
    par["diffuse_fraction"],
    label="PAR diffuse fraction",
    color="chocolate",
    linestyle=":",
)
ax_r.legend(loc="upper right")

plt.show()
```



Total running time of the script: (0 minutes 0.478 seconds)

[Download Jupyter notebook: `plot_diffuse_PAR_Spitters_relationship.ipynb`](#)

[Download Python source code: `plot_diffuse_PAR_Spitters_relationship.py`](#)

[Gallery generated by Sphinx-Gallery](#)

B.5. Modelo de pérdidas por irradiancia no uniforme

Pull Request: #2046, acceso a ejemplo en este link.

Plot Irradiance Non-uniformity Loss

Calculate the DC power lost to irradiance non-uniformity in a bifacial PV array.

The incident irradiance on the backside of a bifacial PV module is not uniform due to neighboring rows, the ground albedo, and site conditions. When each cell works at different irradiance levels, the power produced by the module is less than the sum of the power produced by each cell since the maximum power point (MPP) of each cell is different, but cells connected in series will operate at the same current. This is known as irradiance non-uniformity loss.

Calculating the IV curve of each cell and then matching the working point of the whole module is computationally expensive, so a simple model to account for this loss is of interest. Deline et al. [1] proposed a model based on the Relative Mean Absolute Difference (RMAD) of the irradiance of each cell. They considered the standard deviation of the cells' irradiances, but they found that the RMAD was a better predictor of the mismatch loss.

This example demonstrates how to model the irradiance non-uniformity loss from the irradiance levels of each cell in a PV module.

The function `pvlib.bifacial.power_mismatch_deline()` is used to transform the Relative Mean Absolute Difference (RMAD) of the irradiance into a power loss mismatch. Down below you will find a numpy-based implementation of the RMAD function.

References

- [1][1,2,3] C. Deline, S. Ayala Pelaez, S. MacAlpine, and C. Olalla, 'Estimating and parameterizing mismatch power loss in bifacial photovoltaic systems', Progress in Photovoltaics: Research and Applications, vol. 28, no. 7, pp. 691-703, 2020, DOI: 10.1002/pip.3259.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.cm import ScalarMappable
from matplotlib.colors import Normalize
from pvlib.bifacial import power_mismatch_deline
```

Problem description

Let's set a fixed irradiance to each cell row of the PV array with the values described in Figure 1 (A), [1]. We will cover this case for educational purposes, although it can be achieved with the packages solarfactors and bifacial_radiance.

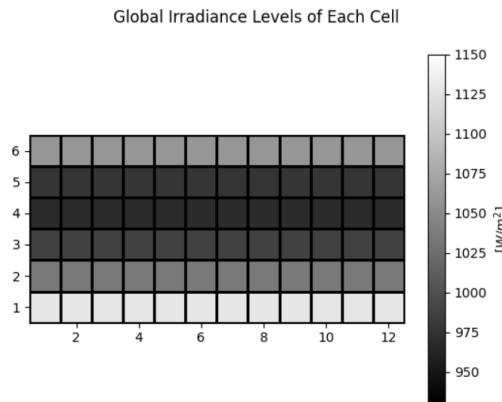
Here we set and plot the global irradiance level of each cell.

```
x = np.arange(12, 0, -1)
y = np.arange(6, 0, -1)
cells_irrad = np.repeat([1059, 976, 967, 986, 1034, 1128], len(x)).reshape(
    len(y), len(x))

color_map = "gray"
color_norm = Normalize(930, 1150)

fig, ax = plt.subplots()
fig.suptitle("Global Irradiance Levels of Each Cell")
fig.colorbar(
    ScalarMappable(cmap=color_map, norm=color_norm),
    ax=ax,
    orientation="vertical",
    label="$[W/m^2]$",
)
ax.set_aspect("equal")
ax.pcolormesh(
    x,
    y,
    cells_irrad,
    shading="nearest",
    edgecolors="black",
    cmap=color_map,
    norm=color_norm,
)
```

