Finite Difference Mode Solver – Solution to the Seminar Tasks

Your task was to implement a finite difference mode solver in one and two dimensions. The results had to be tested with the eigenmodes of a Gaussian permitivity distribution

$$\varepsilon(x) = \varepsilon_s + \Delta\varepsilon\ \exp\left(-\frac{x^2}{w^2}\right) \tag{1}$$

in the 1D case and

$$\varepsilon(x,y) = \varepsilon_s + \Delta\varepsilon\ \exp\left(-\frac{x^2+y^2}{w^2}\right) \tag{2}$$

in the 2D case, respectively.

1 Task 1 – The 1D Case

1.1 Implementation

The implementation of the one-dimensional mode solver requires the discretization of the linear differential operator

$$L = \frac{1}{k_0^2}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + \varepsilon(x) \tag{3}$$

of the eigenvalue equation

$$LE(x) = \varepsilon_{\mathrm{eff}}E(x) \tag{4}$$

in a matrix form using a finite difference scheme to approximate the second order derivative. The eigenvalues and eigenvectors of this matrix, which correspond to the discretized eigenmodes of the permittivity distribution, can then be calculated with built-in Python functions.

Now let us take a look at the source code of the 1D mode solver:

```python
def guided_modes_1DTE(prm, k0, h):
    """Computes the effective permittivity of a TE polarized guided eigenmode.
    All dimensions are in µm.
    Note that modes are filtered to match the requirement that
    their effective permittivity is larger than the substrate (cladding).

    Parameters
    ----------
    prm : 1d-array
        Dielectric permittivity in the x-direction
    k0 : float
        Free space wavenumber
    h : float
```

```
14         Spatial discretization
15 
16     Returns
17     -------
18     eff_eps : 1d-array
19         Effective permittivity vector of calculated modes
20     guided : 2d-array
21         Field distributions of the guided eigenmodes
22     """
23 
24     # set up operator matrix
25     main = -2.0 * np.ones(len(prm)) / (h**2 * k0**2) + prm
26     sec = np.ones(len(prm) - 1) / (h**2 * k0**2)
27     L = np.diag(sec, -1) + np.diag(main, 0) + np.diag(sec, 1)
28 
29     # solve eigenvalue problem
30     eff_eps, guided = np.linalg.eig(L)
```

In the lecture it has been shown that the approximation of the second order derivative with a symmetric finite difference scheme leads to a tridiagonal matrix. The main and secondary diagonals of this matrix are defined on the lines 25 and 26. In line 27 the operator matrix is assembled using the NumPy function `diag` to create diagonal matrices from the vectors. The optional second argument of the `diag` function is used to specify the offset from the main diagonal. Once the matrix is constructed, its eigenvalues and eigenvectors are calculated using NumPy's `eig` function.

The next part of the implementation takes care of separating the truly guided modes from spurious cavity modes. These stem from the perfect electric conducting boundaries which are implicitly introduced by assuming zeros at the edges of the discretized field vector (the secondary diagonals of the operator matrix have one element less than the main diagonal).

```
    # pick only guided modes
    idx = eff_eps > max(prm[0], prm[-1])
    eff_eps = eff_eps[idx]
    guided = guided[:, idx]

    # sort modes from highest to lowest effective permittivity
    # (the fundamental mode has the highest effective permittivity)
    idx = np.argsort(-eff_eps)
    eff_eps = eff_eps[idx]
    guided = guided[:, idx]
    return eff_eps, guided
```

The last two lines of the implementation sort the eigenvalues and eigenvectors in descending order of the effective permittivity. This is desirable as the fundamental mode with the lowest mode number has the highest effective permittivity.

1.2 Convergence Tests

The implemented algorithm depends on two parameters that determine its accuracy. The discretization parameter h determines the accuracy of the geometry representation and the accuracy of the finite difference approximation of the differential operator. Additionally, in our implementation the computational domain does not have open boundaries but is terminated by perfect electric conductors (i.e. perfect mirrors). If these boundaries are too close to the wave-guiding structure, the guided modes can interact with the boundaries and will be altered. Hence, also the size of the computational domain influences the results and must be tested for convergence.

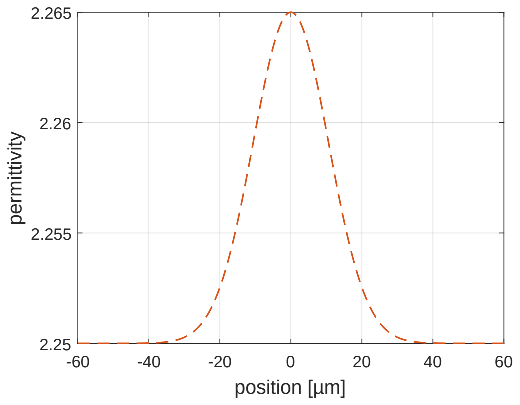

Figure 1: Gaussian permittivity distribution.

