8 Method of Lines

The following sections will introduce the so-called Method of Lines (MOL). It will be initially introduced as a stand-alone approach to handle paraxial beam propagation equations. Then we show how the FFT- and DHT-BPM, together with the Crank-Nicolson techniques can be utilized as components within the method of lines. This context will also provide an opportunity to discuss various representations of beam evolution equations. In particular, MOL approach will leads us in a natural way to what is an exact first-order evolution system that exactly corresponds to the scalar Helmholtz equation.

8.1 Basic formulation

Central idea and rationale behind the Method of Lines (MOL) is to take advantage of the many available "canned" software implementations for Ordinary Differential Equations (ODE) solvers, and to discretize only transverse dimensions of the beam propagation problem. The propagation dimension is preserved in its continuous form, and does not affect considerations concerning the spatial discretization.

The method of lines, as applied to BPM, has several characteristics that are not found in other approaches:

- it selects and "ordinary" grid in x, y space, and performs disretization of spatial derivatives while disregarding issues of stepping in z
- it means that there is no explicit grid in z. Nevertheles, it should be kept in mind that the solution still "lives" on a discrete-set of points along the propagation direction. The big difference from what we have done with the Crank-Nicolson and Maxwell solvers is that this emerging computational z-grid is not controlled in explicit ways.
- ∂_z is retained without discretization — we imagine that the solution proceeds smoothly along z for each discrete point in the space spanned by x, y. The topology of the whole solution then consists of a family of such "world-lines" — hence the name of this method
- this formulation results in a potentially very large system of ODE's, and the task of solving this system is delegated to a dedicated library. The library implementation should fulfill a few requirements that follow from the nature of the problem; In particular, it should provide robust integration step control, and several choices for *explicit* ODE methods. Implicit methods that utilize Jacobian matrix of the system are utterly unsuitable since they require large memory and significant volume of computation at each step.

As an example, consider the paraxial beam propagation equation with an additional interaction term represented as polarization:

$$\frac{\partial \mathcal{E}}{\partial z} = \frac{i}{2k_0} \Delta_\perp \mathcal{E} + \frac{i\omega}{2cn(\omega)} \mathcal{P}(\boldsymbol{r})/\epsilon_0 \ . \tag{8.1}$$

Concrete form of

$$\mathcal{P}(\boldsymbol{r}) = \mathcal{P}(\{\mathcal{E}\})$$

is unimportant for now — it is some functional of the electric field amplitude, and we assume that an algorithm for its evaluation is provided.

In MOL, left hand side of this propagation equation remains untouched, and discretization is applied to the right hand side only. Assuming that discrete Laplacian Δ is given for the transverse computational grid in x, y, we write

$$\frac{\partial \mathcal{E}_j(z)}{\partial z} = \frac{i}{2k_0} \sum_{i \in NN(j)} \Delta_{ji} \mathcal{E}_i(z) + P_j(z) \ . \tag{8.2}$$

Note that indices i, j may represent grid points in one or two dimensions. Thus, we have here one unknown function of z at each (x_j, y_j), and we assume that an ODE solver can give a solution once an initial condition

$$\mathcal{E}_j^0 = \mathcal{E}_j(z = 0)$$

is specified for all grid points j.

As usual, there are several pros and cons of this approach:

Advantages:

- no need to worry about stability and accuracy — the ODE solver is supposed to take care of this as long as the user provides sufficiently stringent accuracy tolerances
- this approach encapsulates a whole family of methods, and gives us a lot of flexibility in how sub-tasks are handled
- the method can compete with implicit methods in the sense that the user need not explicitly know what constitutes a stable integration step
- MOL-based implementation of beam propagation (and to it similar evolution-type computational problems) can be quite fast
- MOL is much more flexible than fully finite-difference methods
- there is little conceptual difference, or difficulty between related problems in one or two transverse dimensions. In other words, upgrading a code from a one-dimensional version to two dimensions is *much* simpler that in finite-difference methods.

Drawbacks:

- stability is ensured only implicitly since users must rely on the ODE solver implementation not to attempt too long integration steps. This is why the initialization of the evolution must be on the conservative side. It can easily happen that the solution fails only because the initial step Δz prescribed by the user happened to be too long.
- this method *must* be coupled with a robust adaptive-step algorithm. The step adjustment algorithm makes or breaks MOL! Relying on constant z-step approach may work for exercises in this course, but it would be an expensive practice for problems without known solutions.
- this method is rather opaque — when it performs poorly, it may be rather difficult to determine the exact reason.

- because the integration step control is normally hidden from our view, and it plays a crucial role, the "same" methods using different ODE libraries may behave differently.
- performance and robustness also depends on how errors in the solutions are weighted and evaluated. It helps to know what exactly is the ODE solver doing to decide the length of the integration step. Experience with a particular family of problems can be also very helpful in choosing "the right parameters."
- there is no universal way to prove or even to predict that this method works in practical sense for a particular problem. Experimentation is the only way to find out for sure.

8.1.1 Typical implementation considerations

The very first step in MOL implementation consists in choosing an ODE solver library. This usually means that a suitable wrapper functions should be written to provide an interface between the main program and the ODE library. While details may depend on the type of the application, it should be possible to do this in a fairly general way. What one desires is as clean as possible separation of the ODE library interface from the rest of the program. It is then not a disaster when we decide to replace one ODE solver library for another.