Anexo B: ejemplos de uso



Out:

```
<matplotlib.collections.QuadMesh object at 0x7f089703a090>
```

Relative Mean Absolute Difference

Calculate the Relative Mean Absolute Difference (RMAD) of the cells' irradiances with the following function, Eq. (4) of [1]:

$$\Delta[\text{unitless}] = \frac{1}{n^2 \bar{G}_{\text{total}}} \sum_{i=1}^n \sum_{j=1}^n |G_{\text{total},i} - G_{\text{total},j}|$$

```
def rmad(data, axis=None):
    """
    Relative Mean Absolute Difference. Output is [Unitless].
    https://stackoverflow.com/a/19472336/19371110
    """
    mean = np.mean(data, axis)
    mad = np.mean(np.abs(data - mean), axis)
    return mad / mean

rmad_cells = rmad(cells_irrad)

# this is the same as a column's RMAD!
print(rmad_cells == rmad(cells_irrad[:, 0]))
```

Out:

```
True
```

Mismatch Loss

Calculate the power loss ratio due to the irradiance non-uniformity with `pvlib.bifacial.power_mismatch_deline()`.

```
mismatch_loss = power_mismatch_deline(rmad_cells)

print(f"RMAD of the cells' irradiance: {rmad_cells:.3} [unitless]")
print(
    "Power loss due to the irradiance non-uniformity: "
    + f"{mismatch_loss:.3} [unitless]"
)
```

Out:

```
RMAD of the cells' irradiance: 0.0475 [unitless]
Power loss due to the irradiance non-uniformity: 0.014 [unitless]
```

B.6. Espectro estándar ASTM G173-03

Pull Request: #1963, acceso a ejemplo en este link.

ASTM G173-03 Standard Spectrum

This example demonstrates how to read the data from the ASTM G173-03 standard spectrum bundled with pvlib and plot each of the components.

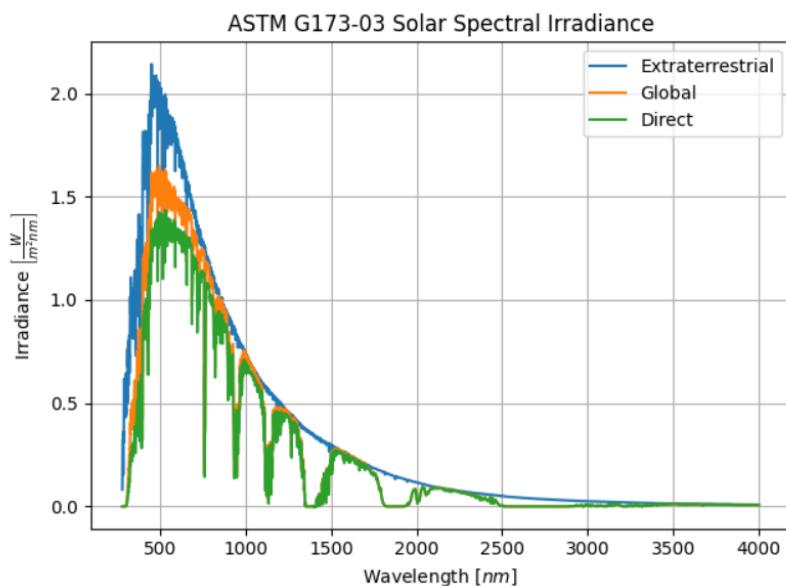
The ASTM G173-03 standard provides reference solar spectral irradiance data.

```
import matplotlib.pyplot as plt
import pvlib
```

