Handin 2

Marcus Malmquist, marmalm, 941022

February 15, 2017

1 Task 1

The beam waist is located at $z = 0$ because that is how I define it. The beam width at $z = z_1$ (which is located at $z = -0.81$) is 296 µm. Since the mirrors are separated by 95 cm and the left mirror is located at $z = -0.81$, the right mirror is naturally located at $z = 0.14$.

2 Task 2

The beam width was numerically calculated to be 300 µm which is very close to the theoretical value. The theoretical value is more likely to be correct as it is theoretical (and if not it would just be rubbish). The simulation uses approximations (particularly when calculating the integral) which probaby has an inpact on the result.

3 Task 3

It is possible for a more incorrect and random starting field to converge into a the fundamental mode as seen in Figure 1

4 Task 4

The hair appears to turn the fundamental mode into a (0,1)-mode as can be seen in Figure 2 although it does not quite converge after 300 iterations.

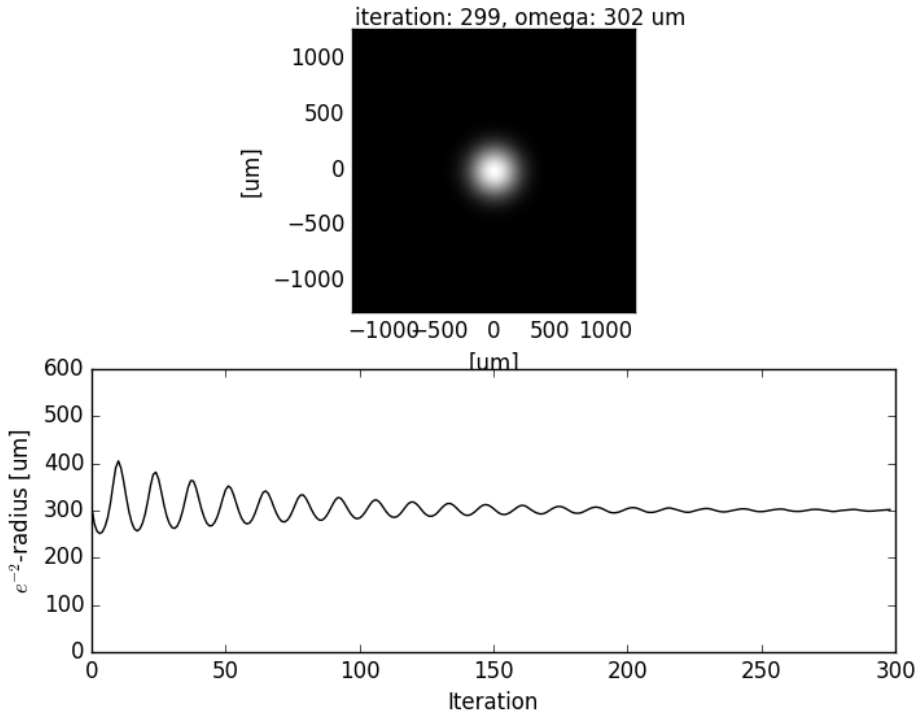

Figure 1: The gaussian beam after 300 iterations (top image) and the e^{-2} radius (bottom graph)

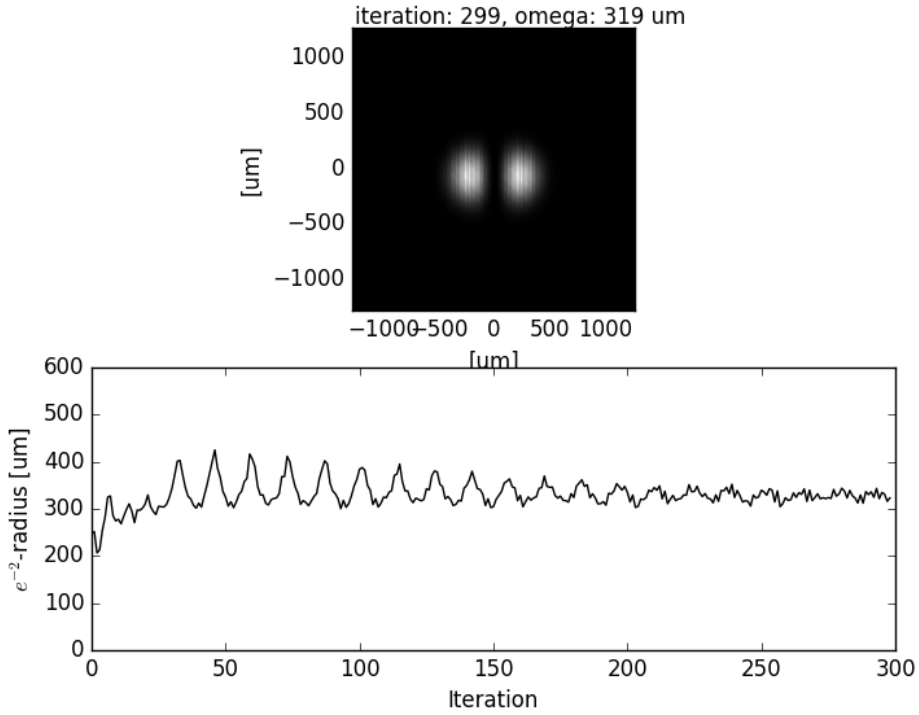

Figure 2: The gaussian beam after 300 iterations (top image) and the e^{-2} radius (bottom graph). The bottom graph should only be used to judge wether or not the beam has converged.

