Implementation of the implicit BPM scheme

It is often convenient, both for derivation and for ease of implementation in a program, to represent the core of the numerical algorithm in the form using matrices, or operators. It is practically a rule in the BPM field...

Here we take the opportunity to practice such an approach in the most simple case. What is done next will be utilized in the subsequent sections, and in particular in different variations on the Crank-Nicolson theme...

Write the implicit BPM-update scheme in the "operator" form

$$E_{old} = L^{(-)} E_{new} \ ,$$

where the matrix acts on the vector (array) storing our one-dimensional complex amplitude representing the profile of the beam:

$$\left(L^{(-)}E\right)_i = \sum_k L^{(-)}_{ik} E_k = \{E_i - \delta(E_{i-1} - 2E_i + E_{i+1})\}$$

with δ standing for the factor

$$\delta = \frac{i\Delta z}{2k_0 \Delta x^2}$$

formed from the "nonphysical" parameters characterizing the discrete form of our BPM equation. It will be illuminating to think of δ as the parameter that controls the departure of operator $L^{(-)}$ from the identity matrix. Similar matrices, and in particular $L^{(+)}$, will appear in our derivations very soon.

In this exercise, we implement the BPM solver in Matlab, and in doing so we take advantage of its matrix-handling functionality. We simply define the above matrix equation, and ask Matlab to apply whatever solver it deems appropriate to solve it. So the core of our main program will contain lines similar to (see instructor's solution in *Main.m*):

```
%% PROPAGATE
Eold = E0;                    % starting from the initial condition E0

   for k=1:nsteps
     z    = k*dz;             % keep track of the actual propagtion distance
     Enew = LM\Eold;          % finds solution to LM Enew = Eold
     Eold = Enew;             % current beam profile always resides in Eold...

     Observer_Report          % output/visualization after each step
   end
%% END PROPAGATE
```

The matrix or operator defining our implicit update scheme happens to be tri-diagonal. This is an important property that in fact influences and guides the design of much of the core BPM methods. However, at this point, we leave it to the linear system solver to recognize the specific type of the matrix and we just hope that the solver will use it to its advantage. We will look at how to solve this kind of the system efficiently later in this course.

The most error-prone step in setting up a BPM simulation is often the stage when the linear system matrix is defined. In this case, $L^{(-)}$ consists of the diagonal and two adjacent upper- and lower-diagonals. The following lines (from *Method.m*) tell Matlab to insert the corresponding matrix elements into a sparse matrix LM holding operator $L^{(-)}$:

```
  delta      = dz/(2*k0*dx^2);
  idelta     = 1i*delta;

  LM_diag = zeros(NX,1);               % NX is the grid dimension
  LM_diag(1:NX) = 1 + 2*idelta;        % this is the vector that constitutes the diagonal

  offdiag = ones(NX-1,1)*idelta;       % both off-diagonals are shorter by one element

 LM   =   sparse(1:NX-1,2:NX,   -offdiag,   NX,NX)+...   % fills in the upper diagonal
          sparse(1:NX,   1:NX,    LM_diag,   NX,NX)+...   % fills in the diagnal
          sparse(2:NX,   1:NX-1,-offdiag,   NX,NX);       % fills in the lower diagonal
```

The reason that we utilize a *sparse* matrix holder here is that most the matrix elements in $L^{(-)}$ are zeros. This fact not only reduces the necessary memory storage, but also makes calculation with such a matrix (e.g. matrix-vector multiplication) much more efficient.

Note that in any other than Matlab programming environment, we would store the elements of the matrix in auxiliary arrays. For now we trust Matlab to execute the operations we ask for in an efficient way. The drawbacks of such an approach is that the user does not enforce the most efficient method explicitly. The obvious advantage of it is that it only takes a few lines of the code to implement what is actually a non-trivial program. This is just what we need at this stage. The above lines of code implement the most important part of the implicit method studied in this exercise. The rest is "support" code that realizes the initial conditions, simulation parameters, and visualization and result outputs. Program *Main.m* calls the corresponding routines and executes the simulation that demonstrates that the implicit BPM method is indeed stable. The run starts from an initially focused Gaussian beam and propagates the solution beyond its focus region. Variation of the "nonphysical" parameters such as *Deltax*, Δz, Lx, ... shows that the instability of the explicit method has been successfully eliminated...