The convergence of the eigenmodes in dependence of the discretization parameter h is tested with the following script:

```
 1  grid_size      = 120
 2  lam            = 0.78
 3  k0             = 2 * np.pi / lam
 4  e_substrate    = 1.5**2
 5  delta_e        = 1.5e-2
 6  num_modes      = 15
 7  w              = 15.0
 8
 9  # 31 points logarithmically spaced between 3 and 0.007
10  h = np.logspace(np.log10(3), np.log10(0.007), 31)
11  eff_eps = np.nan * np.zeros((h.size, num_modes))
12  for i, hi in enumerate(h):
13      Nx = np.ceil(int(0.5 * grid_size / hi))
14      x = np.arange(-Nx, Nx+1) * hi
15      prm = e_substrate + delta_e * np.exp(-(x/w)**2)
16      start = time.clock()
17      mode_eps, guided = guided_modes_1DTE(prm, k0, hi)
18      N = min(num_modes, len(mode_eps))
19      eff_eps[i, :N] = mode_eps[:N]
20      stop = time.clock()
21      print("h = %6.3f, time = %gs" % (hi, stop - start))
22
23  # remove modes that do not exist for all grid discretization
24  # = remove mode number if eff_eps is NaN for at least one grid discretization
25  eff_eps = eff_eps[:, ~np.any(np.isnan(eff_eps), axis=0)]
26
27  # calculate relative error to the value obtained at highest resolution
28  # this should approximate the true error
29  rel_error = np.abs(eff_eps[:-1, :] / eff_eps[-1, :] - 1.0)
```

On line 10 a logarithmically spaced vector of step sizes is created. A `for` loop is used to calculated the eigenvalues for each entry of this vector. Within the loop a grid with the current resolution is created on lines 13 and 14. On this grid a Gaussian permittivity distribution is set up using the parameters of the example structure. The permittivity distribution is shown in figure 1. On line 17 the eigenvalues are calculated using the `guided_modes_1DTE` function. Finally, a subset of the eigenvalues is stored in the array `eff_eps`. In the last line two lines the relative error

$$e = \left| \frac{\varepsilon_{\text{eff}} - \varepsilon^*_{\text{eff}}}{\varepsilon^*_{\text{eff}}} \right| \tag{5}$$

is calculated. The true value $\varepsilon^*_{\text{eff}}$ is unknown. It is approximated by the result for the smallest value of h.

In the next step we determine the exponent k of the power law

$$e = a \cdot h^k \tag{6}$$

that governs the error over a wide range of parameter values. This is done by fitting a linear function to the log-log representation of the dataset:

```python
# fit power law to the linear section of log-log representation
coefficients = np.zeros((rel_error.shape[1], 2))
exclude_large = 6  # number of large h values to exclude from fit
exclude_small = 7  # number of small h values to exclude from fit

x = np.log10(h[exclude_large:-exclude_small-1])
for i in range(rel_error.shape[1]):
    y = np.log10(rel_error[exclude_large:-exclude_small, i])
    coefficients[i, :] = np.polyfit(x, y, 1)
```

The remaining part of the script is dedicated to plotting the results. It is omitted here for brevity. The resulting graph can be seen in figure 2. Over a wide range of the discretization paramter h the effective permittivity exhibits a quadratic convergence. This corresponds exactly to the convergence rate of the finite difference approximation the second derivative.

For very large values of h the convergence becomes irregular, especially for higher order modes with large modes numbers. These modes oscillated rapidly in the lateral direction with oscillation periods in the range of a few microns. If the discretization is chosen too coarse, such features will not be represented correctly and the results will be wrong. The lateral dynamics of the eigenmodes depends on the permittivity, the permittivity contrast, the lateral extent of the permittivity distribution and the wavelength. Hence, a convergence check for the step size h should be performed for every geometry and parameters set.

As mentioned above, the results depend also on the size of the computational domain. The convergence in dependence of the grid size is tested with the following script:

```python
h             = 0.2
lam           = 0.78
k0            = 2 * np.pi / lam
e_substrate   = 1.5**2
delta_e       = 1.5e-2
num_modes     = 15
w             = 15.0

grid_sizes = np.logspace(np.log10(2 * w), np.log10(100 * w), 50)
#grid_sizes = np.logspace(np.log10(2 * w), np.log10(10 * w), 10)
eff_eps = np.nan * np.zeros((grid_sizes.size, num_modes))
```

The script is very similar to the discretization test script. The only differences is that now the grid size is varied inside the loop instead of the discretization. The results are shown in figure 3. Once the computational domain is large enough to accommodate a particular guided mode, this mode