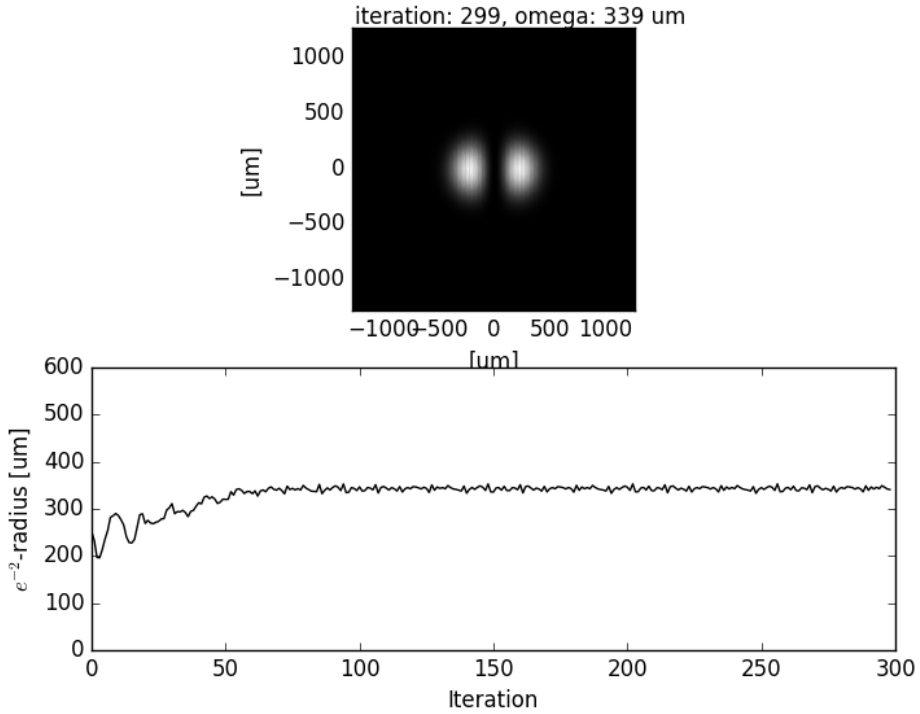

Figure 3: The gaussian beam after 300 iterations (top image) and the e^{-2} radius (bottom graph). The bottom graph should only be used to judge wether or not the beam has converged.

5 Task 5

Then hair appears to turn the fundamental mode into a (0,1)-mode as can be seen in Figure 3. Item should be noted that it actually converges and it does so very quickly compared to the setup in Section 4.

6 Task 6

The hairs appears to turn the fundamental mode into a (1,1)-mode as can be seen in Figure 4 although it does not quite converge after 300 iterations.

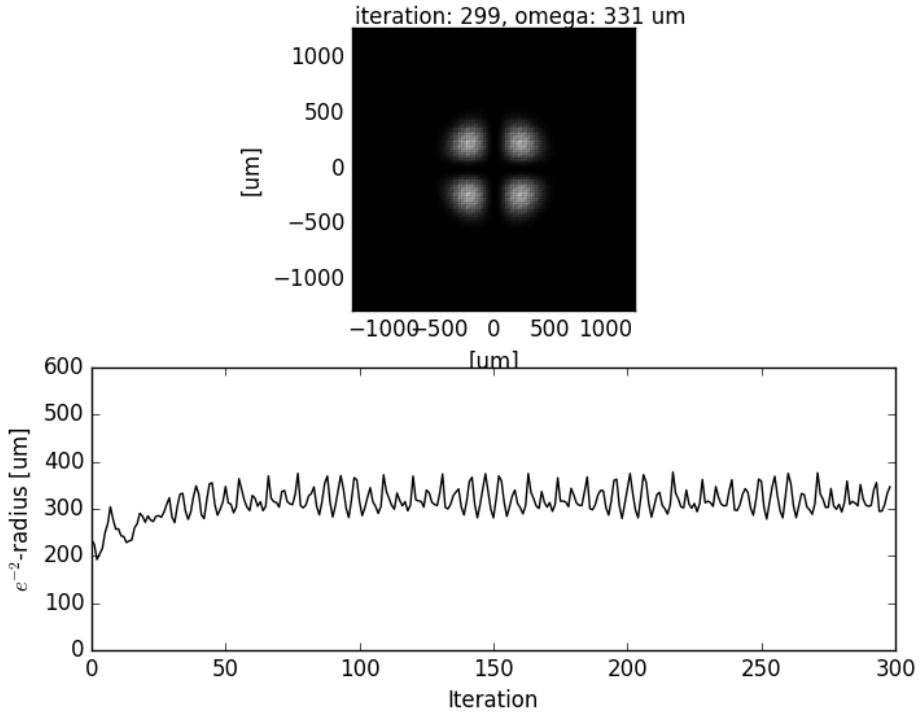

Figure 4: The gaussian beam after 300 iterations (top image) and the e^{-2} radius (bottom graph). The bottom graph should only be used to judge wether or not the beam has converged.