Use `pvlib.spectrum.get_reference_spectra()` to read a spectra dataset bundled with pvlib.

```
am15 = pvlib.spectrum.get_reference_spectra(standard="ASTM G173-03")

# Plot
plt.plot(am15.index, am15["extraterrestrial"], label="Extraterrestrial")
plt.plot(am15.index, am15["global"], label="Global")
plt.plot(am15.index, am15["direct"], label="Direct")
plt.xlabel("Wavelength [nm]")
plt.ylabel("Irradiance $\left[\frac{W}{m^2\ nm}\right]$")
plt.title("ASTM G173-03 Solar Spectral Irradiance")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.155 seconds)

[Download Jupyter notebook: plot_standard_ASTM_G173-03.ipynb](#)

[Download Python source code: plot_standard_ASTM_G173-03.py](#)

B.7. Cálculo geométrico de sombras en 3D

Propuesta en [Pull Request: #2106](#), no aceptada, pero disponible temporalmente en [un ejemplo de sombra estático y una animación de la sombra sobre un campo de cultivo](#).

Spatial shading of a row-to-row shading system

This example demonstrates how to calculate the shading between two rows of panels in a row-to-row shading system. The example will show how to calculate the shaded fraction in 3D and 2D space with the help of the `spatial` module and its classes.

This is a basic example on how to calculate and plot the shaded fraction for an instantaneous time. A more complex task is to calculate the shadows for a time range. This can be done by iterating over the time range and calculating the shadows for each time step. This is done since the `FlatSurface` does not support the calculation of the shaded fraction for a time range. The example `sphx_glr_gallery_plot_par_direct_shading0_fixed_tilt.py` shows how to calculate the shading for a time range for a fixed tilt system.

```
from pvlib.spatial import RectangularSurface
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import shapely
```

Description of the system

Let's start by creating a row-to-row system. We will create a rectangular surface for each of the two rows of panels. Both rows will be parallel to each other and will have the same azimuth angle. The tilt angle of the first row will be 20 degrees, and the tilt angle of the second row will be 30 degrees. The distance between the two rows will be 3 meters. Also, we will assume scalar values for the solar azimuth and zenith angles. Feel free to download the example and change the values to see how the shades change.

```
solar_azimuth = 165 # degrees
solar_zenith = 75 # degrees

row1 = RectangularSurface( # south-most row
    center=[0, 0, 3], azimuth=165, tilt=20, axis_tilt=10, width=2, length=20
)

row2 = RectangularSurface( # north-most row
    center=[0, 3, 3], azimuth=165, tilt=30, axis_tilt=10, width=2, length=20
)
```

Calculating the shadows

The 3D shapely polygons representing the shadows can be calculated with the `get_3D_shades_from()` method. The 2D shapely polygons representing the shadows can be calculated with the `get_2D_shades_from()` method. This method uses either the 3D shadows or calculates them internally if not provided. If the 3D shadows are needed outside its scope, it is recommended to calculate them separately and pass them as an argument for performance reasons.

```
shades_3d = row2.get_3D_shades_from(solar_zenith, solar_azimuth, row1)
shades_2d = row2.get_2D_shades_from(
    solar_zenith, solar_azimuth, shades_3d=shades_3d
)
```

Scene and shades plot

```

row_style = {"color": "darkblue", "alpha": 0.5}
shade_style = {"color": "dimgrey", "alpha": 0.8}
row_style_2d = {**row_style, "add_points": False}
shade_style_2d = {**shade_style, "add_points": False}

fig = plt.figure(figsize=(10, 10))

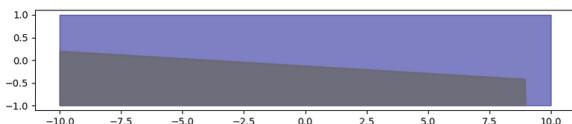
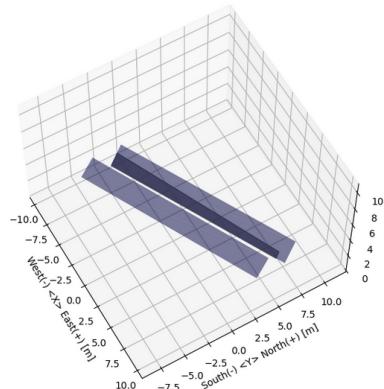
# Split the figure in two axes
gs = fig.add_gridspec(10, 1)
ax1 = fig.add_subplot(gs[0:7, 0], projection="3d")
ax2 = fig.add_subplot(gs[8:, 0])

