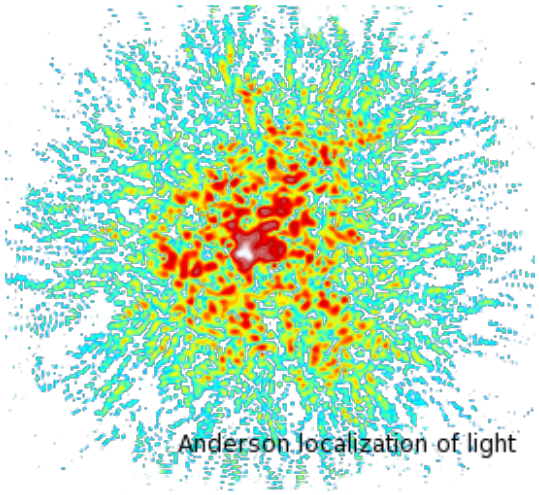
gUPPElab Guide

This document is a guide to the installation and basic usage of gUPPEcore and the associated gUPPElab package. The first part is meant to serve as aid in compiling the software, and in adding user-defined modifications to simulator, especially for users not familiar with work in Linux-like systems. The main part guides the user through several worked-out examples of basic simulator applications.



College of Optical Sciences
University of Arizona

Contents

1 gUPPEcore Setup

1.1 Obtaining gUPPEcore

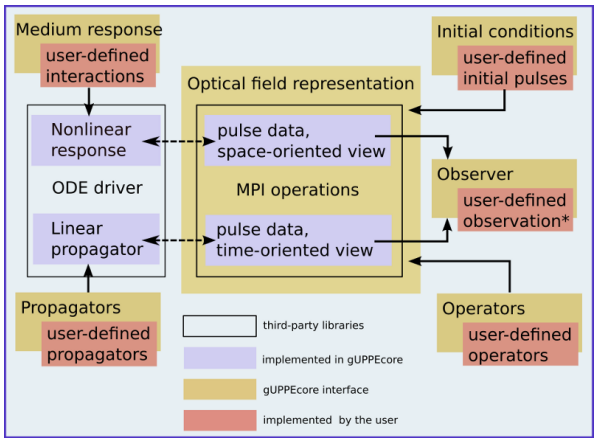
The gUPPEcore archive can be obtained from the gUPPEcore lab download page. The gUPPEcore download is a compressed archive of the source code, tutorials and other files needed to use gUPPEcore. Extract the archive once the download is complete. In the root of the extracted archive there is a README file that describes the content found in the */src* and */templates* directories. The next few sections of this document will describe the compilation and the basic usage of gUUPEcore. Once those sections have been read and compilation is complete it is highly recommended that the new user work their way though the examples in the templates folder. The examples are selected as to illustrate the usage of the simulator from the very basic toward more advanced.

1.2 gUPPEcore and gUPPElab components

This simulation software is meant to be customized by the user. As the name suggests, core capabilities that define the generalized Unidirectional Pulse Propagation Equations are included in the library that comes with the software. This core has several "interfaces" that allow users to define various modules, including initial conditions, medium linear and nonlinear properties, and operators representing apertures, filters, lenses, axicons, etc. Experienced users can also define their own transverse geometry and corresponding linear propagators (e.g. based on Beam-Propagation Methods). These modules, here called user-defined add-ons, are invoked by the core during the simulation. Thus, gUPPEcore is more a framework than a self-contained simulation engine.

Components needed to create a customized simulator executable:

- gUPPEcore library
  ... implements the underlying algorithms. It comes pre-compiled for Linux and OSX operating systems.
- MPI utilities
  ... wrap/hide the parallelization functions. This component is compiled by the user and linked against the locally installed MPI implementation.
- User defined add-ons
  ... express user's idea of the physics to be captured by the simulation. They require simple code writing, and must be compiled before linking all three components into a customized executable.
- Simulation "Lab"
  ... consists of a series of simulation templates that serve two purposes. First, they illustrate the usage of the software, and second, each can be used as a point of departure for a "real" simulation project.



