Summary:

- Computational complexity increases significantly in the two-dimensions: Here we are to appreciate this difficulty with the finite-difference BPM using Crank-Nicolson method.
- Practice construction of sparse matrices to represent discretized differential operators
- Appreciate the advantages and disadvantages of direct linear-system solvers.

0.0.1 Construction of sparse-matrices for differential operators

Task 1: Implementation
Starting from the one-dimensional implementation of Crank-Nicolson-BPM (see Practice-Track package WP08), modify the program for two transverse dimensions. An important aspect of this exercise is to re-use as much as possible from the previous implementation, and thus appreciate the common parts of the algorithm.

Solution:

To reduce the code writing to a minimum, it is probably best to utilize the very first implementation of the Crank-Nicolson based beam propagation as a point of departure for this exercise. In Practice-Track package 08 we defined the operator pair L^+, L^- in terms of sparse matrices, and then the main simulator loop only required a few lines of code. The most significant addition to the code is filling in the sparse matrices representing the L^+ and L^- operators, while the main loop should not require any changes.

The matrix representation of these operators must be based on sparse matrices, because the size of the matrix is significantly larger in two dimension, and a great majority of the matrix elements are zeros. We will look in detail on the construction of a sparse-matrix operator — it is a very common and important task in many contexts.

To keep things simple, Matlab and its sparse-matrix capabilities are used in what follows. In this case it is practical to view the sparse matrix structure as a black box; details of how it is implemented are unimportant. Sparse matrices can be parameterized in several different ways. The simplest one is to provide three lists of numbers specifying the rows, columns, and values of all non-zero elements. The construction of these arrays is the first step.

To simplify the code, and thus reduce the room for errors, one can create first a sparse matrix Δ that represents the discrete Laplacian operator on the given computational grid. If the sparse-matrix library supports matrix algebra, $L^{\pm}$ can be simply derived from Δ.

The computational grid representing the transverse cross-section of the beam is now two-dimensional; indices i and j will be used to specify a grid location along the x and y directions, respectively. It will be assumed that the number N of grid points in both directions is the same. What we need next is a mapping the location in the grid onto an index specifying a column (or a row) in the sparse matrix. In Matlab, the following function realizes such a mapping:

$$\mathrm{M} = @(i,j)\ j * N + i - N$$

It should not be difficult to see that it assigns to (i, j) an index $M(i, j)$ that represents the position in the given grid as counted in the typewriter order. Alternatively, one can recall that there are no "real" matrices in the computer memory. Rather, every array is stored as a linear vector. The above is nothing but a position in such a vector. Note that the definition of this function will be slightly different in different languages. In Matlab, where one starts indexing with one, the first position $(1, 1)$ maps to index 1 as it should. In C, the mapping would be $j * N + i$, with $(0, 0)$ mapped to 0.

Having specified the correspondence between the sparse matrix index and the computational grid location, we are ready to identify the non-zero matrix elements of the discrete Laplacian operator. Recall that the diagonal elements of this operator (on an isotropic grid) are simply the number of nearest neighbors but with a negative sign. For the non-diagonal elements, we have one +1 for each pair of nearest neighbors. In other words, we need

$$\Delta_{ij,ij} \to \Delta_{M(i,j),M(i,j)} = -4 \quad i, j = 1, \ldots, N$$

for the diagonal part of the operator, and

$$\Delta_{ij,ij\pm1} \to \Delta_{M(i,j),M(i,j\pm1)} = +1 \quad i, j = 1, \ldots, N$$

for the non-diagonal part originating in partial derivatives w.r.t. y, and finally

$$\Delta_{ij,i\pm1j} \to \Delta_{M(i,j),M(i\pm1,j)} = +1 \quad i, j = 1, \ldots, N$$

for the non-diagonal part representing partial derivatives w.r.t. x. From here, it is also easy to see that the size of the matrix Δ is $N^2 \times N^2$, and the number of non-zero elements in it is not greater that $5 \times N^2$. We therefore need three arrays of this length that will hold the column, row, and the value of each non-vanishing element of Δ.