In general, ODE solvers require a couple of things from the main program. The first is initialization, in which the user specifies
1. parameters that control the numerical evolution,
2. the initial condition (we solve an initial value problem after all), and
3. the function that stands on the right-hand-side of the ODE system to be solved.

It helps to "map" the BPM evolution problem to the language usually utilized by ODE solvers. In particular, the complex amplitude of the beam maps to a one-dimensional vector of unknowns:

$$\{\mathcal{E}_j(z)\}_{j=1}^{N} \to \{y_k(z)\}_{k=1}^{2N} \ . \tag{8.3}$$

Recall that we have already done this kind of mapping when implementing the two-dimensional Crank-Nicolson method that utilized Matlab's sparse-system solver. We needed to unwrap our two-dimensional grid representation into a one-dimensional array — this is pretty much the same situation. Note that this mapping can be arbitrary as long as we correctly preserve the information about which j is a neighbor of which i. Things get a little more complicated when dealing with several vector components of the optical field, and one has to decide how to best "order" them in the computer memory. Of course, the rule is to keep related quantities close to each other in what is the linear space of computer memory.

There seems to be different number of variables on the two sides of the above mapping — this is because as a rule ODE solvers work with real-valued functions, while we have a complex-valued beam amplitude to calculate. Thus, there is twice as many real items in the vector y. This complex-real issue does not present a problem, all it normally requires is to cast (proclaim) our arrays to be of the required type when they get passed to the ODE solver routine. Similarly, when the solver returns a solution which is a real vector, we can simply cast it back to whatever type is used throughout the main program.

Thus, having decided on how we map our complex beam amplitude $\mathcal{E}$ onto a real vector y, our evolution problem reads

$$\frac{dy_k(z)}{dz} = RHS_k(z, y(z)) \ , \tag{8.4}$$

where RHS is the right-hand-side function ODE solver requires to know. It simply implements the discretization of the propagation equation, and casts the result in the order expected by the solver:

$$\frac{i}{2k_0}\Delta_\perp\mathcal{E} + \frac{i\omega}{2cn(\omega)}\mathcal{P}(\boldsymbol{r})/\epsilon_0 \to RHS(z,\mathcal{E}(z)) \tag{8.5}$$

Note that above the arguments of RHS may not be the array we have allocated for our complex amplitude, and z may not be equal to the current propagation distance. ODE solver decides for what z and $\mathcal{E}$ should be RHS evaluated, and the user must make no assumptions on them.

Also note that in the example used here there is no explicit dependence of RHS on z, but this will change once we generalize the basic formulation of MOL. Moreover, one can have a problem in which material properties depend on the propagation distance, and then the z-dependence of RHS becomes non-trivial even in the present formulation.

Once the system of equations is specified, the ODE solver takes over the control over its solution. The user normally does not know at which concrete propagation distances will the solution be evaluated, because this may depend on how difficult the integration gets. For example, in the self-focusing collapse simulation we considered in WP13, integration step would be long initially, but it would shorten dramatically on approach to the singularity.

This uncertainty requires that the solver library is given certain marching orders. Libraries tend to have functionality that allows to specify that the solution should arrive at an exactly specified value of z. One can use such a capability to achieve a prescribed, say equidistant, sampling of $\mathcal{E}(z_n)$. However, this approach may disturb the natural evolution of the integration step length, and in effect slows down the whole simulation, especially if frequent samples $\mathcal{E}(z_n)$ are requested. This is simply because to end up exactly at z_n the last executed sub-step is typically shorter than really required. In the following integration, the solver will most likely attempt acceleration and will gradually increase subsequent steps. The net result is a locally denser sampling in z than really needed and concomitant performance degradation. This effect can be significant unless the ODE library can be aware of what the optimal local step is irrespectively of the latest shorter Δz update.

A practical alternative in computationally demanding problems is to accept that sampling of the z axis will be non-equidistant. If one can deal with irregular positions at which the solution will be "observed" it is sufficient to ask the ODE solver to interrupt the integration once the next observation point z_{obs} is reached or when the current z landed beyond z_{obs}. The precise location of this observation then occurs close, but not exactly at the requested position. On the upside, there is no slowing down of the integrator.

Finally, a very important point concerns memory usage. In a finite-difference code, it is straightforward to determine how much memory a problem requires for a given discretization density. This is not so transparent in MOL. A good implementation will naturally have the capability to select from several ODE methods the library offers. Different algorithms require varying number of auxiliary arrays to hold intermediate results. Depending on the accuracy order of the method, the ODE solver when initialized will allocate memory space for several copies of the complex amplitude. As a result the total memory needed will be multiple of what may seem to be expected based on the size of the array to hold one sample of the beam. While this is inconsequential in many ODE applications, here we deal with systems that will contain many hundreds, thousands or even millions of unknowns. Then the additional storage allocated does matter a lot, and it may affect the size of the problem that can be solved.

8.1.2 Integration Step Control

The importance of the integration step control can not be overstated for Method of Lines. In some problems it may be quite straightforward to at least estimate what Δz may be acceptable based on some quantities that occur in the course of the numerical solution. For example, in the self-focusing collapse simulation one can rely on the maximal field intensity I_{max} achieved at a given propagation distance, and decide that the product of $\Delta z I_{max}$ remains bounded by some tolerance parameter. Adjusting Δz this way is not totally disastrous, actually. The main advantage is that such a control is easy to implement and it does not impose much of a computational penalty. However, it is based on the specifics of the given problem, which is something a proper implementation of any simulation engine should avoid at all cost. Second, it is not a very robust, and requires some trial and error to tweak the tolerances. In general we may not have such an inexpensive way to identify portions of the propagation problem that are more difficult to integrate...