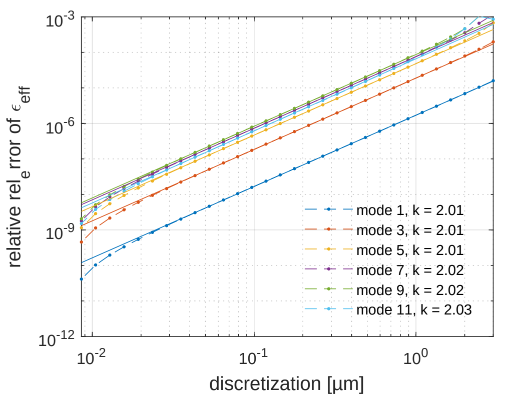

Figure 2: Convergence of the effective permittivity upon variation of the discretization. The solid lines represent a power law fit to the central part of the parameter range.

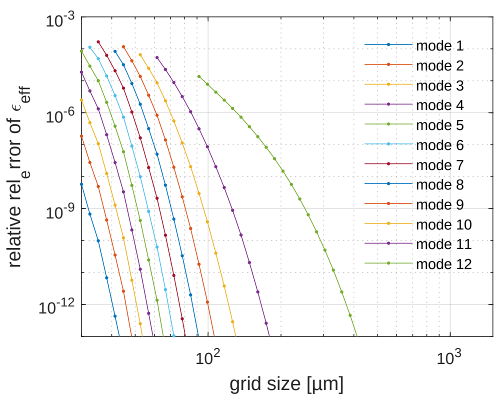

Figure 3: Convergence of the effective permittivity upon variation of the grid size.

converges very rapidly with the grid size as its field decay exponentially with the distance from the guiding structure. However, modes that are guided very weakly can have substantial features in the cladding region before their field decays. They require a quite large computational domain to be represented correctly (e.g. mode number 12 in figure 3). Fortunately, such modes are seldom of practical relevance.

Take-home message: You should test the correctness of any simulation you perform! One important part of that (but not the only one) is to test the convergence of your simulation by varying the critical parameters of the algorithm. For the finite difference scheme discussed here those are the discretization and the grid size. But depending on the numerical algorithm you may have to vary the time step or the number of modes taken into account instead. For the actual simulation you should then select parameters that provide you with the required accuracy.

1.3 Example

Having performed some convergence tests, we can know have a closer look at the guided modes of the example structure. We choose a discretization of $h = 0.2\,\mu\text{m}$ as a good balance between numerical accuracy and computational costs. A grid size of 120 µm will be sufficient to find even very weakly guided modes as can be seen in figure 3.

The test script starts by defining the example parameters ($w = 15\,\mu\text{m}$, $\lambda = 0.78\,\mu\text{m}$, $\varepsilon_s = 2.25$ and $\Delta\varepsilon = 0.01$):

```
 1  grid_size     = 120
 2  number_points = 601
 3  h             = grid_size/(number_points - 1)
 4  lam           = 0.78
 5  k0            = 2*np.pi/lam
 6  e_substrate   = 2.25
 7  delta_e       = 1.5e-2
 8  w             = 15.0
 9  xx            = np.linspace( -grid_size/2, grid_size/2, number_points );
10  prm           = e_substrate + delta_e * np.exp(-(xx/w)**2);
```

On line 10 the Gaussian permittivity distribution is calculated. It is shown in figure 1.