This figure gives a brief overview of how the different components of gUPPEcore feed into each-other. This manual is intended to instruct the user on the gUUPEcore interface and user-defined extensions.

1.3 Prerequisites

SUMMARY:

Mandatory:	Linux or OSX, MPI implementation (e.g. OpenMPI) & g++ compiler
Optional:	Intel icpc compiler, Grace (xmgrace) & Python with numpy & matplotlib for visualization.

gUPPEcore must be compiled before it can be used. Prior to compilation the Linux or OSX system must have an MPI implementation (such as the freely available OpenMPI) and either the GCC g++ or Intel icpc compiler. The Intel compiler will generate more efficient binaries but is not generally freely available outside of an academic setting (see Intel Education Offerings).

Grace and Python are additional (optional) prerequisites used for the visualization of gUPPEcore data. The scripts in the examples/templates folders uses both Grace and Python to quickly plot data.

1.4 Compilation

SUMMARY:

1: Edit /src/loci file
- Specify compiler & set flags if necessary
- Specify mpiCC
- Set WRKDIR_LOC

2: run make in /src directory

The compilation process creates executables for gUPPEcore, mpiutils and any user-created add-ons. The primary executable one will interact with is compiled to *.../bin/guppeCore.out*. As the name suggests, this program will only implement the core capabilities and will contain no customization. It is still a powerful optical pulse simulator, and a large part of the work in the examples folder can be done with it.

Additional executables may need to be compiled for user add-ons. Instructions on how to use the primary and add-on, or customized executables are available in the tutorials located in the */templates* folder and in sections 2.3 & 2.5 of this document. In the */src* folder there is a COMPILATION file that gives a summary of the instructions for the compilation process. That information is also presented here:

Compilation of gUPPEcore assumes the mandatory prerequisites are installed. Prior to compilation the file */src/loci* must be edited. The lines containing the 4 variables CC, MPICC, FLAG & WRKDIR_LOC should be modified to hold the directory paths pointing to the c++ compiler, the MPI c++ compiler, compilations flags passed to the compiler, and the working directory in which the gUPPElab resides, respectively. Example loci files are located in the */src* directory.

The CC variable is expecting the path and executable to be used for C++ compilation. If the compiler you wish to use is visible from the PATH environmental variable then only the executable will need to be specified. See man pages for 'which', 'locate' or 'find' to find the path to the executable. Compatible compilers for gUPPEcore are the GCC g++ and Intel icpc compilers. The Intel icpc compiler is recommended if it is available since in our experience the resulting binary performance is significantly better.

The FLAG variable content will be passed to the compiler specified in the CC variable. If you are using the g++ compiler then the last 2 options (-no-intel-extensions & -static-intel) will need to be commented out.

The MPICC variable is expecting the path and executable to the mpiCC compiler. The location of mpiCC can also be found using 'which', 'locate' or 'find'. Often one has more than one MPI libraries installed on a computer. This is where one specifies which is to be used to compile MPI utilities that come with this software. Note that mpiCC is case sensitive: mpiCC is the MPI compiler for C++ code. mpicc is the compiler for C programs. gUPPEcore requires the C++ compiler.

Finally, the root path of the extracted gUPPElab data will need to be written to the the WRKDIR_LOC variable. For example: *WRKDIR_LOC = /home/username/Downloads/gUPPElab_0.2*. If you have spaces in the directory names enclose the path with quotes or escape properly. The next 3 lines of the loci file use this path to get the locations of the other directories needed during compilation. There is no need to change these unless you have intentionally moved the folders they point to.

Once the initial loci edits have been saved you are ready to compile the gUUPEcore. Simply run the command 'make' from a terminal in the */src* directory. The compilation process will perform 3 compilations:

1. It will compile mpiutils (which contain calls to MPI parallelization functions) with your MPI installation

2. Then it will compile folder named "addons" and place the executable in *WRKDIR_LOC/bin*. This executable is the core, and has no user-defined add-ons. This is used in most of the worked-out examples. The reason we still need to compile (what are in fact "empty" functions) is that the core requires to have a list of add-ons even if the list is empty in this case.