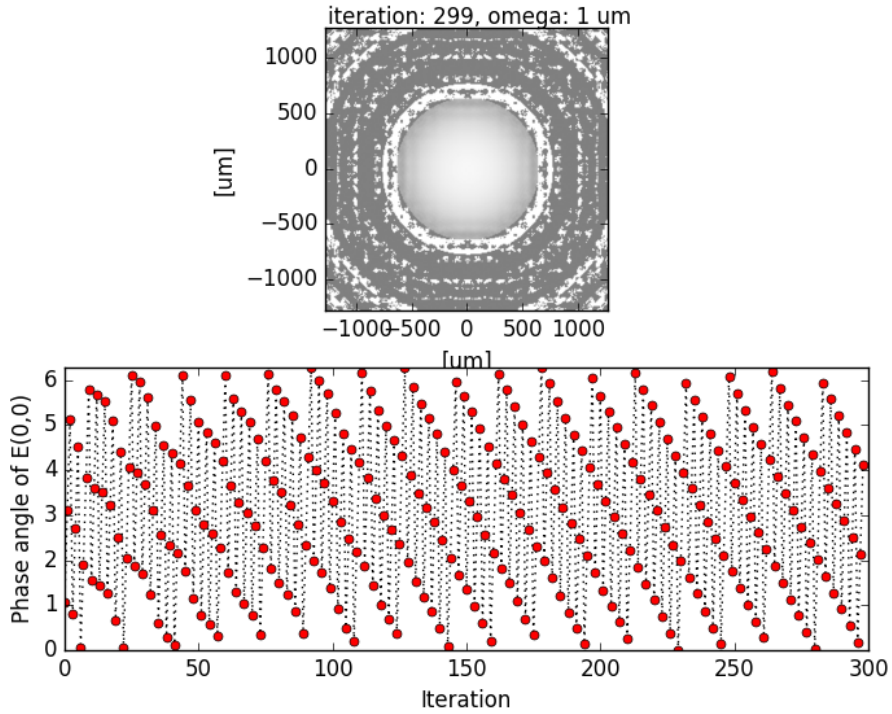

Figure 5: The normalized phase of the gaussian beam after 300 iterations (top image) and the phase of $E(0,0)$ at the left mirror (bottom graph).

7 Task 7

7.1 a

Since we plot the intensity we loose information about the phase, which can cause the field (which has a phase) to look different when compared to the previous roundtrip.

7.2 b

Then graph in Figure 5 confirms that the distance traveled in one roundtrip is not a multiple off λ. From the figure it seems that once the beam has converged the phase changes by just over 4π after three roundtrips.

7.3 c

In order to make the phase of the field equal for two consecutive roundtrips one canadian change the distance between the mirrors slightly in order to make the distance traveled in one roundtrip be a multiple offset λ.