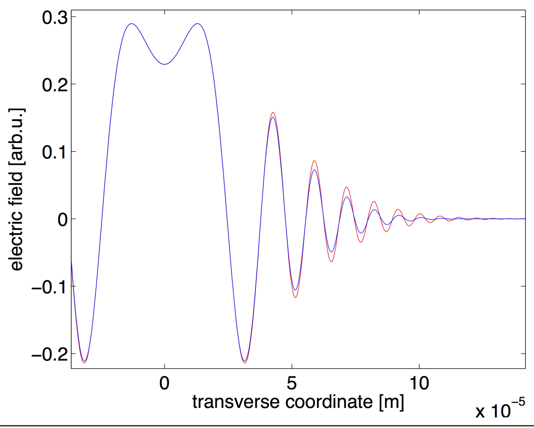
Testing and accuracy

Every program, no matter how trivial, must be tested. Gaussian beam solution (in one transverse dimension) is utilized here to test the BPM implementation. The initial condition is generated by *GaussianBeam1D.m*, and the BPM solver propagates this over certain propagation distance where it is compared to the target solution also generated by *GaussianBeam1D.m.*

Program in *Test.m* executes the numerical-vs-analytic comparison. Readers are invited to explore various parameter settings and get a sense of:

- if the numerical solution agrees with the analytic-formula target
- where (in the real space) the deviations between the two show up
- what parameters control these deviations
- what properties of the numerical method underline this behavior

The following figure illustrates such a comparison run.



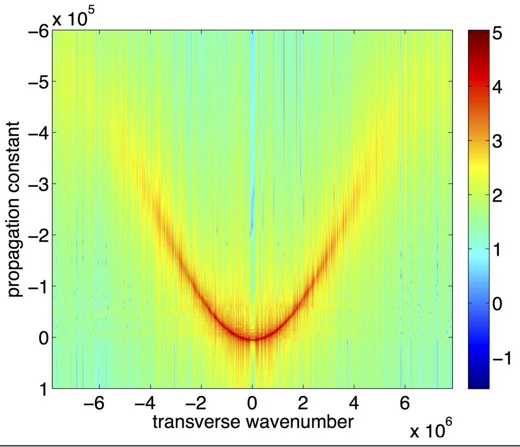
Testing the implicit BPM implementation versus the analytic Gaussian beam solution. Initially collimated Gaussian beam is propagated over several Rayleigh ranges, and the numerical and analytic solutions for the electric field are compared here. The agreement of the two in the on-axis region around $x \approx 0$ indicates that the implementation of the method probably works as it should. The gap between the two solutions is clearly visible further from the axis...

The very first question that this comparative simulation invites is of course if the deviation seen between the analytic and numerical results is due to the limitation of the method itself, or it is caused by how the method was implemented. Since the analytic solution is generated by the formula from the *paraxial* Gaussian beam, one should expect that if the method is properly implemented, one must be able to choose simulation parameters to control (minimize) the error. In other words, one expects this method to converge to the continuum limit of the paraxial beam propagation equation, to which the Gaussian beam is an exact solution. It is left to the reader to verify that the numerical solution can be made closer to its analytic counterpart by choosing a sufficiently fine grid resolution Δx, and/or the integration step Δz.

It is important to give a proper interpretation to the error observed in the numerical solution. In the above figure it becomes obvious that the error appears largest away from axis. This is because it is the wave components that propagate at an angle w.r.t. the axis that contribute most to this portion of the solution. At the same time, we know from our experience with the numerical Maxwell solver that these waves must suffer from the effects of *numerical* dispersion. In other words, they are not even expected to propagate the same way their continuum-limit counterparts do. This is the reason why the error is most visible off-axis. To verify that the reason just put forward indeed applies, the reader could execute a simulation with a single plane wave propagating at an angle, and study the error as a function of its propagation angle. The manifestation of the numerical dispersion, and more specifically of its deviation from the continuum limit, is that the numerical waves propagate at angles different (smaller) from those that should correspond to their transverse wavenumbers.