The following listing illustrates the relevant code that constructs the sparse matrix operator for the discrete Laplacian an a square grid:

Listing 1: Filling in a sparse matrix

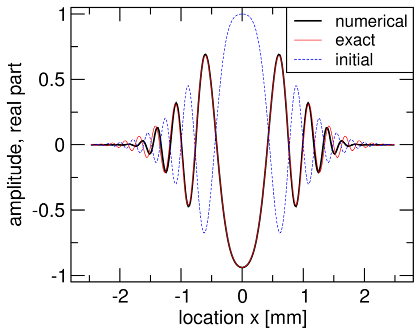
```
% maximal number of non-zero matrix entries
nzmax = N*N*5;

% these arrays will encode sparse matrix of DELTA:
rows = zeros(nzmax,1);  %rows
cols = zeros(nzmax,1);  %columns
valp = zeros(nzmax,1);  %values

% mapping grid location to matrix index
M = @(i,j) j*N + i - N;

% enumerate all non-zero matrix elements
count = 0;    % count how many non-zeros enumerated so far

% loop over each location in the grid
for i=1:N
for j=1:N
```

```

% diagonal elements
loc = M(i,j);  % central stencil point location

count = count + 1;  % update count
rows(count) = loc;  % loc is the row index
cols(count) = loc;  % loc is the column index
valp(count) = -4;   % diagonal element value

% nondiagonal elemens for grid neighbors on the ``right''
if(i<N)  %right neigbor must exist!
  loc = M(i,j);     % central stencil point location
  nnl = M(i+1,j);  % and its nearest neighbor location

  count = count + 1;
  rows(count) = loc;
  cols(count) = nnl;
  valp(count) = 1;  %non-diagonal value
end

% similar code for grid neighbors on ``left''
...
% similar code for grid neighbors ``up''
...
% similar code for grid neighbors ``down''
...
end  % j loop
end  % i loop

% discrete Laplacian operator: insert elements into sparse matrix
DELTA = sparse(rows(1:count),cols(1:count),valp(1:count),N*N,N*N,count);

% auxiliary Kronecked delta (identity) matrix
KD = sparse(1:N*N,1:N*N,1);

% construct Lplus, Lminus
idelta   = 1i*dz/(4*k0*dx^2);

LP =  KD + idelta*DELTA;
LM =  KD - idelta*DELTA;
```

Needless to say, the above could be programmed more efficiently, but this codelet is meant to illustrate the general idea of how sparse-matrix operators are constructed in numerical solutions of partial differential equations. Instructor‘s version for this program is included in *Method.m*.

0.0.2 Issues in 2D: computational complexity increase

Task 2: Testing
Test the implementation of the 2D beam simulator with the analytic Gaussian beam solution. Start with a modest grid size and a not too-tightly focused initial condition in order to make the simulation easier. Make sure to numerically propagate for at least one Rayleigh range so that the beam profile exhibits significant change and thus serves as a basis for a non-trivial test. Reasonable agreement between numerical and analytic solution must be achieved before proceeding to Task 3. While this exercise should be relatively straightforward and is left to the reader, the following figure should give a sense of what kind of agreement between the simulated and analytic solution one can expect for a "modest" simulation with a relatively coarse grid resolution and a long integration step.



Testing 2D Crank-Nicolson BPM algorithm. Only real part of the complex beam amplitude is shown for a simulation starting from an initial beam (dashed blue) focused to a distance of 0.5 m. Simulated (black) and analytic (red) solutions are shown at a distance of $2f$. The computational grid was 300×300 points, representing an area of 5×5 mm. The integration step was chosen to be 1 cm for this beam with wavelength of 800 nm. Note how the near-perfect agreement in the center deteriorates further from the beam axis.

For further exploration:

i) The above illustrated simulation only requires several seconds to execute. Experiment with the simulation parameters in an attempt to improve the numerical-analytic agreement.

ii) Observe the evolution of the numerical solution and identify the propagation distance when the deviation from the analytic target becomes most obvious. Explain your observation.