This is why most of the time it is best to utilize capabilities offered by the ODE library. However, it is also often useful to know how such an adaptive step control may work. In what follows, we show an example based on the implementation adopted in the GNU Scientific Library (GSL) — as GSL is an open source, the code can be inspected to see details. The method is based on comparing two solutions: one obtained with a coarser Δz and one with a finer step. If the two solutions agree well, the one obtained with the finer Δz is accepted. If they do not agree, the current step is abandoned, and a shorter is attempted anew.

Clearly, this two-solution approach is expensive, because it requires typically 1.5 times more calculations in comparison with the simulation performed at the finer integration step (it is less than a factor of two because intermediate results can be sometimes shared between the two solutions). Naturally, adaptive step integration also requires additional memory space for auxiliaries.

The step adaptation is based on the assumption that we want to control the local error introduced from the given current solution which is considered as "exact." In general, there are tree ingredients, or stages:

- estimate error of the numerical solution
- create a desired error bound
- compare the two a decide on the next step size

In the first stage, the observed error is evaluated as a difference between two candidate solutions, $E = |y_{err}|_i$. This vector may arise as a by-product of a method that embeds two algorithms in one, one being higher-order than the other. This is done by the solver, behind the scenes.

For the second part, a desired error bound D_i is created based on the current solution. GSL uses an expression that accepts four different parameters, and allows to tailor the acceptable error along the length of the solution array:

$$D_i = \epsilon_{abs} + \epsilon_{rel} * (a_y|y_i| + a_{dydt}h|y\prime_i|) \ .$$

Here, ϵ_{abs} and ϵ_{rel} are the main absolute and relative tolerance adjustment parameters. In the refined version, the first term may be given a "shape" by specifying a scaling vector. This can be useful if the step control should be driven by certain subset of variables.

Then the observed error is compared with the bound, component by component, and if the first is larger by more then ten percent than the bound, the new solution is rejected. A new, shorter step is then set to

$$\Delta z_{new} = \Delta z_{old} * 0.9 * (E/D)^{-1/q} \quad , \quad (E/D > 1)$$

where q represents the accuracy order of the ODE method used. Thus, the higher the order of the method, the less adjustment it may require. After shortening the step, a new attempt is initiated starting again from the old solution.

On the other hand if the observed error is unnecessarily small, say by fifty percent, we have an opportunity to increase the integration step and thus accelerate the solution progress. A new, longer step can be chosen as

$$\Delta z_{new} = \Delta z_{old} * 0.9 * (E/D)^{-1/(q+1)} \quad , \quad (E/D < 1)$$

Without restrictions on how how fast Δz can change it is possible to envision a situation in which a too short step is suggested, resulting in a local error so small that the subsequent Δz proposal is too big. That in turn will trigger big error and a request for possibly even shorter step than the first, thus starting a vicious circle and violent oscillations. To damp such behavior, step changes are restricted to $1/5 < \Delta z_{new}/\Delta z_{old} < 5$.

From the above description it should be evident that the step control is subject to parameter choice and algorithm "tuning." There is no guarantee that one particular step-adaptive algorithm will perform in all circumstances. Consequently careful and conservative approach is the way to go when setting the error tolerances.

8.2 Combination with spectral methods

When the general framework described in the previous Section is applied to the paraxial beam propagation equation in the presence of weakly guiding structure or under influence of nonlinear interactions, the main numerical effort goes into resolving effects of diffraction. In other words, it is the Laplacian operator that controls how fast the integration can proceed. A very rough estimate of the underlying numerical effort is indicated by a length scale given by a characteristic transverse wavenumber of the beam solution and its relation to the central wavelength

$$\frac{k_0}{k_\perp^2} \ .$$

Of course, this is related to the paraxial beam propagator properties. It is also clear that no matter what method is the ODE solver using, this length-scale must be properly resolved by at least several field samples. This may severely limit the integration step size.

A possible way around starts with transforming the whole equation into spectral representation in which the Laplacian operator becomes diagonal. Then,

$$\frac{\partial \mathcal{E}}{\partial z} = \frac{i}{2k_0}\Delta_\perp \mathcal{E} + \frac{i\omega}{2cn(\omega)}\mathcal{P}(\boldsymbol{r})/\epsilon_0 \ . \tag{8.6}$$

becomes

$$\frac{\partial \hat{\mathcal{E}}}{\partial z} = \frac{-ik^2}{2k_0}\hat{\mathcal{E}} + \frac{i\omega}{2cn(\omega)}\hat{\mathcal{P}}/\epsilon_0 \ . \tag{8.7}$$

where $\hat{\mathcal{E}}(k,z)$ and $\hat{\mathcal{P}}(k,z)$ stand for spatial Fourier transforms. If not for the polarization term, we knew the solution, it would be the free-space beam described in Part 2. That suggests that the solution could be found in the form of the following ansatz based on free-space propagation:

$$\hat{\mathcal{E}}(k,z) = A(k,z)\exp\left[-iz\frac{k^2}{2k_0}\right] \tag{8.8}$$

Spectral amplitude A is designed to evolve only thanks to the effects of interactions with the medium, that are included in the polarization term. It is therefore natural to expect that this relative small term (and it must be such — we have established this earlier in Section XXX), will only induce slow evolution along the propagation distance, and the acceptable integration step will be accordingly longer.