A Code

```
"""
This code is adapted for 80 columns
|--------------------------------------------------------------------------------|
"""
import sys
try:
    import numpy as np
    import numpy.matlib as npm
    import matplotlib.pyplot as plt
    import matplotlib.animation as animation
except ImportError:
    sys.stderr.write("Could not execute program: " + str(sys.exc_info()[1])
                     + ".\nPlease install the module and try again")
    sys.stderr.flush()
    sys.exit()
pi = np.pi


class Handin2():
    """
    This class contains the necessary functions to run the simulations.
    For simplicity and for the sake of not having to pass too many paramters
    arround most variables are class variables.
    """

    def __init__(self):
        """
        class instantiation.
        """
        self.txt_tmpl = "iteration: %.0f"
        self.line_txt_tmpl = "omega: %.0f um"
        self.img_txt = None
        self.N = 0x1 << 8  # number fo sample points
        self.lbda_0 = 633e-9  # wavelength in vacuum
        self.n = 1.0  # refractive index
        self.k = 2*pi*self.n/self.lbda_0
        self.p1_sd = None  # sample distance at animation plane 1
        self.p2_sd = None  # sample distance at animation plane 2
        self.E1 = None  # the starting field for the animation
```

```
        self.img = None  # the animation image
        self.line = None  # the animation 2d plot
        self.task_nbr = 0  # If a circle to block the bright spot
        self.focalLenLeft = 1.25/2  # focal length of first lens
        self.focalLenRight = 1.0/2  # focal length of second lens
        self.d = 0.95  # distance between mirrors/lenses
        self.z0 = None  # radius of waist
        self.z1 = None  # mirror/lense 1 pos
        self.z2 = None  # mirror/lense 2 pos
        self.W0 = None  # beam radius at z0
        self.omega = 200e-6  # [m] exp(-2) radius...
        self.omega_vec = None
        self.iter_vec = None

    @staticmethod
    def calcMirrorGeom(R1=-1.25, R2=-1.0, d=0.95):
        """
        Calculates the location of the mirrors as well as the beam waist.

        Parameters
        ----------
        R1 : float
            Radius of curvature of mirror 1 (left mirror).
        R2 : float
            Radius of curvature of mirror 2 (right mirror).
        d : float
            Distance between the mirrors.

        Returns
        -------
        float
            The beam waist
        float
            Position of left mirror
        float
            Position of right mirror.
        """
        z1 = -d*(R2+d)/(R1+R2+2*d)
        z2 = z1+d
        z0 = np.sqrt(-d*(R1+d)*(R2+d)*(R2+R1+d)/(R1+R2+2*d)**2)
        return z0, z1, z2
```

```
@staticmethod
def fft2c(x):
    """
    Calculates the shifted fourier transform of a 2-dimensional matrix.

    Parameters
    ----------
    x : matrix_like
        The matrix that is to be fourier transformed.

    Returns
    -------
    numpy.ndarray
        The shifted fourier transform of the input.
    """
    return np.fft.fftshift(np.fft.fft2(np.fft.fftshift(x)))

@staticmethod
def ifft2c(x):
    """
    Calculates the shifted inverse fourier transform of a 2-dimensional
    matrix.

    Parameters
    ----------
    x : matrix_like
        The matrix that is to be fourier transformed.

    Returns
    -------
    numpy.ndarray
        The shifted inverse fourier transform of the input.
    """
    return np.fft.fftshift(np.fft.ifft2(np.fft.fftshift(x)))

@staticmethod
def getCoordVect(nPoints, sampleDist):
    """
    A static method that returns an array of nPoints with a spacing of
    sampleDist starting from -nPoints/2 to nPoints/2.
```

```
    Parameters
    ----------
    nPoints : int
        Number of points that the array should contains, i.e. the length of
        the returned array.
    sampleDist : float
        The distance between the points in the array.

    Returns
    -------
    numpy.ndarray
        The array containing nPoints points separated by sampleDist.
    """
    return np.arange(-nPoints/2*sampleDist, (nPoints/2)*sampleDist,
                     sampleDist)

@staticmethod
def TSM(E1, a, b, L, lbda_0, n):
    """
    Implements the Huygens-Fresnel method.

    This function implements the conventional Huygens-Fresnel method (TSM)
    for free space propagation. It does so by propagating from the plane of
    incidence (plane 1) to some dummy plane, and then propagates back to
    plane 2. This means that then two planes can be saparated by any
    distance and still yield a result with high accuracy.
    NOTE: the sampling distance at plane 1 and plane 2 can not be equal as
    it will yield numerical errors.

    Parameters
    ----------
    E1 : matrix_like
        The incident E-field at plane 1.
    a : float
        Sampling distance at plane 1.
    b : float
        Sampling distance at plane 2.
