import cv2  
from pvlib import pvsystem, singlediode  
import pandas as pd  
import numpy as np  
from scipy.interpolate import interp1d  
import matplotlib.pyplot as plt  
import math  
  
from scipy.constants import e as qe, k as kB

# Für alle Berechungungen wurden 25°C Zelltemperatur angenommen  
# kB ist J/K, qe ist C=J/V  
# kB \* T / qe -> V  
Vth = kB \* (273.15+25) / qe  
  
cell\_parameters = {  
 'I\_L\_ref': 18,  
 'I\_o\_ref': 5.16e-9,  
 'R\_sh\_ref': 1000,  
 'a\_ref': 1.3\*Vth,  
 'R\_s': 0.00181,  
 'alpha\_sc': 0.0042,  
 'breakdown\_factor': 2e-3,  
 'breakdown\_exp': 3,  
 'breakdown\_voltage': -15,  
}

def plot\_curves\_current(dfs, labels, title):  
 """plot the forward- and reverse-bias portions of an IV curve"""  
 fig, axes = plt.subplots(figsize=(5, 5))  
 for df, label in zip(dfs, labels):  
 df.plot('v', 'i', label=label, ax=axes)  
 axes.set\_xlim([0, df['v'].max()\*1.5])  
 axes.set\_ylim([0, 20])  
 axes.set\_ylabel('Strom [A]')  
 axes.set\_xlabel('Spannung [V]')  
 fig.suptitle(title)  
 fig.tight\_layout()  
 return axes

def plot\_curves\_power(dfs, labels, title):  
 """plot the forward- and reverse-bias portions of an PV curve"""  
 fig, axes = plt.subplots(figsize=(5, 5))  
 for df, label in zip(dfs, labels):  
 df.plot('v', 'p', label=label, ax=axes)  
 axes.set\_xlim([0, df['v'].max()\*1.5])  
 axes.set\_ylim([0, 750])  
 axes.set\_ylabel('Leistung [W]')  
 axes.set\_xlabel('Spannung [V]')  
 fig.suptitle(title)  
 fig.tight\_layout()  
 return axes

def simulate\_full\_curves(parameters, Geffs, Tcell, ivcurve\_pnts=1000):  
 """  
 Use De Soto and Bishop to simulate a full IV curve with both  
 forward and reverse bias regions.  
 """  
 # adjust the reference parameters according to the operating  
 # conditions using the De Soto model:  
 curves = []  
 for Geff in Geffs:  
 sde\_args = pvsystem.calcparams\_desoto(  
 Geff,  
 Tcell,  
 alpha\_sc=parameters['alpha\_sc'],  
 a\_ref=parameters['a\_ref'],  
 I\_L\_ref=parameters['I\_L\_ref'],  
 I\_o\_ref=parameters['I\_o\_ref'],  
 R\_sh\_ref=parameters['R\_sh\_ref'],  
 R\_s=parameters['R\_s'],  
 )  
 # sde\_args has values:  
 # (photocurrent, saturation\_current, resistance\_series,  
 # resistance\_shunt, nNsVth)  
  
 # Use Bishop's method to calculate points on the IV curve with V ranging  
 # from the reverse breakdown voltage to open circuit  
 kwargs = {  
 'breakdown\_factor': parameters['breakdown\_factor'],  
 'breakdown\_exp': parameters['breakdown\_exp'],  
 'breakdown\_voltage': parameters['breakdown\_voltage'],  
 }  
 v\_oc = singlediode.bishop88\_v\_from\_i(  
 0.0, \*sde\_args, \*\*kwargs  
 )  
 # ideally would use some intelligent log-spacing to concentrate points  
 # around the forward- and reverse-bias knees, but this is good enough:  
 vd = np.linspace(0.99\*kwargs['breakdown\_voltage'], v\_oc, ivcurve\_pnts)  
  
 ivcurve\_i, ivcurve\_v, \_ = singlediode.bishop88(vd, \*sde\_args, \*\*kwargs)  
 curves.append(pd.DataFrame({  
 'i': ivcurve\_i,  
 'v': ivcurve\_v,  
 }))  
   
