NUMERICAL SOLUTION OF THE EXTENDED NONLINEAR SCHRÖDINGER EQUATION

John V.S. Harvey
Richard L. Medina Jr.

Air Force Research Laboratory

SEPTEMBER 2006
August 2005 to September 2006

Approved for public release, distribution unlimited.

Air Force Research Laboratory
Human Effectiveness Directorate
Information Operations and Special Programs Division
Brooks-City-Base, TX 78235
NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 311th Human Systems Wing (311 HSW/PA) Public Affairs Office (PAO) and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-HE-BR-TR-2007-0005 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

THIS TECHNICAL REPORT IS APPROVED FOR PUBLICATION.
Numerical Solution of the Extended Nonlinear Schrödinger Equation

High-resolution mathematical models of the extended nonlinear Schrödinger equation have been designed which include diffraction combined with non-zero second-order group-velocity dispersion (GVD). These models follow a Gaussian pulse as it propagates in air to a large distance (several meters). With diffraction disabled, a pulse quickly collapses to a single singularity on the propagation axis. Alternatively, with diffraction included, a pulse will collapse into a pair of “fins” off the propagation axis. If the GVD is disabled, the fins eventually collapse to singularities. However, if the GVD is set an appropriate non-zero value, the fins can be propagated out to several meters (propagation distance) without singularities forming. In test cases with diffraction plus GVD, we see (A) an initial drop in intensity, followed by (B) a rise at about 2 to 3 meters, and then (C) a gradual drop thereafter. This pattern is most pronounced in our “energy pattern” depictions where we model the distribution of the total energy seen by a target plane as the pulse quickly passes through it. When viewed on a target plane at an optimal distance (roughly 2.5 meters), the energy pattern appears as a bright ring – indicating that the initial Gaussian pulse has collapsed to a very thin cylindrical shape. Our results are based solely on mathematical formulations without any experimental verification. Additionally, these formulations do not attempt to completely ensure energy conservation.
# Contents

1 INTRODUCTION ........................................................................................................................................................................... 1

2 SOFTWARE EVOLUTION DURING THE TASK ........................................................................................................................................ 2
   2.1 Original FORTRAN implementations ................................................................................................................................. 2
   2.2 Early MATLAB implementations ........................................................................................................................................ 5
   2.3 Attempts at Parallelization ..................................................................................................................................................... 6
      2.3.1 Success with our Fast Model (without diffraction) ..................................................................................................... 6
      2.3.2 Problems with our Slower Model (with diffraction) ........................................................................................................... 7
   2.4 Latest MATLAB implementation ........................................................................................................................................ 8

3 LATEST FORMULA GUIDE ............................................................................................................................................................... 9
   3.1 Initial Gaussian Pulse ............................................................................................................................................................. 10
   3.2 Power and Energy ............................................................................................................................................................... 11
   3.3 The Fast Model (without diffraction) .................................................................................................................................. 13
      3.3.1 Spectral Derivation ....................................................................................................................................................... 15
   3.4 The Slower Model (with diffraction) .................................................................................................................................. 19
      3.4.1 Spectral Derivation in Two Dimensions .......................................................................................................................... 21
      3.4.2 Alternate weighted-frequency formulations ....................................................................................................................... 24

4 PARAMETER SETTINGS .................................................................................................................................................................. 27
   4.1 Required Input Parameters .................................................................................................................................................. 27
   4.2 Values Used .......................................................................................................................................................................... 28

5 LATEST RESULTS ......................................................................................................................................................................... 30
   5.1 Description of Output ............................................................................................................................................................ 30
   5.2 Tests of the Fourier Transformations .................................................................................................................................. 31
      5.2.1 Case A: tests with no weighting .................................................................................................................................. 32
      5.2.2 Case B: tests with circular frequency weighting ......................................................................................................... 34
      5.2.3 Case C: tests with quartic frequency weighting .................................................................................................................. 35
      5.2.4 Case D: tests with quintic frequency weighting ........................................................................................................... 36
   5.3 Propagation Tests without Diffraction ................................................................................................................................ 37
      5.3.1 Nonlinear terms only ....................................................................................................................................................... 37
      5.3.2 Nonlinear terms plus second-order GVD .......................................................................................................................... 39
   5.4 Propagation Tests with Diffraction but No Linear Operator .................................................................................................... 41
      5.4.1 Nonlinear only with diffraction and no weighting ....................................................................................................... 41
      5.4.2 Nonlinear only with diffraction and quintic frequency weighting ................................................................................... 42
   5.5 Propagation with Diffraction and Non-Zero Second-Order GVD .......................................................................................... 43
      5.5.1 Singularity Problems with Unweighted Diffraction ....................................................................................................... 43
      5.5.2 Best Results with Quartic Weighting ............................................................................................................................... 45
      5.5.3 Best Results with Quintic Weighting ............................................................................................................................... 48
      5.5.4 Tests with Higher GVD .................................................................................................................................................... 52
   5.6 The Effect of the Chirp Parameter ......................................................................................................................................... 54
      5.6.1 Example with Positive Chirp ........................................................................................................................................ 54
      5.6.2 Example with Negative Chirp ........................................................................................................................................ 58
      5.6.3 Example with Extreme Negative Chirp .......................................................................................................................... 62

CONCLUSIONS ............................................................................................................................................................................... 65
<table>
<thead>
<tr>
<th>Figures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1: pulse evolution model</td>
</tr>
<tr>
<td>Figure 2: the iterative split step approach</td>
</tr>
<tr>
<td>Figure 3: the simple split step approach</td>
</tr>
<tr>
<td>Figure 4: test result for Numerical Recipes FFT</td>
</tr>
<tr>
<td>Figure 5: test result for IMSL FFT</td>
</tr>
<tr>
<td>Figure 6: MATLAB Graphical User Interface</td>
</tr>
<tr>
<td>Figure 7: Distributed Processing without Diffraction</td>
</tr>
<tr>
<td>Figure 8: Timings without Diffraction</td>
</tr>
<tr>
<td>Figure 9: Distributed Processing with Diffraction</td>
</tr>
<tr>
<td>Figure 10: Timings with Diffraction</td>
</tr>
<tr>
<td>Figure 11: Gaussian Pulse</td>
</tr>
<tr>
<td>Figure 12: cylindrical integration</td>
</tr>
<tr>
<td>Figure 13: Default Settings</td>
</tr>
<tr>
<td>Figure 14: Pulse before and after 20,000 transform steps</td>
</tr>
<tr>
<td>Figure 15: maxima silhouette evolution during 20,000 transform steps</td>
</tr>
<tr>
<td>Figure 16: energy pattern evolution during 20,000 transform steps</td>
</tr>
<tr>
<td>Figure 17: maxima silhouette evolution during 20,000 transform steps</td>
</tr>
<tr>
<td>Figure 18: energy pattern evolution during 20,000 transform steps</td>
</tr>
<tr>
<td>Figure 19: maxima silhouette evolution during 20,000 transform steps</td>
</tr>
<tr>
<td>Figure 20: energy pattern evolution during 20,000 transform steps</td>
</tr>
<tr>
<td>Figure 21: maxima silhouette evolution during 20,000 transform steps</td>
</tr>
<tr>
<td>Figure 22: energy pattern evolution during 20,000 transform steps</td>
</tr>
<tr>
<td>Figure 23: pulse at 2 and 2.5 meters</td>
</tr>
<tr>
<td>Figure 24: maxima silhouette up to the singularity</td>
</tr>
<tr>
<td>Figure 25: energy pattern evolution up to the singularity</td>
</tr>
<tr>
<td>Figure 26: pulse shapes at 0.0, 1.5, 3.0, and 4.5 meters</td>
</tr>
<tr>
<td>Figure 27: maxima silhouette evolution</td>
</tr>
<tr>
<td>Figure 28: energy pattern evolution</td>
</tr>
<tr>
<td>Figure 29: pulse shapes at 0.0, 0.5, 1.0, and 1.5 meters</td>
</tr>
<tr>
<td>Figure 30: pulse shapes at 0.5, 1.0, 1.5, and 1.7 meters</td>
</tr>
<tr>
<td>Figure 31: pulse shapes at 0.5, 1.0, 1.5, and 2.0 meters</td>
</tr>
<tr>
<td>Figure 32: pulse shape at 2.5 meters</td>
</tr>
<tr>
<td>Figure 33: pulse corruption around 3 meters</td>
</tr>
<tr>
<td>Figure 34: pulse shapes at 2 and 2.5 meters</td>
</tr>
<tr>
<td>Figure 35: pulse shapes at 3 and 3.5 meters</td>
</tr>
<tr>
<td>Figure 36: maxima silhouette evolution</td>
</tr>
<tr>
<td>Figure 37: energy pattern evolution</td>
</tr>
<tr>
<td>Figure 38: energy pattern as seen by target plane at 2.5 meters</td>
</tr>
<tr>
<td>Figure 39: pulse shapes at 2 and 2.5 meters</td>
</tr>
<tr>
<td>Figure 40: pulse shapes at 3 and 3.5 meters</td>
</tr>
<tr>
<td>Figure 41: maxima silhouette evolution</td>
</tr>
<tr>
<td>Figure 42: energy pattern evolution</td>
</tr>
<tr>
<td>Figure 43: energy pattern as seen by target plane at 2.7 meters</td>
</tr>
<tr>
<td>Figure 44: comparison of composite pulse evolution profiles</td>
</tr>
<tr>
<td>Figure 45: maxima silhouette evolution</td>
</tr>
<tr>
<td>Figure</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>46</td>
</tr>
<tr>
<td>47</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>49</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>51</td>
</tr>
<tr>
<td>52</td>
</tr>
<tr>
<td>53</td>
</tr>
<tr>
<td>54</td>
</tr>
<tr>
<td>55</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>57</td>
</tr>
<tr>
<td>58</td>
</tr>
<tr>
<td>59</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>61</td>
</tr>
<tr>
<td>62</td>
</tr>
<tr>
<td>63</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>65</td>
</tr>
<tr>
<td>66</td>
</tr>
<tr>
<td>67</td>
</tr>
<tr>
<td>68</td>
</tr>
<tr>
<td>69</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>71</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>73</td>
</tr>
<tr>
<td>74</td>
</tr>
<tr>
<td>75</td>
</tr>
<tr>
<td>76</td>
</tr>
<tr>
<td>77</td>
</tr>
<tr>
<td>78</td>
</tr>
</tbody>
</table>
PREFACE

Titan Corporation
Systems Integration Sector
6100 Bandera Road, STE 808
San Antonio, TX 78238

Contract Number – F41624-03-D-6002
Human Systems Center Technical and Management Support II (HSC TMS II).
Task Order 0006, Subtask 01
Bruce D. Bray, Program Manager, (210) 521-1363 x5170, bruce.bray@L-3Com.com.
ACKNOWLEDGEMENTS

The authors are grateful to Dr. William P. Roach, Dr. Robert J. Thomas, Dr. Paul K. Kennedy, Dr. Benjamin A. Rockwell and Dr. Richard A. Albanese for their advice -- especially to Dr. Roach and Dr. Thomas for the many important discussions and helpful suggestions. This work is supported by the Air Force Office Research Laboratory Director Funds.
SUMMARY

The goal of this task was to investigate the theoretical/mathematical feasibility of using femtosecond (pulsed) laser-generated filaments as wave guides for radio-frequency or high-power microwave transmission. More specifically, we were interested in the possible use of laser-induced filaments as waveguides in the atmosphere.

We began by first developing a high-resolution model for electromagnetic wave propagation. This model was initially derived from pre-existing nonlinear Schrödinger (NLS) formulations and software left by Mary Potasek, Sukkeum Kim, and Andrew Paul. Our reference equations extend the "split-step" formula of G. P. Agrawal with a diffraction term used by Potasek as well as Paul. The software was initially a variation of FORTRAN software designed by Andrew Paul. We improved that software by increasing its resolution as well as fixing various scaling errors that we detected. We also supplemented it with MATLAB plotting. We later replaced the FORTRAN software with MATLAB equivalents. Due to a recurring problem with numerical instabilities, this MATLAB implementation was ultimately replaced by our latest MATLAB software which implements a new, faster algorithm.

With settings from actual experimental firings, we have performed a series of tests whereby various terms in our expansion are either enabled or disabled. Most notably, we found that with diffraction disabled, our test pulse quickly collapses to a single singularity on the z axis due to the non-linear operator exploding at a single time-radius point, while all other values go to near zero. In contrast, with diffraction, the pulse tends to collapse more gradually to a pair of "fins" off the z axis before eventually collapsing to a pair of singularities if the propagation is continued too far.

Through extensive testing of the software, we determined that these singularity conditions were primarily a result of the fact that group-velocity dispersion (GVD) was not included. So, in the final months of the task, we combined non-zero GVD values with diffraction and successfully modeled propagation of a Gaussian pulse out to several meters (propagation distance) without singularities forming.

To see the cumulative evolution of the pulse over the analysis distance, the software provides composite surface views. In these views, we see (a) an initial drop, followed by (b) a rise in magnitude at about 2 to 3 meters, and then (c) a gradual drop at larger distances. This pattern is most pronounced in our “energy pattern” depictions where we model the distribution of the total energy seen by a target plane as the pulse quickly passes through it. When viewed on a target plane at a given distance, the energy pattern appears as a bright ring – such that an initial Gaussian pulse has collapsed to a very thin cylindrical shape.

The results in this report are numerical approximations only and actual experiments are needed to validate our predictions.


1 INTRODUCTION

We model the evolution of a radially symmetric pulse – defined as a function of radius \( r \), and pulse duration \( t \) – as it propagates down a \( z \) axis.

Our formulations and software were originally derived from a repository of old computer files – software, slide shows, and technical documents – left by Mary Potasek, Sukkeum Kim, and Andrew Paul. This includes a technical report related to the work of Kim and Potasek [3]. Within this repository was software developed by Andrew Paul which models the propagation of laser pulses according to an approach found in the book by G. P. Agrawal [1]. Although not directly solving the pertinent problem, this implementation served as a good starting point for the development of our approach.

We also assembled published papers relating to the mathematical problem that we hoped to solve. Most of these papers do not directly address the problem that we are solving; but, many contain fundamental formulas and parameter settings that proved useful in our solution. Many of our parameter settings are derived from Sprangle [8].

We used all this material in order to create a reference formula guide. This guide served as the sole document from which we created our software. Within the guide, we extend the “split-step” formula of Agrawal with a diffraction term used by Potasek as well as Paul. Throughout all our testing, we found that if this diffraction term was disabled, the software was generally an order of magnitude faster than with diffraction. Therefore, for all versions, there was always a “fast model” (without diffraction) as well as a “slower model” (with diffraction) in the software.
2 SOFTWARE EVOLUTION DURING THE TASK

Our software evolved in 3 major phases: (A) upgrade of pre-existing FORTRAN code supplemented with MATLAB plotting, (B) complete conversion to MATLAB (with options to use the MATLAB Distributed Computing Toolbox for parallel processing), and (C) the latest MATLAB software which implements a new, faster algorithm. This development occurred from December 2005 through September 2006.

2.1 Original FORTRAN implementations

Our software was initially derived from the FORTRAN “crsplit4” program created by Andrew Paul based on previous software from Mary Potasek. This software uses an iterative (2-part) symmetric split-step approach with sparse coverage. It used FFT transforms in the time direction only – with the Numerical Recipes FFT. It also used the Numerical Recipes tri-diagonal solver.

Initial improvements to the software consisted of upgrades in source readability, execution speed, and graphical displays – including the capability to automatically generate MATLAB display software with the analysis FORTRAN program at each execution. Additionally, parameters were modified to better model propagation through air; Kim and Potasek had modeled the propagation of pulses in reverse saturable absorbers. Subsequently, various adjustments to the mathematics and associated software implementation allowed us to improve the quality of the answer.

First, array sizes were increased and z-axis step sizes were decreased to achieve better answers. This higher density insures that our piece-wise linear approximation of a non-linear phenomenon achieves higher accuracy. Also, in order to minimize end effects, we extended the time axis well beyond the limits of interest.

With this denser coverage, the symmetric split-step approach could safely be replaced by a simple split-step approach that is described in the book: “Nonlinear Fiber Optics” by G. P. Agrawal. This approach is faster and avoids dealing with any concern about convergence of the second (iterative) part of each symmetric approach cycle -- Paul’s software contained no provision for insuring convergence.
Secondly, it was discovered that the Numerical Recipes implementation of the FFT was not accurate enough for our purposes. Simple tests involving repeated forward and reverse FFT cycles showed that the original signal is not preserved. In these tests, the original pulse was first transformed by (a) a forward FFT in the time direction, and, then, (b) the reverse transform. This was repeated for several passes – in order to model the effect of the FFT on our propagation model. Therefore, the FFT calls were immediately replaced with calls to an IMSL implementation which does preserve signals.

For quality assurance, the software has now been converted to SI (MKS) units and enhanced with logic to calculate performance statistics. Using the new statistics, we improved the speed of the software as well as its ability to model propagation at finer z-axis increments. Test runs modeling 1500 z-axis steps involving 2048 by 512 element matrices have been successfully done -- requiring 5 hours on a 2.8 GHz computer. At that time, shorter tests involving 300 steps required roughly 1 hour.

This software suffered from sensitivities to numerical noise such that it ended
prematurely for most test runs. Because of problems with numerical instabilities, the software was modified to use a discrete Hankel transform (DHT) in the radial direction. Although this was later determined to be an invalid modification, it was used in these early implementations as part of the diffraction calculations.

Thus, the full implementation followed this sequence of calculations for each propagation step:

1. initial pulse
2. output of initial intensity and pulse energy
3. propagation loop (per z slice)
   (a) step to next z (by constant increment)
   (b) forward FFT (time)
   (c) forward Hankel Transform (radius)
   (d) apply diffraction linear operator in (full) frequency domain
   Crank-Nicolson propagation
   tridiagonal matrix solver
   (e) inverse Hankel Transform (radius)
   (f) apply dispersion linear operator in (time) frequency domain
   (g) time derivatives of adjusted pulse in (time) frequency domain
   (h) inverse FFT (time)
   (i) apply non-linear operator in time domain
   (j) output intensity and pulse energy (at specified z values)

In response to questions about parallelizability, we did a thorough analysis of the FORTRAN software and concluded that the propagation phase (the time consuming part) of the software can be partially parallelized for significant performance improvements. Timing statistics show that 3/4 of the execution time is spent in the forward/reverse DHT calculations – which are fully parallelizable matrix operations. Parts of the linear and nonlinear operator applications are also parallelizable.
2.2 Early MATLAB implementations

During April, we began investigating the changes necessary to allow the code to be used in a distributed processing environment. Because of restrictions preventing the migration of the IMSL libraries to this new system, as well as the desire to simplify source code, it was decided to convert the program completely to a MATLAB implementation.

We successfully converted the existing FORTRAN program into an equivalent MATLAB application. This new program ran more quickly than the FORTRAN version as a result of the re-expressing of time-consuming parts as matrix operations (in place of loops) with pre-allocated arrays. For the same 300-step calibration test case, the following execution times are seen: (a) 26 minutes with FORTRAN, versus (b) only 4 minutes with MATLAB.

Additionally, this new implementation requires fewer lines of code than the FORTRAN equivalent and takes advantage of the MATLAB library of specialized functions which include Bessel functions, FFT evaluations, and trapezoidal integration. Therefore, we believe that this translation benefited the quality of the software.

In the process of translating the software to MATLAB, we immediately detected and corrected several scaling errors. These errors involved the calculation of the pulse energy plus actual and critical power. With these fixes, model evolution appeared to occur within an order of magnitude of matching the experimental observations 2 to 4 meters versus 20 to 30 meters.

We successfully added a graphical user interface (GUI) to the MATLAB implementation in order to allow the user to quickly control relevant parameters for both calculations and animations – we enhanced the software by adding the automatic generation of AVI files showing animated model evolutions.
2.3 Attempts at Parallelization

During May and June, the software was adapted to run with the MATLAB Distributed Computing Toolbox in a limited parallel-processing environment. With this system, matrix calculations are distributed from a client machine to multiple “workers” on a “Beowulf” cluster and processed in parallel. The new software will also run on a stand-alone MATLAB version 7 client (without connections to the Beowulf) as a conventional MATLAB application.

Although our implementation is faster for some test runs, the software is actually slower for the more realistic case as data transfer penalties overwhelm parallelization savings.

2.3.1 Success with our Fast Model (without diffraction)

By removing the diffraction calculations, all array operations can be sliced by groups of radii such that the propagation can be calculated with one parallel job.

![Distributed Processing without Diffraction](image)

With diffraction disabled, the application shows a modest speed-up when 8 workers on the cluster are used (see below).

<table>
<thead>
<tr>
<th></th>
<th>500 steps</th>
<th>5000 steps</th>
<th>50000 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>client cpu elapsed</td>
<td>client cpu elapsed</td>
<td>client cpu elapsed</td>
</tr>
<tr>
<td>3.6 GHz client only</td>
<td>10.1 sec.</td>
<td>92.5 sec.</td>
<td>820.3 sec.</td>
</tr>
<tr>
<td>3.6 GHz client plus 8 workers (3.6 GHz)</td>
<td>14.6 sec.</td>
<td>42.7 sec.</td>
<td>371.4 sec.</td>
</tr>
</tbody>
</table>

![Timings without Diffraction](image)
2.3.2 Problems with our Slower Model (with diffraction)

As opposed to the fast version, which uses a single parallel job, our implementation with diffraction requires multiple parallel jobs per propagation step.

![Figure 9: Distributed Processing with Diffraction](image)

As a result, this full version actually runs significantly slower (more than 100 times) when distributed this is apparently due to submission costs associated with each parallel job invocation.

<table>
<thead>
<tr>
<th>50 steps</th>
<th>500 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>3.6 GHz client only</strong></td>
<td><strong>3.6 GHz client plus 8 workers (3.6 GHz)</strong></td>
</tr>
<tr>
<td>client cpu elapsed</td>
<td>client cpu elapsed</td>
</tr>
<tr>
<td>32.3 sec.</td>
<td>294.0 sec.</td>
</tr>
<tr>
<td>33.1 sec.</td>
<td>300.9 sec.</td>
</tr>
<tr>
<td>client cpu elapsed</td>
<td>(est. 4+ hours)</td>
</tr>
<tr>
<td>58.6 sec.</td>
<td>1414.9 sec.</td>
</tr>
</tbody>
</table>

![Figure 10: Timings with Diffraction](image)

After contacting MathWorks (the authors of MATLAB), we received a reply which confirmed that efforts to use the MATLAB version 7 Distributed Computing Toolbox for speed improvements will prove fruitless as data transfer penalties (associated with the transmission of megabyte matrices from client to worker set and back) overwhelm parallelization savings; so, efforts to implement a distributed version of the software were abandoned – until a new release of their software provides a satisfactory option.
2.4 Latest MATLAB implementation

In June, while investigating the prospect of adding new terms (multi-photon absorption and the Drude model), it was determined that a fundamental error had been made in the implementation of the diffraction operator. Our approach to mix the DHT with the Crank-Nicolson propagation and its tri-diagonal matrix solver was flawed. When the problem was corrected, the software had difficulty modeling more than a few steps due to numerical noise. Among other weaknesses, the tri-diagonal matrix solver is very sensitive to numerical errors due to the fact that it involves a lot of divisions.

As a result, a new, simpler approach was developed for diffraction. This led to a major revision of the entire algorithm. In the new algorithm, costly (and hard to debug) calculations involving exponentiation, the Crank-Nicolson method, and a tri-diagonal matrix solver were replaced with a simpler approach involving only Fourier and Hankel transformations to evaluate derivatives.

Although this new algorithm appeared to more stable than earlier versions, it suffered from numerical noise problems at large propagation distances. So, the new approach was supplemented to optionally use frequency-weighed derivative operators. These weightings appear to filter out numerical noise such that analyses out to 5 meters (involving 20,000 steps) can be successfully done without any overflow problems.

The resulting software is very fast. We were able to generate results for each 20,000-step analysis in less than 3 hours of software execution on a single 3.6 GHz computer. The optional use of the MATLAB Distributed Computing Toolbox has been included in the new software in updated form; but, it is untested as the software has proven to be sufficiently fast on a conventional computer.

The following section describes the mathematics used in our latest MATLAB software.
3 LATEST FORMULA GUIDE

In the slowly varying envelope approximation, the electric field is modeled as:

\[ E(z, r, \omega) = A(r, \omega - \omega_0) \exp(+ik_0z) + A^*(r, \omega + \omega_0) \exp(-ik_0z) \]

For our analysis, we wish to model the evolution of the field envelope \( A \) as the pulse moves down a propagation axis. Our reference equations extend the "split-step" formula of Agrawal with a diffraction term used by Potasek as well as Paul:

\[
\frac{\partial A(z, r, t)}{\partial z} = \left( \hat{D}_{ds} + \hat{N} \right) A(z, r, t) + \left( \hat{D}_{df} \right) A(z, r, t)
\]

where

\( A \) is the amplitude of the envelope of the electric field,

\( \hat{D}_{ds} \) is the operator which accounts for absorption and dispersion in a linear medium,

\( \hat{N} \) is the operator which governs the nonlinear effects induced by the medium, and

\( \hat{D}_{df} \) is the operator which accounts for diffraction.

We model the evolution of a radially symmetric pulse – defined as a function of radius \( r \), and pulse duration \( t \) – as it moves down a propagation \( z \) axis. The pulse is normalized for numerical stability – we define:

\[ Q(z, r, t) = \frac{A(z, r, t)}{A(0, 0, 0)} = \frac{A(z, r, t)}{A_0} \]

The divisor is the peak value of an initial Gaussian shape.

The analysis appears sensitive to how we specify: (a) peak intensity, (b) maximum power, (c) critical power, and (d) pulse energy. Various published formulations are included as options.

Also, we have noticed that if diffraction is disabled, the software is generally an order of magnitude faster. Therefore, there is a "fast model" (without diffraction) and a "slower model" (with diffraction).
3.1 Initial Gaussian Pulse

The initial pulse is defined as a function of scaled time ($\tau$) and scaled radius ($\rho$).

\[
\tau \equiv \frac{t}{\tau_p}, \\
\rho \equiv \frac{\rho}{\rho_R}
\]

where $\tau_p$ is the “FWHM” temporal width and $\rho_R$ is the “FWHM” radial width.