# 3D plot
ax1.view_init(
    elev=60,
    azim=-30, # matplotlib's azimuth is right-handed to Z+, measured from X+
)
row1.plot(ax=ax1, **row_style)
row2.plot(ax=ax1, **row_style)
for shade in shades_3d.geoms:
    if shade.is_empty:
        continue # skip empty shades; else an exception will be raised
    # use Matplotlib's Poly3DCollection natively since experimental
    # shapely.plotting.plot_polygon does not support 3D
    vertexes = shade.exterior.coords[:, 1]
    ax1.add_collection3d(Poly3DCollection([vertexes], **shade_style))

ax1.axis("equal")
ax1.set_zlim(0)
ax1.set_xlabel("West(-) <x> East(+) [m]")
ax1.set_ylabel("South(-) <y> North(+) [m]")

# 2D plot
row2_2d = row2.representation_in_2d_space()
shapely.plotting.plot_polygon(row2_2d, ax=ax2, **row_style_2d)
for shade in shades_2d.geoms:
    shapely.plotting.plot_polygon(shade, ax=ax2, **shade_style_2d)

```



Calculate the shaded fraction

The shaded fraction can be calculated by dividing the sum of the areas of the shadows by the area of the surface.
The shaded fraction is a scalar value.

```

shaded_fraction = sum(shade.area for shade in shades_2d.geoms) / row2_2d.area
print(f"The shaded fraction is {shaded_fraction:.2f}")

```

Out: The shaded fraction is 0.42

Anexo B: ejemplos de uso

Fixed Tilt System impact on crop shading

This example shows how to calculate the shading of a crop field by a fixed tilt system.

This is the first of a series of examples that will show how to calculate the shading of a crop field by a fixed tilt system, a single-axis tracker, and a two-axis tracker. The examples will show how to calculate the shading in 3D and 2D space, and how to calculate the shading fraction with the help of the `spatial` module and its classes.

Paper Examples

Input parameters	Vertical	Single-axis	Two-axis	Units
Panel width	1	1	1	[m]
Panel length	2	2	2	[m]
Number of panels	40	40	40	[\cdot]
Total panel area	80	80	80	[m^2]
Number of rows	2	2	2	[\cdot]
Row spacing	10	10	10	[m]
Row length	20	20	20	[m]
Crop area	200	200	200	[m^2]
Pitch	-	-	2	[m]
Height	0	3	3	[m]
Fixed tilt angle	90	-	-	[$^\circ$]
Azimuth angle	0	0	0	[$^\circ$]
Maximum tilt angle	-	60	60	[$^\circ$]
Minimum tilt angle	-	-60	-60	[$^\circ$]

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import matplotlib.animation as animation
from matplotlib.dates import DateFormatter
import shapely
try:
    from shapely import Polygon # shapely >= 2
except ImportError:
    from shapely.geometry import Polygon # shapely < 2
import pandas as pd
import numpy as np
from functools import partial
from pvlib.spatial import RectangularSurface
from pvlib import solarposition

# Kärrbo Prästgård, Västerås, Sweden
latitude, longitude, altitude = 59.6099, 16.5448, 20 # °N, °E, m

spring_equinox = pd.date_range("2021-03-20", periods=24, freq="H")
summer_solstice = pd.date_range("2021-06-21", periods=24, freq="H")
fall_equinox = pd.date_range("2021-09-22", periods=24, freq="H")
winter_solstice = pd.date_range("2021-12-21", periods=24, freq="H")
dates = (
    spring_equinox.union(summer_solstice)
    .union(fall_equinox)
    .union(winter_solstice)
)
# dates = spring_equinox
solar_position = solarposition.get_solarposition(
    dates, latitude, longitude, altitude
)
solar Zenith = solar_position["apparent zenith"]
solar azimuth = solar_position["azimuth"]
N = len(solar Zenith)
```

Fixed Tilt

The fixed tilt system is composed of a crop field and two vertical rows of PV panels. Rows are placed at the long sides of the field, with the panels facing east and west. The field is 20 m long and 10 m wide, and the panels are 2 m wide (height since they are vertical) and 20 m long.

```
field = RectangularSurface( # crops surface
    center=[5, 10, 0],
    azimuth=90,
    tilt=0,
    axis_tilt=0,
    width=10,
    length=20,
)
pv_rows = (
    RectangularSurface( # west-most row (lowest X-coordinate)
        center=[-10 / 2 + 5, 10, 2 / 2],
        azimuth=90,
        tilt=90,
        axis_tilt=0,
        width=2,
        length=20,
    ),
    RectangularSurface( # east-most row (highest X-coordinate)
        center=[10 / 2 + 5, 10, 2 / 2],
        azimuth=90,
        tilt=90,
        axis_tilt=0,
        width=2,
        length=20,
    ),
)
```