 return curves

# https://www.microsemi.com/document-portal/doc\_view/125066-rf01055-datasheet  
  
def bypass\_diode():  
 i\_s = 0.00012  
 t = 273.15 + 25  
 v = np.linspace(0,1,1000)  
  
 n = 1.446554  
  
 v\_t = kB \* t / qe  
  
 i = i\_s \* (np.exp(v/(v\_t \* n)) - 1)  
 return pd.DataFrame({'i': i, 'v': -v})

def interpolate\_v(df, i):  
 """convenience wrapper around scipy.interpolate.interp1d"""  
 f\_interp = interp1d(np.flipud(df['i']), np.flipud(df['v']), kind='linear',  
 fill\_value='extrapolate')  
 return f\_interp(i)

def interpolate\_i(df, v):  
 """convenience wrapper around scipy.interpolate.interp1d"""  
 f\_interp = interp1d(np.flipud(df['v']), np.flipud(df['i']), kind='linear',  
 fill\_value='extrapolate')  
 return f\_interp(v)

def assemble\_serial(curves):  
 i\_min = 0  
 i\_max = 0  
  
 for curve in curves:  
 i\_min = min(i\_min, curve['i'].min())  
 i\_max = max(i\_min, curve['i'].max())  
  
 i\_max = min(i\_max, 50)  
  
 i = np.linspace(i\_min, i\_max, 1000)  
  
 v = 0  
   
 for curve in curves:  
 v\_diode = interpolate\_v(curve, i)  
 v += v\_diode  
 return pd.DataFrame({'i': i, 'v': v})

def assemble\_parallel(curves):  
 v\_min = 0  
 v\_max = 0  
  
 for curve in curves:  
 v\_min = min(v\_min, curve['v'].min())  
 v\_max = max(v\_max, curve['v'].max())  
  
 v = np.linspace(v\_min, v\_max, 1000)  
  
 i = 0  
   
 for curve in curves:  
 i\_diode = interpolate\_i(curve, v)  
 i += i\_diode  
   
 return pd.DataFrame({'i': i, 'v': v})

def arr\_to\_module(arr):  
 rotated = np.flip(np.rot90(arr), axis=0)  
 combined = np.concatenate((arr[::2], np.flip(arr[1::2], axis=1)), axis=1)  
 return combined

def addPower(iv):  
 i = iv['i']  
 v = iv['v']  
 p = v \* i  
 return pd.DataFrame({'i': i, 'v': v, 'p': p})

def simulate\_module\_shaded(shadow\_arr, poa\_direct=800, poa\_diffuse=200, Tcell=25, diode\_every\_cell=False):  
   
 shadow\_module = arr\_to\_module(shadow\_arr)  
  
 iv\_strings = []  
 if diode\_every\_cell:  
 for string in shadow\_module:  
 cell\_curves = []  
 for curve in simulate\_full\_curves(cell\_parameters, string \* poa\_direct + poa\_diffuse, Tcell):  
 cell\_curves.append(assemble\_parallel([curve, bypass\_diode()]))  
 iv\_strings.append(assemble\_serial(cell\_curves))  
  
 else:  
 for string in shadow\_module:  
 iv\_strings.append(assemble\_parallel([assemble\_serial(simulate\_full\_curves(cell\_parameters, string \* poa\_direct + poa\_diffuse, Tcell)), bypass\_diode()]))  
  
 iv\_module = assemble\_serial(iv\_strings)  
 return addPower(iv\_module)

def partial\_shadow(sun\_radius):  
 arr = np.linspace(0, 2, sun\_radius \* 2 + 1)  
 arr = np.delete(arr + (arr[1] - arr[0]) / 2, -1)  
 shadow\_arr = (np.arccos(1 - arr) - (1 - arr) \* np.sqrt(2 \* arr - arr \*\* 2)) / np.pi  
  