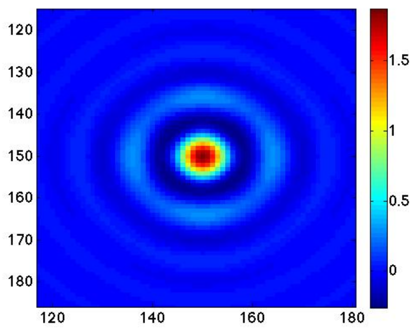
iii) This implementation utilized a direct linear-system solver. This means that the matrix of the linear system of equations must be prepared and passed to the library routine that implements the solver. Initially this is a sparse matrix, but as the direct solution proceeds many of the zero elements become non-zero in the transformed matrix. As a result, more memory is needed to store the matrix and this shows up in the memory usage of the program. Readers should observe, during the simulation, how the memory volume in use increases and subsequently fluctuates. This has two unpleasant effects: First, depending on the solver implementation, memory may be repeatedly requested from and released to the system. This is an expensive operation and may be a cause for poor performance when the size of the problem is large. Second, it is in general difficult to know how much memory is sufficient for the given problem because we do not know before hand what will be the filling effect on the sparse matrix representing the linear system. These are just a few reasons why alternatives to direct solvers are also often used, namely iterative solvers which do not require storage for the matrix at all. On the other hand, they also do not guarantee that a solution will be found... We will have an opportunity to experience and compare the advantages and disadvantages of direct and iterative solvers later in this course.

0.0.3 Issues in 2D: grid anisotropy

Task 3: Grid anisotropy manifestation
Set up a simulation with as tight focusing as feasible for a short simulation (experiment with parameters!), and run the evolution until the focal region is reached. Inspect the central lobe of the solution, and observe how the anisotropy of the grid shows up. Try to modify the integration step and/or the grid resolution in order to reduce the observed anisotropy in the beam pattern. During these simulations, note the memory usage and also the CPU utilization if running on a multiprocessor.

Solution:

To observe numerical anisotropy effects in the simulated solution, one needs to "populate" the initial beam profile with waves that have sufficiently high transverse wavenumbers. Such waves show the strongest difference between their propagation constants when the transverse wave-vector points along the grid axis and along the diagonal. On the other hand, the reader should not impose a too tight focusing geometry, because then the simulation needs shorter step and finer transverse resolution otherwise the accuracy suffers. However, it should become evident with little experimentation with the simulation parameters, that anisotropic wave propagation is actually rather easy to see. The following illustration shows data obtained on a grid of 300×300 points, with the computational domain size of 5×5 mm, for a beam with the wavelength 800 nm, and beam waist of 1 mm. The focal length was chosen $f = 0.4$ and the propagation distance was the same.



Center of a simulated beam (real part of the amplitude) close to its focus. Note the variation of the intensity evident in the rings. The pattern has a four-fold symmetry inherited from the grid-symmetry. The origin of this artifact is that the phase velocity of the wave is different when propagating along the grid axis or along the grid diagonal.

This picture demonstrates that numerical anisotropy shows up readily. Readers should try several sets of input parameters and attempt to reduce the anisotropy of the numerical solution — it will become evident very soon that it is not an easy (or numerically inexpensive) task. In fact, this simulation is an example of how a radially symmetric problem *should not* be solved on a rectangular grid.

Take-away lessons:

- two dimensional beam propagation becomes significantly more demanding, computationally, than the problems in one transverse dimension explored in previous sections

- the idea of the Crank-Nicolson method remains the same in two and in higher dimensions, and the programming is not much different or more complicated if a direct linear solver library routine can be used
- even a small problem may require significant memory with the direct linear solver, because the originally very sparse system matrix fills up with many more non-zero elements during the course of the solution
- numerical dispersion and accuracy issues become significantly more severe, because simulations are in general executed with lesser resolutions and longer integration steps than in “smaller,” one-dimensional problems.
- numerical grid anisotropy shows up readily in the simulated beam solutions