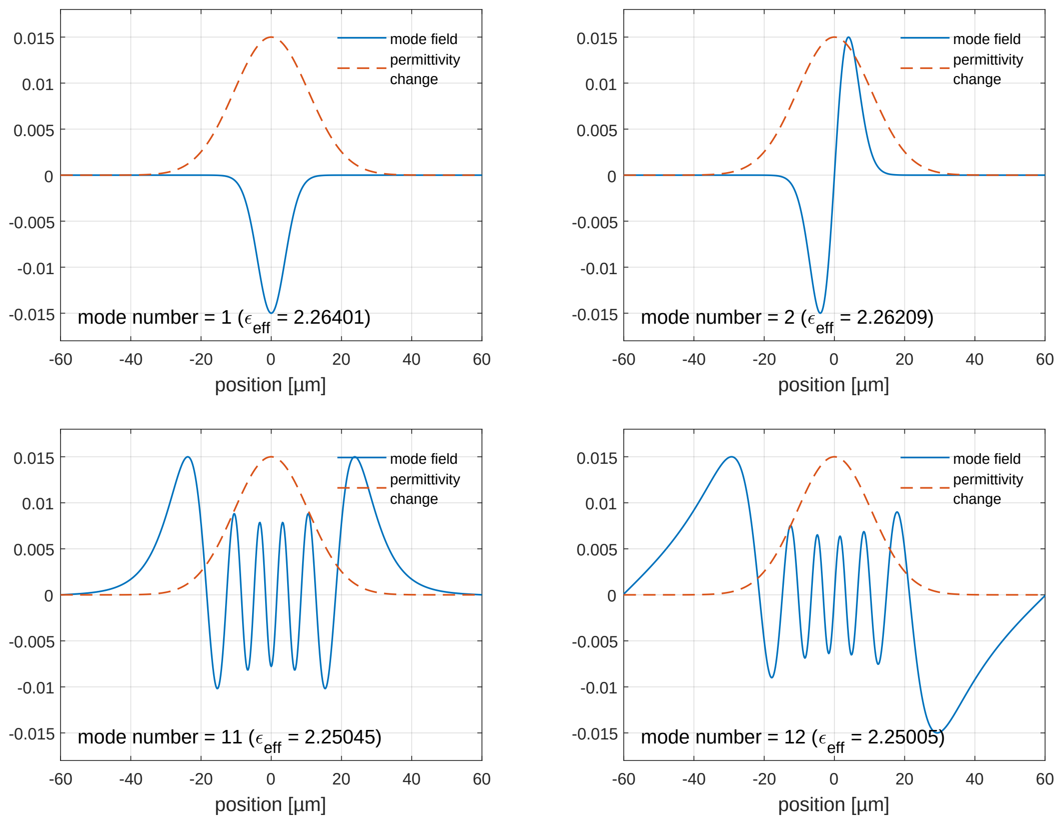

Figure 4: Selected TE eigenmodes of a one-dimensional Gaussian permittivity distribution. Note the increasing delocalization and the increasing number of nodes of higher order modes.

Next we use our mode solver to find the eigenmodes and plot the fields of the different modes:

```
1   eff_eps, guided = guided_modes_1DTE(prm, k0, h);
2
3   for i in range(len(eff_eps)):
4       plotfield = guided[:,i]
5       ind = np.argmax(np.abs(plotfield))
6       plotfield = plotfield / plotfield[ind]*delta_e;
7
8       f = plt.figure()
9       plt.plot(xx, plotfield, color=colors[0], label='mode field')
10      plt.plot(xx, prm - e_substrate, '--', color=colors[1],
11               label='permittivity\nchange')
12      plt.legend(frameon=False, loc='upper right')
13      plt.xlim(xx[[0, -1]])
14      plt.ylim(np.array([-1.2, 1.2])*delta_e)
15      plt.text(xx[0]*0.9, -1.05*delta_e, 'mode number = {0:d} '
16               '($\\epsilon_{{eff}}$ = {1:1.6g})'.format(i+1, eff_eps[i]))
17      plt.xlabel('position [µm]')
18      plt.ylabel(' ')
19      plt.show()
```

The field amplitudes are scaled to the range of the permittivity change in line 6. This allows plotting both in the same axes. A selection of eigenmodes of the example structure is shown in figure 4. Some fundamental properties of the modes can be learned from these plots. Apparently, neighboring modes exhibit opposite field symmetries with respect to the x-axis. Consequently, the

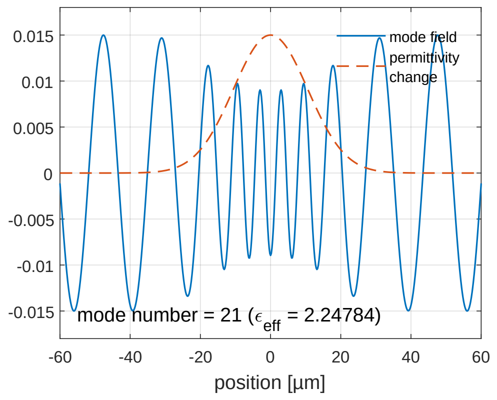

Figure 5: A spurious cavity mode introduced by the conducting boundary conditions and the finite computational domain.

number of nodes (i.e. the number of zero crossings along the x-axis) increases monotonically with the mode number. Higher order modes also have a larger mode area and extend further into the cladding region which is reflected in the decreasing effective permittivity.

What happens if we disable the filtering of guided modes in our function `guided_modes_1DTE`? This can be achieved by changing the block starting in line 2-4 to the following:

```
# pick only guided modes
idx = eff_eps > max(prm [0] , prm [ -1])
eff_eps = eff_eps[ idx ]
guided = guided[: , idx]
```

Our altered function `guided_modes_1DTE_all` now returns all eigenmodes of the operator matrix. If you have a look at modes below cut-off (higher mode number, $\varepsilon_{\text{eff}} < \varepsilon_{\text{s}}$), they do not decay towards the boundaries but show a typical oscillatory behavior that is illustrated in figure 5. It is important to note that these are not real modes of the physical system (which is infinitely large), but cavity modes which are there due to the limited size of the computational domain and the perfect conducting boundary conditions we used in our implementation of the mode solver (you can see that the field tends to zero at the borders of the domain). This is a good example that illustrates how important it is to take care about discretization, boundary conditions and physics of the problem before doing a numerical computation. Not all results of a specific algorithm might be actually reasonable in a physical context.