 return shadow\_arr

def solar\_cell\_mask(cell\_size, distance, pix\_deg, sun\_radius):  
 cell\_x = cell\_size[0]  
 cell\_y = cell\_size[1]  
  
 projection\_x = int(2 \* math.degrees(math.atan(cell\_x / distance / 2)) \* pix\_deg)  
 projection\_y = int(2 \* math.degrees(math.atan(cell\_y / distance / 2)) \* pix\_deg)  
 projection = [projection\_x, projection\_y]  
   
 size\_x = projection\_x + 2 \* sun\_radius  
 size\_y = projection\_y + 2 \* sun\_radius  
  
 mask = np.ones((size\_x, size\_y))  
  
 shadow\_arr = partial\_shadow(sun\_radius)  
  
 mask[0:sun\_radius \* 2, :] \*= shadow\_arr[:, None] # top  
 mask[-sun\_radius \* 2:,:] \*= np.flip(shadow\_arr)[:, None] # bottom  
 mask[:, 0:sun\_radius \* 2] \*= shadow\_arr[None, :] # left  
 mask[:,-sun\_radius \* 2:] \*= np.flip(shadow\_arr)[None, :] # right  
  
 return projection, mask

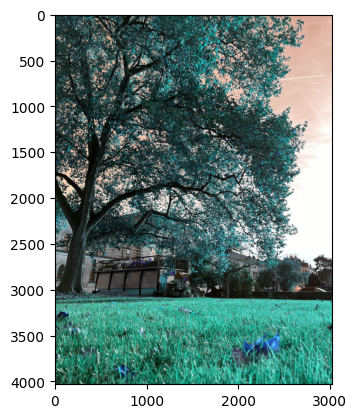
def get\_slices\_from\_coords(mask, mask\_coords):  
 res\_masks = []  
 for row in mask\_coords:  
 res\_row = []  
 for cell in row:  
 res\_row.append(mask[cell[0]:cell[2], cell[1]:cell[3]])  
 res\_masks.append(res\_row)  
 return np.array(res\_masks)

def solar\_module\_masks(cell\_size, shape, spacing, pos, mask, distance, pix\_deg, sun\_radius):  
  
 cell\_projection, cell\_mask = solar\_cell\_mask(cell\_size, distance, pix\_deg, sun\_radius)  
 spacing\_projection = np.int\_(np.multiply(cell\_projection, int(spacing \* 1000000)) / cell\_size / 1000000)  
 unit\_projection = cell\_projection + spacing\_projection  
  
 size = shape \* (unit\_projection)  
 start\_pos = pos - size / 2  
 cell\_mask\_size = np.array(cell\_mask.shape)  
  
 pos\_arr = np.array(np.meshgrid(np.arange(0.5, shape[0] + 0.5), np.arange(0.5, shape[1] + 0.5)))  
 pos\_arr = pos\_arr.T.reshape(shape[0],shape[1],2) \* unit\_projection + start\_pos  
  
 mask\_coords = np.int\_(np.concatenate((pos\_arr - cell\_mask\_size / 2, pos\_arr + cell\_mask\_size / 2), axis=2))  
  
 tree\_mask\_slices = get\_slices\_from\_coords(mask, mask\_coords) / 255  
  
 combined\_masks = tree\_mask\_slices \* cell\_mask  
  
 radiation\_on\_cells = (np.apply\_over\_axes(np.sum, combined\_masks, [2,3]) / np.sum(cell\_mask)).reshape(len(combined\_masks),-1)  
  
 return radiation\_on\_cells, combined\_masks, cell\_mask, mask\_coords

def visualize(mask, cell\_mask, mask\_coords, alpha):  
 beta = 1 - alpha  
  
 overlay = np.zeros\_like(mask, dtype=float)  
 for row in mask\_coords:  
 for coord in row:  
 overlay[coord[0]:coord[2], coord[1]:coord[3]] += cell\_mask  
   
 vis = cv2.addWeighted(np.float64(mask), alpha, (overlay \* 255), beta, 0.0)  
   
 return vis

filename = input("Filename: ")  
tree\_img = cv2.imread('measuredTrees/' + filename)  
plt.imshow(tree\_img)