The evolution equation for the spectral amplitude is obtained as

$$\partial_z A = \exp\left[+iz\frac{k^2}{2k_0}\right]\hat{P}\left(\{A(k,z)\exp\left[-iz\frac{k^2}{2k_0}\right]\}\right) \tag{8.9}$$

which probably deserves a more explicit form in which the Fourier transforms are shown:

$$\partial_z A = \exp\left[+iz\frac{k^2}{2k_0}\right]FT\{P\left(FT^{-1}\{A(k,z)\exp\left[-iz\frac{k^2}{2k_0}\right]\}\right)\} \tag{8.10}$$

This is still a bit abstract; one should keep in mind that P represents an algorithm that calculates the medium response once the local electric field is given. This evaluation is normally done in the real space. That is why we must first transform from the spectral representation of A to the field $\hat{\mathcal{E}}$ and then inverse Fourier transform to $\mathcal{E}$, which finally becomes an input for function $P(.)$ When the latter returns, the result must be Fourier transformed back into spectral space, and multiplied by the inverse of the paraxial propagator.

As a concrete example, consider beam propagation through a Kerr nonlinear medium, where the "medium response function" P is quite simple:

$$\mathcal{P} = 2\epsilon_0 n_0 n_2 |\mathcal{E}|^2 \mathcal{E}(x,z)\ . \tag{8.11}$$

The $RHS(z, A(k))$ function we have to implement and pass to the ODE solver can be evaluated in the following sequence (only one transverse coordinate is shown to keep notation concise):

1. Apply paraxial propagator with step z to A

$$\hat{\mathcal{E}}(k) \to \exp\left[-iz\frac{k^2}{2k_0}\right]A(k)$$

to obtain electric field amplitude in spectral representation.

2. Transform the latter to the real space

$$\mathcal{E}(x) \to FT^{-1}\left\{\hat{\mathcal{E}}(k)\right\}$$

3. Calculate the medium response in the real space as specified by P:

$$\mathcal{P}(x) \to 2\epsilon_0 n_0 n_2 |\mathcal{E}(x)|^2 \mathcal{E}(x)$$

4. Transform back to spectral space

$$\hat{\mathcal{P}}(k) \to FT\{\mathcal{E}(x)\}$$

5. Apply inverse paraxial propagator and return the result as RHS return "value:"

$$RHS \to \exp\left[+iz\frac{k^2}{2k_0}\right]\hat{\mathcal{P}}(k)$$

It should be obvious that the above procedure is general and applies to an arbitrary medium response. The only deviation from the sequence shown occurs in step three where an appropriate implementation of the medium response should be invoked. It is a good practice to write 1. — 5. as a stand-alone procedure which takes a pointer to function P as one of its inputs.

There is one detail in the previous discussion that was omitted. It has to do with the appearance of the propagation variable z in the argument of direct and inverse paraxial propagators. There are situations in which the resulting argument with which the exponential function is called becomes very large. That is something better to avoid in numerics.

However, when an integration step if executed, it does not matter whether it is the very first one, starting from $z = 0$. In other words, we can always reset our frame of reference in z such that the current step starts at $z = 0$. Then all z-values in the above formulas are small and safe to use. This however requires that the relation (8.8) is renewed after each finished ODE integration step. It amounts to re-definition of the point at which $\hat{\mathcal{E}}(k, z)$ and $A(k, z)$ coincide. To shift this point from $z = 0$ to a new position Δz (the latter being the length of a successfully executed ODE integration step), one needs to "propagate" the spectral amplitude by the same amount forward:

6. Re-align the relation between electric field envelope $\mathcal{E}$ and its spectral amplitude A:

$$A_{new}(k) \to A_{current}(k) \exp\left[-i\Delta z \frac{k^2}{2k_0}\right]$$

After this, $\mathcal{E}$ and A coincide at the current propagation distance. This propagator application finishes one full integration step in this generalized MOL method.

The MOL flavor described in this section represents a powerful framework with many applications. For example, it applies equally to the case of radially symmetric BPM driven by the Discrete Hankel Transform instead of FFT. In fact, if we replace the action of the paraxial propagator by and abstract propagator that accurately describes beam propagation in the absence of polarization term P, all else remains in place. Such a generalization is described next.

8.3 Integration factor

Readers have most likely realized that the procedure applied in the previous Section is nothing but application of an integrating factor in the propagation equation. The integration factor happens to be the paraxial propagator of the "free" part of the beam evolution equation. This immediately raises two questions. First, is it possible to improve the above version of MOL by simply replacing the paraxial version of the propagator with the non-paraxial one? Second, what happens if we deal with a structure that is not homogeneous and does not allow direct application of the spectral propagator? The first question will be answered in affirmative in the following section. Here we address a generalization for an "arbitrary" waveguide structure.