2 Task 2 – The 2D Case

2.1 Implementation

The implementation of the two-dimensional mode solver requires the discretization of the linear differential operator

$$H = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + k_0^2 \varepsilon(x, y) \tag{7}$$

of the scalar eigenvalue equation

$$H\,E(x,y) = \beta^2 E(x,y). \tag{8}$$

We implement the 2D mode solver by making use of sparse matrices. Otherwise, the problem will not fit into memory!

```python
import numpy as np
import scipy.sparse as sps
from scipy.sparse.linalg import eigs
```

The actual implementation looks like this:

```python
def guided_modes_2D(prm, k0, h, numb):
    """Computes the effective permittivity of a quasi-TE polarized guided
    eigenmode. All dimensions are in µm.

    Parameters
    ----------
    prm  : 2d-array
        Dielectric permittivity in the xy-plane
    k0 : float
        Free space wavenumber
    h : float
        Spatial discretization
    numb : int
        Number of eigenmodes to be calculated

    Returns
    -------
    eff_eps : 1d-array
        Effective permittivity vector of calculated eigenmodes
    guided : 3d-array
        Field distributions of the guided eigenmodes
    """

    NX, NY = prm.shape
    prm = prm * k0**2
    N = NX * NY
    ihx2 = 1 / h**2
    ihy2 = 1 / h**2

    md = -2.0 * (ihx2 + ihy2) * np.ones(N) + prm.ravel(order='F')
    xd = ihx2 * np.ones(N)
    yd = ihy2 * np.ones(N)
    H = sps.spdiags([yd, xd, md, xd, yd], [-NX, -1, 0, 1, NX], N, N,
                    format='csc')

    # remove the '1' when moving to a new line
    # -> look into script pg. 31, upper and lower blue line in first figure
    for i in range(1, NY):
        n = i * NX
        H[n-1, n] = 0
        H[n, n-1] = 0
```

To bring equation (8) into matrix form, the two-dimensional field and permittivity distributions are flattened into one-dimensional arrayws. The main task is to populate the correct diagonals of the operator matrix. The main diagonal accommodates the flattened permittivity `perm(:)`. It is associated with the field values at the central coordinates (i, j) that appear both in the x- and y

derivatives. The secondary diagonals are associated with the field at the coordinates $(i-1, j)$ and $(i+1, j)$ in the x-derivatives, and the far diagonals which are offset by N_x from the main diagonal are associated with the field at the coordinates $(i, j-1)$ and $(i, j+1)$ in the y-derivatives.

All diagonals are first generated as one-dimensional arrays (lines 30 to 32) and then concatenated to an 2D array that is passed to the `spdiags` function on line 33. This function creates a sparse matrix by using the one-dimensional arrays of the first parameter to fill the diagonals specified by the elements of the second parameter. The last two parameters specify the size of the sparse matrix. Also we specify the format of the sparse matrix (compressed sparse column format) as one that efficiently allows to perform the eigenvalue calculation.

The `for` loop starting in line 38 corrects some errors that have been made in the setup of the secondary diagonals. These diagonals are actually not continuous: Whenever a column of the flattened field matrix ends end the next one starts, there must be a zero on the secondary diagonals to implement the desired electric conducting boundary conditions.

A selected number of eigenvalues and associated eigenvectors of the operator matrix are calculated with the Matlab function `eigs`. The matrix is far too large to calculate all of its eigenvalues.

```
    # solve eigenvalue problem
    eigvals, eigvecs = eigs(H, k=numb, which='LR', maxiter=3000)
    eff_eps = eigvals / (k0**2)
```

The `eigs` function takes a structure with options as its last argument. These options can be used to tune the convergence of the iterative method that is employed internally to find the eigenvalues. For further details please have a look at the documentation. The eigenvalues are calculated beginning from the algrebraically largest eigenvalue as indicated by the string `'LR'` that is passed as third argument to `eigs`. We choose this behavior as the eigenvectors belonging to guided modes have the largest effective permittivity values.