(Note: FWHM denotes “full-width half maximum”).

An optional chirp, $\Delta \varphi$ (applied to phase $\varphi$), is assumed to be quadratic in time, such that:

The instantaneous frequency increases linearly from the leading to trailing edge for $\Delta \varphi > 0$ which is called “up-chirp” while the opposite occurs for $\Delta \varphi < 0$ which is called “down-chirp”. If $\Delta \varphi = 0$, there is no chirp and the pulse is Gaussian.

For our chirped Gaussian pulse, its normalized initial shape is:

\[
Q(0, r, t) = \exp\left(\frac{-\tau^2}{2}\right) \exp(-i\Delta \varphi \frac{\tau^2}{2}) \exp\left(-\frac{\rho^2}{2}\right)
\]

or equivalently

\[
Q(0, r, t) = \exp\left[-\frac{\tau^2}{2} (1 + i\Delta \varphi)\right] \exp\left(-\frac{\rho^2}{2}\right)
\]

Figure 11: Gaussian Pulse
3.2 Power and Energy

Power and energy are calculated as follows:

Let the intensity be:

\[ I(z, r, t) = |Q(z, r, t)|^2 \]

Then, total pulse energy [joules] is:

\[
E_{\text{total}}(z) = \int \int \int r |A(z, r, t)|^2 d\theta dr dt
\]
\[
= 2\pi \int \int r |A(z, r, t)|^2 dr dt
\]
\[
= 2\pi A_0^2 \int \int r I(z, r, t) dr dt
\]

Figure 12: cylindrical integration

Within the FWHM limits, the initial pulse energy for a unit Gaussian is

\[
E_{\text{initial}} = \pi \frac{\delta}{2} \tau_R^2 \tau_p (1 - e^{-1}) \text{erf}(1)
\]
\[
\approx 0.9442\pi \left( \tau_R^2 \right) \tau_p
\]
\[
\approx 2.966 \left( \tau_R^2 \right) \tau_p
\]
Thus, the peak intensity \([\text{Watt/m}^2]\) is
\[
P_0 \equiv A_0^2 = \frac{E_0}{E_{\text{crest}}}
\]

The peak intensity determines the maximum power [Watt]
\[
P_{\text{max}} = \pi (\tau_R^2) P_0
\]

which is compared against the critical power [Watt] estimate:

- Paul’s formula (from FORTRAN software):
  \[
P_{\text{crit}} = \pi \left(\frac{1.22}{4}\right)^2 \left(\frac{\lambda^2}{n_2 n_0}\right)
\]

- formula from Sprangle [8] :
  \[
P_{\text{crit}} = \left(\frac{\lambda^2}{2\pi n_2 n_0}\right)
\]

- formula from Mechain [4]:
  \[
P_{\text{crit}} = \left(\frac{3.37 \lambda^2}{8\pi n_2 n_0}\right)
\]
3.3 The Fast Model (without diffraction)

Assuming diffraction is negligible,

\[ \frac{\partial A(z, r, t)}{\partial z} = \left( \hat{D}_{ds} + \hat{N} \right) A(z, r, t) \]

where

- \( A \) is the field envelope,
- \( \hat{D}_{ds} \) is a differential operator which accounts for absorption and dispersion in a linear medium given by:
  \[ \hat{D}_{ds} = -\frac{i}{2} k^{(2)} \frac{\partial^2}{\partial t^2} + \frac{1}{6} k^{(3)} \frac{\partial^3}{\partial t^3} - \frac{1}{2} \alpha \]
- \( \hat{N} \) is the nonlinear operator which governs the nonlinear effects induced by the medium – given by:
  \[ \hat{N} = i \left( \frac{2\pi n_2}{\lambda} \right) |A|^2 - \frac{n_2}{c} \left( 2A^* \frac{\partial A}{\partial t} + A \frac{\partial A^*}{\partial t} \right) - i \left( \frac{2\pi n_2 T_R}{\lambda} \right) \frac{\partial |A|^2}{\partial t} \]

where:

- \( k^{(2)} \) is the second-order group velocity dispersion
- \( k^{(3)} \) is the third-order group velocity dispersion
- \( \alpha \) is the linear loss (\( \alpha < 0 \) implies gain),
- \( n_2 \) is the nonlinear refractive index,
- \( \lambda \) is the central wavelength of the incident laser pulse,
- \( c \) is the speed of light in the vacuum,
- \( T_R \) is the slope of the Raman gain which is related to the delayed (coherent) time response of the medium. And
- \( A^* \) is the complex conjugate of \( A \).
Equivalently,

\[ \frac{\partial A}{\partial z} = \frac{i}{2} k^{(2)} \frac{\partial^2}{\partial t^2} A \]
\[ + \frac{1}{6} k^{(3)} \frac{\partial^3}{\partial t^3} A \]
\[ - \frac{1}{2} \alpha A \]
\[ + \frac{i}{\lambda} (2\pi n_2) |A|^2 A \]
\[ - \frac{n_2}{c} \left( \frac{2A^* \partial A}{\partial t} + A \frac{\partial A^*}{\partial t} \right) A \]
\[ - \frac{i}{\lambda} (2\pi n_2 T_R) \frac{\partial |A|^2}{\partial t} A \]

For a small propagation step (\(\Delta z\)), approximate:

\[ \frac{\partial A}{\partial z} \approx \frac{\Delta A}{\Delta z} \]

\[ A(z + \Delta z, r, t) \approx A(z, r, t) + \Delta A \]

Then:

\[ \frac{\Delta A}{\Delta z} \approx \frac{i}{2} k^{(2)} \frac{\partial^2}{\partial t^2} A \]
\[ + \frac{1}{6} k^{(3)} \frac{\partial^3}{\partial t^3} A \]
\[ - \frac{1}{2} \alpha A \]
\[ + \frac{i}{\lambda} (2\pi n_2) |A|^2 A \]
\[ - \frac{n_2}{c} \left( \frac{2A^* \partial A}{\partial t} + A \frac{\partial A^*}{\partial t} \right) A \]
\[ - \frac{i}{\lambda} (2\pi n_2 T_R) \frac{\partial |A|^2}{\partial t} A \]
### 3.3.1 Spectral Derivation

Fourier transforms can be used to evaluate derivatives:

\[
\frac{\partial}{\partial t} A = \left[ F_t^{-1} i \omega_t F_t \right] A
\]

\[
\frac{\partial^2}{\partial t^2} A = - \left[ F_t^{-1} \omega_t^2 F_t \right] A
\]

\[
\frac{\partial^3}{\partial t^3} A = - \left[ F_t^{-1} i \omega_t^3 F_t \right] A
\]

where

\( F_t \) denotes the Fourier transform in the temporal direction, and \( \omega_t \) is temporal frequency.

With substitutions:

\[
\Delta A \approx i \left( \frac{1}{2} k^{(2)} \Delta z \right) \left[ F_t^{-1} \omega_t^2 F_t \right] A
\]

\[
- \left( \frac{1}{6} k^{(3)} \Delta z \right) \left[ F_t^{-1} i \omega_t^3 F_t \right] A
\]

\[
- \left( \frac{\alpha}{2} \Delta z \right) A
\]

\[
+ i \left( \frac{2\pi n_2 \Delta z}{\lambda} \right) |A|^2 A
\]

\[
- \left( \frac{2n_2 \Delta z}{c} A^* \right) \left[ F_t^{-1} i \omega_t F_t \right] A
\]

\[
- \left( \frac{n_2 \Delta z}{c} A^2 \right) \left[ F_t^{-1} i \omega_t F_t \right] A^*
\]

\[
- i \left( \frac{2\pi n_2 T_R \Delta z}{\lambda} A \right) \left[ F_t^{-1} i \omega_t F_t \right] |A|^2
\]
Let

\[ Q(z, r, t) = \frac{A(z, r, t)}{A(0, 0, 0)} = \frac{A(z, r, t)}{A_0} \]

Then, if \( A_0 \) is real:

\[ Q(z + \Delta z, r, t) \approx Q(z, r, t) + \Delta Q \]

\[ \Delta Q = \frac{\Delta A}{A_0} \]

Then, if \( A_0 \) is real:

\[
A_0 \Delta Q \approx i \left( \frac{1}{2} k^{(2)} \Delta z \right) \left[ F_i^{-1} i \omega_i^2 F_i \right] A_0 Q \\
- \left( \frac{1}{6} k^{(3)} \Delta z \right) \left[ F_i^{-1} i \omega_i^3 F_i \right] A_0 Q \\
- \left( \frac{\alpha}{2} \Delta z \right) A_0 Q \\
+ i \left( \frac{2 \pi n_2 \Delta z}{\lambda} \right) |A_0 Q|^2 A_0 Q \\
- \left( \frac{2 n_2 \Delta z}{c} A_0 Q^* A_0 Q \right) \left[ F_i^{-1} i \omega_i F_i \right] A_0 Q \\
- \left( \frac{n_2 \Delta z}{c} (A_0 Q)^2 \right) \left[ F_i^{-1} i \omega_i F_i \right] A_0 Q^* \\
- i \left( \frac{2 \pi n_2 T_R \Delta z}{\lambda} A_0 Q \right) \left[ F_i^{-1} i \omega_i F_i \right] |A_0 Q|^2
\]
or, equivalently:

\[
\Delta Q \approx i \left( \frac{1}{2} k^{(2)} \Delta z \right) \left[ F_t^{-1} \omega_t^2 F_t \right] Q - \left( \frac{1}{6} k^{(3)} \Delta z \right) \left[ F_t^{-1} i \omega_t^3 F_t \right] Q - \left( \frac{a}{2} \Delta z \right) Q + i \left( \left( \frac{2\pi n_2 \Delta z}{\lambda} \right) A_0^2 \right) |Q|^2 Q - 2 \left( \left( \frac{n_2 \Delta z}{c} \right) A_0^2 \right) (Q^*) Q \left[ F_t^{-1} i \omega_t F_t \right] Q - \left( \left( \frac{n_2 \Delta z}{c} \right) A_0^2 \right) Q^2 \left[ F_t^{-1} i \omega_t F_t \right] Q^* - i \left( \left( \frac{2\pi n_2 \Delta z}{\lambda} \right) T_{rt} \right) Q \left[ F_t^{-1} i \omega_t F_t \right] |Q|^2
\]

where \( Q^* \) is the complex conjugate of \( Q \).
A Fast Fourier Transform (FFT) is used here. Temporal frequencies (in units of radians/time) are calculated as:

\[ \Delta \omega_t = \frac{2\pi}{N_t(\Delta t)} \]

\[ -\frac{\pi}{\Delta t} \leq \omega_t < \frac{\pi}{\Delta t} \]

The ordering of the frequencies follows standard FFT layout.

\( \omega_t = 0 \) occurs as the first matrix element, and negative frequencies occur in the second half of the matrix.
3.4 The Slower Model (with diffraction)

With diffraction included, a new term is added to the starting equation:

\[
\frac{\partial A}{\partial z}_{\text{dif}} = \frac{\partial A}{\partial z}_{\text{fast}} + (\hat{D}_{\text{df}}) A(z, r, t) \\
= \frac{\partial A}{\partial z}_{\text{fast}} + \left[ \frac{i c}{2 n_0 \omega_0} \left( 1 - \frac{1}{\omega_0 \frac{\partial}{\partial t}} \right) \nabla_r^2 \right] A \\
= \frac{\partial A}{\partial z}_{\text{fast}} + \left[ \frac{i \lambda}{4 \pi n_0} \left( 1 - \frac{\lambda}{2 \pi c \frac{\partial}{\partial t}} \right) \nabla_r^2 \right] A
\]

where

- \( A \) is the field envelope at the beginning of the propagation step,
- \( A_{\text{fast}} \) is the field envelope calculated by the fast model,
- \( A_{\text{dif}} \) is the diffracted field envelope,
- \( n_0 \) is the linear refractive index.
- \( \omega_0 \) is the central angular frequency of the incident laser pulse.
- \( c \) is the speed of light in the vacuum, and
- \( \lambda \) is the central wavelength of the incident laser pulse.

Integrating

\[
A_{\text{dif}}(z, r, t) = A_{\text{fast}}(z, r, t) + \int \left[ \left[ \frac{i \lambda}{4 \pi n_0} \left( 1 - \frac{\lambda}{2 \pi c \frac{\partial}{\partial t}} \right) \nabla_r^2 \right] A \right] dz
\]

As a simple approximation

\[
A_{\text{dif}}(z, r, t) \approx A_{\text{fast}}(z, r, t) + \Delta z \left[ \frac{i \lambda}{4 \pi n_0} \left( 1 - \frac{\lambda}{2 \pi c \frac{\partial}{\partial t}} \right) \nabla_r^2 \right] A
\]

The Laplacian with azimuthal symmetry is given by

\[
\nabla_r^2 = \frac{1}{r} \frac{\partial}{\partial r} + \frac{\partial^2}{\partial r^2}
\]
So

\[ A_{\text{dif}}(z, r, t) \approx A_{\text{fast}}(z, r, t) + \Delta z \left[ \frac{i \lambda}{4 \pi n_0} \left( 1 - \frac{\lambda}{2 \pi c} \frac{\partial}{\partial t} \right) \left( \frac{1}{r} \frac{\partial}{\partial r} + \frac{\partial^2}{\partial r^2} \right) \right] A \]

\[ = A_{\text{fast}}(z, r, t) + \Delta A_{\text{dif}} \]

Where

\[ \Delta A_{\text{dif}} = \left( \frac{i \lambda \Delta z}{4 \pi n_0} \right) \left[ \left( 1 - \frac{\lambda}{2 \pi c} \frac{\partial}{\partial t} \right) \left( \frac{1}{r} \frac{\partial}{\partial r} + \frac{\partial^2}{\partial r^2} \right) \right] A \]

or, equivalently,

\[ \Delta A_{\text{dif}} = \left( \frac{i \lambda \Delta z}{4 \pi n_0} \right) \left[ \left( \frac{1}{r} \frac{\partial}{\partial r} + \frac{\partial^2}{\partial r^2} \right) - \left( \frac{\lambda}{2 \pi c} \right) \left( \frac{1}{r} \frac{\partial}{\partial t} \left( \frac{\partial}{\partial r} \right) - \frac{\lambda}{2 \pi c} \frac{\partial}{\partial t} \left( \frac{\partial^2}{\partial r^2} \right) \right) \right] A \]
3.4.1 Spectral Derivation in Two Dimensions

Fourier transforms can be used to evaluate derivatives in the radial and time directions:

\[ \frac{\partial}{\partial r} A = [F_{r}^{-1}i\omega_{r}F_{r}] A \]
\[ \frac{\partial^2}{\partial r^2} A = -[F_{r}^{-1}\omega_{r}^2F_{r}] A \]
\[ \frac{\partial}{\partial t} A = [F_{t}^{-1}i\omega_{t}F_{t}] A \]

where

- \(F_{r}\) denotes the Fourier transform in the radial direction (the Hankel transform),
- \(\omega_{r}\) is radial frequency, and
- \(F_{t}\) denotes the Fourier transform in the temporal direction,
- \(\omega_{t}\) is temporal frequency.

With substitutions,

\[ \frac{\partial}{\partial t} \left( \frac{\partial}{\partial r} A \right) = [F_{t}^{-1}i\omega_{t}F_{t}] \left[ F_{r}^{-1}i\omega_{r}F_{r} \right] A \]
\[ \frac{\partial}{\partial t} \left( \frac{\partial^2}{\partial r^2} A \right) = -\left[ F_{t}^{-1}i\omega_{t}F_{t} \right] \left[ F_{r}^{-1}\omega_{r}^2F_{r} \right] A \]

After rearranging terms:

\[ \Delta A_{\text{diff}} = \left( \frac{i\lambda \Delta z}{4\pi n_0} \right) \left( \frac{1}{r} \right) \left[ F_{r}^{-1}i\omega_{r}F_{r} \right] A \]
\[ - \left( \frac{i\lambda \Delta z}{4\pi n_0} \right) \left[ F_{r}^{-1}\omega_{r}^2F_{r} \right] A \]
\[ - \left( \frac{i\lambda \Delta z}{4\pi n_0} \right) \left( \frac{\lambda}{2\pi c} \right) \left( \frac{1}{r} \right) \left[ F_{r}^{-1}i\omega_{r}F_{r} \right] \left[ F_{t}^{-1}i\omega_{t}F_{t} \right] A \]
\[ + \left( \frac{i\lambda \Delta z}{4\pi n_0} \right) \left( \frac{\lambda}{2\pi c} \right) \left[ F_{r}^{-1}\omega_{r}^2F_{r} \right] \left[ F_{t}^{-1}i\omega_{t}F_{t} \right] A \]
And the field adjustment for diffraction remains:

\[ A_{\text{dif}}(z, r, t) \approx A_{\text{fast}}(z, r, t) + \Delta A_{\text{dif}} \]

Let

\[ Q(z, r, t) = \frac{A(z, r, t)}{A(0, 0, 0)} = \frac{A(z, r, t)}{A_0} \]

Then:

\[ Q_{\text{dif}}(z, r, t) \approx Q_{\text{fast}}(z, r, t) + \Delta Q_{\text{dif}} \]

where

\[
\Delta Q_{\text{dif}} = \left( \frac{i \lambda \Delta z}{4 \pi n_0} \right) \left( \frac{1}{r} \right) \left[ F_r^{-1} i \omega_r F_r \right] Q
\]
\[
- \left( \frac{i \lambda \Delta z}{4 \pi n_0} \right) \left[ F_r^{-1} \omega_r^2 F_r \right] Q
\]
\[
- \left( \frac{i \lambda \Delta z}{4 \pi n_0} \right) \left( \frac{\lambda}{2 \pi c} \right) \left( \frac{1}{r} \right) \left[ F_r^{-1} i \omega_r F_r \right] \left[ F_t^{-1} i \omega_t F_t \right] Q
\]
\[
+ \left( \frac{i \lambda \Delta z}{4 \pi n_0} \right) \left( \frac{\lambda}{2 \pi c} \right) \left[ F_r^{-1} \omega_r^2 F_r \right] \left[ F_t^{-1} i \omega_t F_t \right] Q
\]

As with the fast model, a fast Fourier transform (FFT) is used in the time direction (\( F_t \)).

A discrete Hankel transform (DHT) is used in the radial direction (\( F_r \)).

For this DHT, radii are non-uniformly spaced:

\[ r_j = R_{\text{max}} \frac{C_j}{C_{N_r+1}} \]

\[ 0 < j < N_r \]
and radial frequencies (in units of radians/distance) are also non-uniformly spaced:

\[(\omega_r)_j = \frac{1}{R_{max}} C_j\]

\[0 < j < N_r\]

where \(C_j\) are reference coefficients of the DHT implementation.
3.4.2 Alternate weighted-frequency formulations

The central difference approximation to a derivative is:

\[
\frac{\partial Q}{\partial r}(z, r, t) = \frac{Q(z, r + \Delta r, t) - Q(z, r - \Delta r, t)}{2\Delta r}
\]

This “leap frog” evaluation can be approximated by a ramped scaling of the spectral operator (in the frequency domain)

\[
F_r\left(\frac{\partial Q}{\partial r}\right) = i\omega \left(\frac{\omega_R - |\omega_r|}{\omega_R}\right) F_r(Q)
\]

where

- \(F_r\) denotes the Fourier transform in the radial direction (the Hankel transform),
- \(\omega_r\) is radial frequency, and
- \(\omega_R\) is the maximum radial frequency in the transform.

Let the “triangular” weighting be defined as:

\[
W_R = \frac{\omega_R - |\omega_r|}{\omega_R}
\]

Then, for the “leap frog” evaluation:

\[
\frac{\partial Q}{\partial r} = [F_r^{-1}i\omega W_R F_r]Q
\]
The diffraction term can be adjusted to include this weighting:

$$
\Delta Q_{\text{diff}} \approx \frac{i \lambda \Delta \omega}{4 \pi \eta_0} \left( \frac{1}{r} \right) \left[ F_r^{-1} i \omega_r W_R F_r \right] Q
- \left( \frac{i \lambda \Delta \omega}{4 \pi \eta_0} \right) \left[ F_r^{-1} \omega_r^2 W_R^2 F_r \right] Q
- \left( \frac{i \lambda \Delta \omega}{4 \pi \eta_0} \right) \left( \frac{\lambda}{2 \pi c} \right) \left( \frac{1}{r} \right) \left[ F_r^{-1} i \omega_r W_R F_r \right] \left[ F_t^{-1} i \omega_t F_t \right] Q
+ \left( \frac{i \lambda \Delta \omega}{4 \pi \eta_0} \right) \left( \frac{\lambda}{2 \pi c} \right) \left[ F_r^{-1} \omega_r^2 W_R^2 F_r \right] \left[ F_t^{-1} i \omega_t F_t \right] Q
$$

Alternately, more general weighting formulations may be used:

$$
W_R \equiv \frac{\omega^n_r - |\omega^n_r|}{\omega^n_R}
$$

where $m$ is an integer exponent (0,1,2,3,4,5,...)

A “circular” weighting is given by:

$$
W_R \equiv \frac{\omega_R^2 - \omega_r^2}{\omega_R^2}
$$

A “quartic” weighting is achieved by:

$$
W_R \equiv \frac{\omega_R^4 - \omega_r^4}{\omega_R^4}
$$
A “quintic” weighting is achieved by:

\[ W_R = \frac{\omega_R^5 - |\omega_R^5|}{\omega_R^5} \]

Again, the **diffraction term** is modeled as:

\[
\Delta Q_{\text{dif}} \approx \left( \frac{i\lambda \Delta z}{4\pi n_0} \right) \left( \frac{1}{r} \right) \left[ F_{r}^{-1} i\omega_r W_R F_r \right] Q \\
- \left( \frac{i\lambda \Delta z}{4\pi n_0} \right) \left[ F_{r}^{-1} \omega_r^2 W_R^2 F_r \right] Q \\
- \left( \frac{i\lambda \Delta z}{4\pi n_0} \right) \left( \frac{\lambda}{2\pi c} \right) \left( \frac{1}{r} \right) \left[ F_{r}^{-1} i\omega_t W_R F_r \right] \left[ F_{t}^{-1} i\omega_t F_t \right] Q \\
+ \left( \frac{i\lambda \Delta z}{4\pi n_0} \right) \left( \frac{\lambda}{2\pi c} \right) \left[ F_{r}^{-1} \omega_r^2 W_R^2 F_r \right] \left[ F_{t}^{-1} i\omega_t F_t \right] Q
\]
4 PARAMETER SETTINGS
Parameters are given to the software through its Graphical User Interface (GUI) implemented in MATLAB version 7 format.

4.1 Required Input Parameters
The following quantities (in MKS units) are specified at start-up:

**Material parameters:**
- $\alpha$: linear loss term (per meter) – note: negative implies gain
- $k^{(2)}$: the second order group velocity derivative $(s^2/m)$
  (note: negative implies anomalous dispersion; while positive implies normal dispersion)
- $k^{(3)}$: the third order group velocity derivative $(s^3/m)$
- $n_0$: the linear refractive index
- $n_2$: the nonlinear refractive index $(m^2/Watt)$
- $T_R$: the slope of the Raman gain (sec)
- $E_0$: the “FWHM” energy of the initial laser pulse (joule)
- $\tau_p$: the “FWHM” temporal width (sec) of the input pulse
- $\tau_R$: the “FWHM” radial width (m) of the input pulse
- $\lambda$: the pulse wavelength (m) for vacuum
- $\Delta \varphi$: the chirp of the pulse (> 0 = up chirp, 0 = none, < 0 = down chirp)

**Analysis dimensions:**
- $W_t$: the number of “FWHM” pulse widths to define min. and max. time
- $N_t$: the number of time grid points
- $W_r$: the number of “FWHM” pulse widths to define maximum radius
- $N_r$: the number of radial grid points
- $z_{max}$: the maximum propagation distance (m)
- $z_{step}$: the propagation step size (m)

(Note: FWHM is the “full-width half maximum”).
### 4.2 Values Used

Here are our parameters based on recent experiments to produce filaments – plus additional values were taken from Sprangle [8], Mlejnek [5], Moloney[6], and Schwarz[7]:

<table>
<thead>
<tr>
<th>parameter</th>
<th>unit</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>m$^{-1}$</td>
<td>0.0</td>
<td>linear loss</td>
</tr>
<tr>
<td>$k^{(2)}$</td>
<td>s$^2$/m</td>
<td>$2.2 \times 10^{-29}$</td>
<td>second order GVD term</td>
</tr>
<tr>
<td>$k^{(3)}$</td>
<td>s$^3$/m</td>
<td>0.0</td>
<td>third order GVD term</td>
</tr>
<tr>
<td>$n_0$</td>
<td></td>
<td>1.0</td>
<td>linear refractive index</td>
</tr>
<tr>
<td>$n_2$</td>
<td>m$^2$/W</td>
<td>$1.0 \times 10^{-23}$</td>
<td>nonlinear refractive index</td>
</tr>
<tr>
<td>$T_R$</td>
<td>s</td>
<td>0.0</td>
<td>slope of Raman gain</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parameter</th>
<th>unit</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_0'$</td>
<td>joule</td>
<td>0.015</td>
<td>initial pulse energy</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>m</td>
<td>$8.05 \times 10^{-9}$</td>
<td>wavelength of input pulse</td>
</tr>
<tr>
<td>$\tau_P$</td>
<td>s</td>
<td>$1.25 \times 10^{-14}$</td>
<td>”FWHM” temporal pulse width (duration)</td>
</tr>
<tr>
<td>$\tau_R$</td>
<td>m</td>
<td>$2.5 \times 10^{-3}$</td>
<td>”FWHM” radius of pulse</td>
</tr>
<tr>
<td>$\Delta \phi$</td>
<td></td>
<td>0.0</td>
<td>chirp of pulse</td>
</tr>
</tbody>
</table>

The following constant is defined internally:

**Speed of light in units of m/sec:**

\[ c = 2.99792456 \times 10^8 \]
### Figure 13: Default Settings

<table>
<thead>
<tr>
<th>Time Axis</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of time points</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>FWHM temporal width</td>
<td>1.25e-014 sec</td>
<td></td>
</tr>
<tr>
<td>Total number of pulse widths</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Radial Axis</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of radial points</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>FWHM radius</td>
<td>0.0025 meters</td>
<td></td>
</tr>
<tr>
<td>Total number of pulse widths</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Deriv. weighting:</td>
<td>quintic</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Propagation Direction</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum propagation distance</td>
<td>5 meters</td>
<td></td>
</tr>
<tr>
<td>Propagation (z-axis) step size</td>
<td>0.00025 meters</td>
<td></td>
</tr>
<tr>
<td>Number (z-axis) slice figures</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Media Parameters</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear loss</td>
<td>0 per meter</td>
<td></td>
</tr>
<tr>
<td>2nd order GVD</td>
<td>2.2e-029 sec*2/m</td>
<td></td>
</tr>
<tr>
<td>3rd order GVD</td>
<td>0 sec*3/m</td>
<td></td>
</tr>
<tr>
<td>Nonlinear refractive index</td>
<td>1e-023 m^2/Watt</td>
<td></td>
</tr>
<tr>
<td>Slope of Raman gain</td>
<td>2e-016 sec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Beam Parameters</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>FWHM pulse energy</td>
<td>0.015 joule</td>
<td></td>
</tr>
<tr>
<td>Wavelength</td>
<td>8.05e-007 meters</td>
<td></td>
</tr>
<tr>
<td>Critical power:</td>
<td>Sprangle</td>
<td></td>
</tr>
</tbody>
</table>

Maximum power is 6.000e+011 Watt
Critical power is 1.031e+010 Watt

---

*Figure 13: Default Settings*
5 LATEST RESULTS

5.1 Description of Output

The following items are calculated at each propagation step:

- intensity surface (normalized) at each propagation step

\[ I(z, r, t) = |Q(z, r, t)|^2 \]

- total pulse energy [joules] at a specific propagation step

\[ E_{total}(z) = 2\pi P_0 \int \int r I(z, r, t) dr dt \]

Integration is limited to within the “FWHM” limits and done by the trapezoidal method.