Let us assume that without the polarization term P, we can approximate the propagating beam with sufficient accuracy. More concretely, let us cast the propagation equation to an abstract form

$$\partial_z \mathcal{E} = i\hat{L}\mathcal{E} + i\mathcal{P}(\mathcal{E}) \ . \tag{8.12}$$

and assume that the action of the "free" propagator on an arbitrary beam represented by a vector ψ

$$\exp\left[i\Delta z L\right]\psi$$

can be calculated. It can be achieved for example through application of the modified Crank-Nicolson method. Note that there is a big difference between evaluating this kind of an operator per se, and its application to a concrete vector the second being an incomparably easier task.

The free propagator plays the role of the integration factor, when the solution is written with the help of an auxiliary in what is a generalization of (8.8):

$$\mathcal{E} = \exp\left[izL\right]A \ . \tag{8.13}$$

The evolution equation for the auxiliary reads

$$\partial_z A = i \exp\left[-izL\right]\mathcal{P}(\exp\left[izL\right]A) \ . \tag{8.14}$$

This is a system of equations that we ask the ODE solver to resolve. Application of the procedure described in the previous Section thus requires definition of the medium response P in the form of call-able function, and an algorithm that approximates the free-propagator action.

This represents a general framework to separate the effects due to the polarization term, and due to free propagation through the waveguide or a bulk medium.

8.4 Spectral representation and exact propagation equations

Now we return to the special case of a homogeneous medium. It seems that the free propagator employed in the previous Section could be the exact non-paraxial beam propagator discussed in the context of spectral methods. Surely it must be better to to replace its paraxial approximation by something that solves the free propagation problem exactly? But then, what is the propagation equation such a scheme should represent and how it can be derived?

Clearly, the evolution equation we need is of the form

$$\partial_z E = i\hat{L}E + \frac{i\omega}{2c\epsilon_0 n(\omega)}P(E) \tag{8.15}$$

It does not matter that there is no simple differential operator that corresponds to the exact non-paraxial propagator through a homogeneous medium. All that is needed, is that we can calculate its action on an arbitrary beam essentially exactly. This is in fact what the FFT-BPM and DHT-BPM do. One can write the above equation formally as

$$\partial_z E = i\sqrt{\frac{\omega^2 n^2(\omega)}{c^2} + \partial_{xx} + \partial_{yy}}\ E + \frac{i\omega}{2c\epsilon_0 n(\omega)}P(E) \tag{8.16}$$

where the square root is a pseudo-differential operator whose action is properly defined in the spectral representation where spatial derivatives correspond to multiplication by transverse wavenumbers.

How can we justify this evolution equation? The answer to this question is related to two important aspects of the paraxial beam propagation considered so far. First, it is the directional nature of the propagation when only wavepackets propagating in the "forward" directions are included in the model. Second, it is the need to keep the polarization term small. Let us first eliminate both of these assumptions and see what the exact evolution system should look like in the first place.

In order to include what we consider forward and backward propagating waves, the total electric field can be written with the help of *two* auxiliaries, each corresponding to one direction:

$$E = e^{+iLz}A + e^{-iLz}B \tag{8.17}$$

Here we have doubled the number of variables that describe the electric field. Taking advantage of this, we have the freedom to impose an additional constraint which we choose to ensure that

$$\partial_z E = iL(e^{+iLz}A - E^{-iLz}B) \ , \tag{8.18}$$

which in turn requires that the following holds:

$$e^{+iLz}\partial_z A + e^{-iLz}\partial_z B \equiv 0 \ . \tag{8.19}$$

Now consider the following evolution system

$$\begin{aligned} \partial_z A &= +\frac{i\omega^2}{2c^2\epsilon_0}\frac{1}{L}e^{-iLz}P(E = e^{+iLz}A + E^{-iLz}B) \\ \partial_z B &= -\frac{i\omega^2}{2c^2\epsilon_0}\frac{1}{L}e^{+iLz}P(E = e^{+iLz}A + E^{-iLz}B) \end{aligned} \tag{8.20}$$

Note that when B is neglected, the first equation reduces to the spectral amplitude evolution system we have seen before. Also, this system clearly satisfies the constraint (16.15). These are indeed the thought after exact evolution equations coupling the forward and backward propagating waves. What we must show is that if they are satisfied, then the *total field* E obeys the Helmholtz equation.

The (scalar-approximation) Helmholtz equation for the electric field and the polarization response of the medium was derived in Section XXX:

$$\left[\partial_{zz} + \partial_{yy} + \partial_{xx} + \frac{\omega^2 n^2(\omega)}{c^2}\right] E + \frac{\omega^2}{\epsilon_0 c^2}P = 0 \tag{8.21}$$

Start with the ∂_{zz} term. First,

$$\partial_z E = iL(e^{+iLz}A - e^{-iLz}B) \tag{8.22}$$

thanks to the constraint imposed on forward-backward amplitudes A, B. Then the second derivative evaluates to

$$\partial_{zz}E = (iL)^2(e^{+iLz}A + e^{-iLz}B) + iL(e^{+iLz}\partial_z A - e^{-iLz}\partial_z B) \tag{8.23}$$

which gives

$$\partial_{zz}E = -L^2 E - \frac{\omega^2}{c^2\epsilon_0}P \ . \tag{8.24}$$

Inserting in the Helmholtz, we see that it is satisfied as long as

$$L^2 = \frac{\omega^2 n^2(\omega)}{c^2} + \partial_{yy} + \partial_{xx} \ . \tag{8.25}$$

This is exactly the dispersion relation of the exact non-paraxial propagator written in the real space. In order to verify this, act with the above operator on an arbitrary plane wave — it turns out to be an eigenfunction of this operator with an eigenvalue that is square of the propagation constant of this plane wave. That means that e^{iLz} is indeed the correct free-space linear propagator.