Run the simulation

The shading fraction is calculated at each instant in time, and the results are stored in the `fixed_tilt_shaded_fraction` array. This is done because the shading calculation API does not allow for vectorized calculations.

```
# Allocate space for the shading results
fixed_tilt_shaded_fraction = np.zeros((N,), dtype=float)

# Shades callback
def simulation_and_plot_callback(
    timestamp_index, *, shade_3d_artists, shade_2d_artists, sun_annotation
):
    # Calculate the shades at an specific instant in time
    solar zenith_instant = solar zenith.iloc[timestamp_index]
    solar azimuth_instant = solar azimuth.iloc[timestamp_index]
    # Update the sun position text
    sun_annotation.set_text(
        f"Sun at y={solar zenith_instant:.2f}, β={solar azimuth_instant:.2f}"
    )
    # skip this instant if the sun is below the horizon
    if solar zenith_instant < 0:
        fixed_tilt_shaded_fraction[timestamp_index] = 0
        return *shade_3d_artists, *shade_2d_artists
    # Calculate the shades, both in 3D and 2D
    shades_3d = field.get_3D_shades_from(
        solar zenith_instant, solar azimuth_instant, *pv_rows
    )
    shades_2d = field.get_2D_shades_from(
        solar zenith_instant, solar azimuth_instant, shades_3d=shades_3d
    )
    # Plot the calculated shades
    for index, shade in enumerate(shades_3d.geoms): # 3D
        if not shade.is_empty:
            shade_3d_artists[index].set_verts(
                [shade.exterior.coords],
                closed=False, # polygon is already closed
            )
    for index, shade in enumerate(shades_2d.geoms): # 2D
        if not shade.is_empty:
            shade_2d_artists[index].set_path(
                shapely plotting._path_from_polygon(shade)
            )

    # Calculate the shaded fraction
    fixed_tilt_shaded_fraction[timestamp_index] = (
        sum(shade.area for shade in shades_2d.geoms) / field2d.area
    )
return *shade_3d_artists, *shade_2d_artists
```

Anexo B: ejemplos de uso

Plot both the 3D and 2D shades

```

fig = plt.figure(figsize=(10, 10))
gs = fig.add_gridspec(10, 1)

# Plotting styles
field_style = {"color": "forestgreen", "alpha": 0.5}
row_style = {"color": "darkblue", "alpha": 0.5}
shade_style = {"color": "dimgray", "alpha": 0.8}
field_style_2d = {**field_style, "add_points": False}
shade_style_2d = {**shade_style, "add_points": False}

ax1 = fig.add_subplot(gs[0:6, 0], projection="3d")
ax2 = fig.add_subplot(gs[8:, 0])

# Upper plot, 3D
ax1.axis("equal")
ax1.view_init(
    elev=45,
    azim=-60, # matplotlib's azimuth is right-handed to Z+, measured from X+
)
ax1.set_xlim(-9, 15)
ax1.set ylim(0, 20)
ax1.set zlim(0, 10)
ax1.set_xlabel("West(-) <x> East(+)/[m]")
ax1.set_ylabel("South(-) <y> North(+)/[m]")
field.plot(ax=ax1, **field_style)
for pv_row in pv_rows:
    pv_row.plot(ax=ax1, **row_style)

# Lower plot, 2D
field2d = field.representation_in_2D_space()
shapely.plotting.plot_polygon(field2d, ax=ax2, **field_style_2d)

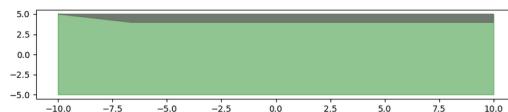
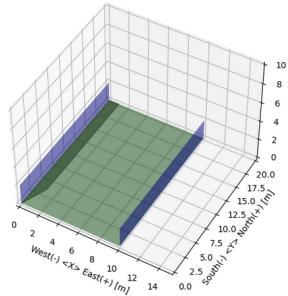
# Add empty shade artists for each shading object, in this case each of the
# PV rows. Artists will be updated in the animation callback later.
shade3d_artists = (
    ax1.add_collection3d(Poly3DCollection([], **shade_style)),
) * len(pv_rows)
shade2d_artists = (
    shapely.plotting.plot_polygon(
        Polygon([[0, 0]] * 4), ax=ax2, **shade_style_2d
    ),
) * len(pv_rows)
sun_text_artist = fig.text(0.5, 0.95, "Sun at y=--, β=--", ha="center")

ani = animation.FuncAnimation(
    fig,
    partial(
        simulation_and_plot_callback,
        shade_3d_artists=shade3d_artists,
        shade_2d_artists=shade2d_artists,
        sun_annotation=sun_text_artist,
    ),
    frames=np.arange(N),
    interval=200,
    blit=True,
)

# uncomment to run and save animation locally
# ani.save("fixed_tilt_shading.gif", writer="pillow")
plt.show()

