Python and MatPlotLib based Open Source Software System for Simulating Images with point Light Sources in Attenuating and Scattering Media

K. Nagesh
Adjunct Faculty,
CUTM, Parlakhemundi,
Gajapati district, Odisha, India

D. Nageswara Rao
Vice Chancellor,
CUTM, Parlakhemundi,
Gajapati district, Odisha, India

Song K. Choi
Professor,
University of Hawaii,
Honolulu, HI, USA

ABSTRACT
Simulating images of various objects in a real world scene has wide applications while testing algorithms for machine vision applications as well as in computer graphics and gaming software industry. Most current algorithms use collimated light sources and assume the medium to be non-scattering and non-attenuating. In real world, most light sources are too near to the objects in the scene and hence cannot be assumed to be collimated. Real world mediums, such as oceans in case of Underwater Robotic Vehicle (URV) applications or smoke and vapor filled air in case of industrial welding applications, scatter and attenuate light. A software system that makes no such assumptions and uses point light sources in scattering and attenuating media has been developed. Another novelty of current work is use of open source Python programming language along with associated 2D graphics and plotting library, MatPlotLib.

General Terms
Underwater Robotic Vehicles, Industrial Robots, Image simulation, Python, MatPlotLib.

Keywords
Point Light Sources, Attenuating Media, Scattering Media.

1. INTRODUCTION
While simulating images for various applications, it is common to assume that the light sources being used are collimated. However, very often, the light sources being used in practice, such as in weld seam inspection and underwater imaging applications, are actually point light sources. When these light sources are approximated as collimated light sources, it causes significant error in the recovered shape and 3D information of the object. The error increases further when the imaging process is undertaken in an attenuating and scattering media. Proposed software system makes no such assumptions and uses point light sources in scattering and attenuating media. Another novelty of current work is use of open source software.

2. MODEL
Consider a coordinate system as shown in Figure 1 in which the origin is located at the camera image plane and the z-axis points along the optical axis toward the object.

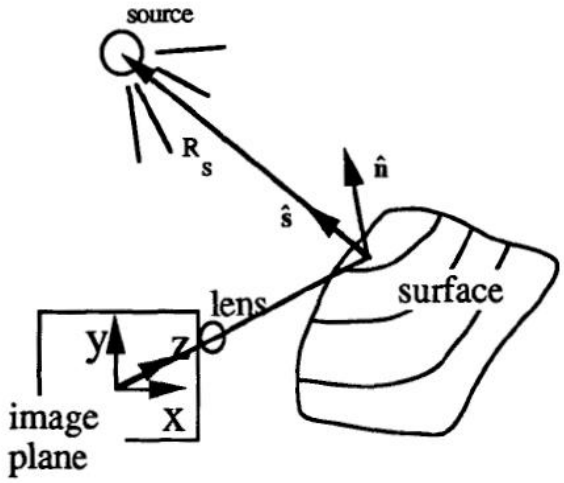

Figure 1. Coordinate system

Using radiometric terminology suggested by the U.S. National Bureau of Standards [2], we define the irradiance (E) as the incident radiant flux per unit area of the receiving surface. The radiant flux density (P) is the radiant power per unit area normal to the ray.

The irradiance (E) of a surface illuminated by a light source is related to the radiant flux density (P) at the surface by Equation 1.

$$E = P\,(\mathbf{n} \cdot \mathbf{s}) \tag{1}$$

Here **n** is the unit outward surface normal and **s** is a unit vector pointing from the surface toward the light source.

Following the procedure suggested by [3], it is convenient to substitute for the dot product in Equation 1 in terms of the gradient angles of the reflecting surface element.

The two unit vectors **n** and **s** can be specified by Equations 2 and 3 respectively.

$$\mathbf{n} = [-1, p, q]^T \tag{2}$$

$$\mathbf{s} = [-1, p_s, q_s]^T \tag{3}$$

Here p and q are the surface gradient components: dz/dx and dz/dy, and p_s and q_s are the surface gradient components of a plane normal to the vector **s**.