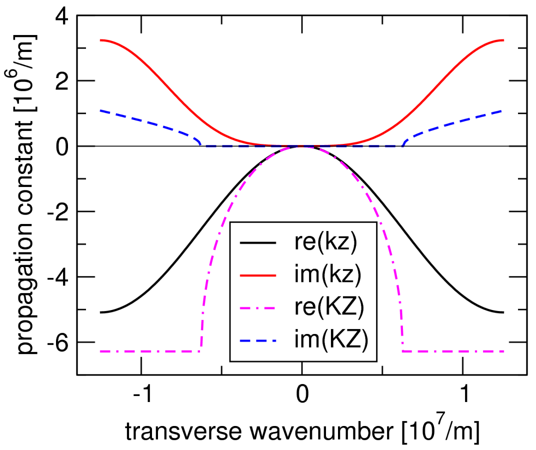
Mapping the numerical dispersion relation

It is instructive to investigate the numerical dispersion with the mapping method previously applied to the one-dimensional Maxwell solver. The initial condition is a white noise for both the real and imaginary part of the solution. This is numerically evolved over some propagation distance, while snapshots of the beam profiles are stored after every integration step. The resulting transverse$\times$longitudinal spatial profile of the complex envelope is Fourier transformed to obtain a spatial-spatial spectrum. This spectrum reveals energy accumulation in the vicinity of the locus that corresponds to the numerical wave dispersion.



Spatial spectrum of the BPM-propagated (initial) white noise reveals the dispersion relation for all possible waves supported by the numerical algorithm. The parabolic structure in the center corresponds to the continuum limit dispersion relation of the paraxial beam propagation equation. Waves with higher transverse wavenumbers are strongly damped — this is the consequence of the implicit nature of the integration scheme.

This figure shows sharply defined longitudinal propagation constants for waves with small transverse wavenumbers. At higher transverse wavenumbers, the imaginary part of the numerical propagation constant causes damping of such waves. This has two consequences, both visible in the figure. First, there is less "energy" in these waves because it was partially dissipated during the propagation. Second, their energy is spread, or blurred over a region of propagation constants, this blurring manifesting the finite lifetimes of the wave.



Comparison of the numerical dispersion curves for the implicit finite-difference BPM, and the exact non-paraxial beam propagation. Curves shown were generated for $\lambda = 1\mu$m, and grid spacing of $\Delta x = 0.25\lambda$. The propagation step is 0^{-8}m. Imaginary parts of the numerical and non-paraxial curves were scaled by a factor of fifty and one tenth, respectively.

That the measured dispersion relation behaves the way it is supposed to can be checked by evaluating the theoretical dispersion formula derived in the class. Figure above shows the real and imaginary parts of the numerical propagation constant versus transverse wavenumber. For comparison, the exact non-paraxial curves are also shown. They indicate the width of the light cone in this figure, and thus show which waves are physical, and which should not be present in the BPM simulation.

The fact that all numerical waves are damped is most important, because dissipation affects not only the noise but also the physical solution we seek to find. While in the context of the beam propagation one does not really need to capture waves with transverse wavenumbers that represent either steep propagation angles or are even beyond the light cone for the given wavelength, even paraxial rays are damped and this is clearly not desirable. The same issue becomes even more serious for example in quantum mechanics, where the preservation of energy in the beam corresponds to the unitary nature of quantum wavefunction evolution.

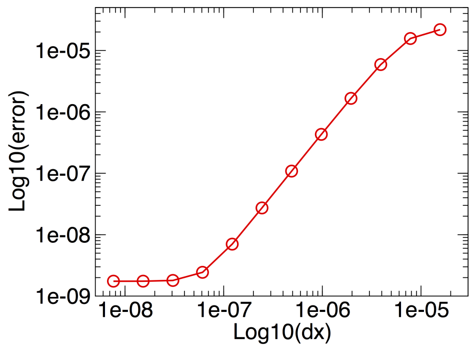
Convergence study

In this exercise, we look at the basics of performing a convergence check of a BPM-method implementation. We have implemented an implicit finite-difference method to solve the basic paraxial beam-propagation equation. The discretization scheme uses a three-point estimated of the second spatial derivative, and as such it should result in a quadratic converge of the numerical solution to the exact one. This rate of convergence will be tested here.

For this simple case, one can take advantage of the fact that an exact solution to the continuum equation exist, namely in the form of a (paraxial) Gaussian beam. This is used to generate both the initial conditions for the numerical simulation, and the target solution that this simulation should approach.