The MATLAB software plots the intensity surface only at intervals selected by the user; we generally produced 20 to 30 slides for each 20,000 step analysis.

To see the cumulative evolution of the pulse over the analysis distance, the software provides composite surface views. In these representations, the 3-dimensional pulse shapes must be reduced to 2-dimensional cross-sectional representations where the time axis is eliminated.

One obvious view is the “maxima silhouette” produced by using the pulse maxima across time at each radius and propagation distance; this is essentially the shape that a target positioned down the propagation axis would see coming at it.

An alternative view is the time integral across the pulse at each radius and distance. This gives the distribution of the total energy seen by a target plane as the pulse quickly passes through it. For display, this surface is normalized relative to its overall maximum value. We call this our “energy pattern” evolution.

For “target plane” views, this "energy pattern" at a specific propagation distance is plotted in Cartesian coordinates – because of the inherent symmetry, concentric circular shapes are seen in this view.
5.2 Tests of the Fourier Transformations

An initial concern was the effect of repeated applications of Fourier transformations (the temporal FFT and the radial DHT) on the model – specifically where accumulated calculation errors would corrupt the answers. Additionally, we needed to know how the alternate frequency weighting would adjust the answers. So, we conducted a series of tests of the net effect of repeated applications of the transforms (FFT and DHT) with only weighting included. For these tests, 20,000 propagation steps were modeled; for each propagation step, the following sequence of transformations was calculated (in order):

A) forward FFT,
B) inverse FFT,
C) forward DHT,
D) apply weighting (if applicable), and
E) inverse DHT.
5.2.1 Case A: tests with no weighting
With no weighting, the transformations do not seem to corrupt the pulse noticeably:

![Figure 14: Pulse before and after 20,000 transform steps](image1)

![Figure 15: maxima silhouette evolution during 20,000 transform steps](image2)
Figure 16: energy pattern evolution during 20,000 transform steps
5.2.2 Case B: tests with circular frequency weighting
With "circular" weighting, the pulse is quickly degraded:

Figure 17: maxima silhouette evolution during 20,000 transform steps

Figure 18: energy pattern evolution during 20,000 transform steps
5.2.3 Case C: tests with quartic frequency weighting

With “quartic” weighting, there is a 20 percent loss of signal during our 20,000 step propagation test.

Figure 19: maxima silhouette evolution during 20,000 transform steps

Figure 20: energy pattern evolution during 20,000 transform steps
5.2.4 Case D: tests with quintic frequency weighting
With “quintic” weighting, there is less than 3 percent loss of signal during our 20,000 step propagation test.

Figure 21: maxima silhouette evolution during 20,000 transform steps

Figure 22: energy pattern evolution during 20,000 transform steps
5.3 Propagation Tests without Diffraction
Without diffraction, the pulse tends to collapse along the propagation axis.

5.3.1 Nonlinear terms only
With only the nonlinear terms active (the linear operator set to zero), the pulse quickly evolves to a singular spike on the propagation axis.

Figure 23: pulse at 2 and 2.5 meters

Figure 24: maxima silhouette up to the singularity
Figure 25: energy pattern evolution up to the singularity
5.3.2 Nonlinear terms plus second-order GVD

By contrast, with the second order group velocity dispersion (GVD) non-zero, the pulse is stretched out in the propagation direction.

Figure 26: pulse shapes at 0.0, 1.5, 3.0, and 4.5 meters

Figure 27: maxima silhouette evolution
Figure 28: energy pattern evolution
5.4 Propagation Tests with Diffraction but No Linear Operator

We performed tests where the linear dispersion operator was set to zero; but, diffraction was enabled. These tests combined a non-zero nonlinear operator with the active diffraction.

5.4.1 Nonlinear only with diffraction and no weighting

At a very short propagation distance, the pulse evolves to a pair of singular spikes alongside the propagation axis; on the target plane, this will actually be a ring.

Because the pulse goes singular at 1.5 meters, the analysis can not continue to larger propagation distances.
5.4.2 Nonlinear only with diffraction and quintic frequency weighting

The incorporation of the alternate frequency weighting did not provide much significant change to our results.

![Figure 30: pulse shapes at 0.5, 1.0, 1.5, and 1.7 meters](image)

Analyses of the results of these tests led us to conclude that a non-zero second-order GVD is required if the pulse evolution is to be modeled at larger propagation distances.
5.5 Propagation with Diffraction and Non-Zero Second-Order GVD

Our best results appear to come from models where diffraction and the second-order GVD are both non-zero and frequency weighting is used.

5.5.1 Singularity Problems with Unweighted Diffraction

Without any frequency weighting, the model progresses up to 2 meters without any numerical problems. However, at around 3 meters, the model becomes corrupted by numerical noise.

Figure 31: pulse shapes at 0.5, 1.0, 1.5, and 2.0 meters
Figure 32: pulse shape at 2.5 meters

Figure 33: pulse corruption around 3 meters
5.5.2 Best Results with Quartic Weighting

By using the “quartic” frequency weighting, we can model the pulse beyond 3 meters. For our tests, we managed to model the evolution out to 5 meters.

Figure 34: pulse shapes at 2 and 2.5 meters
Figure 35: pulse shapes at 3 and 3.5 meters

Figure 36: maxima silhouette evolution
Figure 37: energy pattern evolution

Figure 38: energy pattern as seen by target plane at 2.5 meters
5.5.3 Best Results with Quintic Weighting

As a result of the findings of the transform tests, we replaced all the “quartic-weighted” analyses with “quintic-weighted” equivalents. This new weighting improves the quality of the answer beyond 3 meters. As with the “quartic-weighted” tests, we managed to model the evolution out to 5 meters.

Figure 39: pulse shapes at 2 and 2.5 meters
Figure 40: pulse shapes at 3 and 3.5 meters

Figure 41: maxima silhouette evolution
Figure 42: energy pattern evolution

Figure 43: energy pattern as seen by target plane at 2.7 meters
Figure 44: comparison of composite pulse evolution profiles
5.5.4 Tests with Higher GVD
If the second-order GVD value is doubled, the following results are obtained:

Figure 45: maxima silhouette evolution

Figure 46: energy pattern evolution
Figure 47: comparison of composite pulse evolution profiles

Figure 48: energy pattern as seen by target plane at 3.6 meters
5.6 The Effect of the Chirp Parameter

An optional chirp, $\Delta \phi$ (applied to phase $\phi$), is assumed to be quadratic in time. By our convention, based on that of Agrawal, the instantaneous frequency increases linearly from the leading to trailing edge for $\Delta \phi > 0$ which is called "up-chirp"; while the opposite occurs for $\Delta \phi < 0$ which is called "down-chirp". If $\Delta \phi = 0$, there is no chirp and the pulse is Gaussian.

We repeated our “quintic-weighted” analyses with various values of non-zero chirp.

5.6.1 Example with Positive Chirp

A simple test of positive chirp, where $\Delta \phi = +1.0$, produced the following:

![Figure 49: pulse at 2.5 m. (a) without and (b) with chirp of +1.0](image)

![Figure 50: pulse at 3.0 m. (a) without and (b) with chirp of +1.0](image)
Figure 51: maxima silhouette (a) without and (b) with chirp of +1.0

Figure 52: maxima silhouette (a) without and (b) with chirp of +1.0
Figure 53: energy pattern (a) without and (b) with chirp of +1.0

Figure 54: energy pattern (a) without and (b) with chirp of +1.0
The peak in the energy pattern moves from 2.7 to 2.9 meters when a chirp of +1.0 is used; but a similar pattern is seen.

Figure 55: peak energy pattern as seen by target (a) without and (b) with chirp of +1.0
5.6.2 Example with Negative Chirp
A simple test of negative chirp, where $\Delta \varphi = 1.0$, produced the following:

Figure 56: pulse at 2.5 m. (a) without and (b) with chirp of -1.0

Figure 57: pulse at 3.0 m. (a) without and (b) with chirp of -1.0
Figure 58: maxima silhouette (a) without and (b) with chirp of -1.0

Figure 59: maxima silhouette (a) without and (b) with chirp of -1.0
Figure 60: energy pattern (a) without and (b) with chirp of -1.0

Figure 61: energy pattern (a) without and (b) with chirp of -1.0
The peak in the energy pattern moves from 2.7 to 2.6 meters when a chirp of -1.0 is used; but a similar pattern is seen.

Figure 62: peak energy pattern as seen by target (a) without and (b) with chirp of -1.0
5.6.3 Example with Extreme Negative Chirp

A more extreme test of negative chirp, where $\Delta \varphi = 2.0$, produced the following:

Figure 63: maxima silhouette (a) without and (b) with chirp of -2.0

Figure 64: maxima silhouette (a) without and (b) with chirp of -2.0
Figure 65: energy pattern (a) without and (b) with chirp of -2.0

Figure 66: energy pattern (a) without and (b) with chirp of -2.0
The peak in the energy pattern moves from 2.7 to 2.4 meters when a chirp of -2.0 is used; but a similar pattern is seen.

Figure 67: peak energy pattern as seen by target (a) without and (b) with chirp of -2.0
CONCLUSIONS

In our latest views, we see (a) an initial drop, followed by (b) a rise magnitude at about 2 to 3 meters, and then (c) a gradual drop at larger distances. This pattern is most pronounced in our “energy pattern” depictions where we model the distribution of the total energy seen by a target plane as the pulse quickly passes through it. When viewed on a target plane at a given distance, the energy pattern appears as a bright ring – such that an initial Gaussian pulse has collapsed to a very thin cylindrical shape.

Our results are based solely on mathematical formulations without any experimental verification. In the future, we hope that observations of actual laser experiments will provide results that match our predictions.

Additionally, the software currently lacks adequate safeguards to completely ensure energy conservation with the propagated pulse. Future upgrades should include diagnostics and/or corrections to address this shortcoming.
Appendix A: HOW TO USE THE LATEST MATLAB VERSION

The latest implementation requires MATLAB version 7 to be installed on the computer where the application is to be used. This MATLAB application is started with a special DOS batch file:

![runHighResNLS.bat](image)

This causes the initial pop-up to appear:

![Figure 68: starting the application](image)

Calculation is started with the “Calculate” button at bottom center.
When calculations are activated, the buttons at the bottom turn red; a step counter also appears at the bottom left to indicate status.

![Graphical User Interface (GUI) during active calculations](image)

**Figure 70:** the graphical user interface (GUI) during active calculations
The button turns back to green when calculations are completed; plot buttons activate various display capabilities.

Figure 71: the graphical user interface (GUI) at completion
The first plot button activate pulse animation displays

Figure 72: pulse animation plotting
The second plot button activate composite profile displays

Figure 73: composite profile plotting
The third plot button activate target-plane energy pattern displays

Figure 74: display of energy pattern on target planes
Appendix B: LATEST MATLAB SOURCE CODE

The MATLAB software used in this project consists of two parts: (A) figure files for the graphical user interface, and (B) executable MATLAB textual source code.

B.1 Figures for the Graphical Interface

The graphical user interfaces for this program were created with GUIDE, the MATLAB graphical user interface development environment. GUIDE creates two files, a FIG-file and an M-file. The FIG-file, with extension .fig, is a binary file that contains a description of the layout. The M-file, with extension .m, contains the code that controls the GUI and responds to user actions.

B.1.1 Interface for the main program (highResNLS)

First, there is the interface for the main program, highResNLS.fig:

![GUI definition for the main program](image)

Figure 75: GUI definition for the main program
The following identifiers and callback definitions must be defined:

<table>
<thead>
<tr>
<th>string</th>
<th>tag</th>
<th>style</th>
<th>callback(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>timeGrid512</td>
<td>radiobutton</td>
<td>timeGrid512_Callback</td>
</tr>
<tr>
<td>1024</td>
<td>timeGrid1024</td>
<td>radiobutton</td>
<td>timeGrid1024_Callback</td>
</tr>
<tr>
<td>1.25e-014</td>
<td>taup</td>
<td>edit</td>
<td>taupCallback, taupCreateFcn</td>
</tr>
<tr>
<td>25</td>
<td>widths</td>
<td>edit</td>
<td>widthsCallback, widthsCreateFcn</td>
</tr>
<tr>
<td>256</td>
<td>radialGrid256</td>
<td>radiobutton</td>
<td>radialGrid256_Callback</td>
</tr>
<tr>
<td>512</td>
<td>radialGrid512</td>
<td>radiobutton</td>
<td>radialGrid512_Callback</td>
</tr>
<tr>
<td>0.0025</td>
<td>tauR</td>
<td>edit</td>
<td>tauRCallback, tauCreateFcn</td>
</tr>
<tr>
<td>25</td>
<td>RWidth</td>
<td>edit</td>
<td>RWidthCallback, RWidthCreateFcn</td>
</tr>
<tr>
<td>none</td>
<td>derivFlagRawR</td>
<td>radiobutton</td>
<td>derivFlagRawR_Callback</td>
</tr>
<tr>
<td>quintic</td>
<td>derivFlagQuinticR</td>
<td>radiobutton</td>
<td>derivFlagQuinticR_Callback</td>
</tr>
<tr>
<td>quartic</td>
<td>derivFlagQuarticR</td>
<td>radiobutton</td>
<td>derivFlagQuarticR_Callback</td>
</tr>
<tr>
<td>5</td>
<td>maxPropDist</td>
<td>edit</td>
<td>maxPropDistCallback, maxPropDistCreateFcn</td>
</tr>
<tr>
<td>0.00025</td>
<td>dz</td>
<td>edit</td>
<td>dzCallback, dzCreateFcn</td>
</tr>
<tr>
<td>50</td>
<td>NZSlice</td>
<td>edit</td>
<td>NZSliceCallback, NZSliceCreateFcn</td>
</tr>
<tr>
<td>0</td>
<td>linearLoss</td>
<td>edit</td>
<td>linearLossCallback, linearLossCreateFcn</td>
</tr>
<tr>
<td>2nd order GVD</td>
<td>gvd2onoff</td>
<td>checkbox</td>
<td>gvd2onoffCallback</td>
</tr>
<tr>
<td>2.1e-030</td>
<td>gvd2deriv</td>
<td>edit</td>
<td>gvd2derivCallback, gvd2derivCreateFcn</td>
</tr>
<tr>
<td>0</td>
<td>gvd3deriv</td>
<td>edit</td>
<td>gvd3derivCallback, gvd3derivCreateFcn</td>
</tr>
<tr>
<td>1.0</td>
<td>linearRefract</td>
<td>edit</td>
<td>linearRefract_Callback, linearRefract_CreateFcn</td>
</tr>
<tr>
<td>1e-023</td>
<td>nonlinRefract</td>
<td>edit</td>
<td>nonlinRefractCallback, nonlinRefractCreateFcn</td>
</tr>
<tr>
<td>slope of …</td>
<td>ramanOnOff</td>
<td>checkbox</td>
<td>ramanOnOffCallback</td>
</tr>
<tr>
<td>2e-016</td>
<td>ramanSlope</td>
<td>edit</td>
<td>ramanSlopeCallback, ramanSlopeCreateFcn</td>
</tr>
<tr>
<td>diffraction</td>
<td>diffraction</td>
<td>checkbox</td>
<td>diffraction_Callback</td>
</tr>
<tr>
<td>0</td>
<td>chirp</td>
<td>edit</td>
<td>chirpCallback, chirpCreateFcn</td>
</tr>
<tr>
<td>1.5e-002</td>
<td>pulseEnergy</td>
<td>edit</td>
<td>pulseEnergyCallback, pulseEnergyCreateFcn</td>
</tr>
<tr>
<td>8.05e-007</td>
<td>wavelength</td>
<td>edit</td>
<td>wavelengthCallback, wavelengthCreateFcn</td>
</tr>
<tr>
<td>Paul</td>
<td>powerPaul</td>
<td>radiobutton</td>
<td>powerPaulCallback</td>
</tr>
<tr>
<td>Sprangle</td>
<td>powerSprangle</td>
<td>radiobutton</td>
<td>powerSprangleCallback</td>
</tr>
<tr>
<td>Mechain</td>
<td>powerMechain</td>
<td>radiobutton</td>
<td>powerMechainCallback</td>
</tr>
<tr>
<td>zoomed?</td>
<td>zoomOption</td>
<td>checkbox</td>
<td>zoomOption_Callback</td>
</tr>
<tr>
<td>distributed?</td>
<td>parallel</td>
<td>checkbox</td>
<td>parallel_Callback</td>
</tr>
<tr>
<td>Calculate</td>
<td>calculate</td>
<td>pushbutton</td>
<td>calculateCallback</td>
</tr>
<tr>
<td>Function</td>
<td>Plot</td>
<td>Button Type</td>
<td>Callback</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------</td>
<td>-------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Show Animation</td>
<td>plot1</td>
<td>pushbutton</td>
<td>plot1_Callback</td>
</tr>
<tr>
<td>Show Cumulative</td>
<td>plot2</td>
<td>pushbutton</td>
<td>plot2_Callback</td>
</tr>
<tr>
<td>Show Target Plane</td>
<td>plot3</td>
<td>pushbutton</td>
<td>plot3_Callback</td>
</tr>
<tr>
<td>Status</td>
<td>status</td>
<td>text</td>
<td>(none)</td>
</tr>
<tr>
<td>Maximum power</td>
<td>maxPower</td>
<td>text</td>
<td>(none)</td>
</tr>
<tr>
<td>Critical power</td>
<td>criticalPower</td>
<td>text</td>
<td>(none)</td>
</tr>
</tbody>
</table>
B.1.2 Interface for pulse animation plotting (highResNLSplot1)

For pulse animation plotting, the software uses highResNLSplot1.fig:

![GUI definition for the pulse animation plotter](highResNLSplot1.png)

The following identifiers and callback definitions must be defined:

<table>
<thead>
<tr>
<th>string</th>
<th>tag</th>
<th>style</th>
<th>callback(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>incoming</td>
<td>incoming</td>
<td>radiobutton</td>
<td>incoming_Callback</td>
</tr>
<tr>
<td>outgoing</td>
<td>outgoing</td>
<td>radiobutton</td>
<td>outgoing_Callback</td>
</tr>
<tr>
<td>target</td>
<td>target</td>
<td>radiobutton</td>
<td>target_Callback</td>
</tr>
<tr>
<td>uncompressed</td>
<td>uncompressed</td>
<td>radiobutton</td>
<td>uncompressed_Callback</td>
</tr>
<tr>
<td>compressed</td>
<td>compressed</td>
<td>radiobutton</td>
<td>compressed_Callback</td>
</tr>
<tr>
<td>off</td>
<td>noanimation</td>
<td>radiobutton</td>
<td>noanimation_Callback</td>
</tr>
<tr>
<td>half view</td>
<td>halfview</td>
<td>checkbox</td>
<td>halfview_Callback</td>
</tr>
<tr>
<td>jpeg export</td>
<td>export</td>
<td>checkbox</td>
<td>export_Callback</td>
</tr>
<tr>
<td>plot</td>
<td>plot</td>
<td>pushbutton</td>
<td>plot_Callback</td>
</tr>
<tr>
<td>close all figures</td>
<td>closeall</td>
<td>pushbutton</td>
<td>closeall_Callback</td>
</tr>
<tr>
<td>status</td>
<td>text</td>
<td>(none)</td>
<td></td>
</tr>
</tbody>
</table>
### B.1.3 Interface for composite profile plotting (highResNLsplot2)

For the composite profile displays, the software uses `highResNLsplot2.fig`:

![GUI definition for the composite profile plotter](image)

**Figure 77: GUI definition for the composite profile plotter**

The following identifiers and callback definitions must be defined:

<table>
<thead>
<tr>
<th>string</th>
<th>tag</th>
<th>style</th>
<th>callback(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>incoming</td>
<td>incoming</td>
<td>radiobutton</td>
<td>incoming_Callback</td>
</tr>
<tr>
<td>outgoing</td>
<td>outgoing</td>
<td>radiobutton</td>
<td>outgoing_Callback</td>
</tr>
<tr>
<td>target</td>
<td>target</td>
<td>radiobutton</td>
<td>target_Callback</td>
</tr>
<tr>
<td>maximum</td>
<td>maxVal</td>
<td>radiobutton</td>
<td>maxVal_Callback</td>
</tr>
<tr>
<td>integrated</td>
<td>integrated</td>
<td>radiobutton</td>
<td>integrated_Callback</td>
</tr>
</tbody>
</table>
B.1.4 Interface for target-plane animation plotting (highResNLSplot3)

For animations involving target plane energy patterns, the software requires `highResNLSplot3.fig`:

![GUI definition for the target-plane animation plotter](image)

**Figure 78: GUI definition for the target-plane animation plotter**

The following identifiers and callback definitions must be defined:

<table>
<thead>
<tr>
<th>string</th>
<th>tag</th>
<th>style</th>
<th>callback(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum</td>
<td>maxVal</td>
<td>radiobutton</td>
<td>maxVal_Callback</td>
</tr>
<tr>
<td>integrated</td>
<td>integrated</td>
<td>radiobutton</td>
<td>integrated_Callback</td>
</tr>
<tr>
<td>uncompressed</td>
<td>uncompressed</td>
<td>radiobutton</td>
<td>uncompressed_Callback</td>
</tr>
<tr>
<td>compressed</td>
<td>compressed</td>
<td>radiobutton</td>
<td>compressed_Callback</td>
</tr>
<tr>
<td>off</td>
<td>noanimation</td>
<td>radiobutton</td>
<td>noanimation_Callback</td>
</tr>
<tr>
<td>2-D</td>
<td>draw2D</td>
<td>radiobutton</td>
<td>draw2D_Callback</td>
</tr>
<tr>
<td>3-D</td>
<td>draw3D</td>
<td>radiobutton</td>
<td>draw3D_Callback</td>
</tr>
<tr>
<td>plot</td>
<td>plot</td>
<td>pushbutton</td>
<td>plot_Callback</td>
</tr>
<tr>
<td>close all figures</td>
<td>closeall</td>
<td>pushbutton</td>
<td>closeall_Callback</td>
</tr>
<tr>
<td>status</td>
<td>text</td>
<td>(none)</td>
<td></td>
</tr>
</tbody>
</table>
B.2 Textural Source Code

The figure files are combined with MATLAB textural source code in order to define the complete executable software.

B.2.1 The main program (highResNLS)

First, there is the main program, highResNLS.m:

function varargout = highResNLS( varargin )

Laser Beam Propagation through Air

client-side main program with GUI

14 Sep 2006 version 5.5
with option for
MATLAB 7 Distributed Computing

This program is a simplification of a rewrite of crsplit4 by A. Paul.
which is partially based on the Split Step method described in the
book:
Nonlinear Fiber Optics by G.P. Agrawal.