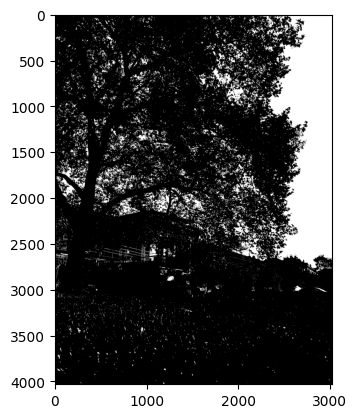
<matplotlib.image.AxesImage at 0x285edbc1060>



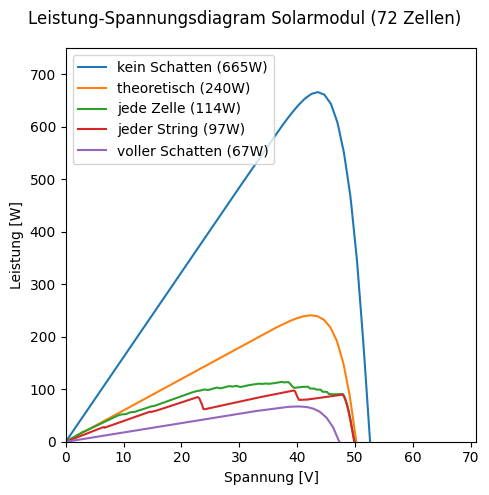
distance = 5  
pos\_az = 10  
  
variation = 200  
  
sun\_radius\_deg = 0.533  
  
res\_x = tree\_img.shape[1]  
res\_y = tree\_img.shape[0]  
pix\_deg = res\_x / 65  
  
sun\_radius = int(sun\_radius\_deg \* pix\_deg) // 2

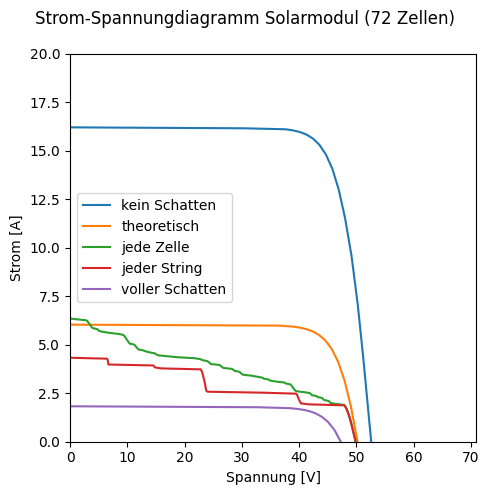
lower\_bound = np.array([170, 0, 0])  
upper\_bound = np.array([360, 255, 255])  
  
mask = cv2.inRange(tree\_img, lower\_bound, upper\_bound)  
  
#cv2.imshow('treemask', imagemask)  
plt.imshow(mask, cmap='gray')

<matplotlib.image.AxesImage at 0x285f448e710>



radiation\_on\_cells1, cells\_transmission1, cell\_mask1, mask\_coords1 = solar\_module\_masks([0.21, 0.21], [12,6], 0.04, [1550, 900], mask, 14, pix\_deg, sun\_radius)  
radiation\_on\_cells2, cells\_transmission2, cell\_mask2, mask\_coords2 = solar\_module\_masks([0.21, 0.21], [12,6], 0.04, [800, 2000], mask, 14, pix\_deg, sun\_radius)  
radiation\_on\_cells3, cells\_transmission3, cell\_mask3, mask\_coords3 = solar\_module\_masks([0.21, 0.21], [12,6], 0.04, [750, 900], mask, 14, pix\_deg, sun\_radius)  
  
radiation\_on\_cells = radiation\_on\_cells2  
  
averaged = np.ones\_like(radiation\_on\_cells) \* np.average(radiation\_on\_cells)  
  