    L : float
        Distance between plane 1 and plane 2. Unit is in metre.
    lbda_0 : float
```

```
    Wavelength of incident wave in vacuum. Unit is in metre.
n : float
    Refractive index inside the medium in which the field propagates.

Returns
-------
numpy.ndarray
    The E-field at plane 2
float
    The distance between plane 1 and the dummy plane.
float
    The distance between plane 2 and the dummy plane.
numpy.ndarray
    A 1-dimensional array containing the sampling locations along the
    x-axis at the dummy plane.
numpy.ndarray
    A 1-dimensional array containing the sampling locations along the
    y-axis at the dummy plane.
numpy.ndarray
    A 2-dimensional array containing the sampling locations along the
    x-axis at the dummy plane for each sampling location along the
    y-axis.
numpy.ndarray
    A 2-dimensional array containing the sampling locations along the
    y-axis at the dummy plane for each sampling location along the
    x-axis.
"""
N = np.size(E1, axis=0)  # Square matrix size
lambda_medium = lbda_0/n
k = 2*pi/lambda_medium

# Plane 1 coordinates:
[xmat, ymat] = np.meshgrid(Handin2.getCoordVect(N, a),
                           Handin2.getCoordVect(N, a))

# Plane 2 coordinates
uvect = Handin2.getCoordVect(N, b)
vvect = Handin2.getCoordVect(N, b)
[umat, vmat] = np.meshgrid(uvect, vvect)

# Distance to dummy plane
```

```
    L1 = L*a/(a-b)  # plane 1 to dummy plane
    L2 = L*b/(a-b)  # plane 2 to dummy plane
    # Sampling distance in Dummy plane
    c = lambda_medium/(N*a)*L1

    # Dummy plane coordinates
    xiVect = Handin2.getCoordVect(N, c)
    etaVect = Handin2.getCoordVect(N, c)
    [xiMat, etaMat] = np.meshgrid(xiVect, etaVect)

    # Calculating the field in Plane 2
    fctr0 = a**2 / b**2 * L2/L1 * np.exp(1j*k*(L1-L2))
    fctr1 = np.exp(-1j*k *
                   (umat**2 + vmat**2) / (2*L2))
    prefactor = np.exp(
        1j*k * (xiMat**2 + etaMat**2) / 2 * (1/L1 - 1/L2))
    fft_part = Handin2.fft2c(
        E1*np.exp(1j*k * (xmat**2 + ymat**2) / (2*L1)))
    fctr2 = Handin2.ifft2c(prefactor * fft_part)
    E2 = fctr0*fctr1*fctr2
    return E2, L1, L2, uvect, vvect, umat, vmat

def getBeamRadius(self, z):
    """
    Calculates the beam waist at some position along the optical axis.

    Parameters
    ----------
    z : float
        The point on then optical axis.

    Returns
    -------
    float
        The beam waist at the point on then optical axis.
    """
    return self.W0*np.sqrt(1+(z/self.z0)**2)

def mainFun(self, L):
    """
    The purpose of this function is to calculcate the next field
```

```
    during the animation process. It can be used to modify the variables
    returned from the TSM function.

    Parameters
    ----------
    L : float
        The distance between plane 1 and plane 2.

    Returns
    -------
    numpy.ndarray
        The E-field in plane 2 (after propagating from plane 1).
    """
    out = Handin2.TSM(self.E1, self.p1_sd, self.p2_sd,
                      L, self.lbda_0, self.n)
    return out[0]

def simData(self):
    """
    This method propagates between the mirrors/lenses and applies the
    appropriate transfer functions such as aperture, lense and hair.
    Since this function does not return any values (it only yields them)
    it should only be used with the FuncAnimation from the
    matplotlib.animation module.
    What task is performed depends on the class variable task_nbr.

    The function yields different values for different tasks:
    Yields
    ------
    numpy.ndarray
        The E-field in plane 1.
    float
        The beam radius in plane 1. Only for task 2-7.
    int
        The current iteration number.
    """
    if (self.task_nbr == 1):
        for i in self.iter_vec:
            # E1 is currently located just after left lense
            self.p1_sd = 10e-6
            self.p2_sd = 20e-6
```

```
            # propagate and use lense transfer function
            self.E1 = self.mainFun(self.d) *\
                self.getAperture(self.p2_sd, fracRadi=0.5)*0.972 *\
                self.getLensTrsfFunc(self.p2_sd, self.focalLenRight)
            # E1 is not locaded after right lense
            self.p1_sd = 20e-6
            self.p2_sd = 10e-6
            # propagate and use lense transfer function
            self.E1 = self.mainFun(self.d) *\
                self.getAperture(self.p2_sd, fracRadi=0.5) *\
                self.getLensTrsfFunc(self.p2_sd, self.focalLenLeft)
            yield self.E1, i
    elif (self.task_nbr == 2 or self.task_nbr == 3 or self.task_nbr == 7):
        for i in self.iter_vec:
            # E1 is currently located just after left lense
            self.p1_sd = 10e-6
            self.p2_sd = 20e-6
            # propagate and use lense transfer function
            self.E1 = self.mainFun(self.d) *\
                self.getAperture(self.p2_sd, fracRadi=0.5)*0.972 *\