The remaining part of the implementation is dedicated to filtering the guided modes and to sorting the eigenvalues and eigenvectors like in the 1D case:

```
   # pick only guided modes
    eps_clad = max([prm[:, 0].max(), prm[:, -1].max(),
                    prm[0, :].max(), prm[-1, :].max()])
    ind = eff_eps > eps_clad/k0**2
    eff_eps = eff_eps[ind]
    eigvecs = eigvecs[:, ind]

    # sort modes from highest to lowest effective permittivity
    # (the fundamental mode has the highest effective permittivity)
    idx = np.argsort(-eff_eps)
    eff_eps = eff_eps[idx]
    eigvecs = eigvecs[:, idx]
```

Finally the eigenmodes are cast from flat vector form back into matrix form using the `reshape` function:

```
    # reshape eigenvectors to a 2D matrix and store them in a 3D array
    guided = np.zeros((len(eff_eps), NX, NY), dtype=eigvecs.dtype)
    for i in range(len(eff_eps)):
        guided[i, :, :] = np.reshape(eigvecs[:, i], (NX, NY), order='F')

    return eff_eps, guided
```

2.2 Alternative Implementation

A glance at the documentation of SciOPy's help reveals that the function `eigs` actually does not need the complete operator matrix. To find the largest eigenvalues it is actually sufficient to supply a function handle that calculates the result $r = H \cdot f$ of the application of the discretized 2D mode operator matrix H to a given vector of field values f. With this knowledge we can simplify the implementation of our 2D mode solver:

```
1    """Computes the effective permittivity of a quasi-TE polarized guided
2      eigenmode. All dimensions are in µm.
3
4      Parameters
5      ----------
6      prm  : 2d-array
7          Dielectric permittivity in the xy-plane
8      k0 : float
9          Free space wavenumber
10     h : float
11         Spatial discretization
12     numb : int
13         Number of eigenmodes to be calculated
14
15     Returns
16     -------
17     eff_eps : 1d-array
18         Effective permittivity vector of calculated eigenmodes
19     guided : 3d-array
20         Field distributions of the guided eigenmodes
21     """
22
23     NX, NY = prm.shape
24     N = NX * NY
25
26     matvec = lambda x: mode_operator_2D(x, prm, k0, h)
27     H = sps.linalg.LinearOperator(dtype=np.float64, shape=(N, N),
28                                   matvec=matvec)
29
30     # solve eigenvalue problem
31     eigvals, eigvecs = eigs(H, k=numb, which='LR', maxiter=3000)
32     eff_eps = eigvals / (k0**2)
```

The construction of the sparse operator matrix has been replaced by the definition of a function handle on line 27. The specific syntax defines an anonymous function with a single parameter by binding the remaining 3 parameters of the function `mode_opeator_2d` to already existing local variables. The function `mode_opeator_2d` takes care of the application of the mode operator. The call to the `eigs` function on line 29 is slightly different from the matrix version: Instead of the sparse operator matrix the function handle `mode_fun` and the size of the field vector `N` are passed as the first two arguments. The remaining arguments are the same as in the original version. The rest of the implementation is basically the same as in `guided_modes_2D.m`.

But how does the function `mode_operator_2D` look like?

```
def mode_operator_2D(field, prm, k0, h):
    """ Calculates the scalar finite differnece mode operator in 2D.

    Parameters
    ----------
    field : 1d-array
```

```
 7            Array containing the unwrapped field.
 8        prm : 2d-array
 9            Dielectric permittivity in the xy-plane.
10        h : float
11            Spatial discretization
12        """
13        field = field.reshape(prm.shape, order='F')
14        res = -4 * field
15        res[:-1, :] += field[1:, :]
16        res[1:, :] += field[:-1, :]
17        res[:, :-1] += field[:, 1:]
18        res[:, 1:] += field[:, :-1]
19        res /= h * h
20        res += prm * k0 * k0 * field
21        return np.ravel(res, order='F')
```

The function is actually a straight forward translation of the finite difference scheme that was given in the seminar into vectorized NumPy expressions using in-place operations for speed and memory efficiency. There is no hint of a sparse matrix anymore. Line 13 restores the original matrix form of the field from the vector that will be supplied by the `eigs` function. Line 14 and line 20 correspond to the main diagonal of the operator matrix. Lines 15 and 16 belong to the x-derivative of the Laplacian and correspond to the secondary diagonals of the operator matrix, lines 17 and 18 belong to the y-derivative and correspond to the far diagonals that are offset from the main diagonal by N_x in the operator matrix. Finally, on line 21 the result is reshaped into a vector again as it is expected by the `eigs` function.

Ordering:the Python code above specifies `order="F"` on reshaping and flattening. This has historical reasons as we tried to match the Python code in its behavior to the Matlab solutions. And Matlab by default uses Fortran ordering and concatenates the columns of a matrix upon flattening. NumPy by default uses C ordering which does the opposite except when the ordering is specified by a keyword.

2.3 Convergence Tests

The mathematical structure of the two-dimensional mode solver is closely related to the one-dimensional case. Consequently, the influence of step size h and the grid size onto the solution will be very similar. As the geometries of the 1D and the 2D examples are comparable (same permittivity contrast, same width of the Gaussian, same wavelength) we can rely on the results of the first task.