This method for the extended nonlinear Schrodinger equation (NLS)
models the propagation of optical pulses in a nonlinear material.

input files

---

c.mat -- coefficients for DHT

output files

---

plott.mat -- time axis grid
plotr.mat -- radial axis grid
distances.mat -- propagation distances
energies.mat -- pulse energies
intensity.mat -- intensity surfaces

Control Parameters (inputs)

media parameters:

linear loss term per meter [1/m]
2nd-order group vel. deriv. [sec^2/m]
3rd-order group vel. deriv. [sec^3/m]
% linear refractive index (unitless)
% nonlinear refractive index [m^2/Watt]
% slope of the Raman gain [sec]
% diffraction on/off flag
% beam parameters:
% chirp (-1 = down chirp, +1 = up chirp, 0 = No chirp)
% pulse energy [joule]
% wavelength for vacuum [m]
% self-focusing power option
% 1 = Paul's formula (AFRL Tech. Report)
% 2 = Sprangle formula (Physical Review E 66, 046418 [2002])
% 3 = Mechain formula (Teramobile)
% time-axis parameters:
% pulse width [sec]
% total number of time points
% total number of pulse widths
% radial-axis parameters:
% FWHM radius [m]
% total number of radial points
% total number of radial pulse widths
% derivative option
% propagation-axis parameters:
% maximum propagation distance [m]
% propagation (z-axis) step size [m]
% number (z-axis) slice figures
% output:
% zoom option:
% 0 for wide view (512x256 plotted with 4x4 averaging)
% 1 for zoomed view (center 128x64 plotted)

% Here are a few parameters from recent experiments to produce filaments:
% Wavelength:  805 nm (Center) +/- 10 nm Bandwidth
% Beam Radius: ~15 mm
% Pulse Energy: 430mJ
% Pulse Duration: 50fs
% Chirp: No chirp
% Beam Collimated (non-focused)
% Filament in 2-3 meters from output aperture
% Number of filaments (60 +) (AKA A BUNCH)
% Difficult to tell if pattern remains the same shot-to-shot

% Wavelength:  805 nm (Center) +/- 10 nm Bandwidth
% Beam Radius: ~5 mm
% Pulse Energy: 15mJ (8 mJ is threshold for occasional filament)
% Pulse Duration: 50fs
% Chirp: No Chirp
% Beam Collimated (non-focused)
% Filament in 4 meters from output aperture
% Number of filaments: 1

% The original software has been extended to use the Hankel Transform of
% M. Guizar-Sicairos and J. C. Gutierrez-Vega -- which implements Hankel
% transforms of integer order based on a Fourier-Bessel series expansion
% as described in the recently published work:
% M. Guizar-Sicairos and J. C. Gutierrez-Vega, Computation of
% quasi-discrete Hankel transforms of integer order for propagating
% The numerical method features great accuracy and is energy preserving by
% construction, it is especially suitable for iterative transformation
% processes. Its implementation, requires the computation of zeros of
% the m-th order Bessel function of the first kind where m is the
% transformation order. An array of the first 3001 Bessel functions of
% order from zero to four can be found in the "c.mat" array. If a greater
% transformation order is required the zeros may be found numerically.
% With the c.mat array, as included, Hankel transforms of order 0-4 may be
% computed, with up to 3000 sampling points.
%
%==========================================================================
%============================================
%          executes just before highResNLS is made visible.
%============================================
%

format long;

% Begin initialization code - DO NOT EDIT

gui_Singleton = 1;

gui_State = struct( 'gui_Name', mfilename, ...
                     'gui_Singleton', gui_Singleton, ...
                     'gui_OpeningFcn', @highResNLS_OpeningFcn, ...
                     'gui_OutputFcn', @highResNLS_OutputFcn, ...
                     'gui_Ca...

if( nargin & isstr(varargin{1}) )
    gui_State.gui_Callback = str2func( varargin{1} );
end

if( nargout )
    [varargout{1:nargout}] = gui_mainfcn( gui_State, varargin{:} );
else
    gui_mainfcn( gui_State, varargin{:} );
end

% End initialization code - DO NOT EDIT

%==========================================================================
%============================================

function   highResNLS_OpeningFcn( hObject, eventdata, handles, varargin )

% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

80
function varargout = highResNLS_OutputFcn( hObject, eventdata, handles )

% varargout cell array for returning output args (see VARARGOUT);
% hObject handle to figure
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

%=========================================================================%
% outputs from this function are returned to the command line.
%=========================================================================%

% Initialize graphical interface
function initialize_gui( fig_handle, handles )

%---------------------------
% media parameters
%
linearLoss = 0.0;
gvd2deriv = 2.2*(10^(-29));
gvd3deriv = 0.0;
linearRefract = 1.0;
nonlinRefract = 10^(-23);
% nonlinRefract = 3*(10^(-23)); % from Mechani
ramanSlope    = 2.0e-016;
diffraction   = 1;

set( handles.linearLoss,    'Value',  linearLoss     );
set( handles.gvd2deriv,     'Value',  gvd2deriv     );
set( handles.gvd3deriv,     'Value',  gvd3deriv     );
set( handles.linearRefract, 'Value',  linearRefract );
set( handles.nonlinRefract, 'Value',  nonlinRefract );
set( handles.ramanSlope,     'Value',  ramanSlope    );
set( handles.diffraction,    'Value',  diffraction   );

set( handles.linearLoss,    'String', linearLoss    );
set( handles.gvd2deriv,     'String', gvd2deriv     );
set( handles.gvd3deriv,     'String', gvd3deriv     );
set( handles.linearRefract, 'String', linearRefract );
set( handles.nonlinRefract, 'String', nonlinRefract );
set( handles.ramanSlope,     'String', ramanSlope   );

chirp       = 0.0;
pulseEnergy = 1.5*(10^(-2));
wavelength  = 805.0*(10^(-9));

set( handles.chirp,       'Value',  chirp        );
set( handles.pulseEnergy, 'Value',  pulseEnergy  );
set( handles.wavelength,  'Value',  wavelength   );

set( handles.chirp,       'String', chirp      );
set( handles.pulseEnergy, 'String', pulseEnergy );
set( handles.wavelength,  'String', wavelength  );

taup   = 1.25*(10^(-14));
widths = 25.0;

set( handles.taup,        'Value',  taup         );
set( handles.widths,      'Value',  widths       );
set( handles.taup,        'String', taup       );
set( handles.widths, 'String', widths );

%---------------------------
%
% radial-axis parameters
%

set( handles.radialGrid256, 'Value', 1 );
set( handles.radialGrid512, 'Value', 0 );
set( handles.derivFlagRawR, 'Value', 0 );
set( handles.derivFlagQuarticR, 'Value', 0 );
set( handles.derivFlagQuinticR, 'Value', 1 );

tauR = 2.5*(10^(-3));

RWidth = 25.0;

set( handles.tauR, 'Value', tauR );
set( handles.RWidth, 'Value', RWidth );
set( handles.tauR, 'String', tauR );
set( handles.RWidth, 'String', RWidth );

%---------------------------
%
% propagation axis parameters
%

maxPropDist = 5.0;
dz = 0.00025;
NZSlice = 50;

set( handles.maxPropDist, 'Value', maxPropDist );
set( handles.dz, 'Value', dz );
set( handles.NZSlice, 'Value', NZSlice );

set( handles.maxPropDist, 'String', maxPropDist );
set( handles.dz, 'String', dz );
set( handles.NZSlice, 'String', NZSlice );

%---------------------------
%
% viewing option
%

zoomOption = 1;

set( handles.zoomOption, 'Value', zoomOption );

%---------------------------
%
% show power estimates
%

showPower( handles );
function showPower( handles )

% handles structure with handles and user data (see GUIDATA)

%---------------------------
% % ref. parameters
%
linearRefract = get( handles.linearRefract, 'Value' );
onlinRefract = get( handles.nonlinRefract, 'Value' );
wavelength   = get( handles.wavelength, 'Value' );
pulseEnergy   = get( handles.pulseEnergy, 'Value' );
tau          = get( handles.taup, 'Value' );
tauR         = get( handles.tauR, 'Value' );

%---------------------------
% % peak intensity
%
denom       = pi^(3/2)*(tauR^2)*taup*erf( 1.0 )*( 1.0 - exp( -1.0 ) );
peakIntens  = pulseEnergy/denom;

%---------------------------
% % maximum power
%
Pmax       = pulseEnergy/(2*taup);
text       = sprintf( ' maximum power is %10.3e Watt', Pmax );
set( handles.maxPower, 'String', text );

%---------------------------
% % Paul formula for critical power
%
if( get( handles.powerPaul, 'Value' ) > 0 )

    Pcrit = pi*((1.22*wavelength)^2)/(32.0*nonlinRefract*linearRefract);
    text  = sprintf( ' critical power is %10.3e Watt', Pcrit );
    set( handles.criticalPower, 'String', text );

%---------------------------
% Sprangle formula for critical power
%
elseif( get( handles.powerSprang, 'Value' ) > 0 )
Pcrit = (wavelength^2)/(2*pi*nonlinRefract*linearRefract);

text = sprintf( ' critical power is %10.3e Watt', Pcrit );

set( handles.criticalPower, 'String', text );

%---------------------------
%
% Mechans formula for critical power
%
else
Pcrit = ((3.37*wavelength)^2)/(8*pi*nonlinRefract*linearRefract);

text = sprintf( ' critical power is %10.3e Watt', Pcrit );

set( handles.criticalPower, 'String', text );
end

%==========================================================================
% total number of time points
%==========================================================================

function timeGrid512_Callback(hObject, eventdata, handles)

% hObject handle to timeGrid512 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

disp( 'temporal grid size 512 selected' );
set( handles.timeGrid1024, 'Value', 0 );

%==========================================================================

function timeGrid1024_Callback(hObject, eventdata, handles)

% hObject handle to timeGrid1024 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

disp( 'temporal grid size 1024 selected' );
set( handles.timeGrid512, 'Value', 0 );

%==========================================================================

% temporal pulse width

function taupCreateFcn(hObject, eventdata, handles)

% hObject handle (see GCBO)
function taupCallback(hObject, eventdata, handles)

  taup = str2double(get(hObject, 'String'));

  if isnan(taup)
    set(hObject, 'String', 0);
    errordlg('temporal pulse width must be a number', 'Error');
  else
    set(handles.taup, 'Value', taup);
  end

function widthsCreateFcn(hObject, eventdata, handles)

function widthsCallback(hObject, eventdata, handles)

function widthsCallback(hObject, eventdata, handles)
function derivFlagRawR_Callback(hObject, eventdata, handles)

% hObject handle to derivFlagRawR (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

disp('raw (spectral) derivative selected for radial direction');

set(handles.derivFlagQuarticR, 'Value', 0);
set(handles.derivFlagQuinticR, 'Value', 0);

function derivFlagQuarticR_Callback(hObject, eventdata, handles)

% hObject handle to derivFlagQuarticR (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

set(handles.derivFlagRawR, 'Value', 0);
set(handles.derivFlagQuinticR, 'Value', 0);

function derivFlagQuinticR_Callback(hObject, eventdata, handles)

% hObject handle to derivFlagQuinticR (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

disp('circularly-weighted derivative selected in radial direction');

set(handles.derivFlagRawR, 'Value', 0);
set(handles.derivFlagQuarticR, 'Value', 0);

function radialGrid256_Callback(hObject, eventdata, handles)

% hObject handle to radialGrid256 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

disp('radial grid size 256 selected');
function radialGrid512_Callback(hObject, eventdata, handles)
% hObject    handle to radialGrid512 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

disp('radial grid size 512 selected');
set(handles.radialGrid512, 'Value', 0);

%==========================================================================
%                            radial pulse width
%==========================================================================

function taurCreateFcn(hObject, eventdata, handles)
% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

set(hObject, 'BackgroundColor', 'black');
set(hObject, 'ForegroundColor', 'green');

%----------------------------------------------------
function taurCallback(hObject, eventdata, handles)
% hObject    handle (see GCBO)
%
% handles    structure with handles and user data (see GUIDATA)

tauR = str2double(get(hObject, 'String'));
if(isnan(tauR))
    set(hObject, 'String', 0);
    errordlg('radial pulse width must be a number','Error');
else
    set(handles.tauR, 'Value', tauR);
    showPower(handles);
end

%==========================================================================
%                    total number of radial pulse widths
%==========================================================================

function RWidthCreateFcn(hObject, eventdata, handles)
% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB

88
% handles     empty - handles not created until after all CreateFcns called

set( hObject, 'BackgroundColor', 'black' );
set( hObject, 'ForegroundColor', 'green' );

function  RWidthCallback( hObject, eventdata, handles )

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

RWidth = str2double( get( hObject, 'String' ) );

if( isnan( RWidth ) )
    set( hObject, 'String', 0 );
    errordlg( 'number of radial pulse widths must be a number', 'Error' );
else
    set( handles.RWidth, 'Value', RWidth );
end

%===============================================================================
%                           maximim propagation distance
%===============================================================================

function  maxPropDistCreateFcn( hObject, eventdata, handles )

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

set( hObject, 'BackgroundColor', 'black' );
set( hObject, 'ForegroundColor', 'green' );

function  maxPropDistCallback( hObject, eventdata, handles )

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

maxPropDist = str2double( get( hObject, 'String' ) );

if( isnan( maxPropDist ) )
    set( hObject, 'String', 0 );
    errordlg( 'maximim propagation distance must be a number', 'Error' );
else
    set( handles.maxPropDist, 'Value', maxPropDist );
end
function dzCreateFcn( hObject, eventdata, handles )

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

set( hObject, 'BackgroundColor', 'black' );
set( hObject, 'ForegroundColor', 'green' );

function dzCallback( hObject, eventdata, handles )

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

dz = str2double( get( hObject, 'String' ) );
if( isnan( dz ) )
    set( hObject, 'String', 0 );
    errordlg( 'z-axis step size must be a number', 'Error' );
else
    set( handles.dz, 'Value', dz );
end

function NZSliceCreateFcn( hObject, eventdata, handles )

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

set( hObject, 'BackgroundColor', 'black' );
set( hObject, 'ForegroundColor', 'green' );

function NZSliceCallback( hObject, eventdata, handles )

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

NZSlice = str2double( get( hObject, 'String' ) );
if( isnan( NZSlice ) )
set(hObject, 'String', 0);

errordlg( 'number z-axis slice figures must be a number', 'Error' );

else
    set(handles.NZSlice, 'Value', NZSlice);
end

%========================================================================
%                          linear loss per meter
%========================================================================

function linearLossCreateFcn(hObject, eventdata, handles)

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

set(hObject, 'BackgroundColor', 'black');
set(hObject, 'ForegroundColor', 'green');

%========================================================================
%                          linear loss per meter
%========================================================================

function linearLossCallback(hObject, eventdata, handles)

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

nearLoss = str2double(get(hObject, 'String'));

if isnan(nearLoss)
    set(hObject, 'String', 0);
    errordlg( 'near loss must be a number', 'Error' );
else
    set(handles.linearLoss, 'Value', nearLoss);
end

%========================================================================
%                          on/off toggler of 2nd-order group vel. deriv.
%========================================================================

function gvd2onoffCallback(hObject, eventdata, handles)

% hObject    handle to checkbox2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

if(get(hObject, 'Value') > 0)
    disp( '2nd-order group vel. deriv. on' );

    % gvd2deriv = 2.0*(10^(-28)); % from Schwarz
    % gvd2deriv = 2.2*(10^(-29)); % from Sprangle
    % gvd2deriv = 0.7*(10^(-29)); % best for "leap-frog" deriv. (low res.)
% gvd2deriv = 0.2075*(10^(-29)); % best for raw deriv. (low res.)
% gvd2deriv = 0.125*(10^(-29)); % too-low value

gvd2deriv = 2.2*(10^(-29));

else
    disp('2nd-order group vel. deriv. off');
    gvd2deriv = 0.0;
end

set(handles.gvd2deriv, 'Value', gvd2deriv);
set(handles.gvd2deriv, 'String', gvd2deriv);

function gvd2derivCreateFcn(hObject, eventdata, handles)
    % hObject handle (see GCBO)
    % eventdata reserved - to be defined in a future version of MATLAB
    % handles empty - handles not created until after all CreateFcns called
    gvd2deriv = str2double(get(hObject, 'String'));
    if(isnan(gvd2deriv))
        set(hObject, 'String', 0);
        errordlg('2nd-order group vel. deriv. must be a number', 'Error');
    else
        set(handles.gvd2deriv, 'Value', gvd2deriv);
    end

function gvd2derivCallback(hObject, eventdata, handles)
    % hObject handle (see GCBO)
    % eventdata reserved - to be defined in a future version of MATLAB
    % handles structure with handles and user data (see GUIDATA)
    gvd2deriv = str2double(get(hObject, 'String'));
    if(isnan(gvd2deriv))
        set(hObject, 'String', 0);
        errordlg('2nd-order group vel. deriv. must be a number', 'Error');
    else
        set(handles.gvd2deriv, 'Value', gvd2deriv);
    end

function gvd3derivCreateFcn(hObject, eventdata, handles)
    % hObject handle (see GCBO)
    % eventdata reserved - to be defined in a future version of MATLAB
    % handles empty - handles not created until after all CreateFcns called
function gvd3derivCallback(hObject, eventdata, handles)

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

gvd3deriv = str2double(get(hObject, 'String'));

if (isnan(gvd3deriv))
    set(hObject, 'String', 0);
    errordlg('3rd-order group vel. deriv. must be a number', 'Error');
else
    set(handles.gvd3deriv, 'Value', gvd3deriv);
end

%========================================================
% linear refractive index
%========================================================

function linearRefract_CreateFcn(hObject, eventdata, handles)

% hObject    handle to linearRefract (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

set(hObject, 'BackgroundColor', 'black');
set(hObject, 'ForegroundColor', 'green');

%---------------------------------

function linearRefract_Callback(hObject, eventdata, handles)

% hObject    handle to linearRefract (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

linearRefract = str2double(get(hObject, 'String'));

if (isnan(linearRefract))
    set(hObject, 'String', 0);
    errordlg('linear refractive index must be a number', 'Error');
else
    set(handles.linearRefract, 'Value', linearRefract);
    showPower(handles);
end

93
function nonlinRefractCreateFcn(hObject, eventdata, handles)

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

set(hObject, 'BackgroundColor', 'black');
set(hObject, 'ForegroundColor', 'green');

%--------------------------------------------------

function nonlinRefractCallback(hObject, eventdata, handles)

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

nonlinRefract = str2double(get(hObject, 'String'));

if( isnan(nonlinRefract) )
    set(hObject, 'String', 0);
    errordlg( 'nonlinear refractive index must be a number', 'Error' );
else
    set(handles.nonlinRefract, 'Value', nonlinRefract);
    showPower(handles);
end

%--------------------------------------------------

function ramanOnOffCallback(hObject, eventdata, handles)

% hObject    handle to checkbox2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

if( get(hObject, 'Value') > 0 )
    disp( 'Raman on' );

    ramanSlope = 7.1429e-014;    % from Sprangle
    ramanSlope = 3.4507e-016;    % from Zemyanov
    ramanSlope = 2.0e-016;
else
    disp( 'Raman off' );

    ramanSlope = 0.0;

end
end

set( handles.ramanSlope, 'String', ramanSlope );
set( handles.ramanSlope, 'Value', ramanSlope );

function ramanSlopeCreateFcn( hObject, eventdata, handles )

% hObject handle (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles empty - handles not created until after all CreateFcns called

set( hObject, 'BackgroundColor', 'black' );
set( hObject, 'ForegroundColor', 'green' );

function ramanSlopeCallback( hObject, eventdata, handles )

% hObject handle (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

ramanSlope = str2double( get( hObject, 'String' ) );

if( isnan( ramanSlope ) )
    set( hObject, 'String', 0 );
    errordlg( 'Raman slope must be a number', 'Error' );
else
    set( handles.ramanSlope, 'Value', ramanSlope );
end

function diffraction_Callback( hObject, eventdata, handles )

% hObject handle to checkbox2 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

if( get( hObject, 'Value' ) > 0 )
    disp( 'diffraction on' );
else
    disp( 'diffraction off' );
end

function chirp

95
function chirpCreateFcn(hObject, eventdata, handles)

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

set(hObject, 'BackgroundColor', 'black');
set(hObject, 'ForegroundColor', 'green');

function chirpCallback(hObject, eventdata, handles)

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

chirp = str2double(get(hObject, 'String'));
if(isnan(chirp))
    set(hObject, 'String', 0);
    errordlg('chirp must be a number', 'Error');
else
    set(handles.chirp, 'Value', chirp);
end

function pulseEnergyCreateFcn(hObject, eventdata, handles)

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

set(hObject, 'BackgroundColor', 'black');
set(hObject, 'ForegroundColor', 'green');

function pulseEnergyCallback(hObject, eventdata, handles)

% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

pulseEnergy = str2double(get(hObject, 'String'));
if(isnan(pulseEnergy))
    set(hObject, 'String', 0);
    errordlg('pulse energy must be a number', 'Error');
else
    set( handles.pulseEnergy, 'Value', pulseEnergy );
    showPower( handles );
end

%==========================================================================
%==========================================================================
%======================================
function  wavelengthCreateFcn( hObject, eventdata, handles )
% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
set( hObject, 'BackgroundColor', 'black' );
set( hObject, 'ForegroundColor', 'green' );

%==========================================================================
%==========================================================================
%======================================
function  wavelengthCallback( hObject, eventdata, handles )
% hObject    handle (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
wavelength = str2double( get( hObject, 'String' ) );
if( isnan( wavelength ) )
    set( hObject, 'String', 0 );
    errordlg( 'wavelength must be a number','Error' );
else
    set( handles.wavelength, 'Value', wavelength );
    showPower( handles );
end

%==========================================================================
%==========================================================================
%======================================
function  powerPaulCallback( hObject, eventdata, handles )
% hObject    handle to radio button (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
disp( 'Paul''s formula self-focusing power option selected' );
set( handles.powerSprangle, 'Value', 0 );
set( handles.powerMechain, 'Value', 0 );
showPower( handles );

%========================================================================
% Sprangle formula self-focusing power
%========================================================================

function powerSprangleCallback(hObject, eventdata, handles)
    hObject = handle to radio button (see GCBO)
    eventdata = reserved - to be defined in a future version of MATLAB
    handles = structure with handles and user data (see GUIDATA)

disp( 'Sprangle formula self-focusing power option selected' );
set( handles.powerPaul, 'Value', 0 );
set( handles.powerMechain, 'Value', 0 );
showPower( handles );

%========================================================================
% Mechain formula (Teramobile) self-focusing power
%========================================================================

function powerMechainCallback(hObject, eventdata, handles)
    hObject = handle to radio button (see GCBO)
    eventdata = reserved - to be defined in a future version of MATLAB
    handles = structure with handles and user data (see GUIDATA)

disp( 'Mechain formula self-focusing power option selected' );
set( handles.powerPaul, 'Value', 0 );
set( handles.powerSprangle, 'Value', 0 );
gvd2deriv = get( handles.gvd2deriv, 'Value' );
showPower( handles );

%========================================================================
% zoom option
%========================================================================

function zoomOption_Callback(hObject, eventdata, handles)
    hObject = handle to checkbox2 (see GCBO)
    eventdata = reserved - to be defined in a future version of MATLAB
    handles = structure with handles and user data (see GUIDATA)

if( get( hObject, 'Value' ) > 0 )
    disp( 'zoom on' );
else
    disp( 'zoom off' );
end

%========================================================================
% distributed computing option
%========================================================================

98
function parallel_Callback(hObject, eventdata, handles)

% hObject    handle to parallel (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

if( get(hObject, 'Value') > 0 )
    disp( 'distributed computing on' );
else
    disp( 'distributed computing off' );
end

%===============================================================
% respond to button press (calculate)
%=======================================================================

function calculateCallback(hObject, eventdata, handles)

% hObject    handle to pushbutton2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

set(handles.plot3, 'BackgroundColor', 'green');
set(handles.plot3, 'Enable', 'off');
set(handles.plot2, 'BackgroundColor', 'red');
set(handles.plot2, 'Enable', 'off');
set(handles.plot1, 'BackgroundColor', 'red');
set(handles.plot1, 'Enable', 'off');
set(handles.calculate, 'BackgroundColor', 'red');
set(handles.calculate, 'Enable', 'off');

pause( 0.1 );
diffraction = get(handles.diffraction, 'Value');

if( diffraction > 0 )
    highResNLSfullCalc(handles);
else
    highResNLSfastCalc(handles);
end

set(handles.calculate, 'BackgroundColor', 'green');
set(handles.calculate, 'Enable', 'on');
set(handles.plot1, 'BackgroundColor', 'green');
set(handles.plot1, 'Enable', 'on');
set(handles.plot2, 'BackgroundColor', 'green');
set(handles.plot2, 'Enable', 'on');
set(handles.plot3, 'BackgroundColor', 'green');
set(handles.plot3, 'Enable', 'on');
function plot1_Callback(hObject, eventdata, handles)

set(handles.plot3, 'BackgroundColor', 'red');
set(handles.plot3, 'Enable', 'off');
set(handles.plot2, 'BackgroundColor', 'red');
set(handles.plot2, 'Enable', 'off');
set(handles.plot1, 'BackgroundColor', 'red');
set(handles.plot1, 'Enable', 'off');
set(handles.plot1, 'BackgroundColor', 'red');
set(handles.plot1, 'Enable', 'off');
set(handles.plot2, 'BackgroundColor', 'green');
set(handles.plot2, 'Enable', 'on');
set(handles.plot3, 'BackgroundColor', 'green');
set(handles.plot3, 'Enable', 'on');
set(handles.calculate, 'BackgroundColor', 'red');
set(handles.calculate, 'Enable', 'off');
set(handles.calculate, 'BackgroundColor', 'green');
set(handles.calculate, 'Enable', 'on');

pause(0.1);

highResNLSplot1;

set(handles.calculate, 'BackgroundColor', 'green');
set(handles.calculate, 'Enable', 'on');
set(handles.plot1, 'BackgroundColor', 'green');
set(handles.plot1, 'Enable', 'on');
set(handles.plot2, 'BackgroundColor', 'green');
set(handles.plot2, 'Enable', 'on');
set(handles.plot3, 'BackgroundColor', 'green');
set(handles.plot3, 'Enable', 'on');

%==============================================================================
% respond to button press (cumm. plot1)
%==============================================================================

function plot2_Callback(hObject, eventdata, handles)

set(handles.plot3, 'BackgroundColor', 'red');
set(handles.plot3, 'Enable', 'off');
set(handles.plot2, 'BackgroundColor', 'red');
set(handles.plot2, 'Enable', 'off');
set(handles.plot1, 'BackgroundColor', 'red');
set(handles.plot1, 'Enable', 'off');
set(handles.plot1, 'BackgroundColor', 'red');
set(handles.plot1, 'Enable', 'off');
set(handles.plot2, 'BackgroundColor', 'green');
set(handles.plot2, 'Enable', 'on');
set(handles.plot3, 'BackgroundColor', 'green');
set(handles.plot3, 'Enable', 'on');
set(handles.calculate, 'BackgroundColor', 'red');
set(handles.calculate, 'Enable', 'off');
set(handles.calculate, 'BackgroundColor', 'green');
set(handles.calculate, 'Enable', 'on');

pause(0.1);

highResNLSplot2;
set( handles.plot2, 'Enable', 'on' );
set( handles.plot3, 'Enable', 'on' );
set( handles.plot3, 'BackgroundColor', 'green' );

set( handles.plot3, 'Enable', 'on' );
set( handles.plot3, 'BackgroundColor', 'red' );
set( handles.plot2, 'BackgroundColor', 'red' );
set( handles.plot2, 'Enable', 'off' );
set( handles.plot1, 'BackgroundColor', 'red' );
set( handles.plot1, 'Enable', 'off' );
set( handles.calculate, 'BackgroundColor', 'red' );
set( handles.calculate, 'Enable', 'off' );
set( handles.plot1, 'BackgroundColor', 'green' );
set( handles.plot1, 'Enable', 'on' );
set( handles.plot2, 'BackgroundColor', 'green' );
set( handles.plot2, 'Enable', 'on' );
set( handles.plot3, 'BackgroundColor', 'green' );
set( handles.plot3, 'Enable', 'on' );