We have thus demonstrated that solutions of the coupled evolution system (8.20) provide exact solutions of the Helmholtz equation. This is therefore an exact replacement of Helmholtz with a first-order propagation system.

One approximation that has been tacitly assumed above is that the field is scalar and that $\nabla.\boldsymbol{E}$ can be neglected. These deficiencies are easy to correct, and we will do this later in the course when we address situations with sharp material interfaces that will cause mixing of various polarization components of the electric field.

For now, the coupled system is exact for scalar fields. Let us see what approximations will take us back to the non-paraxial propagation equation we guessed in the beginning of this section. Clearly, we need to drop the backward propagating field component B. This is what is called uni-directional approximation. It requires that if the beam is initialized with the forward component alone, the interaction with the medium is such that the backward propagating field will not be generated, and that

$$P(E = e^{+iLz}A + E^{-iLz}B) \approx P(E = e^{+iLz}A) \ . \tag{8.26}$$

Under uni-directional approximation we have a single equation for the forward amplitude, namely

$$\partial_z A = +\frac{i\omega^2}{2c^2\epsilon_0}\frac{1}{L}e^{-iLz}P(E = e^{+iLz}A) \ . \tag{8.27}$$

The nontrivial problem here is the inverse of L. This can be easily evaluated in the spectral representation, but let us assume we insist on the real-space language; we have to assume one more approximation, and replace

$$\frac{1}{L}P \equiv \frac{1}{\sqrt{\frac{\omega^2 n^2(\omega)}{c^2} + \partial_{xx} + \partial_{yy}}}P \approx \frac{c}{\omega n(\omega)}P \tag{8.28}$$

In other words it is required that the polarization term is such that it is paraxial, and its spatial spectrum is narrow. If it is the case A obeys

$$\partial_z A = +\frac{i\omega}{2cn(\omega)\epsilon_0}e^{-iLz}P(E = e^{+iLz}A) \ , \tag{8.29}$$

and the electric field satisfies the equation we started with

$$\partial_z E = i\hat{L}E + \frac{i\omega}{2c\epsilon_0 n(\omega)}P(E) \ . \tag{8.30}$$

Unfortunately, now it is evident that improving the paraxial approximation in the first term and handling LE exactly in the spectral space is most likely inconsistent with the assumption that the second term is paraxial. Indeed, if the field itself was non-paraxial, containing waves that propagate at steep angles, it is hard to believe that the polarization term would still have narrow spatial spectrum. The only excuse here would be that this term is so small in the first place that its non-paraxial corrections are negligible. However, since the exact propagator must be dealt with in the spectral space, it would be only a minor complication to retain the equation which requires solely the uni-directional approximation, namely

$$\partial_z E = iLE + \frac{i\omega^2}{2c^2\epsilon_0}\frac{1}{L}P(E) \ . \tag{8.31}$$

This equation admits non-paraxial fields, but requires that the polarization term does not generate backward propagating waves. This is not easy to verify without actually solving a given problem, and we have to rely more on our intuition and experience.

To conclude this section, we have seen that to tackle the beam propagation problem exactly, operators such as L, its exponential, and its inverse must be handled. This is one of the main issues in what is the so-called Wide-Angle Beam Propagation Method (WA-BPM).

8.5 Practice track: Implementations of MOL

Summary:

- Practice implementation of MOL in its simplest version.
- Appreciate the “black-box” nature of this approach by comparing nominally equal implementations using different ODE-solver libraries.

This work-package directory contains two sub-folders, each with a simple implementation of a BPM method employing the Method of Lines approach. One is written in C++ (*wp15-C-MOL-Radial-Arago*), and the other is in Matlab (*wp15-M-MOL-Radial-Arago*). The two templates set up a simulation of the Arago bright spot with equivalent numerical parameters to enable comparative simulations.

The Method of Lines implementations invariably utilize canned libraries for the underlying Ordinary Differential Equation (ODE) solvers. Having two different implementations, even if they are written to be in one-to-one correspondence, gives us the opportunity to appreciate how much the MOL depends on the ODE library used and how much it behaves as a black-box object that requires careful, so to speak experimental approach. We will see shortly that what might be two nominally equivalent implementations can potentially produce different results. So, one important take-away message will be that in the MOL, it is up to the user to watch for potential problems, and tune the ODE-solver controlling parameters accordingly.

8.5.1 Simple test of a simple MOL code

The first task of this Practice-track session is to code up a simple MOL-based BPM, and test its function in our standard test, namely by comparing the results of numerical and analytic Gaussian beam propagation.