2.4 Example

The generalization of the Gaussian permittivity distribution to two dimensions is straight forward. The parameters are the same is in the 1D case. As before the test script starts with defining the example parameters and then generates the permittivity distribution:

```
grid_size     = 120
number_points = 601
h             = grid_size/(number_points - 1)
lam           = 0.78
k0            = 2*np.pi/lam
e_substrate   = 1.5**2
delta_e       = 1.5e-2
```

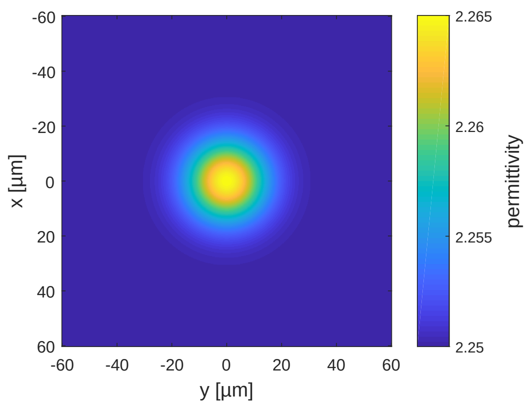

Figure 6: Two-dimensional Gaussian permittivity distribution.

```
 8  w                = 15.0
 9  xx               = np.linspace(-grid_size/2-h,grid_size/2+h,number_points+2);
10  yy               = np.linspace(-grid_size/2,grid_size/2,number_points);
11  prm              = e_substrate + delta_e * np.exp(-(xx/w)**2);
12
13  [x,y] = np.meshgrid(xx,yy, indexing='ij');
14
15  # make permittivity slightly asymmetric to enforce consistent orientation
16  # of eigenmodes
17  prm = e_substrate + delta_e*np.exp(-(x**2 + (1-1e-6)*y**2)/w**2)
18
19  plt.figure()
20  plt.pcolormesh(x, y, prm, cmap='inferno', zorder=-5)
21  plt.xlabel('x [µm]');
22  plt.ylabel('y [µm]');
23  plt.tight_layout()
24  plt.gca().set_aspect('equal')
25  plt.gca().set_rasterization_zorder(0)
26  cb = plt.colorbar(fraction=0.2)
27  cb.set_label('permittivity')
28  plt.show()
```

The result is shown in figure 6. The only new thing to learn is how to generate a 2D grid. This is achieved with the command `meshgrid` in line 13. It generates a x-matrix whose entries are constant along the rows and change along the rows and change along the collumns and a y-matrix whose entries are constant along the columns and change along the rows. Note that by default `meshgrid` uses the inverse convention, which is why we specified `index="ij"`. The permittivity distribution is plotted in line 19-28.

A closer look at the source code above reveals two little details: On line 9 we make the x-axis two points wider than the y-axis. This will turn our grid into a rectangular matrix instead of a square one and is a good way of testing for implementation errors. It may reveal many common mistakes, which are due to mixing up the axes or indices. On line 17 the permittivity distribution is made slightly asymmetric by squeezing the y-axis by a tiny fraction. This asymmetry breaks the rotational symmetry of our test geometry and thus lifts potential degeneracies of guided modes. The change is so small that it will hardly be noticeable in the results but it causes a consistent ordering of the modes and a consistent orientation of the mode fields.

After the setup of the example parameters and the permittivity distribution we solve the eigenmodes with our 2D mode solver.

```
t = time.time()
eff_eps, guided = guided_modes_2D(prm, k0, h, 10)
print('Elapsed time: {0}s'.format(time.time() - t))
```

Finally we plot the eigenmodes:

```
for i in range(len(eff_eps)):
    f = plt.figure()
    plotfield = guided[i,...].real
    plotfield /= plotfield.flat[np.argmax(np.abs(plotfield))]
    plt.pcolormesh(x, y, plotfield, cmap='bluered_dark',
                   zorder=-5, vmin=-1, vmax=1)
    plt.xlabel('x [µm]')
    plt.ylabel('y [µm]')
    plt.xlim(xx[[0,-1]])
    plt.ylim(yy[[0,-1]])
    plt.gca().set_aspect('equal')
    plt.gca().set_rasterization_zorder(0)
    plt.gca().add_artist(plt.Circle((0, 0), w, facecolor='none',
                         edgecolor='w', lw=0.5, ls='--'))
    plt.text(0, yy[0]*0.9, 'mode number = {0:d} '
             '($\\epsilon_{{eff}}$ = {1:1.6g})'.format(i+1, eff_eps[i].real),
             ha = 'center')
    plt.show()
```

The amplitudes of the modes are normalized before plotting and the command `caxis` is used to set a symmetric colormap range. A custom diverging colormap (`'bluered_dark'`) is used, as the mode amplitudes assume both positive and negative values but the brightness of the plots should be related to the magnitude of the field independently of the sign. Using the sequential default color map (`'virids'` in recent versions of matplotlib) would cause positive and negative values of the same magnitude to be rendered with different brightness values. The function `plt.Circle` is used to overlay a circle with radius w onto the plots.