Equation 1 can be combined with Equations 2 and 3 and rewritten as Equation 4.

$$E = P\frac{(1 + pp_s + qq_s)}{\sqrt{(1+p^2+q^2)}\sqrt{(1+p_s^2+q_s^2)}} \quad (4)$$

If the surface is Lambertian, the image brightness (F) produced when observing the object from any direction is proportional to the irradiance (E). The constant of proportionality depends only on the reflectivity of the surface (ρ) and the optics of the imaging system. The dependence on the optics is generally ignored since it is specific to the imaging system and is easily obtained through calibration. Hence, the image brightness (F) is related to the irradiance (E) by Equation 5.

$$F = \rho E \quad (5)$$

Equations 5 can be combined with Equation 4 and rewritten as Equation 6.

$$F = \rho \ P\frac{(1 + pp_s + qq_s)}{\sqrt{(1+p^2+q^2)}\sqrt{(1+p_s^2+q_s^2)}} \quad (6)$$

If the direction (**s**) and radiant flux density (P) of the illumination are known at each point on the surface, three independent evaluations of Equation 6 for each surface element are sufficient to solve for the two unknown surface gradient components (p, q) and the unknown surface reflectance (ρ) at the surface element.

Once the surface gradient components (p, q) are computed for each surface element, shape and 3D depth (z) map of the object can be computed using Equation 7.

$$z = z_0 + \int (p\,dx + q\,dy) \quad (7)$$

Here z_0 is the depth of a known point on the object i.e. 'datum depth'.

[1] accomplishes this by assuming that the light source is collimated and uniform so that s and P are spatially invariant. Three images using same imaging system but three different light sources, as shown in Figure 2, provide the required three independent evaluations of Equation 6. This technique is known as 'photometric stereo'.

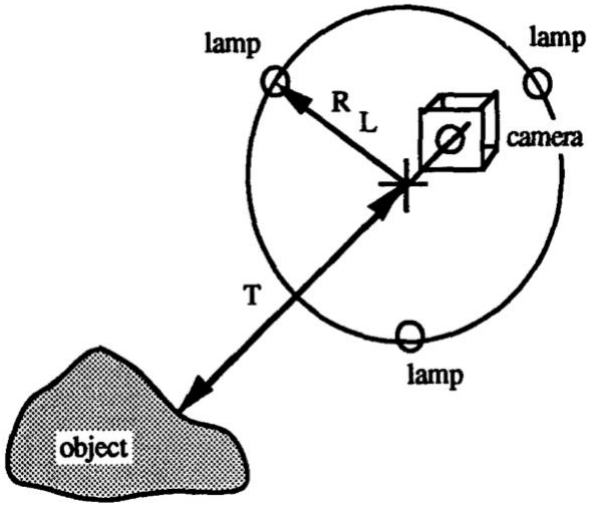

Figure 2. Light source arrangement

In practice this can be expected to work well as long as the difference in the depth between the nearest and furthest surface elements on the object being observed is small compared to the distance of the object from the light source.

In the case of a mobile robot carrying its own lamps, or cooperating robots trying to minimize backscatter into the camera [4], as in weld seam inspection and underwater applications, the light source is likely to be close to the objects being viewed. In such cases, the light sources are better approximated as point light sources. To assume they are collimated light sources causes significant error in the recovered shape and 3D information of the object. The error increases further when the imaging process is undertaken in an attenuating and scattering media.

2.1 Point Light Sources

When using point light sources, the local radiant flux density (P) can be expressed in terms of the distance from the point light source (Rs) as given in Equation 8.

$$P = \frac{I_0}{R_s^2} \quad (8)$$

Here I_0 is the radiant intensity of an isotropic point light source.

Since distance and direction to point light source (R_s) varies from one surface element to another, p_s and q_s themselves vary from one surface element to another. Hence, having three images is no longer sufficient to solve Equation 6 and compute p, q and ρ at each surface element.

2.2 Attenuating Media

In addition, if the operation takes place in a turbid or smoky medium, attenuation will be present so that the radiant flux density (P) will vary as a function of position for each of the surface elements on the object as described in Equation 9.

$$P = I_0 \frac{e^{-\frac{R_s}{\beta}}}{R_s^2} \quad (9)$$

Here β is the characteristic attenuation length.