                self.getLensTrsfFunc(self.p2_sd, self.focalLenRight)
            # E1 is not locaded after right lense
            self.p1_sd = 20e-6
            self.p2_sd = 10e-6
            # propagate and use lense transfer function
            self.E1 = self.mainFun(self.d) *\
                self.getAperture(self.p2_sd, fracRadi=0.5) *\
                self.getLensTrsfFunc(self.p2_sd, self.focalLenLeft)
            I_max = np.max(np.abs(self.E1)**2)
            # the integral over the gaussian function
            itgl_val = np.sum(np.abs(self.E1)**2*self.p2_sd**2)
            self.omega = np.sqrt(2*itgl_val/(I_max*pi))
            yield self.E1, self.omega, i
    elif (self.task_nbr >= 4 and self.task_nbr <= 6):
        hair_clip = 0
        if (self.task_nbr == 5):
            hair_clip = 1
        if (self.task_nbr == 6):
            hair_clip = 2
        for i in self.iter_vec:
            # E1 is currently located just after left lense
```

```
            self.p1_sd = 10e-6
            self.p2_sd = 20e-6
            hair_w = 50e-6
            # propagate and use lense transfer function
            self.E1 = self.mainFun(self.d) *\
                self.getHair(self.p2_sd, hair_w, hair_clip) *\
                self.getAperture(self.p2_sd, fracRadi=0.5)*0.972 *\
                self.getLensTrsfFunc(self.p2_sd, self.focalLenRight)
            # E1 is not locaded after right lense
            self.p1_sd = 20e-6
            self.p2_sd = 10e-6
            # propagate and use lense transfer function
            self.E1 = self.getHair(self.p1_sd, hair_w, hair_clip) *\
                self.mainFun(self.d) *\
                self.getAperture(self.p2_sd, fracRadi=0.5) *\
                self.getLensTrsfFunc(self.p2_sd, self.focalLenLeft)
            I_max = np.max(np.abs(self.E1)**2)
            # the integral over the gaussian function
            itgl_val = np.sum(np.abs(self.E1)**2*self.p2_sd**2)
            self.omega = np.sqrt(2*itgl_val/(I_max*pi))
            yield self.E1, self.omega, i

def updatefig(self, simData):
    """
    Handles setting the new image/graph and info text for the simulation.

    Parameters
    ----------
    simData : tuple
        A tuple containing the E-field in plane 1.
        A float containing the beam width (only for task 2-7)
        An int containing the current iteration number.

    Returns
    -------
    matplotlib.image.AxesImage
        The grayscale intensity (image) at plane 1.
    matplotlib.lines.Line2D
        The beam width plot as a function of the iteration number.
        Only for task 2-7.
    matplotlib.text.Text
```

```
        The text containing information about what is being shown.
    """
    if (self.task_nbr == 1):
        E1,  iter_idx = simData[0], simData[1]
        E1_plt = np.array(np.abs(E1)/np.max(np.abs(E1)),
                          dtype=np.float64)**2
        dst = self.txt_tmpl % iter_idx
        self.img_txt.set_text(dst)
        self.img.set_array(E1_plt)
        return self.img, self.img_txt
    elif (self.task_nbr >= 2 and self.task_nbr <= 6):
        E1, omga,  iter_idx = simData[0], simData[1], simData[2]
        E1_plt = np.array(np.abs(E1)/np.max(np.abs(E1)),
                          dtype=np.float64)**2
        dst = self.txt_tmpl % iter_idx + ", " +\
            self.line_txt_tmpl % (omga*1e6)
        self.img_txt.set_text(dst)
        self.img.set_array(E1_plt)
        self.omega_vec[iter_idx] = omga*1e6
        self.line.set_data(self.iter_vec[:iter_idx],
                           self.omega_vec[:iter_idx])
        return self.img, self.line, self.img_txt
    elif (self.task_nbr == 7):
        E1, omga,  iter_idx = simData[0], simData[1], simData[2]
        E1_plt = np.array((np.angle(E1)+np.pi)/np.max(np.angle(E1)+np.pi),
                          dtype=np.float64)
        dst = self.txt_tmpl % iter_idx + ", " +\
            self.line_txt_tmpl % (E1_plt[int(self.N/2+1), int(self.N/2+1)])
        self.img_txt.set_text(dst)
        self.img.set_array(E1_plt)
        self.omega_vec[iter_idx] = np.angle(E1)[int(self.N/2+1),
                                                int(self.N/2+1)]+pi
        self.line.set_data(self.iter_vec[:iter_idx],
                           self.omega_vec[:iter_idx])
        return self.img, self.line, self.img_txt

def getLensTrsfFunc(self, sampleDist, focalLen):
    """
    Creates the lense transfer function.

    Parameters
```

```
    ----------
    sampleDist : float
        The sample distance at the lence.
    focalLen : float
        The focal length of the lense.

    Returns
    -------
    numpy.ndarray
        The lense transfer function, i.e. the elementwise product of the
        E-field before the lense and the lense transfer function yields
        the E-field after the lense.
    """
    [xmat, ymat] = np.meshgrid(Handin2.getCoordVect(self.N, sampleDist),