%-------------------------------------------------------------------------------------------------
% respond to button press (cumm. plot1)
%-------------------------------------------------------------------------------------------------

function   plot3_Callback( hObject, eventdata, handles )

% hObject    handle to plot1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

set( handles.plot3, 'BackgroundColor', 'red' );
set( handles.plot3, 'Enable', 'off' );
set( handles.plot2, 'BackgroundColor', 'red' );
set( handles.plot2, 'Enable', 'off' );
set( handles.plot1, 'BackgroundColor', 'red' );
set( handles.plot1, 'Enable', 'off' );
set( handles.calculate, 'BackgroundColor', 'red' );
set( handles.calculate, 'Enable', 'off' );

pause( 0.1 );

highResNLSplot3;

set( handles.calculate, 'BackgroundColor', 'green' );
set( handles.calculate, 'Enable', 'on' );
set( handles.plot1, 'BackgroundColor', 'green' );
set( handles.plot1, 'Enable', 'on' );
set( handles.plot2, 'BackgroundColor', 'green' );
set( handles.plot2, 'Enable', 'on' );
set( handles.plot3, 'BackgroundColor', 'green' );
set( handles.plot3, 'Enable', 'on' );

%-------------------------------------------------------------------------------------------------
% end
%-------------------------------------------------------------------------------------------------
B.2.2 The main function for the “fast model” (highResNLSfastCalc)

For the “fast model”, the main program references highResNLSfastCalc.m:

function highResNLSfastCalc( handles )

% Laser Beam Propagation through Air
% fast calculations without diffraction
% 14 Sep 2006 version 5.5
% with option for MATLAB 7 Distributed Computing

% handles structure with handles and user data

format long;

tic;
startCPU = cputime;

% control parameters

speedLight = 2.99792456*(10^8);

% media parameters

linearLoss = get( handles.linearLoss, 'Value' );
gvd2deriv = get( handles.gvd2deriv, 'Value' );
gvd3deriv = get( handles.gvd3deriv, 'Value' );
linearRefract = get( handles.linearRefract, 'Value' );
nonlinRefract = get( handles.nonlinRefract, 'Value' );
ramanSlope = get( handles.ramanSlope, 'Value' );

disp( 'media parameters:' );
linear loss term per meter [1/m] = %12.6e', linearLoss);
disp( sprintf('  2nd-order group vel. deriv. [sec^2/m] = %12.6e', gvd2deriv));
disp( sprintf('  3rd-order group vel. deriv. [sec^3/m] = %12.6e', gvd3deriv));
disp( sprintf('  linear refractive index (unitless) = %12.6e', linearRefract));
disp( sprintf('  nonlinear refractive index [m^2/Watt] = %12.6e', nonlinRefract));
disp( sprintf('  slope of the Raman gain [sec] = %12.6e', ramanSlope));

%----------------------
% beam parameters
%
chirp = get( handles.chirp, 'Value' );
pulseEnergy = get( handles.pulseEnergy, 'Value' );
wavelength = get( handles.wavelength, 'Value' );

disp( 'beam parameters:' );
disp( sprintf('  chirp = %4.2f', chirp));
disp( '(-1 = down chirp, +1 = up chirp, 0 = no chirp)');
disp( sprintf('  initial pulse energy [joule] = %12.6e', pulseEnergy));
disp( sprintf('  wavelength for vacuum [m] = %12.6e', wavelength));

if( get( handles.powerPaul, 'Value' ) > 0 )
  disp('  self-focusing power option of Paul');
elseif( get( handles.powerSprangle, 'Value' ) > 0 )
  disp('  self-focusing power option of Sprangle');
else
  disp('  self-focusing power option of Mechain');
end

%----------------------
% time-axis control
%
if( get( handles.timeGrid1024, 'Value' ) > 0 )
  Ntemporal = 1024;
else
  Ntemporal = 512;
end

taup = get( handles.taup, 'Value' );
widths = get( handles.widths, 'Value' );
tMax = widths*taup;
dt = 2.0*tMax/(1.0*Ntemporal);

disp( 'time-axis parameters:' );
disp( sprintf( '  pulse width [sec] = %12.6e', taup ) );
disp( sprintf( '  total number of time points = %d', Ntemporal ) );
disp( sprintf( '  total number of pulse widths = %12.6e', tMax ) );
disp( sprintf( '  time step [sec] = %12.6e', dt ) );

NhalfTime = Ntemporal/2;
NhalfPlus = NhalfTime + 1;
data.Ntemporal = Ntemporal;

if( get( handles.radialGrid512, 'Value' ) > 0 )
    Nradial = 512;
else
    Nradial = 256;
end

tauR = get( handles.tauR, 'Value' );
RWidth = get( handles.RWidth, 'Value' );
Rmax = RWidth*tauR;
NradialMinus1 = Nradial - 1;
NradialPlus1 = Nradial + 1;

%---------------------------
% propagation-axis control
%---------------------------
maxPropDist = get( handles.maxPropDist, 'Value' );
propStep = get( handles.dz, 'Value' );
NZSlice = get( handles.NZSlice, 'Value' );

data.numStep = floor( (maxPropDist/propStep) + 0.5 );
data.plotted = floor( ((1.0*data.numStep)/(1.0*NZSlice)) + 0.5 );
data.numPlots = NZSlice + 1;

disp( 'propagation-direction parameters:' );
disp( sprintf( ' maximim propagation distance [m] = %12.6e', maxPropDist ));
disp( sprintf( ' z-axis step size [m] = %12.6e', propStep ));
disp( sprintf( ' total number of steps in z = %d', data.numStep ));
disp( sprintf( ' number z-axis slice figures = %d', data.numPlots ));
disp( sprintf( ' (plot after every %dth slice)', data.plotted ));

%---------------------------
% viewing option
% data.zoomOption = get( handles.zoomOption, 'Value' );

disp( 'plotting parameters:' );
disp( sprintf( ' zoom option (1=zoomed,0=wide) = %d', data.zoomOption ));

% temporal plotting
%
if( data.zoomOption == 0 )
    data.Ntplot = Ntemporal/4;
    data.itplot1 = 0;
    data.itplot2 = Ntemporal;
else
    if( Ntemporal == 1024 )
        data.Ntplot = 257;
        data.itplot1 = 385;
        data.itplot2 = 641;
    else
        data.Ntplot = 129;
        data.itplot1 = 193;
        data.itplot2 = 321;
    end
end

% radial plotting
%
if( data.zoomOption == 0 )
    Nrplot = Nradial/4;
else
    if( Nradial == 512 )
        Nrplot = 128;
    else
        Nrplot = 64;
    end

    Nradial        = Nrplot;
    NradialMinus1 = Nradial - 1;
    NradialPlus1  = Nradial + 1;
end

%---------------------------
%                        
%  distribution option     
%                        
%---------------------------

data.parallel = get( handles.parallel, 'Value' );

if( data.parallel == 0 )
    disp( '*** serial mode ***' );
else
    disp( '*** parallel mode ***' );
end

numWorkers    = 8;

rchunk        = int16( Nradial/numWorkers );
plchunk       = int16( Nrplot/numWorkers );

startRadius   = 1;
endRadius     = rchunk;

startPlotted  = 1;
endPlotted    = plchunk;

for worker = 1:numWorkers
    firstPlotted(worker) = startPlotted;
    lastPlotted(worker)  = endPlotted;

    startPlotted     = startPlotted + plchunk;
    endPlotted       = endPlotted + plchunk;

    if( data.zoomOption > 0 )
        firstRadius(worker) = firstPlotted(worker);
        lastRadius(worker)  = lastPlotted(worker);
    else
        firstRadius(worker) = startRadius;
        lastRadius(worker)  = endRadius;
    end

    startRadius       = startRadius + rchunk;
    endRadius         = endRadius  + rchunk;
end
end
end

%----------------------------------------------------------
% intensity, power, and energy
%----------------------------------------------------------
denom = \pi^{(3/2)}*(\tau_R^2)*\tau_p*\text{erf}(1.0)*(1.0 - \exp(-1.0));

peakIntens = \text{pulseEnergy}/\text{denom};

%-------------------------
% maximum power
%
P_{\text{max}} = \pi*(\tau_R^2)*\text{peakIntens};

%--------------------------
% Paul formula for critical power
%
if( \text{get( handles.powerPaul, 'Value' ) > 0 )}
P_{\text{crit}} = \pi*((1.22*\text{wavelength})^2)/(32.0*\text{nonlinRefract}*\text{linearRefract});

%--------------------------
% Sprangle formula for critical power
%
elseif( \text{get( handles.powerSprangle, 'Value' ) > 0 )}
P_{\text{crit}} = (\text{wavelength}^2)/(2*\pi*\text{nonlinRefract}*\text{linearRefract});

%--------------------------
% Mechain formula for critical power
%
else
P_{\text{crit}} = ((3.37*\text{wavelength})^2)/(8*\pi*\text{nonlinRefract}*\text{linearRefract});
end

%---------------------------
% power ratio for nonlinear terms
%
powerRatio = P_{\text{max}}/P_{\text{crit}};

disp('power:');
disp( sprintf( '  P_{\text{max}} [Watt] = %12.6e', P_{\text{max}} ));
disp( sprintf( '  P_{\text{crit}} [Watt] = %12.6e', P_{\text{crit}} ));

%----------------------------------------------------------
energyFactor = 2.0*pi*(peakIntens)*dr*dt;

radial axis

disp('Forming radial axis arrays');
scaledRadii = zeros( Nradial, 1 );
trueRadii   = zeros( Nradial, 1 );
scaledDr    = dr/tauR;
rvalue      = 0.0;
for jradial = 1:Nradial
    scaledRadii(jradial) = rvalue;
    rvalue = rvalue + scaledDr;
end
rscaling   = 1.0/scaledRadii(Nradial);
trueRadii  = scaledRadii*tauR;

% case A: wide-view plotting axis

if( data.zoomOption == 0 )
    plotr = zeros( Nrplot, 1 );
    jplot = 1;
    for j4 = 4:4:Nradial
        j3   = j4 - 1;
        j2   = j3 - 1;
        plotr(jplot) = 0.5*( trueRadii(j2) + trueRadii(j3) );
        jplot   = jplot + 1;
    end
else
    plotr = zeros( Nrplot, 1 );
    for j1 = 1:Nrplot
        plotr(j1) = trueRadii(j1);
    end
end

% case B: zoomed plotting axis

%
end
end

%---------------------------------------------------------------------
%                 temporal axis
%---------------------------------------------------------------------

disp( 'Forming temporal axis arrays' );

scaledTimes = zeros( Ntemporal, 1 );
trueTimes   = zeros( Ntemporal, 1 );
scaledDt    = dt/taup;
tvalue      = 0.0 - ( NhalfTime - 0.5 )*scaledDt;

for itime = 1:Ntemporal
    scaledTimes(itime) = tvalue;
    tvalue             = tvalue + scaledDt;
end

trueTimes   = scaledTimes*taup;

%---------------------------
% case A:  wide-view plotting axis
%
if( data.zoomOption == 0 )
    plott = zeros( data.Ntplot, 1 );
    i3    = data.itplot1;
    for iplot = 1:data.Ntplot
        i3 = i3 + 4;
        i2 = i3 - 1;
        i1 = i2 - 1;
        plott(iplot) = 0.5*( trueTimes(i1) + trueTimes(i2) );
    end

%---------------------------
% case B:  zoomed plotting axis
%
else
    plott = zeros( data.Ntplot, 1 );
    iplot = 1;
    for i1 = data.itplot1:data.itplot2
        plott(iplot) = trueTimes(i1);
        iplot       = iplot + 1;
    end
mask = zeros( Ntemporal, Nradial );
for jradial = 1:Nradial
    rvalue = trueRadii(jradial);
    if( rvalue <= tauR )
        for itime = 1:Ntemporal
            tvalue = abs( trueTimes(itime) );
            if( tvalue <= taup )
                mask(itime,jradial) = rvalue;
            end
        end
    end
end

% initial Gaussian pulse definition
% The pulse is given in terms of scaled units of taup
% where taup is the full width at half maximum in intensity.
% The chirp is controlled by the parameter chirp and is assumed to be
% quadratic in time, i.e., exp(-i*chirp*(t/t_e)^2). This implies that
% the instantaneous frequency increases linearly from the leading to
% trailing edge for chirp > 0 which is called "up-chirp" while the
% opposite occurs for chirp < 0 which is called "down-chirp".

disp( 'Forming Gaussian pulse...' );

radexpons   = -0.5*scaledRadii.*scaledRadii;
radialCurve = exp( radexpons );

tratios     = -0.5*scaledTimes.*scaledTimes;
timexpons   = tratios*( 1.0 + (i*chirp));
timeCurve   = exp( timexpons );
field = complex(1.0,0.0).*timeCurve*radialCurve.

omegas(itime2) = -omegas(itime1);
iv  = i.*omegas.*omegas.*omegas;

a.factor1 = 0.5*i*gvd2deriv*propStep/vd3deriv*propStep/6.0;

field       = complex(1.0,0.0).*(timeCurve*(radialCurve.'));

% derivative operators (time)
disp( 'Pre-calculating temporal derivative operators...' );

% temporal frequencies
omegas     = zeros( Ntemporal,1);
deltaOmega = 2*pi/( dt*Ntemporal );
omega      = 0.0;
for itime = 1:NhalfPlus
  omegas(itime) = omega;
  omega       = omega + deltaOmega;
denime2     = Ntemporal;
for itime1 = 2:NhalfPlus
  omegas(itime2) = -omegas(itime1);
  itime2        = itime2 - 1;
end

data.firstDeriv  = i.*omegas;
data.secondDeriv = omegas.*omegas;
data.thirdDeriv  = i.*omegas.*omegas.*omegas;

disp( 'Pre-calculating constants...' );
data.factor1 = 0.5*i*gvd2deriv*propStep;
data.factor2 = gvd3deriv*propStep/6.0;
data.factor3 = 0.5*linearLoss*propStep;
data.factor4 = i*( 2*pi*nonlinRefract*propStep*peakIntens/wavelength );
data.factor7 = data.factor4*ramanSlope;
data.factor6 = nonlinRefrac*propStep*peakIntens/speedLight;
data.factor5 = 2*data.factor6;

%-------------------------------------------------------------
% propagation loop
%-------------------------------------------------------------

disp( sprintf( 'Beginning propagation loop of %d steps', data.numStep ));
timer = fix( clock );
hr = timer(4);
mnv = timer(5);
sec = timer(6);
disp( sprintf( '--- started at %d:%d:%d', hr, mnv, sec ));

%--------------------------
% % pre-allocate arrays
%--------------------------

intensity = zeros( Nrplot, data.Ntplot, data.numPlots );
radialEnergy = zeros( Nrplot, data.numPlots );
distances = zeros( data.numPlots, 1 );
energies = zeros( data.numPlots, 1 );

%--------------------------
% distances
%--------------------------

figno = 1;
distances(1) = 0.0;

for step = 1:data.numStep
    propagDist = step*propStep;
    if( mod( step, data.plotted ) == 0 )
        figno = figno + 1;
        distances(figno) = propagDist;
    end
end

%--------------------------
% set counters
%--------------------------

lastStep = data.numStep;
lastFrame = data.numPlots;

%------------------------
%
% case A: parallel jobs
%
if( data.parallel > 0 )
    set( handles.status, 'String', '-(distributed)-' );
    pause( 0.001 );
    try
        for worker = 1:numWorkers
            first = firstRadius(worker);
            last = lastRadius(worker);
            data.input = field(:,first:last);
            data.size = last - first + 1;
            data.mask = mask(:,first:last);
            first = firstPlotted(worker);
            last = lastPlotted(worker);
            data.sizeplot = last - first + 1;
            inputs(worker) = data;
        end
        pjob = dfevalasync( @highResNLSfastPropD, 1, inputs, ...
            'LookupURL', 'Win1CM', ...
            'FileDependencies', {'highResNLSfastPropD'}, ...
            'StopOnError', true );
        results = getAllOutputArguments( pjob );
        errmsgs = get( pjob.Tasks, 'ErrorMessage' );
        nonempty = ~cellfun( @isempty, errmsgs );
        celldisp( errmsgs( nonempty ) );
        for worker = 1:numWorkers
            frames = results{worker};
            first = firstPlotted(worker);
            last = lastPlotted(worker);
            intensity(first:last,:) = frames.intensity;
            radialEnergy(first:last,:) = frames.energy;
            if( frames.lastStep < lastStep )
                lastStep = frames.lastStep;
            end
        end
    end
end

if( frames.lastFrame < lastFrame )
    lastFrame = frames.lastFrame;
end
end

destroy( pjob );

catch
    data.parallel = 0;
    disp( '*** serial mode (failed connection) ***' );
end
end

%--------------------------
% % case B: serial calls
% %
if( data.parallel == 0 )
    data.input     = field;
    data.size      = Nradial;
    data.mask      = mask;
    data.sizeplot  = Nrplot;
    frames         = highResNLSfastProp( data, handles );
    intensity      = frames.intensity;
    radialEnergy   = frames.energy;
    lastStep       = frames.lastStep;
    lastFrame      = frames.lastFrame;
end

%--------------------------
% % note singularity
%
if( lastStep < data.numStep )
    disp( '--- field went near-infinite ---' );
    disp( sprintf( 'last step: %d of %d', lastStep, data.numStep ) );
    disp( sprintf( 'last frame: %d of %d', lastFrame, data.numPlots ) );
end

%--------------------------
% % pulse energies
%
for figno = 1:lastFrame
    energies(figno) = trapz( radialEnergy(:,figno) )*energyFactor;
end

%-----------------------------------------------------------
save data

disp( 'Saving data for output...' );

save 'plott.mat' plott;
save 'plotr.mat' plotr;
save 'distances.mat' distances;
save 'energies.mat' energies;
save 'intensity.mat' intensity;

% display execution times

disp( 'analysis complete' );

timeDiff = cputime - startCPU;

disp( sprintf( ' total client cpu time = %9.3f SEC.', timeDiff ));
disp( sprintf( ' total elapsed time = %9.3f SEC.', toc ));

end
B.2.3 Distributed routine for the “fast model” (highResNLSfastPropD)

If the “fast model” is used in a distributed computing environment, the program uses highResNLSfastPropD.m:

```matlab
function frames = highResNLSfastPropD( data )

%**************************************************************************
% Laser Beam Propagation through Air
% fast propagation without diffraction -- distributed
% 14 Sep 2006 version 5.5
% for
% MATLAB 7 Distributed Computing
% data     input parameters
% frames   updated structure
%     intensity = intensity matrices (per figure)
%     energy    = pulse energy       (per figure)
%     lastStep  = last step calculated
%     lastFrame = last frame number
%**************************************************************************

% pre-allocate arrays
frames.intensity = zeros( data.sizeplot, data.Ntplot, data.numPlots );
frames.energy    = zeros( data.sizeplot, data.numPlots );

% initial (plotted) frame
field = data.input;
figno = 1;
frames = fastFrame( figno, data, field, frames );

% propagation loop
frames.lastStep = 0;
for step = 1:data.numStep
    frames.lastStep = step;
```
for item = 1:data.size
    timeBuffer    = field(:,item);
    timeBufferSq  = timeBuffer.*timeBuffer;
    conjBuffer    = conj( timeBuffer );
    conjProd      = conjBuffer.*timeBuffer;
    realpart      = real( timeBuffer );
    imagpart      = imag( timeBuffer );
    absIntens     = (realpart.*realpart) + (imagpart.*imagpart);
    intensProd    = absIntens.*timeBuffer;
    
    % % forward FFT
    
    transField    = fft( timeBuffer );
    transConj     = fft( conjBuffer );
    transIntens   = fft( absIntens );
    
    % % applu derivative operators
    
    firstDerivFr  = data.firstDeriv.*transField;
    secondDerivFr = data.secondDeriv.*transField;
    thirdDerivFr  = data.thirdDeriv.*transField;
    conjDerivFr   = data.firstDeriv.*transConj;
    intensDerivFr = data.firstDeriv.*transIntens;
    
    % % reverse FFT
    
    firstDeriv    = ifft( firstDerivFr  );
    secondDeriv   = ifft( secondDerivFr );
    thirdDeriv    = ifft( thirdDerivFr  );
    conjDeriv     = ifft( conjDerivFr   );
    intensDeriv   = ifft( intensDerivFr );
    
    % % change
    
    part1         = data.factor1.*secondDeriv;
    part2         = data.factor2.*thirdDeriv;
    part3         = data.factor3.*timeBuffer;
    part4         = data.factor4.*intensProd;
part5 = data.factor5.*conjProd.*firstDeriv;
part6 = data.factor6.*timeBufferSq.*conjDeriv;
part7 = data.factor7.*timeBuffer.*intensDeriv;

deltaQ = part1 - part2 - part3 ... 
+ part4 - part5 - part6 - part7;

field(:,item) = timeBuffer + deltaQ;
end

% check for overflow
% test = sum( sum( isnan( field )));
if( test > 0 )
    break;
end

% new (plotted) frame
if( mod( step, data.plotted ) == 0 )
    figno = figno + 1;

    frames = fastFrame( figno, data, field, frames );
    % check for overflow
    % test = sum( isnan( frames.energy ));
    if( test > 0 )
        break;
    end
end
end

generate new plot frame

function frames = fastFrame( figno, data, field, frames )

% figno figure number
% data input parameters
% field latest field matrix
% frames updated stucture
    intensity = intensity matrices (per figure)
    energy   = energy matrices (per figure)
lastStep = last step calculated
lastFrame = last frame number

frames.lastFrame = figno;

%---------------------------
%  get intensity
%---------------------------

absField = abs(field);
sqrField = absField.*absField;

% case A: wide view

if( data.zoomOption == 0 )
  i4 = data.itplot1;

  for iplot = 1:data.Ntplot
    i4 = i4 + 4;
    i3 = i4 - 1;
    i2 = i3 - 1;
    i1 = i2 - 1;

    jplot = 1;

    for j4 = 4:4:data.size
      j3 = j4 - 1;
      j2 = j3 - 1;
      j1 = j2 - 1;

      magnit = sqrField(i1,j1);
      magnit = magnit + sqrField(i2,j1);
      magnit = magnit + sqrField(i3,j1);
      magnit = magnit + sqrField(i4,j1);
      magnit = magnit + sqrField(i1,j2);
      magnit = magnit + sqrField(i2,j2);
      magnit = magnit + sqrField(i3,j2);
      magnit = magnit + sqrField(i4,j2);
      magnit = magnit + sqrField(i1,j3);
      magnit = magnit + sqrField(i2,j3);
      magnit = magnit + sqrField(i3,j3);
      magnit = magnit + sqrField(i4,j3);
      magnit = magnit + sqrField(i1,j4);
      magnit = magnit + sqrField(i2,j4);
      magnit = magnit + sqrField(i3,j4);
      magnit = magnit + sqrField(i4,j4);

      frames.intensity(jplot,iplot,figno) = 0.0625*magnit;
jplot = jplot + 1;
end

sum1 = zeros( data.Ntemporal, 1 );
sum2 = zeros( data.Ntemporal, 1 );
sum3 = zeros( data.Ntemporal, 1 );
sum4 = zeros( data.Ntemporal, 1 );

jplot = 1;
for j4 = 4:4:data.size
    j3 = j4 - 1;
    j2 = j3 - 1;
    j1 = j2 - 1;
    sum1 = sqrField(:,j1).*data.mask(:,j1);
    sum2 = sqrField(:,j2).*data.mask(:,j2);
    sum3 = sqrField(:,j3).*data.mask(:,j3);
    sum4 = sqrField(:,j4).*data.mask(:,j4);
    frames.energy(jplot,figno) = trapz( sum1 ) ... 
        + trapz( sum2 ) ... 
        + trapz( sum3 ) ... 
        + trapz( sum4 );
    jplot = jplot + 1;
end

%--------------------------

%  case B:  zoomed view

else
    iplot = 1;
    for itime = data.itplot1:data.itplot2
        for jplot = 1:data.sizeplot
            frames.intensity(jplot,iplot,figno) = sqrField(itime,jplot);
        end
        iplot = iplot + 1;
    end
    sum1 = zeros( data.Ntemporal, 1 );
    for j1 = 1:data.sizeplot
        sum1 = sqrField(:,j1).*data.mask(:,j1);
        frames.energy(j1,figno) = trapz( sum1 );
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
end
B.2.4 Serial-mode routine for the “fast model” (highResNLSfastProp)

If the “fast model” is used in a non-distributed computing environment, the program uses highResNLSfastProp.m:

function frames = highResNLSfastProp( data, handles )

% Laser Beam Propagation through Air
% fast propagation without diffraction
% (client-only version)
% 14 Sep 2006 version 5.5

% data input parameters
% handles structure with handles and user data
% frames updated structure
%      intensity = intensity matrices (per figure)
%      energy = pulse energy (per figure)
%      lastStep = last step calculated
%      lastFrame = last frame number

% pre-allocate arrays
frames.intensity = zeros( data.sizeplot, data.Ntplot, data.numPlots );
frames.energy = zeros( data.sizeplot, data.numPlots );

% initial (plotted) frame
field = data.input;
figno = 1;
frames = fastFrame( figno, data, field, frames );