iv\_module1 = simulate\_module\_shaded(radiation\_on\_cells, poa\_diffuse=100, diode\_every\_cell=True)  
iv\_module2 = simulate\_module\_shaded(radiation\_on\_cells, poa\_diffuse=100)  
iv\_module3 = simulate\_module\_shaded(np.ones\_like(radiation\_on\_cells), poa\_diffuse=100)  
iv\_module4 = simulate\_module\_shaded(np.zeros\_like(radiation\_on\_cells), poa\_diffuse=100)  
iv\_module5 = simulate\_module\_shaded(averaged, poa\_diffuse=100)  
  
p\_max1 = iv\_module1['p'].max()  
p\_max2 = iv\_module2['p'].max()  
p\_max3 = iv\_module3['p'].max()  
p\_max4 = iv\_module4['p'].max()  
p\_max5 = iv\_module5['p'].max()  
  
  
curves = [iv\_module3, iv\_module5, iv\_module1, iv\_module2, iv\_module4]  
labels\_p = ['kein Schatten (' + str(int(p\_max3)) + 'W)', 'theoretisch (' + str(int(p\_max5)) + 'W)', 'jede Zelle (' + str(int(p\_max1)) + 'W)', 'jeder String (' + str(int(p\_max2)) + 'W)', 'voller Schatten (' + str(int(p\_max4)) + 'W)']  
labels\_i = ['kein Schatten', 'theoretisch', 'jede Zelle', 'jeder String', 'voller Schatten']  
  
ax = plot\_curves\_power(curves, labels=labels\_p, title='Leistung-Spannungsdiagram Solarmodul (72 Zellen)')  
ax = plot\_curves\_test(curves, labels\_i, 'Strom-Spannungdiagramm Solarmodul (72 Zellen)')





radiation\_on\_cells = radiation\_on\_cells3  
  
res\_table = [["Diffuse Einstahlung [W/m^2]", "voller Schatten [W/m^2]", "Diode für jeden String [W/m^2]", "Diode für jede Zelle [W/m^2]", "direkte Einstahlung jeder String [W/m^2]", "direkte Einstahlung jede Zelle [W/m^2]", "Leistungsanstieg bei jede Zelle"]]  
for i, diffuse in enumerate(range(0, 225, 25)): #0-200  
 res\_row = [diffuse]  
 full\_shadow = simulate\_module\_shaded(np.zeros\_like(radiation\_on\_cells), poa\_diffuse=diffuse)['p'].max()  
 every\_string = simulate\_module\_shaded(radiation\_on\_cells, poa\_diffuse=diffuse)['p'].max()  
 every\_cell = simulate\_module\_shaded(radiation\_on\_cells, poa\_diffuse=diffuse, diode\_every\_cell=True)['p'].max()  
   
 direct\_every\_string = every\_string - full\_shadow  
 direct\_every\_cell = every\_cell - full\_shadow  
  
 direct\_power\_increase = direct\_every\_cell / direct\_every\_string - 1  
  
 res\_table.append([diffuse, full\_shadow, every\_string, every\_cell, direct\_every\_string, direct\_every\_cell, direct\_power\_increase])

df\_table = pd.DataFrame(res\_table)  
  
writer = pd.ExcelWriter('test.xlsx')  
  
df\_table.to\_excel(writer, sheet\_name='2', index=False)  
  
writer.save()

C:\Users\patri\AppData\Local\Temp\ipykernel\_31192\1808722512.py:7: FutureWarning: save is not part of the public API, usage can give unexpected results and will be removed in a future version  
 writer.save()

#vis\_filename = input('visualisation filename')  
#mask\_coords = [\*mask\_coords1, \*mask\_coords2, \*mask\_coords3]  
vis = visualize(mask, cell\_mask2, mask\_coords2, 0.4)  
plt.imshow(vis, cmap='gray')  
#cv2.imwrite('doc\_pictures/' + vis\_filename + 'png', vis)  
cv2.imwrite('test.png', vis)

True