                               Handin2.getCoordVect(self.N, sampleDist))
    return np.exp(-1j*self.k*(xmat**2 + ymat**2)/(2*focalLen))

def getAperture(self, sampleDist, fracRadi=1):
    """
    Calculates the aperture transfer matrix

    Parameters
    ----------
    sampleDist : float
        The sampling distance at the apterture.
    fracRadi : float
        The fractional radius of the aperture. If the value is 1.0 then
        the aperture will have a diameter has the same size as the side
        of the matrix. If the value is 0.5 then the aperture will have a
        diameter has the half size as the side of the matrix.

    Returns
    -------
    numpy.ndarray
        The apterture transfer function, i.e. the elementwise product of
        the E-field before the aperture and the aperture transfer function
        yields the E-field after the aperture.
    """
    [xmat, ymat] = np.meshgrid(Handin2.getCoordVect(self.N, sampleDist),
                               Handin2.getCoordVect(self.N, sampleDist))
    rmat = np.sqrt(xmat**2 + ymat**2)
```

```
    return np.ones((self.N, self.N))*(rmat < sampleDist*self.N/2*fracRadi)

def getHair(self, sampleDist, hairWidth, hairClip):
    """
    Calculates the hair transfer matrix.

    Parameters
    ----------
    sampleDist : float
        The sampling distance at them hair.
    hairWidth : float
        The width (diameter) of them hair. Unit is in metre.
    hairClip : int
        Determines what type of hair is used:
        0 means ones vertical hair placed in the middle.
        1 means half a vertical hair placed in the middle.
        2 means half a vertical hair placed in the middle and half a
          horizontal hair placed in them middle.

    Returns
    -------
    numpy.ndarray
        The hair transfer function, i.e. the elementwise product of the
        E-field before the hair and the hair transfer function yields
        the E-field after the hair.
    """
    [xmat, ymat] = np.meshgrid(Handin2.getCoordVect(self.N, sampleDist),
                               Handin2.getCoordVect(self.N, sampleDist))
    if (hairClip == 1):  # half a vertical hair
        bool_mat = 1 - (np.abs(xmat) < hairWidth/2) * (ymat > 0)
    elif (hairClip == 2):  # half a vertical and horizontal hair
        bool_mat = 1 - (((np.abs(xmat) < hairWidth/2) & (ymat > 0)) |
                        ((np.abs(ymat) < hairWidth/2) & (xmat > 0)))
    else:  # one vertical hair
        bool_mat = 1 - (np.abs(xmat) < hairWidth/2)
    return np.ones((self.N, self.N))*bool_mat

def initializeImage(self, I0):
    """
    initialzes the grayscale intesity image.
```

```
    Parameters
    ----------
    I0 : array_like
        The intensty of the (starting) field.
    """
    self.img_txt = ax1.text(-1256, 1.3e3, "", color="#000000")
    self.img = ax1.imshow(I0, animated=True, cmap=plt.get_cmap('gray'),
                          interpolation='quadric', vmin=0, vmax=1,
                          extent=[-(self.N/2)*self.p1_sd*1e6,
                                  (self.N/2-1)*self.p1_sd*1e6,
                                  -(self.N/2)*self.p1_sd*1e6,
                                  (self.N/2-1)*self.p1_sd*1e6])
    ax1.set_xlabel("[um]")
    ax1.set_ylabel("[um]")
    ax1.set_aspect(1)

def initializePlot(self):
    """
    Initializes the 2D plot of them beam diameter vs iteration number.
    """
    self.omega_vec = np.zeros(np.size(self.iter_vec))
    self.line, = ax2.plot([], [], linestyle="-", color="black")
    ax2.set_ylim(0, 600)
    ax2.set_xlabel("Iteration")
    ax2.set_ylabel("$e^{-2}$-radius [um]")
    ax2.set_xlim(0, np.size(self.iter_vec))

def createInitialField(self):
    """
    Creates the starting field for the different tasks.
    Tasks 1, 2 and 7 has a single gaussian beam.
    Tasks 3, 4, 5 and 6 has a sum of (30) gaussian fields with random
    phase and a random offset from them origin of the frame of reference.

    Returns
    -------
    numpy.ndarray
        The starting E-field for the current task.
    """
    [xmat, ymat] = np.meshgrid(Handin2.getCoordVect(self.N, self.p1_sd),
                               Handin2.getCoordVect(self.N, self.p1_sd))
```

```
    if (self.task_nbr == 1 or self.task_nbr == 2 or self.task_nbr == 7):
        self.E1 = np.exp(-(xmat**2 + ymat**2)/self.omega**2)
    if (self.task_nbr > 2 and self.task_nbr < 7):
        n_beams = 30
        """
        Creates n_beams copies of xmat and ymat and adds a unique
        offset for each copy.
        """
        xmat = npm.repmat(np.reshape(xmat, (1, -1)), n_beams, 1) +\
            300e-6*(2*np.random.random((n_beams, 1))-1)
        ymat = npm.repmat(np.reshape(ymat, (1, -1)), n_beams, 1) +\
            300e-6*(2*np.random.random((n_beams, 1))-1)
        E1_tmp = np.exp(-(xmat**2 + ymat**2)/self.omega**2)
        rand_phase = 2j*np.pi*np.random.random((n_beams, 1))