2.3 Scattering Media

When the medium is also scattering, if we assume it is homogeneous and not highly dense, as in smoky welding environment or murky water, we can use single scattering model (Figure 3) [5].

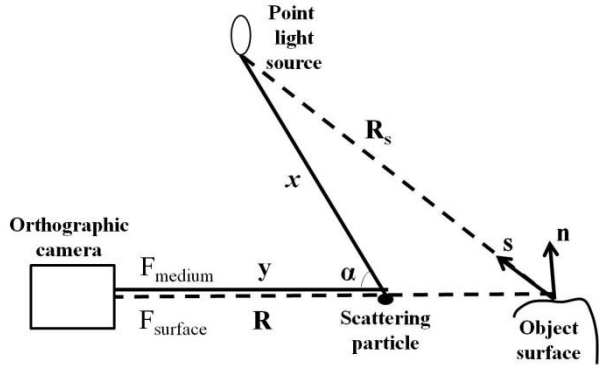

Figure 3. Single scattering model

Using single scattering model, the image brightness (F) due to a particle in the medium is given by Equation 10.

$$F_{medium} = \left(P\, e^{-\frac{x}{\beta}}\right)(\gamma\, S(g,\alpha))\left(e^{-\frac{y}{\beta}}\right) \quad (10)$$

Here γ is scattering coefficient i.e. fraction of the incident flux scattered by a unit volume of the medium in all directions, S(g,α) is phase function i.e. angular scattering distribution of the incident flux, x is the distance of the scattering particle from the point light source and y is the distance of the scattering particle from the origin of the coordinate system.

S(g,α) is assumed to be a smooth, low-order polynomial of cos α, where α is the angle between incident and scattered directions. Its first-order approximation is given by Equation 11.

$$S(g, \alpha) = \frac{1}{4\pi} (1 + g \cos \alpha) \quad (11)$$

Here g is the forward scattering parameter that controls the shape of the phase function and € (-1,1).

The total image brightness of a pixel corresponding to a surface element due to scattering by all particles is obtained by integrating Equation 10 along viewing direction to surface element from 0 to R.

This together with irradiance due to the surface element produces the final image brightness (F) as given by Equation 12.

$$F = F_{medium}\ \delta(x<R) + F_{surface\ element}\ \delta(x=R) \quad (12)$$

Here δ is Dirac delta function.

3. SOFTWARE PROGRAM

The software system was developed using Python 3 along with its associated 2D graphics and plotting package, MatPlotLib.

The coordinate system was defined with center of camera image plane as the origin of coordinate system. Center of the camera lens is at 'f' meters from origin along z axis. Center of object in the scene is at a distance of (z_0-f) meters from lens center further along z axis i.e. z_o meters from origin along z axis. Positive z-axis points along the optical axis towards the object in the scene. Object in the scene intersects z-axis at 'z_0' meters ('datum depth') from origin.