% propagation loop
frames.lastStep = 0;
for step = 1:data.numStep
    if( mod( step, 10 ) == 0 )
        text = sprintf( 'step %d of %d', step, data.numStep );
    end
end
set( handles.status, 'String', text );

pause( 0.0000001 );
end

frames.lastStep = step;

for item = 1:data.size
    timeBuffer    = field(:,item);
    timeBufferSq  = timeBuffer.*timeBuffer;
    conjBuffer    = conj( timeBuffer );
    conjProd      = conjBuffer.*timeBuffer;
    realpart      = real( timeBuffer );
    imagpart      = imag( timeBuffer );
    absIntens     = (realpart.*realpart) + (imagpart.*imagpart);
    intensProd    = absIntens.*timeBuffer;

    transConj     = fft( conjBuffer );
    absIntens    = (realpart.*realpart) + (imagpart.*imagpart);
    intensProd    = absIntens.*timeBuffer;

    transField    = fft( timeBuffer );
    transConj     = fft( conjBuffer );
    transIntens   = fft( absIntens );

    firstDerivFr  = data.firstDeriv.*transField;
    secondDerivFr = data.secondDeriv.*transField;
    thirdDerivFr  = data.thirdDeriv.*transField;
    conjDerivFr   = data.firstDeriv.*transConj;
    intensDerivFr = data.firstDeriv.*transIntens;

    firstDeriv    = ifft( firstDerivFr  );
    secondDeriv   = ifft( secondDerivFr );
    thirdDeriv    = ifft( thirdDerivFr  );
    conjDeriv     = ifft( conjDerivFr  );
    intensDeriv   = ifft( intensDerivFr );
% % change %

part1 = data.factor1.*secondDeriv;
part2 = data.factor2.*thirdDeriv;
part3 = data.factor3.*timeBuffer;
part4 = data.factor4.*intensProd;
part5 = data.factor5.*conjProd.*firstDeriv;
part6 = data.factor6.*timeBufferSq.*conjDeriv;
part7 = data.factor7.*timeBuffer.*intensDeriv;

deltaQ = part1 - part2 - part3 ... 
+ part4 - part5 - part6 - part7;

field(:,item) = timeBuffer + deltaQ;
end
%
% check for overflow 
%

%---
% test = sum( sum( isnan( field )))
if( test > 0 )
  break;
end

%---------------------------------------------------------------------
% new (plotted) frame
%
if( mod( step, data.plotted ) == 0 )
  figno = figno + 1;
  frames = fastFrame( figno, data, field, frames );

  % % check for overflow
  %
  test = sum( isnan( frames.energy ));
  if( test > 0 )
    break;
  end
end

%========================================================================
% generate new plot frame
%========================================================================

function frames = fastFrame( figno, data, field, frames )
frames.lastFrame = figno;

%---------------------------
% get intensity
%
absField = abs(field);
sqrField = absField.*absField;

%---------------------------
% case A: wide view
%
if(data.zoomOption == 0)
    i4 = data.itplot1;
    for iplot = 1:data.Ntplot
        i4 = i4 + 4;
        i3 = i4 - 1;
        i2 = i3 - 1;
        i1 = i2 - 1;

        jplot = 1;
        for j4 = 4:4:data.size
            j3 = j4 - 1;
            j2 = j3 - 1;
            j1 = j2 - 1;

            magnit = sqrField(i1,j1);
            magnit = magnit + sqrField(i2,j1);
            magnit = magnit + sqrField(i3,j1);
            magnit = magnit + sqrField(i4,j1);

            magnit = magnit + sqrField(i1,j2);
            magnit = magnit + sqrField(i2,j2);
            magnit = magnit + sqrField(i3,j2);
            magnit = magnit + sqrField(i4,j2);

            magnit = magnit + sqrField(i1,j3);
            magnit = magnit + sqrField(i2,j3);
            magnit = magnit + sqrField(i3,j3);
magnit = magnit + sqrField(i4,j3);
magnit = magnit + sqrField(i1,j4);
magnit = magnit + sqrField(i2,j4);
magnit = magnit + sqrField(i3,j4);
magnit = magnit + sqrField(i4,j4);
frames.intensity(jplot,iplot,figno) = 0.0625*magnit;
   jplot = jplot + 1;
end
end
sum1 = zeros( data.Ntemporal, 1 );
sum2 = zeros( data.Ntemporal, 1 );
sum3 = zeros( data.Ntemporal, 1 );
sum4 = zeros( data.Ntemporal, 1 );
jplot = 1;
for j4 = 4:4:data.size - 1;
   j3 = j4 - 1;
   j2 = j3 - 1;
   j1 = j2 - 1;
   sum1 = sqrField(:,j1).*data.mask(:,j1);
   sum2 = sqrField(:,j2).*data.mask(:,j2);
   sum3 = sqrField(:,j3).*data.mask(:,j3);
   sum4 = sqrField(:,j4).*data.mask(:,j4);
   frames.energy(jplot,figno) = trapz( sum1 ) ...
      + trapz( sum2 ) ...
      + trapz( sum3 ) ...
      + trapz( sum4 );
   jplot = jplot + 1;
end

%--------------------------
% % case B: zoomed view
% %
else
   iplot = 1;
   for itime = data.itplot1:data.itplot2
      for jplot = 1:data.sizeplot
         frames.intensity(jplot,iplot,figno) = sqrField(itime,jplot);
      end
      iplot = iplot + 1;
   end
   sum1 = zeros( data.Ntemporal, 1 );
for j1 = 1:data.sizeplot
    sum1 = sqrField(:,j1).*data.mask(:,j1);
    frames.energy(j1,figno) = trapz(sum1);
end
end

%========================================
%========================================
B.2.5 The main function for the “slower model” (highResNLsfullCalc)

For the “slower model”, the main program references highResNLsfullCalc.m:

```matlab
function highResNLsfullCalc( handles )
%****************************************************************************
% % Laser Beam Propagation through Air
% % full calculations with diffraction
% % 14 Sep 2006 version 5.5
% % with option for
% % MATLAB 7 Distributed Computing
% %
% handles structure with handles and user data (see GUIDATA)
%****************************************************************************
format long;

%--------------------------------------------
% ---
% initialize timing statistics
%--------------------------------------------
tic;
startCPU = cputime;

%--------------------------
% ---
% control parameters
%--------------------------
% speed of light in units of m/sec

speedLight = 2.99792456*(10^8);

%--------------------------
% ---
% media parameters

linearLoss = get( handles.linearLoss, 'Value' );
gvd2deriv = get( handles.gvd2deriv, 'Value' );
gvd3deriv = get( handles.gvd3deriv, 'Value' );
linearRefract = get( handles.linearRefract, 'Value' );
nonlinRefract = get( handles.nonlinRefract, 'Value' );
ramanSlope = get( handles.ramanSlope, 'Value' );

disp( 'media parameters:');
disp( sprintf( ' linear loss term per meter [1/m] = %12.6e', linearLoss ));
```

128
disp( sprintf( '  2nd-order group vel. deriv. [sec^2/m] = %12.6e', gvd2deriv ));
disp( sprintf( '  3rd-order group vel. deriv. [sec^3/m] = %12.6e', gvd3deriv ));
disp( sprintf( '  linear refractive index (unitless) = %12.6e', linearRefract ));
disp( sprintf( '  nonlinear refractive index [m^2/Watt] = %12.6e', nonlinRefract ));
disp( sprintf( '  slope of the Raman gain [sec] = %12.6e', ramanSlope ));

%------------------------
%  beam parameters
%------------------------
chirp        = get( handles.chirp,    'Value' );
pulseEnergy  = get( handles.pulseEnergy, 'Value' );
wavelength   = get( handles.wavelength, 'Value' );
disp( 'beam parameters:' );
disp( sprintf( '  chirp (-1 = down chirp, +1 = up chirp, 0 = no chirp)' ));
disp( sprintf( '  initial pulse energy [joule] = %12.6e', pulseEnergy ));
disp( sprintf( '  wavelength for vacuum [m] = %12.6e', wavelength ));

if( get( handles.powerPaul, 'Value' ) > 0 )
    disp( '  self-focusing power option of Paul' );
elseif( get( handles.powerSprangle, 'Value' ) > 0 )
    disp( '  self-focusing power option of Sprangle' );
else
    disp( '  self-focusing power option of Mechain' );
end

%------------------------
%  time-axis control
%------------------------
if( get( handles.timeGrid1024, 'Value' ) > 0 )
    Ntemporal = 1024;
else
    Ntemporal = 512;
end

taup         = get( handles.taup, 'Value' );
widths       = get( handles.widths, 'Value' );
tMax         = widths*taup;
dt           = 2.0*tMax/(1.0*Ntemporal);
disp( 'time-axis parameters:' );
disp( sprintf( ' pulse width [sec] = %12.6e', taup ));
disp( sprintf( ' total number of time points = %d', Ntemporal ));
disp( sprintf( ' total number of pulse widths = %6.3f', widths ));
disp( sprintf( ' maximum time [sec] = %12.6e', tMax ));
disp( sprintf( ' time step [sec] = %12.6e', dt ));

NhalfTime   = Ntemporal/2;
NhalfPlus   = NhalfTime + 1;
dataB.Ntemporal = Ntemporal;

%---------------------------
% radial-axis control
%---------------------------

if( get( handles.radialGrid512, 'Value' ) > 0 )
    Nradial = 512;
else
    Nradial = 256;
end

if( get( handles.derivFlagRawR, 'Value' ) > 0 )
    derivFlagR = 1;
elseif( get( handles.derivFlagQuarticR, 'Value' ) > 0 )
    derivFlagR = 2;
else
    derivFlagR = 3;
end

tauR       = get( handles.tauR,   'Value' );
RWidth     = get( handles.RWidth, 'Value' );
Rmax       = RWidth*tauR;
NradialPlus1 = Nradial + 1;

data( 'radial-axis parameters:' );
disp( sprintf( ' FWHM radius [m] = %12.6e', tauR ));
disp( sprintf( ' total number of radial points = %d', Nradial ));
disp( sprintf( ' total number of radial pulse widths = %6.3f', RWidth ));
disp( sprintf( ' maximum radial distance [m] = %12.6e', Rmax ));
disp( sprintf( ' derivative option = %d', derivFlagR ));

dataA.Nradial = Nradial;

%---------------------------
% propagation-axis control
%---------------------------

maxPropDist   = get( handles.maxPropDist, 'Value' );
propStep       = get( handles.dz, 'Value' );
NZSlice           = get( handles.NZSlice,     'Value' );
numStep           = floor( (maxPropDist/propStep) + 0.5 );
plotted           = floor( ((1.0*numStep)/(1.0*NZSlice)) + 0.5 );
plotting.numPlots = NZSlice + 1;

disp( 'propagation-direction parameters:' );
disp( sprintf( ' maximim propagation distance [m]      = %12.6e', maxPropDist ));
disp( sprintf( ' z-axis step size [m]                  = %12.6e', propStep ));
disp( sprintf( ' total number of steps in z            = %d',     numStep ));
disp( sprintf( ' number z-axis slice figures           = %d', plotting.numPlots ));
disp( sprintf( '     (plot after every %dth slice)',               plotted ));

%---------------------------
%  viewing option
%
plotting.zoomOption = get( handles.zoomOption, 'Value' );

disp( 'plotting parameters:' );
disp( sprintf( ' zoom option (1=zoomed,0=wide)         = %d', plotting.zoomOption ));

% % temporal plotting
%
if( plotting.zoomOption == 0 )
    plotting.Ntplot  = Ntemporal/4;
    plotting.itplot1 =         0;
    plotting.itplot2 = Ntemporal;
else
    if( Ntemporal == 1024 )
        plotting.Ntplot  = 257;
        plotting.itplot1 = 385;
        plotting.itplot2 = 641;
    else
        plotting.Ntplot  = 129;
        plotting.itplot1 = 193;
        plotting.itplot2 = 321;
    end
end

%
% radial plotting

if( plotting.zoomOption == 0 )
    plotting.Nrplot = Nradial/4;
else
    if( Nradial == 512 )
        plotting.Nrplot = 128;
    else
        plotting.Nrplot = 64;
    end
end

%---------------------------
% distribution option
%
parallel = get( handles.parallel, 'Value' );

if( parallel == 0 )
    disp( '*** serial mode ***' );
else
    disp( '*** parallel mode ***' );
end

numWorkers   = 8;
tchunk       = int16( Ntemporal/numWorkers );
rchunk       = int16( Nradial/numWorkers );
startTime    = 1;
endTime      = tchunk;

startRadius = 1;
endRadius   = rchunk;

for worker = 1:numWorkers
    firstTime(worker)   = startTime;
    lastTime(worker)    = endTime;

    startTime           = firstTime(worker) + tchunk;
    endTime             = lastTime(worker)   + tchunk;

    startRadius(worker) = endRadius - rchunk;
    endRadius(worker)   = startRadius + rchunk;
end

%---------------------------------------------------------------------
% intensity, power, and energy
%---------------------------------------------------------------------
denom = pi^(3/2)*(tauR^2)*taup*erf( 1.0 )*( 1.0 - exp( -1.0 ) );
peakIntens = pulseEnergy/denom;

%maximum power

Pmax = pi*(tauR^2)*peakIntens;

%Paul formula for critical power
if( get( handles.powerPaul, 'Value' ) > 0 )
    Pcrit = pi*((1.22*wavelength)^2)/(32.0*nonlinRefract*linearRefract);
elseif( get( handles.powerSprangle, 'Value' ) > 0 )
    Pcrit = (wavelength^2)/(2*pi*nonlinRefract*linearRefract);
else
    Pcrit = ((3.37*wavelength)^2)/(8*pi*nonlinRefract*linearRefract);
end

powerRatio = Pmax/Pcrit;

disp( 'power:' );
disp( sprintf( '  P_max [Watt] = %12.6e', Pmax ) );
disp( sprintf( '  P_crit [Watt] = %12.6e', Pcrit ) );

%factor for energy calc.
plotting.factor = 2.0*pi*(peakIntens)*dt;
% DHT initialization

disp( 'Pre-calculating arrays for DHT...' );

%----------------------
% read in coefficients matrix
%

disp( '          (loading coefficients matrix)' );
load c.mat;
coefs = c(1,1:Nradial);
cscaling = 1.0/c(1,NradialPlus1);

%----------------------
% radii vector (non-uniform spacing)
%
trueRadii = (coefs')*Rmax*cscaling;

%----------------------
% frequency vector (non-uniform spacing)
%
rscaling = 1.0/Rmax;
radOmegas = (coefs')*rscaling;

%----------------------
% XM is the transformation matrix
%
disp( '          (transformation matrix)' );
[Jn,Jm] = meshgrid( coefs, coefs );
xdenom = abs( besselj( 1, Jn )).*abs( besselj( 1, Jm ) );
xarg = Jn.*Jm.*cscaling;
dataA.XM = (2.0*cscaling)*besselj( 0, xarg )./xdenom;

%----------------------
% DHT scalings
%
disp( '          (scaling matrices)' );
dataA.ml = ( abs( besselj( 1, coefs ) ).*rscaling );
dataA.mlrecip = 1.0./dataA.ml;
%--------------------------------------------------------------------------
%                              radial axis
%--------------------------------------------------------------------------

disp( 'Forming radial axis arrays' );

plotting.radii = trueRadii;
scaledRadii    = trueRadii./tauR;

%---------------------------
%
%  case A:  wide-view plotting axis
%

if( plotting.zoomOption == 0 )
    plotr = zeros( plotting.Nrplot, 1 );
    jplot = 1;
    for j4 = 4:4:Nradial
        j3 = j4 - 1;
        j2 = j3 - 1;
        plotr(jplot) = 0.5*( trueRadii(j2) + trueRadii(j3) );
        jplot = jplot + 1;
    end

%---------------------------
%
%  case B:  zoomed plotting axis
%
else
    plotr = zeros( plotting.Nrplot, 1 );
    for j1 = 1:plotting.Nrplot
        plotr(j1) = trueRadii(j1);
    end
end

%--------------------------------------------------------------------------
%                               temporal axis
%--------------------------------------------------------------------------

disp( 'Forming temporal axis arrays' );

scaledTimes = zeros( Ntemporal, 1 );
tureTimes   = zeros( Ntemporal, 1 );
scaledDt    = dt/taup;
tvalue      = 0.0 - ( NhalfTime - 0.5 )*scaledDt;

for itime = 1:Ntemporal
scaledTimes(itime) = tvalue;
    tvalue = tvalue + scaledDt;
end

trueTimes = scaledTimes*taup;

%------------------------------------------------------------
% case A: wide-view plotting axis
%
if( plotting.zoomOption == 0 )
    plott = zeros( plotting.Ntplot, 1 );
       i3   = plotting.itplot1;
    for iplot = 1:plotting.Ntplot
       i3   = i3 + 4;
       i2   = i3 - 1;
       i1   = i2 - 1;
       plott(iplot) = 0.5*( trueTimes(i1) + trueTimes(i2) );
    end

%------------------------------------------------------------
% case B: zoomed plotting axis
%
else
    plott = zeros( plotting.Ntplot, 1 );
       iplot = 1;
    for i1 = plotting.itplot1:plotting.itplot2
       plott(iplot) = trueTimes(i1);
       iplot = iplot + 1;
    end
end

%------------------------------------------------------------
% pulse energy mask

plotting.Nradial = Nradial;
plotting.mask    = zeros( Ntemporal, Nradial );

for jradial = 1:Nradial
    rvalue = trueRadii(jradial);
    if( rvalue <= tauR )
        for itime = 1:Ntemporal
            tvalue = abs( trueTimes(itime) );
        end
if( tvalue <= taup )
    plotting.mask(itime,jradial) = rvalue;
end
end
end

%--------------------------------------------------------------------------
%                     initial Gaussian pulse definition
%--------------------------------------------------------------------------
%
% The pulse is given in terms of scaled units of taup
% where taup is the full width at half maximum in intensity.
% The chirp is controlled by the parameter chirp and is assumed to be
% quadratic in time, i.e., exp(-i*chirp*(t/t_e)^2). This implies that
% the instantaneous frequency increases linearly from the leading to
% trailing edge for chirp > 0 which is called "up-chirp" while the
% opposite occurs for chirp < 0 which is called "down-chirp".
%
disp( 'Forming Gaussian pulse...' );
%
% radial part
%
radexpons   = -0.5*scaledRadii.*scaledRadii;
radialCurve = exp( radexpons );

% temporal part
%
tratios     = -0.5*scaledTimes.*scaledTimes;
timexpons   = tratios*( 1.0 + (i*chirp));
timeCurve   = exp( timexpons );

% full pulse
%
field       = complex(1.0,0.0).*(timeCurve*(radialCurve.'));

%----------------------
% derivative operators (time)
%----------------------

disp( 'Pre-calculating temporal derivative operators...' );

%----------------------
% temporal frequencies
omegas = zeros( Ntemporal, 1 );
deltaOmega = 2*pi/( dt*Ntemporal );
omega = 0.0;
for itime = 1:NhalfPlus
    omegas(itime) = omega;
    omega = omega + deltaOmega;
end
itime2 = Ntemporal;
for itime1 = 2:NhalfPlus
    omegas(itime2) = -omegas(itime1);
    itime2 = itime2 - 1;
end
%----------------------
%  time-derivative operators
%----------------------
dataB.firstDeriv = i.*omegas;
dataB.secondDeriv = omegas.*omegas;
dataB.thirdDeriv = i.*omegas.*omegas.*omegas;
%--------------------------------------------------------------------------
%                          pre-calculated constants
%--------------------------------------------------------------------------
disp( 'Pre-calculating temporal constants...' );
dataB.factor1 = 0.5*i*gvd2deriv*propStep;
dataB.factor2 = gvd3deriv*propStep/6.0;
dataB.factor3 = 0.5*linearLoss*propStep;
dataB.factor4 = i*( 2*pi*nonlinRefract*propStep*peakIntens/wavelength );
dataB.factor7 = dataB.factor4*ramanSlope;
dataB.factor6 = nonlinRefract*propStep*peakIntens/speedLight;
dataB.factor5 = 2*dataB.factor6;
%--------------------------------------------------------------------------
%                           diffraction operator
%--------------------------------------------------------------------------
disp( 'Pre-calculating factors for diffraction...' );
%-----------------------------------------------
% radial derivative operator
%
dataA.derivOp = i.*radOmegas;

%----------------------
%
% weighting functions for radial derivative
%
if( derivFlagR == 2 )
   rNqu  = radOmegas(Nradial)^4;
   denom = 1.0/rNqu;
   for jradial = 1:Nradial
      rqu   = radOmegas(jradial)^4;
      ratio = ( rNqu - rqu )*denom;
      dataA.derivOp(jradial) = dataA.derivOp(jradial).*ratio;
   end
elseif( derivFlagR == 3 )
   rNqu  = abs( radOmegas(Nradial)^5 );
   denom = 1.0/rNqu;
   for jradial = 1:Nradial
      rqu   = abs( radOmegas(jradial)^5 );
      ratio = ( rNqu - rqu )*denom;
      dataA.derivOp(jradial) = dataA.derivOp(jradial).*ratio;
   end
end

%----------------------
%
% scaling factors for diffraction
%
dataA.divRad = 1.0./trueRadii;

factorV      = (i*wavelength*propStep)/( 4*pi*linearRefract );
factorD      = factorV*( wavelength/( 2*pi*speedLight ));

dataB.scale1 = factorV;
dataB.scale2 = factorV;
dataB.scale3 = factorD;
dataB.scale4 = factorD;

%--------------------------------------------------------------------------
%                              initial frame
%--------------------------------------------------------------------------
disp( 'Generating initial (plotted) frame...' );

frames.intensity = zeros( plotting.Nrplot, ... 
                      plotting.Ntplot, ... 
                      plotting.numPlots );

frames.energy    = zeros( plotting.numPlots, 1 );
distances        = zeros( plotting.numPlots, 1 );

figno            = 1;
distances(1)     = 0.0;
frames           = generateFrame( figno, plotting, field, frames );

lastStep         = 0;

%--------------------------------------------------------------------------
%                           pre-allocate arrays
%--------------------------------------------------------------------------

if( parallel > 0 )
    disp( 'pre-allocating arrays for distributed processing...' );

    firstDeriv  = complex(1.0,0.0).*zeros( Ntemporal, Nradial );
    secondDeriv = complex(1.0,0.0).*zeros( Ntemporal, Nradial );
end

%--------------------------------------------------------------------------
%                            propagation loop
%--------------------------------------------------------------------------

disp( sprintf( 'Beginning propagation loop of %d steps', numStep ));

timer  = fix( clock );

hr     = timer(4);
minv   = timer(5);
sec    = timer(6);

disp( sprintf( '--- started at %d:%d:%d', hr, minv, sec ));

broken = 0;

for step = 1:numStep
    text     = sprintf( 'step %d of %d', step, numStep );

    set( handles.status, 'String', text );

    pause( 0.00001 );

    lastStep = step;

    %----------------------

%----------------------
% case A: parallel jobs
% if( parallel > 0 )
    try
        for worker = 1:numWorkers
            first           = firstTime(worker);
            last            = lastTime(worker);
            dataA.size      = last - first + 1;
            dataA.input     = field(first:last,:);
            inputsA(worker) = dataA;
        end
        pjobA = dfevalasync( @highResNLSfullPartA, 1, inputsA,             ...
            'LookupURL',          'Win1CM',               ...
            'FileDependencies',  {'highResNLSfullPartA'}, ...
            'StopOnError',        true                   );
    waitforState( pjobA, 'finished' );
    resultsA  = getAllOutputArguments( pjobA );
    errmsgsA  = get( pjobA.Tasks, {'ErrorMessage'} );
    nonemptyA = ~cellfun( @isempty, errmsgsA );
    celldisp( errmsgsA( nonemptyA ) );
    for worker = 1:numWorkers
        returnedA                 = resultsA(worker);
        first                     = firstTime(worker);
        last                      = lastTime(worker);
        firstDeriv(first:last,:)  = returnedA.firstDeriv;
        secondDeriv(first:last,:) = returnedA.secondDeriv;
    end
    destroy( pjobA );
end

%--------
for worker = 1:numWorkers
    first                = firstRadius(worker);
    last                 = lastRadius(worker);
    dataB.size           = last - first + 1;
    dataB.input          = field(:,first:last);
    dataB.firstRadDeriv  = firstDeriv(:,first:last);
    dataB.secondRadDeriv = secondDeriv(:,first:last);
    inputsB(worker)      = dataB;
end
pjobB = dfevalasync( @highResNLSfullPartB, 1, inputsB,             ...
waitForState( pjobB, 'finished' );

resultsB  = getAllOutputArguments( pjobB );

errmsgsB = get( pjobB.Tasks, {'ErrorMessage'} );
onemptyB = ~cellfun( @isempty, errmsgsB );
celldisp( errmsgsB( nonemptyB ) );

for worker = 1:numWorkers
    first = firstRadius(worker);
    last = lastRadius(worker);

    field(:,first:last) = resultsB{worker};
end

destroy( pjobB );

%---------
catch
    parallel = 0;

    disp( '*** serial mode (failed connection) ***' );
end
end

%----------------------
%
%  case B:  serial calls
%
if( parallel == 0 )
    dataA.input         = field;
    dataA.size          = Ntemporal;
    returnedA           = highResNLSfullPartA( dataA );

    dataB.input         = field;
    dataB.firstRadDeriv = returnedA.firstDeriv;
    dataB.secondRadDeriv = returnedA.secondDeriv;

    dataB.size          = Nradial;

    field               = highResNLSfullPartB( dataB );
end

test = sum( sum( isnan( field )));
if( test > 0 )
    broken = 1;
    break;
end
if( mod( step, plotted ) == 0 )
    figno = figno + 1;
    distances(figno) = step*propStep;
    frames = generateFrame( figno, plotting, field, frames );
    test = sum( isnan( frames.energy ));
    if( test > 0 )
        broken = 1;
        break;
    end
end

if( broken == 1 )
    disp( '---> field went near-infinite <---' );
    disp( sprintf( 'last step: %d of %d', lastStep, numStep ) );
    disp( sprintf( 'last frame: %d of %d', frames.lastFrame, plotting.numPlots ) );
end

disp( 'Saving data for output...' );

energies = frames.energy;
intensity = frames.intensity;

save 'plott.mat' plott;
save 'plotr.mat' plotr;
save 'distances.mat' distances;
save 'energies.mat' energies;
save 'intensity.mat' intensity;

disp( 'analysis complete' );
timeDiff = cputime - startCPU;

disp( sprintf( '   total client cpu time = %9.3f sec.', timeDiff ));
disp( sprintf( '   total elapsed time    = %9.3f sec.', toc      ));