This example highlights the conceptual simplicity of MOL, and also its flexibility. For example, the simple program created here for the radially symmetric situation would be very easy to modify for two transverse dimensions. The same task could be much more demanding with other approaches, for example for the Crank-Nicolson method. The following is a listing for a function that “propagates” a given beam amplitude over a given distance; as one can see a few lines suffice to produce a functioning MOL code:

Listing 8.1. Method of lines BPM propagator

```
function [zsp enews] =  Propagate(Ain,nr,dr,k0,zfin)
% Radially symmetric beam propagator
% Uses Method of Lines
% Implements PEC boundary
% Ain = input array (beam profile)
% nr  = number of points in the radial grid
% dr  = radial grid spacing
% k0  = propagation wavenumber
% zfin= final propagation distance to reach

idelta = 1i/(2*k0*dr^2);

options = odeset('AbsTol', 1.0e-03, 'RelTol',0.0e-03);
```

```
14
15 [zsp enews] = ode23(@rhs2,[0, zfin/2, zfin],Ain);
16
17   function df = rhs2(z,y)
18
19     laux = circshift(y,+1);
20     raux = circshift(y,-1);
21     indx = linspace(1,nr,nr).' - 1;%'
22
23     df     = idelta*(laux -2*y + raux + (raux-laux)./(2.0*indx));
24     df(1)  = 4*idelta*(y(2)-y(1));
25     df(nr) = 0;
26 end
27
28 end
```

In line 13, we set the options that control how the ODE solver of Matlab integrates our evolution equations. In this case, it is essentially only the value of the absolute-error tolerance that we use.

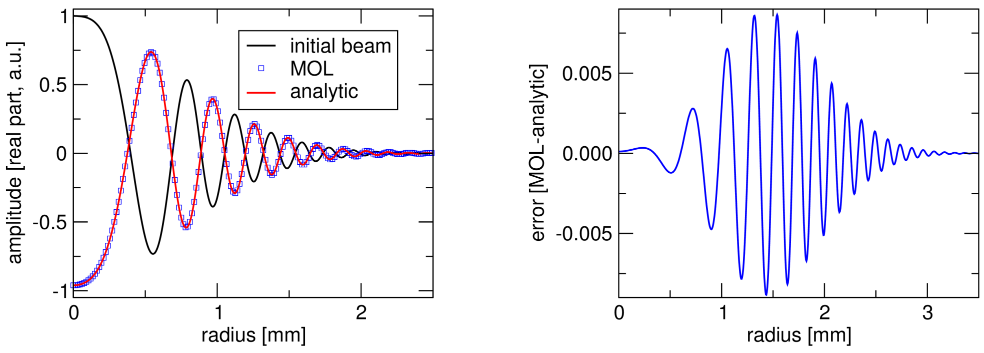
In line 15, the ODE solver is invoked and returns its results. Here we specify (in the order from the last to the first argument) the initial condition, the interval over which we ask to find a solution, with how many intermediate points this interval is to be sampled, and finally we supply the name of the function that defines the BPM problem.

The latter is where the given simulation problem, and in particular the discretized evolution equation is specified. The few lines of the function evaluate the right-hand-side of the differential equation system. Here it is realized in the vector form characteristic of Matlab.

The implementation in C++ is somewhat more complex, but this has to do more with the nature of the language than with the problem being solved. The reader should inspect and compare both codes in order to appreciate both their common structure, and the differences of invoking the ODE solver.

To verify that the BPM actually works, we start with an initial condition representing a focused beam, propagate it through its focal region, about twice the focal distance. Qualitatively, what one expects to obtain as the output is the same shape of the beam amplitude but with opposite sign. The latter is the consequence of the Gouy phase accumulated during the propagation. The following figure shows the result, compared to the analytic formula target, and showing the resulting error.

Now, with the program passing the standard test, it is left to the reader to experiment with the ODE solver tolerance settings to see how much numerical effort does it take to compress the error below some target. One should also note that as a function of the tolerance the integration step will decrease and the computational time increases correspondingly. It is also interesting to compare result and program running times for different ODE methods orders. One should see readily that a higher order method may not always achieve the pre-set target accuracy of the final-z solution faster than a method using a lower-order scheme that requires less function evaluations per one step. Thus, the choice of the most efficient method to solve the ODE system within MOL is most often a matter of experimentation.



Gaussian beam propagation test. Initially focuses Gaussian beam (black, left) is propagated by the MOL technique through its focus (symbols) and is compared with the analytic target solution (red). The right panel shows the error of the numerical solution.

8.5.2 Using different ODE solvers in MOL programs

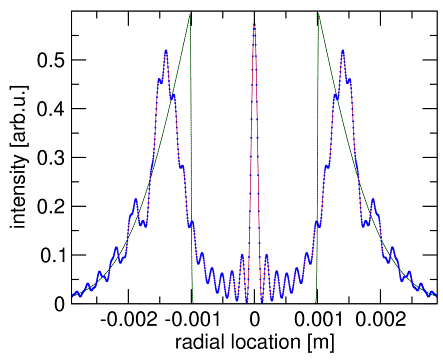
For a more stringent test, we turn to the simulation of the bright spot of Arago — we have already established in the previous exercises that this particular problem is ideal in the sense that it stresses all aspects of a BPM method.

Task 1: Inspect both codes. In particular, compare how both codes access ODE-solver library routines. In the ODE library used in the C++ code, the system to be solved is assumed to be real. This is however not a practical limitation, as one can cast the type of the array depending on its use; in the RHS function it is more convenient to treat it as an array of complex amplitude values, while it may be passed to the ODE solver as an array of reals. However, one has to keep in mind that the number of equations to be solved is twice the number of grid points.

Another point worthwhile to note is the control of where or at what propagation distances the solver produces its "outputs," and this may concern the Matlab users. Because the MOL systems in the context of BPM tend to be large, the Matlab user must pay attention to how the ODE routine is invoked, and avoid too many solution snapshots be included in the output. In fact the way default way Matlab ODE solver passes the solution is not optimal for very large ODE systems. In BPM, one tends to process a single solution snapshot at a time, before moving to the next integration step.