Focal length of the camera lens was assumed to be 0.02 meters. Distance of the object from lens was assumed to be 3.0 meters. Simulated image resolution was 201 pixels x 201 pixels, corresponding to its size of 2 meters x 2 meters. For simulating images, object data was inputted, followed by information about radiant intensity and various locations of the light source. Corresponding images were simulated using this data with equation 12. The core program is given below:

```
#-----------------
# Import libraries
#-----------------
# Support functions
from support import support_class
import matplotlib.pyplot as plt
# Numerical analysis
from numpy import mean
from numpy import median
from numpy import std
from numpy import corrcoef
from numpy import ones
from numpy import zeros
from numpy import float64
from numpy import arange
import math
#-----------------
# Main class
#-----------------
class main_class:
 def __init__(self):
  # Constants
  # DATA DIRECTORIES
  #MAC#
  BASE_FILE_DIR = '/Users/turiya/Desktop'
  SEP='/'
  #WINDOWS#
  #BASE_FILE_DIR = 'C:\\Users\\KV_DoLR\\Desktop'
  #SEP='\\'
  DATA_FOLDER='ips'
  INPUT_FILE_DIR=BASE_FILE_DIR+SEP+
DATA_FOLDER
  # INPUT and OUTPUT FILENAMES
   self.INPUT_FILE_1 = INPUT_FILE_DIR + SEP + '1.txt'
   #--------------------------------------------------
   # Coordinate system definition:
   #  Center of camera image plane: Origin of coordinate system
   #  Center of camera lens:  'f' meters from origin along z axis
   #  Center of object in the scene: Further (z0-f) meters from lens center along z axis
   #      zo meters from origin along z axis
   # Positive z-axis:
   #  Points along the optical axis towards the object in the scene
   # Object in the scene:
   #  Intersects z-axis at 'z0' meters ('datum depth') from origin
   #--------------------------------------------------
   # Focal length of the camera Lens
   self.f = 0.02     # in meters
   self.obj_dist_from_lens = 3.0                # in meters
   #--------------------------------------------------
   # Camera image plane
   #--------------------------------------------------
   # 201 pixels x 201 pixels (i.e. -100 (via 0) to 100)
   self.IMG_RES_X = 11
   self.IMG_RES_Y = 11
   # 0.0201 m x 0.0201 m (i.e. 2.01 cm x 2.01 cm)
   self.IMG_SIZE_X = 0.0201              # in meters
   self.IMG_SIZE_Y = 0.0201              # in meters
   self.IMG_RES_X_HALF = (self.IMG_RES_X - 1)/2
   self.IMG_RES_Y_HALF = (self.IMG_RES_Y - 1)/2
   self.MIN_I = 0
   self.MAX_I = (self.IMG_RES_X - 1)
   self.MIN_J = 0
   self.MAX_J = (self.IMG_RES_Y - 1)
   self.IMGPIXEL_SIZE_X=self.IMG_SIZE_X                /
self.IMG_RES_X  # 0.0001 meters
```

```
self.IMGPIXEL_SIZE_Y=self.IMG_SIZE_Y / self.IMG_RES_Y  # 0.0001 meters
  # Object in the scene
  # one image pixel = one object surface element
self.z0 = self.obj_dist_from_lens + self.f  # in meters
self.MAGNIFICATION = (self.z0 - self.f) / self.f
self.OBJELEM_SIZE_X=self.IMGPIXEL_SIZE_X * self.MAGNIFICATION
self.OBJELEM_SIZE_Y=self.IMGPIXEL_SIZE_Y * self.MAGNIFICATION
self.OBJ_SIZE_X=self.IMG_SIZE_X * self.MAGNIFICATION
self.OBJ_SIZE_Y=self.IMG_SIZE_Y * self.MAGNIFICATION
  # Light source
  self.Power = 60  # in Watts
  self.I0 = ( self.Power / (4 * 3.14) ) # in Watts per Steradian
  # Attenuating and scattering media
  #  Beta: Characteristic attenuation length
  #  Gamma: Scattering coefficient
  #  g: Forward scattering parameter: (-1,1)
  self.beta = 1000000.0  # negligible attenuation
  self.gamma = 0000001.0  # negligible scattering
  self.g = 0.0
  self.SCATTERING_STEP = 0.1
  # Get access to classes containing support functions
  self.class2 = support_class()
  # Lists of lists
  self.matrix_list = []
  # Lists
  self.list = []
  # Counts
  self.count = 0
 #---------------------------------------------------
 # Simulate images
 #---------------------------------------------------
