

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. 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.

0.0.1 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. A few lines suffice to produce a functioning MOL code:

Listing 1: Method of lines BPM propagator

```

1 function [zsp enews] = Propagate(Ain,nr,dr,k0,zfin)
2 % Radially symmetric beam propagator
3 % Uses Method of Lines
4 % Implements PEC boundary
5 % Ain = input array (beam profile)
6 % nr = number of points in the radial grid
7 % dr = radial grid spacing
8 % k0 = propagation wavenumber
9 % zfin= final propagation distance to reach
10
11 idelta = 1i/(2*k0*dr^2);
12
```

```

13 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.

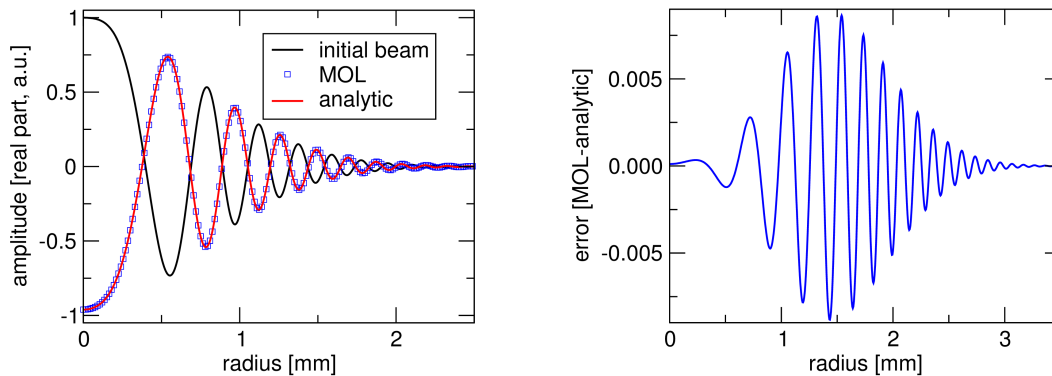
In line 15, the ODE solver is invoked and returns its results. Here we specify (in the order from the last to the first argument) the initial condition, the interval over which we ask to find a solution, with how many intermediate points this interval is to be sampled, and finally we supply the name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

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.

0.0.2 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.

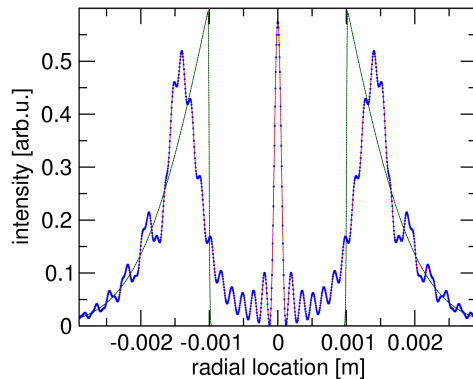
Task 1: Inspect both codes. In particular, compare how both codes access ODE-solver library routines. In the ODE library used in the C++ code, the system to be solved is assumed to be real. This is however not a practical limitation, as one can cast the type of the array depending on its use; in the 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:

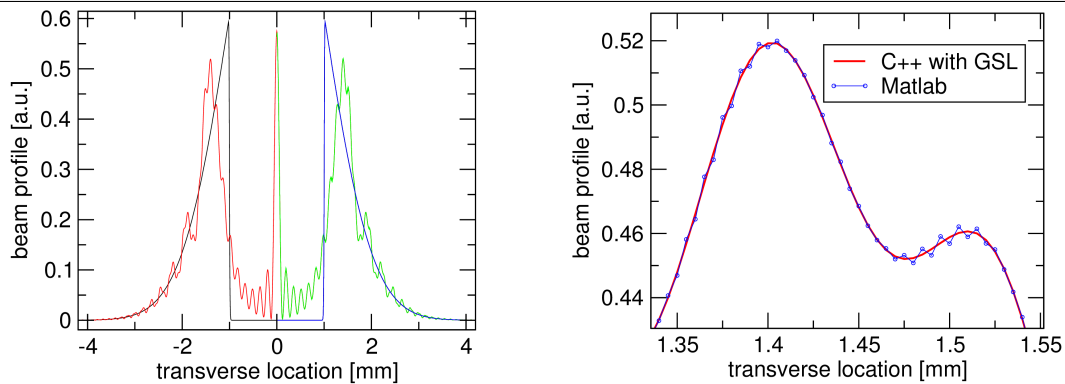


Comparisons of MOL-BPM results for rk2 and rkf45 ODE-integrators. Full lines represent the rkf45 method, while the dotted lines and symbols are for the rk2 integrator. Green line shows the initial condition - a Gaussian beam which encountered a circular obstacle. The intensity peak in the center is a “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 two 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 it 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: Next we look at a similar comparison, but with a Matlab-based MOL implementation. 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 method 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 numerical artifacts. Surprisingly, it is the higher-order solver ode45 that runs into problems!

Take home lesson: Matlab and c++ implementation may use 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.