3. Finally it will visit all remaining folders addons... and compile executables within. These are user-add-on examples and templates. For each, there is a corresponding templates/wrk_04X... folder which contains the corresponding source and a simulation example.

2 gUPPEcore Usage

2.1 Input Files

gUPPEcore utilizes input files to initialize simulation runs. The input file consists of several hierarchically organized sections. These sections are populated with parameters that define the simulation. Parameters used will vary between input files base on the simulation being run. The following sections/subsections, however, are mandatory in any input file, even if not used:

gUPPE_Parameters: All input files start with this line. This lets gUPPEcore know that it is to expect all mandatory parameters below.

ODE_Driver: Lets ODE solver know how to behave. Specifies output file base-name, ODE method, ODE tolerances and propagation distance parameters.

UPPE_observer: The UPPE observer is the simulation output controller. Options are set to control the type an frequency of output report.

gUPPE_Domain_Parameters: This section defines the spatial & temporal "box" (and associated properties) that the simulation exists in.

Domain_Temporal: Mandatory subsection of gUPPE_Domain_Parameters containing temporal extent and properties.

Domain_Spatial: Mandatory subsection of gUPPE_Domain_Parameters containing spatial extent and properties. May be 0, 1, 2 or 3 dimensional.

Initial_conditions: A variable length vector of objects that define the initial simulation conditions. For example, initial Gaussian beam definition.

Medium_response: A variable length vector of objects that define the simulated nonlinear medium response. For example, Kerr and Plasma responses.

UPPE_Operator_Prolog: Operators applied to initial field, as defined in the Initial_conditions section, prior to simulation run.

UPPE_Operator_Epilog: Operators applied to propagated field (after simulation).

The structure of the input file is intentionally rigid. The upside is that users can quickly copy-and-modify previously used input files, and also compare (diff) different inputs and thus easily detect their differences. This makes the work with inputs easier. The downside of this is that inputs may contain parameters that are not used in the given simulation. See */templates/000_Inputs_Explained* or appendix B for an in-depth explanation of the input file format.

2.2 Output Files

gUPPEcore produces output files that contain either information about the simulation or the data results of the simulation. Output files that contain information about the simulation can be used to repeat or continue the simulation. Files that contain the results of the simulation can be analyzed (through plotting or further data processing) to examine the results of the simulation.

Output files are either specified in the mpirun command (Section 2.3) or in the input file UPPE_observer section (Appendix B.2). Output files that are specified in the UPPE_observer section are referred to as reports. There are 4 levels of reports. Each level of report is increasingly more computationally costly to produce. Level 1 reports are global variables that do not take much preparation to write. Level 2 reports are 1D reports for a quick overview of the simulation. Level 3 reports are 2D reports that are a little more computationally intensive to generate. Level 4 reports are essentially the entire field an are very computationally expensive to generate.

If a report is generated and how often it is generated is also specified in the UPPE_observer section of the input file. Each report has a *Report#_period* distance value assigned that instructs gUUPE to write that report once the propagation distance has been reached. For example, the Report2 Spatial Profile in Appendix B.2 will be prepared and written to disk after every centimeter of propagation. See */templates/000_Outputs_Explained* or appendix C for an in-depth explanation of the various reports.

2.3 Running gUPPEcore

gUPPEcore is initiated by passing */bin/guppeCore.out* and an input file to mpirun. The running program writes into standard output, and this is usually directed to (or split by tee) a record file of the run. For example, a run using 4 parallel processes started from a */template* folder containing file *inputA* can be executed like this:

```
$ mpirun -n 4 ../../bin/guppeCore.out inputA | tee recA
```

The -n 4 tells mpirun to use 4 CPUs (or rather processor cores). The relative path to guppeCore.out is needed to let mpirun know which of the executables to run (naturally, customized executables are run in the same way).

inputA is assumed to reside in the working directory, a path can be given if it does not. The standard output of the simulation is piped into a tee which splits and directs the output onto screen and into a file called recA. recA will be written in the working directory unless an alternate path is specified.