%==========================================================================
%                         generate new plot frame
%==========================================================================

function  frames = generateFrame( figno, plotting, field, frames )

% figno    figure number
% plotting plotting control parameters
% field    field value matrix
% frames   updated structure
%           intensity = intensity matrices (per figure)
%           energy    = pulse energy       (per figure)
%           lastFrame = last frame number
%
frames.lastFrame     = figno;

%---------------------------
% calculate pulse energy
%---------------------------

absField             = abs( field );
sqrField             = absField.*absField;
powerMatrix          = sqrField.*plotting.mask;
firstIntegral        = trapz( powerMatrix );
secondIntegral       = trapz( plotting.radii, firstIntegral' );
frames.energy(figno) = secondIntegral*plotting.factor;

%---------------------------
% case A:  wide view
%---------------------------

if( plotting.zoomOption == 0 )
    i4 = plotting.itplot1;
    for iplot = 1:plotting.Ntplot
        i4    = i4 + 4;
        i3    = i4 - 1;
        i2    = i3 - 1;
        i1    = i2 - 1;
        jplot = 1;
    end
for j4 = 4:4:plotting.Nradial
    j3 = j4 - 1;
    j2 = j3 - 1;
    j1 = j2 - 1;
    magnit = sqrField(i1,j1);
    magnit = magnit + sqrField(i2,j1);
    magnit = magnit + sqrField(i3,j1);
    magnit = magnit + sqrField(i4,j1);
    magnit = magnit + sqrField(i1,j2);
    magnit = magnit + sqrField(i2,j2);
    magnit = magnit + sqrField(i3,j2);
    magnit = magnit + sqrField(i4,j2);
    magnit = magnit + sqrField(i1,j3);
    magnit = magnit + sqrField(i2,j3);
    magnit = magnit + sqrField(i3,j3);
    magnit = magnit + sqrField(i4,j3);
    magnit = magnit + sqrField(i1,j4);
    magnit = magnit + sqrField(i2,j4);
    magnit = magnit + sqrField(i3,j4);
    magnit = magnit + sqrField(i4,j4);
    frames.intensity(jplot,iplot,figno) = 0.0625*magnit;
    jplot = jplot + 1;
end
end

%-------------------------------
% case B: zoomed view
%-------------------------------
else
    iplot = 1;
    for itime = plotting.itplot1:plotting.itplot2
        for jplot = 1:plotting.Nrplot
            frames.intensity(jplot,iplot,figno) = sqrField(itime,jplot);
        end
        iplot = iplot + 1;
    end
end

%==========================================================================
%                                  end
%==========================================================================
B.2.7 Part A of the the “slower model” (highResNLSfullPartA)

For the first part of the “slower model”, the software uses highResNLSfullPartA.m:

```matlab
function returned = highResNLSfullPartA( data )
%
%**************************************************************************
% \Laser Beam Propagation through Air
% \part A of propagation step -- distributed (DHT processing)
% \14 Sep 2006 version 5.5
% \for MATLAB 7 Distributed Computing
% \data input parameters
% \input = original field values
% \returned returned data
%**************************************************************************
%
% pre-allocate arrays
%
returned.firstDeriv = complex(1.0,0.0).*zeros( data.size, data.Nradial );
returned.secondDeriv = complex(1.0,0.0).*zeros( data.size, data.Nradial );
%
% radial derivatives
%
for item = 1:data.size
    radialBuffer2 = data.input(item,:).';
    scaled = radialBuffer2.*data.m1recip;
    transformed = data.XM*scaled;
    onceScaled = data.derivOp.*transformed;
    firstTrans = data.XM*onceScaled;
    firstTerm = firstTrans.*data.m1;
    returned.firstDeriv(item,:) = (data.divRad.*firstTerm).';
%
```
% second derivative
%

    twiceScaled = data.derivOp.*onceScaled;

    secondTrans = data.XM*twiceScaled;

    secondTerm = secondTrans.*data.m1;

    returned.secondDeriv(item,:) = secondTerm.);

end

%==========================================================================
%                                  end
%==========================================================================
B.2.8 Part B of the “slower model” (highResNLSfullPartB)

For the second part of the “slower model”, the software uses `highResNLSfullPartB.m`:

```matlab
function  returned = highResNLSfullPartB( data )
%**************************************************************************
%                     Laser Beam Propagation through Air
%                     part B of propagation step -- distributed
%                        (FFT processing)
%  14 Sep 2006 version 5.5
%                        for
%                       MATLAB 7 Distributed Computing
%  data       input parameters
%  returned   returned structure
%                 adjusted field values
%**************************************************************************
% pre-allocate arrays
%
returned = complex(1.0,0.0).*zeros( data.Ntemporal, data.size );
%**************************************************************************
% propagation step
%
for item = 1:data.size
    timeBuffer       = data.input(:,item);
    timeBufferSq     = timeBuffer.*timeBuffer;
    conjBuffer       = conj( timeBuffer );
    conjProd         = conjBuffer.*timeBuffer;
    realpart         = real( timeBuffer );
    imagpart         = imag( timeBuffer );
    absIntens        = (realpart.*realpart) + (imagpart.*imagpart);
    intensProd       = absIntens.*timeBuffer;
    firstRadDrv      = data.firstRadDeriv(:,item);
    secondRadDrv     = data.secondRadDeriv(:,item);
    %
```
% forward FFT
%
transField       = fft( timeBuffer );
transConj        = fft( conjBuffer );
transIntens      = fft( absIntens  );
transFirstR      = fft( firstRadDrv );
transSecondR     = fft( secondRadDrv );
%
% apply derivative operators
%
firstDerivFr     = data.firstDeriv.*transField;
secondDerivFr    = data.secondDeriv.*transField;
thirdDerivFr     = data.thirdDeriv.*transField;
conjDerivFr      = data.firstDeriv.*transConj;
intensDerivFr    = data.firstDeriv.*transIntens;
derivFirstFr     = data.firstDeriv.*transFirstR;
derivSecondFr    = data.firstDeriv.*transSecondR;
%
% reverse FFT
%
firstDeriv       = ifft( firstDerivFr  );
secondDeriv      = ifft( secondDerivFr );
thirdDeriv       = ifft( thirdDerivFr  );
conjDeriv        = ifft( conjDerivFr   );
intensDeriv      = ifft( intensDerivFr );
derivFirstRD     = ifft( derivFirstFr  );
derivSecondRD    = ifft( derivSecondFr );
%
% updated field envelope
%
part1            = data.factor1.*secondDeriv;
part2            = data.factor2.*thirdDeriv;
part3            = data.factor3.*timeBuffer;
part4            = data.factor4.*intensProd;
part5            = data.factor5.*conjProd.*firstDeriv;
part6            = data.factor6.*timeBufferSq.*conjDeriv;
part7            = data.factor7.*timeBuffer.*intensDeriv;

partD1           = data.scale1.*firstRadDrv;
pard2            = data.scale2.*secondRadDrv;
pard3            = data.scale3.*derivFirstRD;
pard4            = data.scale4.*derivSecondRD;
deltaQ = part1 - part2 - part3 ... 
+ part4 - part5 - part6 - part7 ... 
+ partD1 + partD2 - partD3 - partD4;

newField = timeBuffer + deltaQ;

%save results%

returned(:,item) = newField;
end

%==========================================================================
%                                  end 
%==========================================================================
B.2.9 Pulse animation plotting (highResNLSplot1)

For pulse animation plotting, the software uses highResNLSplot1.m:

```matlab
function varargout = highResNLSplot1(varargin)

%**************************************************************************
%                           Laser Beam Propagation through Air
%                           pulse animation plotting
%  14 Sep 2006 version 5.5
%     for MATLAB 7
%==========================================================================
%
%                                input files
%                                -----------
%  plott.mat     -- time axis grid
%  plotr.mat     -- radial axis grid
%  distances.mat -- propagation distances
%  energies.mat  -- pulse energies
%  intensity.mat -- intensity surfaces
%==========================================================================
%
%                        Control Parameters (inputs)
%                        ---------------------------
% plotting parameters:
%    perspective
%        1 for incoming perspective
%        2 for outgoing perspective
%        3 for target   perspective
%    animation option
%        0 = off (multiple figures displayed)
%        1 = on with compression (low-quality AVI file generated)
%             (single figure updated)
%        2 = on with no compression (high-quality AVI file generated)
%             (single figure updated)
%    halfview
%        0 for full-surface view
%        1 for half-surface view
%    export flag
%        1 for on  (jpeg files generated per frame or figure)
%        0 for off
%==========================================================================
%
%                                main program
%==========================================================================

format long;
```
% Begin initialization code - DO NOT EDIT

gui_Singleton = 1;

gui_State = struct(...
    'gui_Name',       mfilename,                   ...
    'gui_Singleton',  gui_Singleton,               ...
    'gui_OpeningFcn', @highResNLSplot1_OpeningFcn, ...
    'gui_OutputFcn',  @highResNLSplot1_OutputFcn,  ...
    'gui_LayoutFcn',  [],                          ...
    'gui_Callback',   []                          );

if( nargin & isstr( varargin{1} ) )
    gui_State.gui_Callback = str2func( varargin{1} );
end

if( nargout )
    [varargout{1:nargout}] = gui_mainfcn( gui_State, varargin{:} );
else
    gui_mainfcn( gui_State, varargin{:} );
end

% End initialization code - DO NOT EDIT

%==========================================================================
%         executes just before highResNlSplot is made visible.
%==========================================================================

function highResNLSplot1_OpeningFcn( hObject, eventdata, handles, varargin )

    hObject handle to figure
    eventdata reserved - to be defined in a future version of MATLAB
    handles structure with handles and user data (see GUIDATA)
    varargin command line arguments to highResNlSplot (see VARARGIN)

% This function has no output args, see OutputFcn.

% Choose default command line output for highResNlSplot

handles.output = hObject;

% Update handles structure

guidata( hObject, handles );

% UIWAIT makes highResNlSplot wait for user response (see UIRESUME)
% uiwait(handles.figure1);

%==========================================================================
% outputs from this function are returned to the command line.
%==========================================================================

function varargout = highResNLSplot1_OutputFcn( hObject, eventdata, handles )
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargout  cell array for returning output args (see VARARGOUT);

% Get default command line output from handles structure
varargout{1} = handles.output;

%==========================================================================
%                            incoming perspective
%==========================================================================

function incoming_Callback(hObject, eventdata, handles)

% hObject    handle to incoming (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of incoming

disp( 'incoming perspective selected' );
set( handles.target, 'Value', 0 );
set( handles.outgoing, 'Value', 0 );

%==========================================================================
%                            outgoing perspective
%==========================================================================

function outgoing_Callback(hObject, eventdata, handles)

% hObject    handle to outgoing (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of outgoing

disp( 'outgoing perspective selected' );
set( handles.target, 'Value', 0 );
set( handles.incoming, 'Value', 0 );

%==========================================================================
%                            target perspective
%==========================================================================

function target_Callback(hObject, eventdata, handles)

% hObject    handle to target (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of target
disp( 'target perspective selected' );
set( handles.incoming, 'Value', 0 );
set( handles.outgoing, 'Value', 0 );

%==========================================================================
%                          uncompressed animation
%==========================================================================

function uncompressed_Callback( hObject, eventdata, handles )
    % hObject    handle to uncompressed (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
    % Hint: get(hObject,'Value') returns toggle state of uncompressed
    disp( 'uncompressed selected' );
    set( handles.compressed, 'Value', 0 );
    set( handles.noanimation, 'Value', 0 );

%==========================================================================
%                           compressed animation
%==========================================================================

function compressed_Callback( hObject, eventdata, handles )
    % hObject    handle to compressed (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
    % Hint: get(hObject,'Value') returns toggle state of compressed
    disp( 'compressed selected' );
    set( handles.uncompressed, 'Value', 0 );
    set( handles.noanimation, 'Value', 0 );

%==========================================================================
%                      no animation (multiple figures)
%==========================================================================

function noanimation_Callback( hObject, eventdata, handles )
    % hObject    handle to noanimation (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
    % Hint: get(hObject,'Value') returns toggle state of noanimation
    disp( 'noanimation selected' );
    set( handles.uncompressed, 'Value', 0 );
    set( handles.compressed, 'Value', 0 );
function export_Callback(hObject, eventdata, handles)

% hObject    handle to export (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of export

disp('export selected');

function halfview_Callback(hObject, eventdata, handles)

% hObject    handle to halfview (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of halfview

disp('halfview selected');

function plot_Callback(hObject, eventdata, handles)

% hObject    handle to plot (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

set(handles.closeall, 'BackgroundColor', 'red');
set(handles.closeall, 'Enable', 'off');
set(handles.plot, 'BackgroundColor', 'red');
set(handles.plot, 'Enable', 'off');

disp('plot activated');

format long;

% parameters for plotting

if( get(handles.target, 'Value') > 0 )
    perspective = 3;
elseif( get(handles.outgoing, 'Value') > 0 )
    perspective = 2;

155
else
    perspective = 1;
end

if( get( handles.uncompressed, 'Value' ) > 0 )
    animation = 2;
elseif( get( handles.compressed, 'Value' ) > 0 )
    animation = 1;
else
    animation = 0;
end

halfview   = get( handles.halfview, 'Value' );
exportFlag = get( handles.export, 'Value' );

disp( 'plotting parameters:' );
disp( sprintf( '    perspective (1=incoming,2=outgoing,3=target) = %d', perspective ));
disp( sprintf( '    animation   (2=uncompr,1=compr,0=off)        = %d', animation ));
disp( sprintf( '    halfview flag (1=half,0=full)                = %d', halfview ));
disp( sprintf( '    export flag   (1=on,0=off)                   = %d', exportFlag ));

%----------------------------------------------------------------------------------
%                               load data
%----------------------------------------------------------------------------------

disp( 'loading output data...' );
load 'plott.mat'    plott;
load 'plotr.mat'    plotr;
load 'distances.mat' distances;
load 'energies.mat' energies;
load 'intensity.mat' intensity;

% % radial axis
%
lengths   = size( plotr );
Nrdata    = lengths(1);
radialMax = plotr(Nrdata);
if( plotr(1) == 0.0 )
    Nrplot = 2*Nrdata - 1;
else
    Nrplot = 2*Nrdata;
end

% % temporal axis
%
lengths = size( plott );

Ntplot = lengths(1);

timeMax = plott(Ntplot);

% adjust plotting units
%
if( radialMax < (10^(-3)) )
    plotr = plotr*(10^6);
    radialMax = radialMax*(10^6);
    radialLabel = 'radius (micron)';
elseif( radialMax < 1.0 )
    plotr = plotr*(10^3);
    radialMax = radialMax*(10^3);
    radialLabel = 'radius (mm)';
else
    radialLabel = 'radius (m)';
end

if( timeMax < (10^(-12)) )
    plott = plott*(10^15);
    timeMax = timeMax*(10^15);
    timeLabel = 'time (fs)';
elseif( timeMax < (10^(-9)) )
    plott = plott*(10^12);
    timeMax = timeMax*(10^12);
    timeLabel = 'time (ps)';
elseif( timeMax < (10^(-6)) )
    plott = plott*(10^9);
    timeMax = timeMax*(10^9);
    timeLabel = 'time (ns)';
else
    timeLabel = 'time (sec)';
end

if(( Ntplot == 128 ) || ( Ntplot == 256 ))
    verticalLabel = 'rel. intensity (4x4 Avg.)';
else
    verticalLabel = 'rel. intensity';
end

% mirror r about z axis
%
lengths = size( intensity );
Numplots = length(N);  
lastFigure = 0;  
for figno = 1:Numplots  
    if( energies(figno) == 0.0 )  
        break;  
    end  
    lastFigure = figno;  
end  
Numplots = lastFigure;  
fullr = zeros( Nrplot, 1 );  
fullSurf = zeros( Nrplot, Ntplot, Numplots );  
for figno = 1:Numplots  
    for iplot = 1:Ntplot  
        jplot = 1;  
        for jradial = Nrdata:-1:1  
            fullr(jplot) = -plotr(jradial);  
            fullSurf(jplot,iplot,figno) = intensity(jradial,iplot,figno);  
            jplot = jplot + 1;  
        end  
        if( plotr(1) == 0.0 )  
            for jradial = 2:Nrdata  
                fullr(jplot) = plotr(jradial);  
                fullSurf(jplot,iplot,figno) = intensity(jradial,iplot,figno);  
                jplot = jplot + 1;  
            end  
        else  
            for jradial = 1:Nrdata  
                fullr(jplot) = plotr(jradial);  
                fullSurf(jplot,iplot,figno) = intensity(jradial,iplot,figno);  
                jplot = jplot + 1;  
            end  
        end  
    end  
set( handles.closeall, 'UserData', Numplots );

%---------------------------------------------------------------------  
%                         output initial pulse  
%---------------------------------------------------------------------


disp('plotting...');

propagDist = 0.0;
figno = 1;

handle = figure(1);

pulseEnergy = energies(figno);
etext = sprintf('pulse energy (FWHM) = %8.6f J', pulseEnergy);
text = sprintf('prop. distance = 0.0, %s', etext);

axis([-timeMax timeMax -radialMax radialMax]);
title(text, 'FontWeight', 'bold');
xlabel(timeLabel, 'FontWeight', 'bold');
ylabel(radialLabel, 'FontWeight', 'bold');
zlabel(verticalLabel, 'FontWeight', 'bold');

if (perspective == 3)
    view([90 0]);
elseif (perspective == 2)
    view([-50 30]);
else
    view([50 30]);
end

pause(0.1);

hold on;
grid on;

if (halfview == 1)
    surfl(plott, plotr, intensity(:,:,figno));
else
    surfl(plott, fullr, fullSurf(:,:,figno));
end

shading interp;
colormap(gray);
hold off;

refresh(1);

if (animation ~= 0)
    frames(figno) = getframe(handle);
end

if (exportFlag ~= 0)
    filename = sprintf('figure%d.jpg', figno);
    print(handle, '-djpeg', filename);
end
for figno = 2:Numplots
    propagDist = distances(figno);

    text = sprintf(' frame %d of %d', figno, Numplots);
    set(handles.status, 'String', text);

    if( propagDist > 0 )
        pulseEnergy = energies(figno);

        if( animation == 0 )
            handle = figure(figno);
        else
            clf;
        end

        if( propagDist < (10^-6) )
            dtext = sprintf('prop. distance = %5.3f nm', 100000000.0*propagDist);
        elseif( propagDist < (10^-3) )
            dtext = sprintf('prop. distance = %5.3f micron', 1000000.0*propagDist);
        elseif( propagDist < 1.0 )
            dtext = sprintf('prop. distance = %5.3f mm', 1000.0*propagDist);
        else
            dtext = sprintf('prop. distance = %5.3f m', propagDist);
        end

        etext = sprintf('pulse energy (FWHM) = %8.6f J', pulseEnergy);
        text = sprintf('%s, %s', dtext, etext);

        axis([-timeMax timeMax -radialMax radialMax]);
        title(text, 'FontWeight', 'bold');
        xlabel(timeLabel, 'FontWeight', 'bold');
        ylabel(radialLabel, 'FontWeight', 'bold');
        zlabel(verticalLabel, 'FontWeight', 'bold');

        if( perspective == 3 )
            view([90 0]);
        elseif( perspective == 2 )
            view([-50 30]);
        else
            view([50 30]);
        end
    end
end
if( halfview == 1 )
    surfl( plott, plotr, intensity(:,:,figno) );
else
    surfl( plott, fullr, fullSurf(:,:,figno) );
end

shading interp;
colormap( gray );
hold off;

if( animation == 0 )
    refresh( figno );
else
    drawnow;
    refresh( 1 );

    frames(figno) = getframe( handle );
end

if( exportFlag ~= 0 )
    filename = sprintf( 'figure%d.jpg', figno );
    print( handle, '-djpeg', filename );
end
end

%---------------------------------------------------------------------------------
%                              save movie
%---------------------------------------------------------------------------------

if( animation ~= 0 )
    disp( 'saving to avi...' );

    text  = sprintf( 'saving to AVI (%d frames)', Numplots );
    set( handles.status, 'String', text );

    if( perspective == 3 )
        if( animation == 2 )
            movie2avi( frames, 'target.avi', 'FPS', 3, 'COMPRESSION', 'None' );
        else
            movie2avi( frames, 'target.avi', 'FPS', 3, 'COMPRESSION', 'Indeo5', 'QUALITY', 100 );
        end
        disp( '   target.avi saved' );
        set( handles.status, 'String', 'target.avi saved' );
    elseif( perspective == 2 )
        if( animation == 2 )
            movie2avi( frames, 'outgoing.avi', 'FPS', 3, 'COMPRESSION', 'None' );
        else
    elseif( perspective == 2 )
        if( animation == 2 )
            movie2avi( frames, 'outgoing.avi', 'FPS', 3, 'COMPRESSION', 'None' );
        else
movie2avi( frames, 'outgoing.avi', 'FPS', 3, 'COMPRESSION', 'Indeo5', 'QUALITY', 100 );
end

disp( ' outgoing.avi saved' );
set( handles.status, 'String', 'outgoing.avi saved' );

else
if( animation == 2 )
    movie2avi( frames, 'incoming.avi', 'FPS', 3, 'COMPRESSION', 'None' );
else
    movie2avi( frames, 'incoming.avi', 'FPS', 3, 'COMPRESSION', 'Indeo5', 'QUALITY', 100 );
end

disp( ' incoming.avi saved' );
set( handles.status, 'String', 'incoming.avi saved' );
end

set( handles.plot,     'BackgroundColor', 'green' );
set( handles.plot,     'Enable',          'on'    );
set( handles.closeall, 'BackgroundColor', 'green' );
set( handles.closeall, 'Enable',          'on'    );

%==========================================================================
%                  respond to button press (close all)
%==========================================================================

function closeall_Callback( hObject, eventdata, handles )

    % hObject    handle to closeall (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)

    if( get( handles.noanimation, 'Value' ) > 0 )
        % close all;

        Numplots = get( handles.closeall, 'UserData' );

        for figno = 1:Numplots
            handle = figure( figno );

            close( handle );
        end
    else
        handle = figure( 1 );

        close( handle );
    end

set( handles.closeall, 'Enable', 'off' );
B.2.10 Composite profile plotting (highResNLSplot2)

For the composite profile displays, the software uses `highResNLSplot2.m`:

```matlab
function varargout = highResNLSplot2(varargin)
%**************************************************************************
% Laser Beam Propagation through Air
% cummulative profile plotting
14 Sep 2006 version 5.5
for MATLAB 7
%==========================================================================
%
% input files
% -----------
% plott.mat -- time axis grid
% plotr.mat -- radial axis grid
% distances.mat -- propagation distances
% energies.mat -- pulse energies
% intensity.mat -- intensity surfaces
%
%==========================================================================
%
% Control Parameters (inputs)
% ---------------------------
% plotting parameters:
% perspective
% 1 for incoming perspective
% 2 for outgoing perspective
% 3 for target perspective
%
%==========================================================================
%
format long;
% Begin initialization code - DO NOT EDIT

gui_Singleton = 1;

gui_State = struct( 'gui_Name', mfilename, ... 
'gui_Singleton', gui_Singleton, ... 
'gui_OpeningFcn', @highResNLSplot2_OpeningFcn, ... 
'gui_OutputFcn', @highResNLSplot2_OutputFcn, ... 
'gui_LayoutFcn', [], ... 
'gui_Callback', [] );

if( nargin & isstr(varargin{1}) )
```

164
gui_State.gui_Callback = str2func( varargin{1} );
end

if( nargout )
    [varargout{1:nargout}] = gui_mainfcn( gui_State, varargin{:} );
else
    gui_mainfcn( gui_State, varargin{:} );
end

% End initialization code - DO NOT EDIT

%==========================================================================
% executes just before highResNLSplot2 is made visible.
%==========================================================================

function highResNLSplot2_OpeningFcn( hObject, eventdata, handles, varargin )

% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to highResNLSplot (see VARARGIN)
%
% This function has no output args, see OutputFcn.
%
% Choose default command line output for highResNLSplot

handles.output = hObject;

% Update handles structure

guidata( hObject, handles );

% UIWAIT makes highResNLSplot wait for user response (see UIRESUME)
% uiwait(handles.figure1);

plot_Callback( handles );

%==========================================================================
% outputs from this function are returned to the command line.
%
function varargout = highResNLSplot2_OutputFcn( hObject, eventdata, handles )

% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargout  cell array for returning output args (see VARARGOUT);

% Get default command line output from handles structure

varargout{1} = handles.output;
function incoming_Callback(hObject, eventdata, handles)

% hObject    handle to incoming (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of incoming

disp( 'incoming perspective selected' );

set( handles.target, 'Value', 0 );
set( handles.outgoing, 'Value', 0 );
plot_Callback( handles );

function outgoing_Callback(hObject, eventdata, handles)

% hObject    handle to outgoing (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of outgoing

disp( 'outgoing perspective selected' );

set( handles.target, 'Value', 0 );
set( handles.incoming, 'Value', 0 );
plot_Callback( handles );

function target_Callback(hObject, eventdata, handles)

% hObject    handle to target (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of target

disp( 'target perspective selected' );

set( handles.incoming, 'Value', 0 );
set( handles.outgoing, 'Value', 0 );
plot_Callback( handles );
function maxVal_Callback(hObject, eventdata, handles)

% hObject    handle to maxVal (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

disp( 'maximum value mode selected' );
set( handles.integrated, 'Value', 0 );
plot_Callback( handles );


function integrated_Callback(hObject, eventdata, handles)

% hObject    handle to integrated (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

disp( 'integrated mode selected' );
set( handles.maxVal, 'Value', 0 );
plot_Callback( handles );


function plot_Callback(hObject)

% hObject    handle to plot (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

disp( 'plot activated' );

format long;

if( get( handles.target, 'Value' ) > 0 )
    perspective = 3;
elseif( get( handles.outgoing, 'Value' ) > 0 )
    perspective = 2;
end
else
    perspective = 1;
end

if( get( handles.integrated, 'Value' ) > 0 )
    integrated = 1;
else
    integrated = 0;
end

disp( 'plotting parameters:' );
disp( sprintf( '  perspective = %d', perspective ) );
disp( sprintf( '  integration = %d', integrated ) );