The lowest 8 guided modes of the example structure are shown in figure 7. As in the 1D case, the mode area increases with increasing mode number while the effective permittivity decreases. Due to the symmetry of the permittivity distribution some modes are degenerate. They have the same effective permittivity and differ only in the orientation of their fields.[1] With increasing mode number the number of nodes in azimuthal and radial direction increases.

[1]The slight artificial asymmetry that has been introduced intentionally is not noticeable in the displayed number of digits.

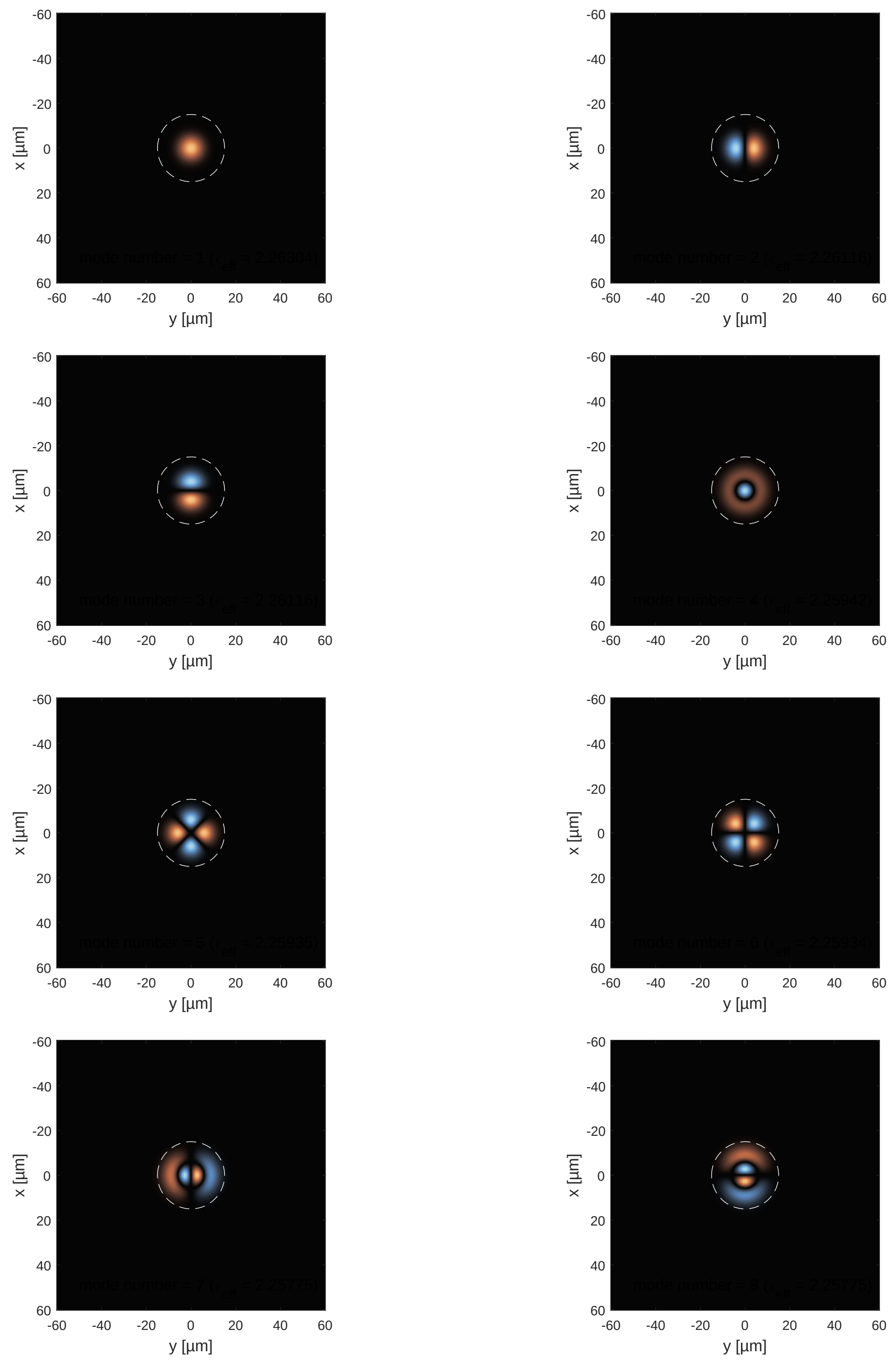

Figure 7: The first 8 guided modes of the two-dimensional Gaussian permittivity distribution. Some modes are degenerate. They have the same effective permittivity and differ only in their orientation. Note the increasing mode area and the increasing number of nodes in radial and azimuthal direction with increasing mode number.