The record file recA is complete simulation record that can be used to recreate the simulation. Specifically, the top of the record can be used as an input for an identical simulation. gUPPEcore additionally produces output files that contain the results of the simulation (see section C for details).

Longer simulation runs are better executed without the standard output on the screen:

```
$ mpirun -n 4 ../../bin/guppeCore.out inputA > recA
```

2.4 Scripts

The examples in the */templates* directory can be initiated through a pre-written script. These scripts are usually named *runthis*. They are provided to show a new user the order of action taken to execute a given example. Of course the sripts can be used, but they may need to be edited prior to execution. Most of the scripts have 'MYmpirun' and 'MYmpiplot' variables defined. MYmpirun is simply the path to the 'mpirun' that you want to use on your system. This is similar to specifying mpiCC in section 1.4.

The worked-out example scripts will sometimes call a plotting program to visualize results. We use Grace (Linux), QtGrace (OSX) or our own Python script for this. Of course, users need not use the scripts, they are not part of gUPPEcore/lab and are only included for convenience. But if desired, the script should be edited to call an appropriate plotting program.

MYmpiplot is the path to your xmgrace (a.k.a. Grace, QtGrace) executable. If you are not using xmgrace then you will need to comment out lines that refer to Mympiplot.

Similarly, Mypython is the path to the python executable you wish to use. The python executable specified must have access to the numpy & matplotlib modules to generate plots.

The other item to edit is the number of threads available for mpirun to use. This is set with the -n option (ex: -n 4). Nominally this option should be set to match the number of cores on the computers processor (see Linux command nproc). There may be other situations or scenarios where this is not the case. One scenario that comes up in the template examples is the case where there are not enough data points to spread out to multiple processors (or threads). In this case -n will be limited to 1. It may take some experimentation to determine the optimal -n value for your configuration.

Other than setting these variables the scripts are fairly self-explanatory. The scripts usually run guppeCore.out, optionally process the data and then produce some sort of plot to visualize the data.

2.5 User-defined Addons

The core capabilities of gUUPEcore target simulations in the field of optical filamentation. However, it is applicable to many other general situations, such as nonlinear optics in waveguides. The most typical filamentaion scenarios do not require any modifications, and the simulator can be utilized "as is." However, gUPPEcore is not meant to be an "application," or an all-encomapasing simulation engine for femtosecond pulse propagation. Instead, it is left to the user to define specific features of his/her simulation, such as initial conditions and nonlinear medium response models. Yet, these modifications, or user-addons as we call them, should not require extensive coding experience, as gUPPEcore provides an easy to use interface to do this. The figure in section 1.2 shows gUUPEcore scheme and where user-defined addons fit into the scheme. Note that user-defined observables is not yet implemented. As of version 0.2 users can define custom addons for initial conditions, propagators, material responses and pre/post propagation operators.

The user will be required to edit text files that contain C++ code, but knowledge of C++ is not required to edit these files. The user will mainly be defining variables and equations to be used. This will need to be done in a specific format but shouldn't be complicated once the process is understood.There are worked out examples in the templates folder that show the implementation of each type of addon. What follows here is a general overview of the user-defined addon implementation process.

There are 4 basic steps in defining a user-addon.

1. Define name for the addon in *user_addons.cc* in the appropriate section.
2. Create an associated header (*.h) file that contains constants, variables & equations to be used.
3. Edit a Makefile and then compile the addon from the code files above.
4. Call the addon in the script that runs the simulation.

The following sections will go through each of these steps for a simple linear axicon. In this case we will treat the axicon as a thin optical element that effects the phase of the propagating beam.

WARNING: gUPPEcore does not have a method of verifying that user-defined addons correspond to real, physical situations. It is up to the users to ensure that what they are implementing is physical and coded correctly.