Task 2: Execute simulation as set up in the templates, compare run times (these codes measure the net time spent in the main beam-evolution loop for fair comparison), and resulting simulated intensity profiles. Nice agreement between the two implementation should be evident.

With MOL implementations utilizing libraries of ODE solvers, one can easily switch from one ODE-method to another. This is done in the *wp15-C-MOL-...* template, where the executable accepts the name of the method as its first argument (inspect the solver header to see what are the names of acceptable methods). This figure compares results obtained with rk2 and rkf45 solvers, and demonstrates the close agreement one should expect:

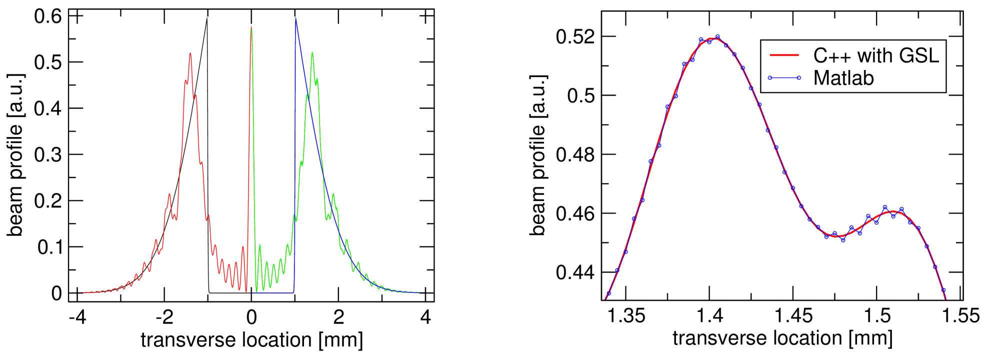


Comparisons of MOL-BPM results for rk2 and rkf45 ODE-integrators. Full lines represent the rkf45 method, while the dotted lines and symbols are for the rk2 integrator. Green line shows the initial condition - a Gaussian beam which encountered a circular obstacle. The intensity peak in the center is a "degraded" bright spot of Arago. The lower intensity of the central spot is a consequence of illumination with a finite-diameter beam.

It is interesting to compare the running times required to execute the above simulations. With the tolerance set to the same value(s), and with the to methods solving the same problem to a desired level of accuracy, the wall-clock time is the ultimate measure for the method comparison.

The higher order rkf45 method required 1366 steps to propagate the beam solution over the distance of 0.5 meter. The lower order rk2 method needed 2766 steps to achieve the same,i.e. almost twice as many. However, the running times for the two methods were 493 ms and 620 ms, respectively, which is not a big improvement for the rkf45 solver. So, the higher-order method does what is promises, namely it calculates the solution in a smaller number of integration steps. However, if we counted the number of evaluations of the RHS of the ODE system solved, the numbers would be much more comparable, and this explains why the actual gain in using the higher order method is not as big as one might expect. Note that this is not an uncommon behavior, it is often more efficient to utilize a simpler method with an inexpensive update scheme than a sophisticated one. This also illustrates that with MOL, one should experiment and find by trial which of the ODE solvers made available by the used library performs best for the given problem.

Task 3: Now we look a similar comparison, but with the Matlab MOL implementation. Namely, we invoke rk2 and rkf45 ODE solvers within our MOL code to see if their relative performance is similar as we have seen in the C++ code. For this purpose, we modify the Matlab *Propagation.m* as to call either ode23 or the ode45 solver. Re-running the simulations, we note the timing, and will inspect the result again. The following figure illustrates the surprising outcome. Paying attention to the small-scale details of the new solution, one can easily see that the ode45 solver produced a significantly worse result. Moreover, while it took 5.2 seconds to execute with the ode23 solver, ode45 needed more than 11 seconds! Not only that the higher-order methods runs significantly slower, but it also gives results that are worse than those from the lower-order ode23 method. Almost perfect agreement with the c++ based solution is also lost; there are high-frequency spatial oscillations evident in the new solution which are obviously unphysical.



Comparison of MOL performance utilizing ode23 and ode45 solvers in a Matlab implementation of beam propagator, applied to the problem of Poisson's bright spot. The overall agreement is illustrated on the left. However, zooming into the vicinity of the most prominent peak in the solution, one uncovers that one of the solution suffers from pronounced numerical artifact. Surprising, it is the higher-order solver ode45 that runs into problems!

Take home lesson: Matlab and c++ implementation may use the very similar methods with the same embedded accuracy orders of five and four, but the results are disturbingly different both in execution time and smoothness of the solution. This illustrates the fact that MOL often requires some experimentation as to what ODE method and what accuracy-control parameters work best for the problem at hand. This also demonstrates that when MOL runs into a problem, it may be very difficult to diagnose.

That being said, MOL represents an extremely useful numerical tool. While not too commonly used as a BPM method per se, it can serve as a very flexible component of many methods, optical pulse propagation simulation being one such example.

Yet another worthwhile issue to note here is that the Matlab MOL solutions ran an order of magnitude slower to compute the nominally equivalent problem. The lesson the reader should take away is that while general-purpose compute environments like Mathematica or Matlab may do an excellent job in any specific task, if a new simulation engine must push an envelope of the state of the art, then it will be most likely written in an "old-fashioned" way, using a proven compiled language.