```

Sun at $y=119.46, \beta=16.89$



Shaded Fraction vs. Time

Create a handy pandas series to plot the shaded fraction vs. time.

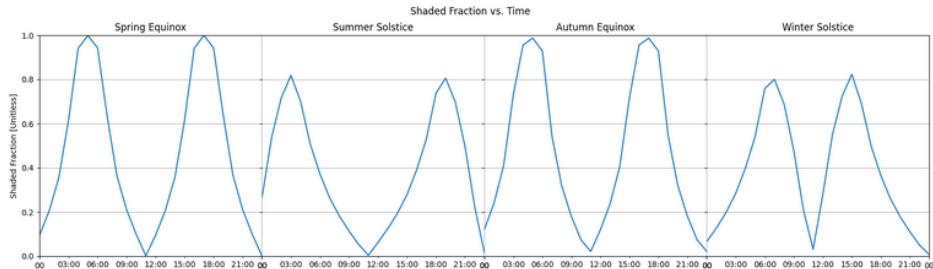
```
fixed_tilt_shaded_fraction = pd.Series(fixed_tilt_shaded_fraction, index=dates)

fig, axs = plt.subplots(ncols=4, sharey=True, figsize=(20, 5))
fig.suptitle("Shaded Fraction vs. Time")
fig.subplots_adjust(wspace=0)

for ax, day_datetimes, title in zip(
    axs,
    (spring_equinox, summer_solstice, fall_equinox, winter_solstice),
    ("Spring Equinox", "Summer Solstice", "Autumn Equinox", "Winter Solstice"),
):
    fixed_tilt_shaded_fraction[day_datetimes].plot(ax=ax)
    ax.xaxis.set_major_formatter(DateFormatter("%H"))
    ax.set_title(title)
    ax.grid(True)
for ax_a, ax_b in zip(axs[:-1], axs[1:]):
    ax_a.spines.right.set_visible(False)
    ax_b.spines.left.set_visible(False)
    ax_a.tick_params(labelright=False)
    ax_b.tick_params(labelleft=False)

axs[0].set_ylabel("Shaded Fraction [Unitless]")
axs[0].set_ylim(0, 1)

plt.show()
```



References

- 1 S. Zainali et al., 'Direct and diffuse shading factors modelling for the most representative agrivoltaic system layouts', Applied Energy, vol. 339, p. 120981, Jun. 2023, DOI: [10.1016/j.apenergy.2023.120981](https://doi.org/10.1016/j.apenergy.2023.120981).
- 2 Y. Cascone, V. Corrado, and V. Serra, 'Calculation procedure of the shading factor under complex boundary conditions', Solar Energy, vol. 85, no. 10, pp. 2524-2539, Oct. 2011, DOI: [10.1016/j.solener.2011.07.011](https://doi.org/10.1016/j.solener.2011.07.011).
- 3 Kevin S. Anderson, Adam R. Jensen; Shaded fraction and backtracking in single-axis trackers on rolling terrain. J. Renewable Sustainable Energy 1 March 2024; 16 (2): 023504. DOI: [10.1063/5.0202220](https://doi.org/10.1063/5.0202220).