2.5.1 File: user_addons.cc

The first step is implementing the user addon is to let gUPPEcore know where to find information about the addon. This is done in the *user_addon.cc* file. There are examples of *user_addon.cc* files in the addon subdirectories of */src*. In this file the user will need to specify 3 parameters. The first is the name of the header (*.h) file that will contain the actual operator code. An example will be shown in the next section, for now we just need the name of the file. The next parameter is the IDstring. The IDstring is the name that will be assigned to type_id_string in the relevant section of the input file (see Appendix B). The final parameter is the className. The className will be used in the *.h file that is discussed in the next section. For this example an axicon custom operator is being defined. It will be defined using the following values:

IDstring: "Axicon"
className: User_operator_axicon
file: user_operator_axicon.h

The name of the file will simply be included as shown in the code below on line 7. The IDstring and className will have to be placed in the following bit of code:

```
_getVirtualClassPtr(s,"IDstring",className);
```

This bit of code will need to be placed in the appropriate class definition section of the code. In this example we are defining an operator so it will go in the UPPE_Operator section (see line 27 of the code listing below). If it were a medium response class it would need to be inserted between lines 19 & 20. Or if it were an initial condition class it would need to be inserted between lines 12 & 13. The return(0); must appear in all sections, regardless of the situation.

user_addon.cc for the axicon operator

```
1  #include<uppe_params_incond.h>
2  #include<uppe_params_response.h>
3  #include<uppe_params_operators.h>
4  #include<domain.h>
5
6  // include here all your user-defined class headers
7  #include"user_operator_axicon.h"
8
9  // user-defined initial condition classes:
10 template<>
11 UPPEinitial_condition* NameIdentifiedClassPtr_usr<UPPEinitial_condition>(char const* s) {
12     // in this case we have no user-defined initial conditions
13     return(0);
14 }
15
16 // user-defined medium response classes:
17 template <>
18 UPPE_params_nlresponse* NameIdentifiedClassPtr_usr<UPPE_params_nlresponse>(char const* s) {
19     // in this case we have no user-defined medium responses
20     return(0);
21 }
22
23 // user-defined operator classes:
24 template <>
25 UPPE_operator* NameIdentifiedClassPtr_usr<UPPE_operator>(char const* s) {
26     // in this case we have one user-defined operator
27     _getVirtualClassPtr(s,"Axicon",User_operator_axicon);
28     return(0);
29 }
30
31 // user-defined SpatialDomain classes:
32 template <>
33 gUPPE_Domain_Spatial* NameIdentifiedClassPtr_usr<gUPPE_Domain_Spatial>(char const* s) {
34     // in this case we have no user-defined SpatialDomains
35     return(0);
36 }
37
38 Params_Domain_UserDefined* GetClassPtr_Params_Domain_UserDefined(char const* s) {
39     // in this case we have no user-defined SpatialDomain-Parameters
40     return(0);
41 }
```

user_addon.cc code notes:

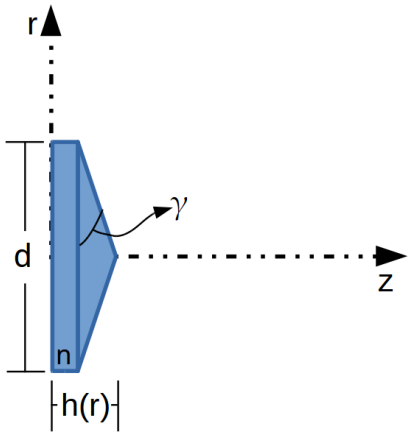
Line(s):	Comment
1-4:	Necessary header files from gUPPEcore. Should not be changed.
7:	Name of header file that user addon will be written in.
10-14:	Empty definition of intial condition addon. Must be included even though not used.
17-21:	Empty medium response addon. Must be included.
24-29:	Properly defined operator addon for axicon example. IDstring and className assignment code from discussion above.
32-41:	Empty spatial domain addon. Note there are 2 sections of code. See worked-out example 3.8.

In summary, user_addon.cc requires minimal changes for each addon, and serves to provide to gUPPEcore with a list of all user-defined classes.

2.5.2 File: *.h

This is the header file where the user addon is computationally defined. See addon directories in the */src* folder for other header file examples. If you are familiar with C++ programming you know that this file may be broken down in to several files. For this minimal example will put all of the code that is required to define our axicon in this file.