% load data

load( 'plott.mat' )     plott;
load( 'plotr.mat' )     plotr;
load( 'distances.mat' ) distances;
load( 'energies.mat' )  energies;
load( 'intensity.mat' ) intensity;

% radial axis

lengths   = size( plotr );
Nrdata    = lengths(1);
radialMax = plotr(Nrdata);
if( plotr(1) == 0.0 )
    Nrplot = 2*Nrdata - 1;
else
    Nrplot = 2*Nrdata;
end

% distance axis

lengths   = size( distances );
Nzplot    = lengths(1);
lastFigure = 0;

for figno = 1:Nzplot
    if( energies(figno) == 0.0 )
        break;
    end
end
lastFigure = figno;
end

distMax = distances(lastFigure);

% % temporal axis

lengths = size(plott);
Ntplot = lengths(1);

% % adjust plotting units

if( radialMax < (10^(-3)) )
plotr = plotr*(10^6);
radialMax = radialMax*(10^6);
radialLabel = 'radius (micron)';
elseif( radialMax < 1.0 )
plotr = plotr*(10^3);
radialMax = radialMax*(10^3);
radialLabel = 'radius (mm)';
else
    radialLabel = 'radius (m)';
end
distLabel = 'distance (m)';

% % mirror r about z axis

lengths = size(intensity);
Numplots = lengths(3);
lastFigure = 0;
for figno = 1:Numplots
    if( energies(figno) == 0.0 )
        break;
    end
    lastFigure = figno;
end

fullr = zeros(Nrplot, 1);
jplot = 1;
for jradial = Nrdata:-1:1
fullr(jplot) = -plotr(jradial);
jplot        = jplot + 1;
end

if( plotr(1) == 0.0 )
  for jradial = 2:Nrdata
    fullr(jplot) = plotr(jradial);
    jplot        = jplot + 1;
  end
else
  for jradial = 1:Nrdata
    fullr(jplot) = plotr(jradial);
    jplot        = jplot + 1;
  end
end

fullSurf = NaN( Nrplot, Nzplot );

if( integrated > 0 )
  maxval = 0.0;

  values = zeros( Ntplot, 1 );
  deltat = plott(2) - plott(1);

  for figno = 1:lastFigure
    jplot = 1;

    for jradial = Nrdata:-1:1
      values                = intensity(jradial,:,figno);
      fullSurf(jplot,figno) = deltat*trapz( values );
      if( fullSurf(jplot,figno) > maxval )
        maxval = fullSurf(jplot,figno);
      end
      jplot                 = jplot + 1;
    end

    if( plotr(1) == 0.0 )
      for jradial = 2:Nrdata
        values                = intensity(jradial,:,figno);
        fullSurf(jplot,figno) = deltat*trapz( values );
        jplot                 = jplot + 1;
      end
    else
      for jradial = 1:Nrdata
        values                = intensity(jradial,:,figno);
      end
    end
  end
end
\[ \text{fullSurf}(j\text{plot},f\text{igno}) = \text{deltat} \cdot \text{trapz} (\text{values}); \]
\[ j\text{plot} = j\text{plot} + 1; \]
\end{verbatim}

\begin{verbatim}
normalizer = 1.0/\text{maxval};

\text{for figno = 1:} \text{lastFigure} \]
\[ \text{for jradial = 1:Nr\text{plot} } \]
\[ \quad \text{fullSurf}(j\text{radial},f\text{igno}) = \text{fullSurf}(j\text{radial},f\text{igno}) \ast \text{normalizer}; \]
\[ \text{end} \]
\[ \text{end} \]
\text{else} \]
\[ \text{for figno = 1:} \text{lastFigure} \]
\[ \quad j\text{plot} = 1; \]
\[ \text{for jradial = Nr\text{data}:-1:1} \]
\[ \quad \text{plotted} = 0.0; \]
\[ \text{for iplot = 1:Nt\text{plot} } \]
\[ \quad \text{value} = \text{intensity}(j\text{radial},i\text{plot},f\text{igno}); \]
\[ \quad \text{if} (\text{value} > \text{plotted}) \]
\[ \quad \quad \text{plotted} = \text{value}; \]
\[ \quad \text{end} \]
\[ \text{end} \]
\[ \text{fullSurf}(j\text{plot},f\text{igno}) = \text{plotted}; \]
\[ j\text{plot} = j\text{plot} + 1; \]
\[ \text{end} \]
\text{if (plotr(1) == 0.0)} \]
\[ \text{for jradial = 2:Nr\text{data} } \]
\[ \quad \text{plotted} = 0.0; \]
\[ \text{for iplot = 1:Nt\text{plot} } \]
\[ \quad \text{value} = \text{intensity}(j\text{radial},i\text{plot},f\text{igno}); \]
\[ \quad \text{if} (\text{value} > \text{plotted}) \]
\[ \quad \quad \text{plotted} = \text{value}; \]
\[ \quad \text{end} \]
\[ \text{end} \]
\[ \text{fullSurf}(j\text{plot},f\text{igno}) = \text{plotted}; \]
\[ j\text{plot} = j\text{plot} + 1; \]
\[ \text{end} \]
\text{else} \]
\[ \text{for jradial = 1:Nr\text{data} } \]
\[ \quad \text{plotted} = 0.0; \]
for iplot = 1:Ntplot
    value = intensity(jradial,iplot,figno);
    if( value > plotted )
        plotted = value;
    end
end

fullSurf(jplot,figno) = plotted;

jplot = jplot + 1;
end
end
end

disp( 'plotting...' );
figure(1);
clf;
axis( [0.0 distMax -radialMax radialMax] );
if( integrated > 0 )
    title( 'normalized (time) integrated propagated pulse', 'FontWeight', 'bold' );
    if(( Ntplot == 128 ) || ( Ntplot == 256 ))
        verticalLabel = 'integrated rel. intensity (4x4 Avg.)';
    else
        verticalLabel = 'integrated rel. intensity';
    end
else
    title( 'maxima of propagated pulse', 'FontWeight', 'bold' );
    if(( Ntplot == 128 ) || ( Ntplot == 256 ))
        verticalLabel = 'maximum rel. intensity (4x4 Avg.)';
    else
        verticalLabel = 'maximum rel. intensity';
    end
end
xlabel( distLabel, 'FontWeight', 'bold' );
ylabel( radialLabel, 'FontWeight', 'bold' );
zlabel( verticalLabel, 'FontWeight', 'bold' );
if( perspective == 3 )
    view( [90 0] );
elseif( perspective == 2 )
view([-50 30]);
else
  view([50 30]);
end
pause(0.1);
hold on;
grid on;
surfl(distances,fullr,fullSurf);
shading interp;
colormap(gray);
hold off;
refresh(1);

%**************************************************************************
%                                  END
%**************************************************************************

%**************************************************************************
%                                  END
%**************************************************************************
B.2.11 Target-plane animation plotting (highResNLSpot3)

For animations involving target plane energy patterns, the software uses `highResNLSpot3.m`:

```matlab
function varargout = highResNLSpot3(varargin)
%**************************************************************************
% Laser Beam Propagation through Air
% target plane animation
% 14 Sep 2006 version 5.5
% for
% MATLAB 7

%==========================================================================
% input files
% -----------
% plott.mat -- time axis grid
% plotr.mat -- radial axis grid
% distances.mat -- propagation distances
% energies.mat -- pulse energies
% intensity.mat -- intensity surfaces

%==========================================================================
% Control Parameters (inputs)
% ---------------------------
% plotting parameters:
% mode
% 1 for maximum values
% 2 for integrated values
% animation option
% 0 = off (multiple figures displayed)
% 1 = on with compression (low-quality AVI file generated)
% (single figure updated)
% 2 = on with no compression (high-quality AVI file generated)
% (single figure updated)

%==========================================================================
% main program
%==========================================================================

format long;
% Begin initialization code - DO NOT EDIT

gui_Singleton = 1;

gui_State = struct('gui_Name', mfilename, ...
    'gui_Singleton', gui_Singleton, ...
    'gui_OpeningFcn', @highResNLSpot3_OpeningFcn, ...
```

174
if( nargin & isstr(varargin{1}) )
  gui_State.gui_Callback = str2func( varargin{1} ) ;
end

if( nargout )
  [varargout{1:nargout}] = gui_mainfcn( gui_State, varargin{:} ) ;
else
  gui_mainfcn( gui_State, varargin{:} ) ;
end

% End initialization code - DO NOT EDIT

%==========================================================================
%                                                                                                                                 
%         executes just before highResNlSplot is made visible. 
%                                                                                                                                 
%function highResNLSplot3_OpeningFcn( hObject, eventdata, handles, varargin )
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to highResNlSplot (see VARARGIN)
%
% This function has no output args, see OutputFcn.
%
% Choose default command line output for highResNlSplot
handles.output = hObject;

% Update handles structure
guidata( hObject, handles );

% UIWAIT makes highResNlSplot wait for user response (see UIRESUME)
% uiwait(handles.figure1);

%==========================================================================
%                                                                                                                                 
% outputs from this function are returned to the command line. 
%                                                                                                                                 
%function varargout = highResNLSplot3_OutputFcn( hObject, eventdata, handles )
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
%
% varargout cell array for returning output args (see VARARGOUT);
% Get default command line output from handles structure
varargout{1} = handles.output;

%==========================================================================
%                        maximum (time) value mode
%==========================================================================

function maxVal_Callback(hObject, eventdata, handles)

% hObject    handle to maxVal (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

disp('maximum value mode selected');
set(handles.integrated,'Value',0);

%==========================================================================
%                           integrated mode
%==========================================================================

function integrated_Callback(hObject, eventdata, handles)

% hObject    handle to integrated (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

disp('integrated mode selected');
set(handles.maxVal,'Value',0);

%==========================================================================
%                          uncompressed animation
%==========================================================================

function uncompressed_Callback(hObject, eventdata, handles)

% hObject    handle to uncompressed (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of uncompressed

disp('uncompressed selected');
set(handles.compressed,'Value',0);
set(handles.noanimation,'Value',0);

%==========================================================================
%                           compressed animation
%==========================================================================

function compressed_Callback(hObject, eventdata, handles)

% hObject    handle to compressed (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB

% handles       structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of compressed

disp( 'compressed selected' );

set( handles.uncompressed, 'Value', 0 );
set( handles.noanimation, 'Value', 0 );

%==========================================================================
%                      no animation (multiple figures)
%==========================================================================

function noanimation_Callback( hObject, eventdata, handles )
    % hObject    handle to noanimation (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
    % Hint: get(hObject,'Value') returns toggle state of noanimation

disp( 'noanimation selected' );

set( handles.uncompressed, 'Value', 0 );
set( handles.compressed,   'Value', 0 );

%==========================================================================
%                            draw in 2-D view
%==========================================================================

function draw2D_Callback( hObject, eventdata, handles )
    % hObject    handle to draw2D (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)

set( handles.draw3D, 'Value', 0 );

%==========================================================================
%                            draw in 3-D view
%==========================================================================

function draw3D_Callback( hObject, eventdata, handles )
    % hObject    handle to draw3D (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)

set( handles.draw2D, 'Value', 0 );

%==========================================================================
%              respond to button press (activate plotting)
%==========================================================================

function plot_Callback( hObject, eventdata, handles )
```matlab
% hObject handle to plot (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

set( handles.closeall, 'BackgroundColor', 'red' );
set( handles.closeall, 'Enable', 'off' );
set( handles.plot, 'BackgroundColor', 'red' );
set( handles.plot, 'Enable', 'off' );

disp( 'plot activated' );

format long;

%---------------------------
%
% parameters for plotting
%

if( get( handles.integrated, 'Value' ) > 0 )
    integrated = 1;
else
    integrated = 0;
end

if( get( handles.uncompressed, 'Value' ) > 0 )
    animation = 2;
elseif( get( handles.compressed, 'Value' ) > 0 )
    animation = 1;
else
    animation = 0;
end

disp( 'plotting parameters:' );
disp( sprintf( '  integration = %d', integrated ) );
disp( sprintf( '  animation   = %d', animation ) );

%--------------------------------------------------------------------------
%
%                               load data
%--------------------------------------------------------------------------

disp( 'loading output data...' );

load 'plott.mat'    plott;
load 'plotr.mat'    plotr;
load 'distances.mat' distances;
load 'energies.mat'  energies;
load 'intensity.mat' intensity;

% % theta axis
%
Ntheta    = 361;
%
% radial axis
```
% lengths   = size( plotr );
Nrdata    = lengths(1);
Nrhalf    = floor( Nrdata/3 );
radialMax = plotr(Nrhalf);
Nrplot    = 2*Nrhalf;
Nraxial   = Nrhalf + 1;
%
% distance axis
%
lengths   = size( distances );
Nzplot    = lengths(1);
%
% temporal axis
%
lengths   = size( plott );
Ntplot    = lengths(1);
%
% adjust plotting units
%
if( radialMax < (10^(-3)) )
    plotr       =     plotr*(10^6);
    radialMax   = radialMax*(10^6);
    radialLabel = 'radius (micron)';
elseif( radialMax < 1.0 )
    plotr       =     plotr*(10^3);
    radialMax   = radialMax*(10^3);
    radialLabel = 'radius (mm)';
else
    radialLabel = 'radius (m)';
end
distLabel = 'distance (m)';

%---------------------------
% find last figure
%
lengths   = size( intensity );
Numplots  = lengths(3);
lastFigure = 0;
for figno = 1:Numplots  
    if( energies(figno) == 0.0 )
        end
    lastFigure = figno;
end

set( handles.closeall, 'UserData', Numplots );

%---------------------------
%  radial axis (mirrored about z axis)
%
fullr = zeros( Nrplot, 1 );
jplot = 1;
for jradial = Nrhalf:-1:1 
    fullr(jplot) = -plotr(jradial);
    jplot = jplot + 1;
end
for jradial = 1:Nrhalf 
    fullr(jplot) = plotr(jradial);
    jplot = jplot + 1;
end

%---------------------------
%  surface of revolution
%
spun = NaN( Ntheta, Nraxial, Nzplot );

for itheta = 1:Ntheta 
    xx(itheta,1) = 0.0;
    yy(itheta,1) = 0.0;
end

j1 = 1;
for j2 = 2:Nraxial 
    rvalue = plotr(j1);
    for itheta = 1:Ntheta
        theta = itheta*pi/180.0;
\[ xx(\text{itheta}, j2) = \text{rvalue} \times \cos(\text{theta}); \]
\[ yy(\text{itheta}, j2) = \text{rvalue} \times \sin(\text{theta}); \]
end

\[ j1 = j2; \]
end

\% case A: integrated
\%

if( \text{integrated} > 0 )
    zMax = 1.0;
    verticalLabel = 'norm. integrated intensity';
    maxval = 0.0;
    values = zeros( \text{Ntplot}, 1 );
    deltat = \text{plott}(2) - \text{plott}(1);
    for \text{figno} = 1: \text{lastFigure}
        j1 = 1;
        for j2 = 2: \text{Nraxial}
            values = \text{intensity}(j1,:,\text{figno});
            axval = deltat*\text{trapz}(values);
            if( axval > maxval )
                maxval = axval;
            end
            for \text{itheta} = 1: \text{Ntheta}
                spun(\text{itheta}, j2, \text{figno}) = axval;
            end
        end
        j1 = j2;
    end
    deltar = \text{plotr}(2) - \text{plotr}(1);
    deltas = spun(1,3,\text{figno}) - spun(1,2,\text{figno});
    axval = spun(2,2,\text{figno}) - (\text{plotr}(1) \times \text{deltas/deltar});
    if( axval > maxval )
        maxval = axval;
    end
    for \text{itheta} = 1: \text{Ntheta}
        spun(\text{itheta},1,\text{figno}) = axval;
    end
end
normalizer = 1.0/maxval;
for figno = 1:lastFigure
    for j1 = 1:Nraxial
        for itheta = 1:Ntheta
            spun(itheta,j1,figno) = normalizer*spun(itheta,j1,figno);
        end
    end
end

% case B: maxima

else
    zMax = 1.0;

    if(( Ntplot == 128 ) || ( Ntplot == 256 ))
        verticalLabel = 'maximum rel. intensity (4x4 Avg.)';
    else
        verticalLabel = 'maximum rel. intensity';
    end

    for figno = 1:lastFigure
        j1 = 1;

        for j2 = 2:Nraxial
            plotted = 0.0;

            for iplot = 1:Ntplot
                value = intensity(j1,iplot,figno);

                if( value > plotted )
                    plotted = value;
                end
            end

            for itheta = 1:Ntheta
                spun(itheta,j2,figno) = plotted;
            end

            j1 = j2;
        end
    end

    deltar = plotr(2)        - plotr(1);
    deltas = spun(1,3,figno) - spun(1,2,figno);
    axval = spun(2,2,figno) - (plotr(1)*deltas/deltar);

    for itheta = 1:Ntheta
        spun(itheta,1,figno) = axval;
    end
end

%------------------------------------------------------------
% color map
map = zeros(25,3);

map(01,1) = 0.0000; map(01,2) = 0.0000; map(01,3) = 0.0000;
map(02,1) = 0.0400; map(02,2) = 0.0000; map(02,3) = 0.0077;
map(03,1) = 0.0600; map(03,2) = 0.0000; map(03,3) = 0.0192;
map(04,1) = 0.0800; map(04,2) = 0.0000; map(04,3) = 0.0307;
map(05,1) = 0.1000; map(05,2) = 0.0000; map(05,3) = 0.0499;
map(06,1) = 0.1200; map(06,2) = 0.0000; map(06,3) = 0.0691;
map(07,1) = 0.1400; map(07,2) = 0.0000; map(07,3) = 0.0960;
map(08,1) = 0.1600; map(08,2) = 0.0000; map(08,3) = 0.1229;
map(09,1) = 0.1800; map(09,2) = 0.0000; map(09,3) = 0.1575;
map(10,1) = 0.2000; map(10,2) = 0.0000; map(10,3) = 0.1920;
map(11,1) = 0.2018; map(11,2) = 0.0000; map(11,3) = 0.2160;
map(12,1) = 0.2035; map(12,2) = 0.0000; map(12,3) = 0.2400;
map(13,1) = 0.1936; map(13,2) = 0.0000; map(13,3) = 0.2600;
map(14,1) = 0.1837; map(14,2) = 0.0000; map(14,3) = 0.2800;
map(15,1) = 0.1661; map(15,2) = 0.0000; map(15,3) = 0.3000;
map(16,1) = 0.1485; map(16,2) = 0.0000; map(16,3) = 0.3200;
map(17,1) = 0.1232; map(17,2) = 0.0000; map(17,3) = 0.3400;
map(18,1) = 0.0979; map(18,2) = 0.0000; map(18,3) = 0.3600;
map(19,1) = 0.0650; map(19,2) = 0.0000; map(19,3) = 0.3800;
map(20,1) = 0.0320; map(20,2) = 0.0000; map(20,3) = 0.4000;
map(21,1) = 0.0160; map(21,2) = 0.0296; map(21,3) = 0.4200;
map(22,1) = 0.0000; map(22,2) = 0.0493; map(22,3) = 0.4400;
map(23,1) = 0.0000; map(23,2) = 0.0976; map(23,3) = 0.4600;
map(24,1) = 0.0000; map(24,2) = 0.1459; map(24,3) = 0.4800;
map(25,1) = 0.0000; map(25,2) = 0.2019; map(25,3) = 0.5000;
map(26,1) = 0.0000; map(26,2) = 0.2579; map(26,3) = 0.5200;
map(27,1) = 0.0000; map(27,2) = 0.3216; map(27,3) = 0.5400;
map(28,1) = 0.0000; map(28,2) = 0.3853; map(28,3) = 0.5600;
map(29,1) = 0.0000; map(29,2) = 0.4567; map(29,3) = 0.5800;
map(30,1) = 0.0000; map(30,2) = 0.5280; map(30,3) = 0.6000;
map(31,1) = 0.0000; map(31,2) = 0.5840; map(31,3) = 0.5970;
map(32,1) = 0.0000; map(32,2) = 0.6400; map(32,3) = 0.5939;
map(33,1) = 0.0000; map(33,2) = 0.6600; map(33,3) = 0.5472;
map(34,1) = 0.0000; map(34,2) = 0.6800; map(34,3) = 0.5005;
map(35,1) = 0.0000; map(35,2) = 0.7000; map(35,3) = 0.4461;
map(36,1) = 0.0000; map(36,2) = 0.7200; map(36,3) = 0.3917;
map(37,1) = 0.0000; map(37,2) = 0.7400; map(37,3) = 0.3296;
map(38,1) = 0.0000; map(38,2) = 0.7600; map(38,3) = 0.2675;
map(39,1) = 0.0000; map(39,2) = 0.7800; map(39,3) = 0.1978;
map(40,1) = 0.0000; map(40,2) = 0.8000; map(40,3) = 0.1280;
map(41,1) = 0.0134; map(41,2) = 0.8200; map(41,3) = 0.0640;
map(42,1) = 0.0269; map(42,2) = 0.8400; map(42,3) = 0.0000;
map(43,1) = 0.1120; map(43,2) = 0.8600; map(43,3) = 0.0000;
map(44,1) = 0.1971; map(44,2) = 0.8800; map(44,3) = 0.0000;
map(45,1) = 0.2899; map(45,2) = 0.9000; map(45,3) = 0.0000;
map(46,1) = 0.3827; map(46,2) = 0.9200; map(46,3) = 0.0000;
map(47,1) = 0.4832; map(47,2) = 0.9400; map(47,3) = 0.0000;
map(48,1) = 0.5837; map(48,2) = 0.9600; map(48,3) = 0.0000;
map(49,1) = 0.6919; map(49,2) = 0.9800; map(49,3) = 0.0000;
map(50,1) = 0.8000; map(50,2) = 1.0000; map(50,3) = 0.0000;
% output initial pulse
%--------------------------------------------------------------------------

disp('plotting...');

propagDist = 0.0;
figno = 1;

handle = figure(1);

pulseEnergy = energies(figno);

etext = sprintf('pulse energy (FWHM) = %8.6f J', pulseEnergy);

text = sprintf('prop. distance = 0.0, %s', etext);

axis([-radialMax radialMax -radialMax radialMax 0.0 zMax]);

caxis([0.0 zMax]);

title(text, 'FontWeight', 'bold');

xlabel(radialLabel, 'FontWeight', 'bold');

ylabel(radialLabel, 'FontWeight', 'bold');

zlabel(verticalLabel, 'FontWeight', 'bold');

if( get(handles.draw3D, 'Value') > 0 )
    view([45 70]);
else
    view([90 90]);
end

hold on;

surf(xx, yy, spun(:,:,figno));

shading interp;

colormap(map);

colorbar;

if( get(handles.draw3D, 'Value') > 0 )
    grid on;
end

hold off;

refresh(1);

if( animation ~= 0 )
    frames(figno) = getframe(handle);
end

%--------------------------------------------------------------------------
% propagation loop
%--------------------------------------------------------------------------
for figno = 2:Numplots
    propagDist = distances(figno);
    text = sprintf( 'frame %d of %d', figno, Numplots );
    set( handles.status, 'String', text );
    if( propagDist > 0 )
        pulseEnergy = energies(figno);
        if( animation == 0 )
            handle = figure( figno );
        else
            clf;
        end
        if( propagDist < (10^-6) )
            dtext = sprintf( 'prop. distance = %5.3f nm', 100000000.0*propagDist);
        elseif( propagDist < (10^-3) )
            dtext = sprintf( 'prop. distance = %5.3f micron', 1000000.0*propagDist);
        elseif( propagDist < 1.0 )
            dtext = sprintf( 'prop. distance = %5.3f mm', 1000.0*propagDist);
        else
            dtext = sprintf( 'prop. distance = %5.3f m', propagDist);
        end
        etext = sprintf( 'pulse energy (FWHM) = %8.6f J', pulseEnergy );
        text = sprintf( '%s, %s', dtext, etext );
        axis( [radialMax radialMax -radialMax radialMax 0.0 zMax] );
        caxis( [0.0 zMax] );
        title( text, 'FontWeight', 'bold' );
        xlabel( radialLabel, 'FontWeight', 'bold' );
        ylabel( radialLabel, 'FontWeight', 'bold' );
        zlabel( verticalLabel, 'FontWeight', 'bold' );
        if( get( handles.draw3D, 'Value' ) > 0 )
            view( [45 70] );
        else
            view( [90 90] );
        end
        hold on;
        surf( xx, yy, spun(:,:,figno) );
        shading interp;
end
 colormap( map );
 colorbar;

 if( get( handles.draw3D, 'Value' ) > 0 )
   grid on;
 end

 hold off;

 if( animation == 0 )
   refresh( figno );
 else
   drawnow;
   refresh( 1 );

   frames(figno) = getframe( handle );
 end
 end

%--------------------------------------------------------------------------
%                              save movie
%--------------------------------------------------------------------------

 if( animation ~= 0 )
   disp( 'saving to avi...' );

   text  = sprintf( 'saving to AVI (%d frames)', Numplots );

   set( handles.status, 'String', text );

   if( animation == 2 )
     movie2avi( frames, 'spot.avi', 'FPS', 3, 'COMPRESSION', 'None' );
   else
     movie2avi( frames, 'spot.avi', 'FPS', 3, 'COMPRESSION', 'Indeo5',
                  'QUALITY', 100 );
   end

   disp( '   spot.avi saved' );

   set( handles.status, 'String', 'spot.avi saved' );
 end

 set( handles.plot, 'BackgroundColor', 'green' );
 set( handles.plot, 'Enable', 'on' );
 set( handles.closeall, 'BackgroundColor', 'green' );
 set( handles.closeall, 'Enable', 'on' );

%==========================================================================
%                  respond to button press (close all)
%==========================================================================

 function closeall_Callback( hObject, eventdata, handles )
if( get( handles.noanimation, 'Value' ) > 0 )
    Numplots = get( handles.closeall, 'UserData' );

    for figno = 1:Numplots
        handle = figure( figno );
        close( handle );
    end
else
    handle = figure( 1 );
    close( handle );
end

set( handles.closeall, 'Enable', 'off' );
REFERENCES