To avoid accumulation of errors, and thus probe only the local error controlled by the discretization scheme, the propagation distance should not be too long. At the same time it should be long enough to create sufficient difference between the initial and target beam profiles.

In this particular example, the goal is to verify the methods properties as the discretization of the spatial grid becomes finer and finer. To this end, a series of runs is executed, doubling the resolution (or the number of grid points) every time. The distance between the numerical and target solutions is evaluated for each resolution and the result is plotted in the log-log scale. The following figure shows an example generated by the instructor‘s solution *ConvergenceStudy.m*.

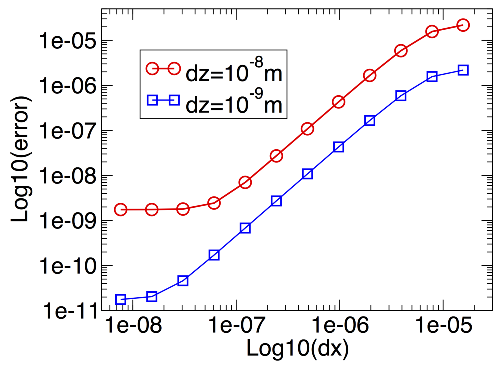


The error of the numerical solution as a function of Δx, the spatial grid‘s spacing. The slope of the linear part of the plot is close to two, corresponding to the order of accuracy of the discretized Laplacian in the paraxial beam propagation equation. The error is measured as the maximum (over the spatial extent of the grid) of $|E(x)_{num} - E(x,)_{exact}|$. Other metrics could be used, leading to a qualitatively similar result.

There are three regions in the curve shown above. The first is controlled by relatively large Δx when the error decreases with decreasing Δx but at a rate slower than one corresponding to the accuracy order one expects based on the discretization scheme. This is simply the manifestation of the fact that error$\approx \Delta x$ behavior is only reached in the limit of small grid spacing.

The middle of the plot is where Δx is sufficiently small to reveal quadratic decrease of the error. This is the regime we look for.

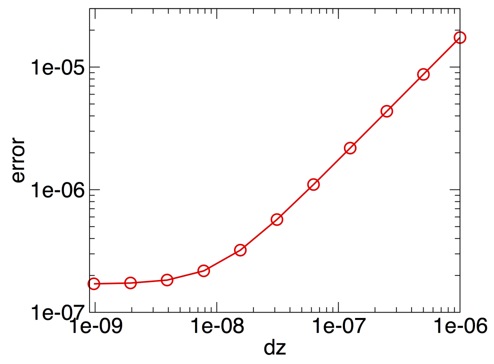
For very small Δx, the spatial grid discretization error become so small that some other error source starts to dominate. That is why decreasing Δx even further does not actually reduce the error any more. In this case, the residual error is controlled by the discreteness of the integration step Δz. This can be checked by running the same series of simulations with a smaller Δz. The reader is invited to check that it extends the region over which error decreases quadratically with Δx. Eventually, for a very small Δz, the error will behave erratically as the machine number accuracy starts to affect comparison of the exact and numerical solutions.



Convergence w.r.t. Δx for two fixed values of Δz. The minimal error achieved for small Δx obviously decreases with Δz, and this is in line with "two" sources or discretization errors. However, first-order accuracy in Δz suggests that the improvement should only be by about an order of magnitude, while the figure shows two orders...

Figure above compares convergence curves for two values of the integration step. As pointed out in the caption, the minimal error improvement seen at the bottom part of the curve is better than the expected one order of magnitude. This occurs because of the setup of this simulation. It is not suitable for measuring convergence w.r.t. Δz, because it does not ensure that "absolutely everything" except the controlled quantity (Δz in this case) remains unchanged during the whole convergence study.

The following figure shows a properly executed convergence check w.r.t. the integration step.



Convergence of the implicit beam propagation methods with respect to the integration step Δz. The first linear part of the plot has a slope close to unity, and this is expected since the accuracy order of the method is one for this parameter. For very small integration steps, the error levels off — this is when it becomes limited by the grid spacing Δx.

To conclude the convergence exercise, let us note that it makes it obvious that refining resolution in one dimension only does not necessarily result in an improved solution. The interplay between the non-physical numerical parameters becomes important, and very often decreasing one requires a corresponding adjustment in the other. Obviously this has an unpleasant consequence that the numerical work required increases even faster.