 def simulate_images(self):
  # DEBUG_print to output messages based on DEBUG level
  self.class2.DEBUG_print(2, 'Staring to simulate images...')
  # Read object data:
  #  For each surface element of the object, get its:
  #   z, rho
  #   Read them into two 2-D arrays
  #   each of size IMG_RES_X, IMG_RES_Y
  object_z, object_rho = self.get_object_data_plane()
  #self.class2.DEBUG_print(2, 'object_z:')
  #self.class2.DEBUG_print_array(2, object_z)
  #self.class2.DEBUG_print(2, 'object_rho:')
  #self.class2.DEBUG_print_array(2, object_rho)
  # Get I0 (radiant intensity) and x, y, z (position) of the three isotropic point light sources
  lights_I0, lights_x, lights_y, lights_z = self.get_light_sources_data()
  # simulate three images, one for each of the three light sources
  image_1 = self.compute_image(object_z, object_rho, lights_I0[0], lights_x[0], lights_y[0], lights_z[0])
  image_2 = self.compute_image(object_z, object_rho, lights_I0[1], lights_x[1], lights_y[1], lights_z[1])
  image_3 = self.compute_image(object_z, object_rho, lights_I0[2], lights_x[2], lights_y[2], lights_z[2])
  plt.imshow(image_1)
  plt.gray()
  plt.show()
  plt.imshow(image_2)
  plt.gray()
  plt.show()
  plt.imshow(image_3)
  plt.gray()
  plt.show()
  self.write_image_to_file(image_1)
  self.class2.DEBUG_print(2, 'image_1:')
  self.class2.DEBUG_print_array(2, image_1)
  self.write_image_to_file(image_2)
  self.class2.DEBUG_print(2, 'image_2:')
  self.class2.DEBUG_print_array(2, image_2)
  self.write_image_to_file(image_3)
  self.class2.DEBUG_print(2, 'image_3:')
  self.class2.DEBUG_print_array(2, image_3)
  return 1
 #---------------------------------------------------
 # Recover shape
 #---------------------------------------------------
 def recover_shape(self):
  # Not working from within code
  # Use this from within python console
  # qgis.console.clearConsole()
  # Get access to classes containing support functions
  self.class2 = support_class2()
```

```
  # DEBUG_print to output messages based on DEBUG level
  self.class2.DEBUG_print(2, 'Starting to recover shape from images...')
  return
 def get_object_data_plane(self):
  # Copy z, rho into two 2-D arrays
  #  each of size IMG_RES_X, IMG_RES_Y
  object_z = []
  object_z = zeros( (self.IMG_RES_X, self.IMG_RES_Y), float64)
  object_rho = []
  object_rho = zeros( (self.IMG_RES_X, self.IMG_RES_Y), float64)
  # assume part of a sphere spanning the whole image/object space:
  #  r_min = max (self.OBJ_SIZE_X, self.OBJ_SIZE_Y) = 3.01 meters
  #  (0, 0, z0) is its closest point
  r = max (self.OBJ_SIZE_X, self.OBJ_SIZE_Y)  # meters
  # for each image pixel / object surface element :
  for i in range(self.MIN_I, (self.MAX_I+1)):
   for j in range(self.MIN_J, (self.MAX_J+1)):
    obj_elem_x = (i - self.IMG_RES_X_HALF) * self.OBJELEM_SIZE_X
    obj_elem_y = (self.IMG_RES_Y_HALF - j) * self.OBJELEM_SIZE_Y
    object_z[i,j] = self.z0
    object_rho[i,j] = 1.0
  return object_z, object_rho
 def get_object_data(self):
  # Copy z, rho into two 2-D arrays
  #  each of size IMG_RES_X, IMG_RES_Y
  object_z = []
  object_z = zeros( (self.IMG_RES_X, self.IMG_RES_Y), float64)
  object_rho = []
  object_rho = zeros( (self.IMG_RES_X, self.IMG_RES_Y), float64)
  # assume part of a sphere spanning the whole image/object space:
  #  r_min = max (self.OBJ_SIZE_X, self.OBJ_SIZE_Y) = 3.01 meters
  #  (0, 0, z0) is its closest point
  r = max (self.OBJ_SIZE_X, self.OBJ_SIZE_Y)  # meters
  # for each image pixel / object surface element :
  for i in range(self.MIN_I, (self.MAX_I+1)):
   for j in range(self.MIN_J, (self.MAX_J+1)):
    obj_elem_x = (i - self.IMG_RES_X_HALF) * self.OBJELEM_SIZE_X
    obj_elem_y = (self.IMG_RES_Y_HALF - j) * self.OBJELEM_SIZE_Y
    object_z[i,j] = self.z0 + r - math.sqrt(r**2 - obj_elem_x**2 - obj_elem_y**2)