In this example we will create a simple thin axicon. It changes the phase of the incident pulse as it passes through the axicon. The amount of phase change is dependent on where the beam interacts with the axicon along its radius. The geometry used in this implementation is shown in the figure below.



Simple axicon geometry where:

- γ is the "axicon angle"
- d is the axicon diameter
- n is the axicon refractive index
- $r = \sqrt{x^2 + y^2}$ is radial position in the x-y plane
- $h(r)$ is the phase accumulating distance
- ω is the pulse frequency
- c is the speed of light

From this geometry we find that the total phase change for a ray passing through at radius r is:

$$\phi(r) = k \cdot h(r) = \frac{\omega}{c} \cdot [R + (R - r)(n - 1)] \tan(\gamma)$$

Implementation of this phase change is shown in the function starting on line 22 below. Line 29 below shows the code implementation of $h(r)$. Line 33 shows $\phi(r)$. The basic algorithm for applying this phase change is shown in lines 31-34. On line 31 we iterate over all polarization states (defined by numC passed to us by gUPPEcore). Then, on line 32, for each polarization we loop through all of the frequencies present in the pulse. Line 33 shows the calculation of the phase change as the imaginary part of a complex exponential. Finally line 34 stores the result of each polarization & frequency specific phase change in a block of memory to be used by gUPPEcore.

user_operator_axicon.h

```
1  #ifndef __operator_axicon_h__
2  #define __operator_axicon_h__
3
4  #include<stdlib.h>
5  #include<uppe_constants.h>
6
7  // every operator is derived from UPPE_operator:
8  class User_operator_axicon : public UPPE_operator {
9    public:
10
11     namdbl t;
12     namdbl d;
13     namdbl n;
14
15     User_operator_axicon() {
16       SetName("Axicon");
17       IncludeParameter(t,"Axicon_Angle[radian]");
18       IncludeParameter(d,"Diameter");
19       IncludeParameter(n,"Index");
20     }
21
22     void Operate(comple* target, int numC, int dim0, const double *omega, double x, double y){
23       int o,c;
24       comple phs;
25       double hr;
26       double r = sqrt(x*x + y*y);
27       double R = d.v()/2;
28
29       hr = (R + (R-r)*(n.v()-1))*tan(t.v());
30
31       for(c=0;c<numC;c++) {
32         for(o=0;o<dim0;o++) {
33           phs = exp(comple(0.0, - omega[o]*hr/cnst_c));
34           target[o*numC + c] *= phs;
35         }
36       }
37     }
38 };
```

```
39
40 #endif// __operator_axicon_h__
```

user_operator_axicon.h code notes:

Line(s):	Comment
1,2 & 46:	These define the axicon operator to gUPPEcore. The name used with #ifndef and #define needs to be the same. The name used with #endif is an optional comment but will minimize confusion if it is kept the same or omitted.
4:	Should always be included.
5:	Optional. This file contains commonly used constants such as cnst_c in line 39. If you want to be sure to use the same constants that the rest of gUPPEcore uses then include this file.
8:	The class name "User_operator_axicon" must be className that is defined in user_addons.cc. The rest of this line must remain the same.
9:	Everything below this public declaration is available outside of this class.
11-13:	Public variables. In this case we create constant doubles d, t & n which will be used to hold the axicon parameters.
15-20:	This section defines the named variables that are used by the addon. Specifically these are the items that are required to be in the input file if the user-addon is being used. This will be seen in the usage section of this example.
22-37:	This is where the operator implementation algorithm is defined.

As the listing shows, the core of the add-on action is encoded in the function called *Operate*. Every addon that belongs to the operator class must have this function defined. Its parameters have the following meaning:

target is the pointer to the array(s) on which the operator acts
numC is the number of (polarization) components
dimO is the number of active frequencies the spectral field amplitude is sampled on
omega is the array of active frequencies
x, y are transverse coordinates

Line 34 shows how one can access the spectral amplitude corresponding to the angular frequency *omega[o]* and the polarization component *c*. So in the body of *Operate*, simply visit all spectral amplitudes and modify them to reflect the action of your operator. In this case, we merely add propagation phase onto the spectral amplitude, this phase being dependent on the radius calculated form *x,y*.