        # adds a unique random phase for each E-field.
        self.E1 = np.reshape(np.sum(E1_tmp*rand_phase, axis=0),
                             (self.N, self.N))
    return np.array(np.abs(self.E1)/np.max(np.abs(self.E1)),
                    dtype=np.float64)**2

def calcBeamDiameter(self):
    """
    calculates the beam waist and prints the beam diameter at them left
    mirror.
    """
    self.z0, self.z1, self.z2 = Handin2.calcMirrorGeom()
    self.W0 = np.sqrt(self.lbda_0*self.z0/pi)
    print("Beam diameter at z1: %.0f microns"
          % (self.getBeamRadius(self.z1)*1e6))

def task1(self):
    """
    Initializes task 1
    """
    self.task_nbr = 1
    self.calcBeamDiameter()
    self.p1_sd = 10e-6
    self.p2_sd = 20e-6
    self.initializeImage(self.createInitialField())
    self.iter_vec = np.arange(0, 300, dtype=np.uint32)
```

```
def task2(self):
    """
    Initializes task 2
    """
    self.task_nbr = 2
    self.calcBeamDiameter()
    self.p1_sd = 10e-6
    self.p2_sd = 20e-6
    self.initializeImage(self.createInitialField())
    self.iter_vec = np.arange(0, 300, dtype=np.uint32)
    self.initializePlot()

def task3(self):
    """
    Initializes task 3
    """
    self.task_nbr = 3
    self.calcBeamDiameter()
    print(self.z0, self.z1, self.z2)
    self.p1_sd = 10e-6
    self.p2_sd = 20e-6
    self.initializeImage(self.createInitialField())
    self.iter_vec = np.arange(0, 300, dtype=np.uint32)
    self.initializePlot()

def task4(self):
    """
    initializes task 4
    """
    self.task_nbr = 4
    self.calcBeamDiameter()
    self.p1_sd = 10e-6
    self.p2_sd = 20e-6
    self.initializeImage(self.createInitialField())
    self.iter_vec = np.arange(0, 300, dtype=np.uint32)
    self.initializePlot()

def task5(self):
    """
    initializes task 5
    """
```

```
        self.task_nbr = 5
        self.calcBeamDiameter()
        self.p1_sd = 10e-6
        self.p2_sd = 20e-6
        self.initializeImage(self.createInitialField())
        self.iter_vec = np.arange(0, 300, dtype=np.uint32)
        self.initializePlot()

    def task6(self):
        """
        initializes task 6
        """
        self.task_nbr = 6
        self.calcBeamDiameter()
        self.p1_sd = 10e-6
        self.p2_sd = 20e-6
        self.initializeImage(self.createInitialField())
        self.iter_vec = np.arange(0, 300, dtype=np.uint32)
        self.initializePlot()

    def task7(self):
        """
        initializes task 7
        """
        self.task_nbr = 7
        self.calcBeamDiameter()
        self.p1_sd = 10e-6
        self.p2_sd = 20e-6
        self.initializeImage(self.createInitialField())
        self.iter_vec = np.arange(0, 300, dtype=np.uint32)
        self.initializePlot()
        self.line, = ax2.plot([], [], marker="o", markersize="5",
                              markerfacecolor="red",
                              linestyle=":", color="black")
        ax2.set_ylim(0, 2*pi)
        ax2.set_xlim(0, np.size(self.iter_vec))
        ax2.set_ylabel("Phase angle of E(0,0)")


if (__name__ == "__main__"):
    """
```

```
This function is called when the program executes
"""
fig = plt.figure()  # new figure
hi1 = None
if (len(sys.argv) > 1):
    args = sys.argv[1:]
    if (args[0] == "1"):
        ax1 = fig.add_subplot(111)
        hi1 = Handin2()  # the simulation class
        hi1.task1()  # method for task 1
    elif (args[0] == "2"):
        ax1 = fig.add_subplot(211)
        ax2 = fig.add_subplot(212)
        hi1 = Handin2()  # the simulation class
        hi1.task2()
    elif (args[0] == "3"):
        ax1 = fig.add_subplot(211)
        ax2 = fig.add_subplot(212)
        hi1 = Handin2()  # the simulation class
        hi1.task3()
    elif (args[0] == "4"):
        ax1 = fig.add_subplot(211)
        ax2 = fig.add_subplot(212)
        hi1 = Handin2()  # the simulation class
        hi1.task4()  # method for task 4
    elif (args[0] == "5"):
        ax1 = fig.add_subplot(211)
        ax2 = fig.add_subplot(212)
        hi1 = Handin2()  # the simulation class
        hi1.task5()  # method for task 5
    elif (args[0] == "6"):
        ax1 = fig.add_subplot(211)
        ax2 = fig.add_subplot(212)
        hi1 = Handin2()  # the simulation class
        hi1.task6()  # method for task 6
    elif (args[0] == "7"):
        ax1 = fig.add_subplot(211)
        ax2 = fig.add_subplot(212)
        hi1 = Handin2()  # the simulation class
        hi1.task7()  # method for task 7
    else:
```

```
        sys.stdout.write("Usage: python <filename.py> <task_nbr>")
        sys.stdout.flush()
        sys.exit()
else:
    sys.stdout.write("Usage: python <filename.py> <task_nbr>")
    sys.stdout.flush()
    sys.exit()
# the function handling the the animation itself
ani = animation.FuncAnimation(fig, hi1.updatefig, hi1.simData,
                              interval=100, blit=False, repeat=False)
plt.show()  # plot figure
```