    # To overwrite as a plane at distance (z0+1) meters
    object_z[i,j] = self.z0 + 1.0
    object_rho[i,j] = 1.0
  return object_z, object_rho
 def get_light_sources_data(self):
  lights_I0 = [ self.I0, self.I0, self.I0 ]
  # assume light sources to be at top-left, bottom-mid and top-right of lens
  lights_x=[ -(0.5*self.OBJ_SIZE_X), 0.0, (0.5*self.OBJ_SIZE_X) ]
  lights_y=[ (0.5*self.OBJ_SIZE_X), -(0.5*self.OBJ_SIZE_X), (0.5*self.OBJ_SIZE_X) ]
  lights_z = [self.f, self.f, self.f]
  return lights_I0, lights_x, lights_y, lights_z
 def compute_image(self, object_z, object_rho, light_I0, light_x, light_y, light_z):
  image = []
  image = zeros( (self.IMG_RES_X, self.IMG_RES_Y), float64)
  # for each image pixel / object surface element :
  for i in range(self.MIN_I, (self.MAX_I+1)):
   for j in range(self.MIN_J, (self.MAX_J+1)):
    objelem_x = (i - self.IMG_RES_X_HALF) * self.OBJELEM_SIZE_X
    objelem_y = (self.IMG_RES_Y_HALF - j) * self.OBJELEM_SIZE_Y
    objelem_z = object_z[i, j]
    # compute distance of object element from light source: Rs
    Rs = math.sqrt( ((light_x-objelem_x)**2) + ((light_y-objelem_y)**2) + ((light_z-objelem_z)**2))
    #self.class2.DEBUG_print(2, str(light_x)+'; '+str(light_y)+'; '+str(light_z))
    #self.class2.DEBUG_print(2, str(objelem_x)+'; '+str(objelem_y)+'; '+str(objelem_z))
    #self.class2.DEBUG_print(2, 'Rs:'+str(Rs))
    # compute local radiant flux density: P
    P = light_I0 * math.exp(-Rs / self.beta) / (Rs*Rs)
    # compute unit outward surface normal: n
    if (i < self.MAX_I):
  p = (object_z[(i+1), j] - object_z[i, j]) / self.OBJELEM_SIZE_X
    else:
```

```
p = (object_z[i, j] - object_z[(i-1), j]) / self.OBJELEM_SIZE_X
if (j < self.MAX_J):
q = (object_z[i, (j+1)] - object_z[i, j]) / self.OBJELEM_SIZE_Y
else:
q = (object_z[i, j] - object_z[i, (j-1)]) / self.OBJELEM_SIZE_Y
n_x = (-1) / math.sqrt( 1 + (p**2) + (q**2) )
n_y = p / math.sqrt( 1 + (p**2) + (q**2) )
n_z = q / math.sqrt( 1 + (p**2) + (q**2) )
# compute unit vector pointing from the surface toward the light source: s
s_x = (light_x - objelem_x) / Rs
s_y = (light_y - objelem_y) / Rs
s_z = (light_z - objelem_z) / Rs
# compute n_dot_s
n_dot_s = ( (n_x * s_x) + (n_y * s_y) + (n_z * s_z) ) * math.exp(-Rs / self.beta)
#self.class2.DEBUG_print(2, 'P:'+str(P)+'; n_dot_s:'+str(n_dot_s))
# compute distance of object element from center of camera lens: Rl
# to account for attenuation from object element to camera
Rc = math.sqrt( ((0-objelem_x)**2) + ((0-objelem_y)**2) + ((self.f-objelem_z)**2))
# compute image brightness: F
# ToDo: Remove (-1)
image[i, j] = (-1) * P * n_dot_s * object_rho[i, j] * math.exp(-Rc / self.beta)
# add brightness due to scattering media, F_medium
start_x = 0
start_y = 0
start_z = self.f
end_x = objelem_x
end_y = objelem_y
end_z = objelem_z
for t in arange(0, 1, self.SCATTERING_STEP):
particle_x = ((end_x-start_x)*t) + start_x
particle_y = ((end_y-start_y)*t) + start_y
particle_z = ((end_z-start_z)*t) + start_z
Rs = math.sqrt( ((light_x-particle_x)**2) + ((light_y-particle_y)**2) + ((light_z-particle_z)**2))
Rc = math.sqrt( ((0-particle_x)**2) + ((0-particle_y)**2) + ((self.f-particle_z)**2))
#compute cosine of angle between particle-light source and particle-camera lines: cos_alpha
a1 = (start_x - particle_x)
b1 = (start_y - particle_y)
c1 = (start_z - particle_z)
a2 = (light_x - particle_x)
b2 = (light_y - particle_y)
c2 = (light_z - particle_z)
cos_alpha = ((a1 * a2) + (b1 * b2) + (c1 * c2)) / ( math.sqrt((a1*a1)+(b1*b1)+(c1*c1)) * math.sqrt((a2*a2)+(b2*b2)+(c2*c2)) )
P = light_I0 * math.exp(-Rs / self.beta) / (Rs*Rs)
S = (1 + (self.g * cos_alpha) ) / (4 * math.pi)
F_medium = P * self.gamma * S * math.exp(-Rc / self.beta)
image[i, j] = image[i, j] + F_medium
return image
def write_image_to_file(self, image):
# plt.savefig('test.png')
return
a = main_class()
a.simulate_images()
```

4. RESULTS AND DISCUSSION

The software was tested by simulate images of a hemispherical object of radius 1 meter using a 60 Watt point light source. Three images were simulated by moving the light source to three different locations.

The image of the object simulated by placing the light source at top-left corner of the object is shown in Figure 4.

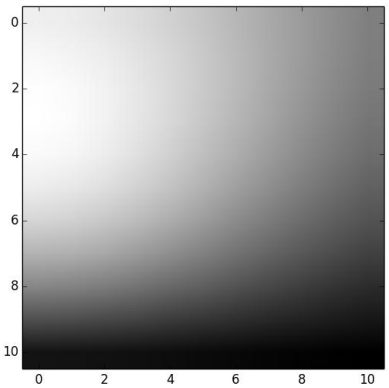

Figure 4. Image with light source at top-left corner

The image of the object simulated by placing the light source at bottom-middle corner of the object is shown in Figure 5.

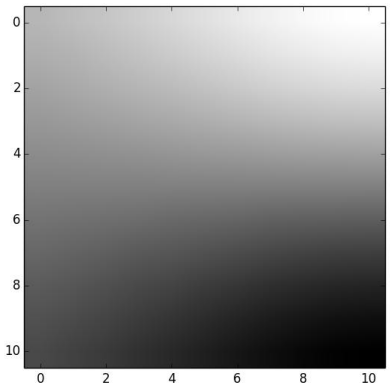

Figure 5. Image with light source at bottom-mid location

The image of the object simulated by placing the light source at top-right corner of the object is shown in Figure 6.

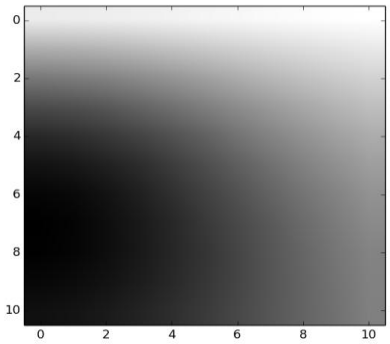

Figure 6. Image with light source at top-right corner

5. CONCLUSIONS

A open source Python and MatPlotLib software system was developed for simulating images using point light sources in scattering and attenuating media. It was tested by demonstrating its use for a simple object using three point light sources.

6. ACKNOWLEDGMENTS

We thank Prof. Joel S. Fox and the reviewers for their insightful comments.

7. REFERENCES

[1] Woodham, R.J., 1980 . Photometric method for determining surface orientation from multiple images. OptEng, 19(1). .

[2] Nicodemus, F.E.. Richmond, J.C., Hsia, Ginsberg, I.W. and Limperk, T., 1977. NBS Monograph 160, National Bureau of Standards, Washington, D.C.

[3] Horn, B.K.P., 1975. Obtaining Shape from Shading Information. In: Winston, P.H. (ed.). Psychology of Computer Vision. McGraw-Hill Book Co., New York.

[4] Turner, RM., Turner, E.H., Fox, J.S., Blidberg, D.R., 1991. Multiple Autonomous Vehicle Imaging System. 7th Int. Symp. on Unmanned Untethered Submersible Technology. Sept., Durham, New Hampshire.

[5] Narasimhan, S. G., Nayar, S. K., Sun, B., Koppal, S. J., 2005. Structured light in scattering media. October.