2.5.3 Addon Compilation

To begin compilation gather *user_addon.cc*, **.h* and a Makefile (from the */src/addon** directory) all into the same folder. The Makefile will look similar to the one below:

```
1  # specify appropriate values here:
2
3  CC     = /usr/bin/g++
4  MPICC = /usr/bin/mpiCC
5  FLAG   = -O3  -Wall -Wno-deprecated -pedantic
6
7  INCL    = -I ../../include -I .
8  XLIBS   = -L ../../bin -lguppecore -lfftw3  -lgsl -lgslcblas
9  LIBS    = -L ../../lib -lguppempiutils -lguppecore -lfftw3  -lgsl -lgslcblas
10
11 ###################################################################
12
13 OBJS =  user_addons.o
14
15 all:  clean exe cleanall
16
17 clean:
18         rm -f *.o *~ core* *.out
19
20 cleanall:
21         rm -f x* *.o lib*.a
22
23
```

```
24  %.o : %.cc %.h
25          ${CC} ${FLAG} -c $< -o $@ ${ILOC} ${INCL}
26
27  %.o : %.cc
28          ${CC} ${FLAG} -c $< -o $@ ${ILOC} ${INCL}
29
30  exe:
31          ${MPICC} ${FLAG}  -o guppeA.out user_addons.cc ${INCL} ${LIBS}
```

Makefile notes:

Line(s):	Comment
3 & 4:	These 2 lines will be familiar if you have compiled gUPPEcore. They should be set to the same values in section 1.4.
5:	Compiler options that probably do not need to be changed. If these options seem to be giving you trouble then refer to section 1.4 for the compiler options you previously used in your loci file.
7-9:	These are path definitions that are used later in the Makefile. They most likely will not need to be changed. The primary situation where they would be changed is if the indicated folders are not in the ../../ relative path. You may have to change the relative path in these lines so the compiler can see the folders.
13:	This should only be changed if you did not use the user_addons.cc naming convention.
15 - 30:	Does not need to be edited in most situations.
31:	This line only needs to be changed if you want a custom name for you executable or if you did not name your file user_addons.cc.

Ones the Makefile is saved run "make" from within this directory. This will compile the addon to the specified name in the current directory. If you wish you may copy the compiled executable to the simulation directory that contains your input file. The next section will assume that the executable and the input file are in the same folder.

2.5.4 Addon Usage

Once you have the addon compiled you must instruct gUPPEcore to use it. This is done by editing your script that runs gUPPE to use the newly compiled addon and placing the newly required values into your Input file. To use your newly created addon replace "guppeCore.out" in your script with the name of the addon executable. For example, if your input file and your executable are in the same directory you would change:

```
${MYmpirun} -n 16 guppeCore.out input | tee rec
```

to:

```
${MYmpirun} -n 16 guppeA.out input | tee rec
```

You will also need to edit the input file. At a minimum you will have to assign your "IDstring" to type_id_string under the section of the input file where your addon is used. There may be other values that you need to set in this section depending on how you implemented your addon. To use the axicon operator with a 1/2° axicon angle we add the operator and associated name parameters to the UPPE_Operator_Prolog section of the Input file as shown:

```
UPPE_Operator_Prolog
  num._of_items                 1
    type_id_string              Axicon
      Axicon
      Axicon_Angle[radian]      0.0087
      Diameter                  5
      Index                     1.5
```

Once the input file and script are updated you are ready to perform the custom simulation. Note that it is entirely up to the user to ensure that they results they get are valid.

For further examples of user-addons please see the user-addon worked-out examples and the */src* directory in the gUPPEcore download.

3 Worked-Out Examples

The */templates* directory contains worked out simulation examples. It is highly recommended that users familiarize themselves with Inputs Explained (Appendix B) & Outputs Explained (Appendix C) prior to working through the examples. Each of these directories has an informative README text file or pdf which describes a few aspects of the input and output control, simulation execution and result interpretation.

The example folders are broken up into two groups. The first group concentrates on the basic usage (i.e. without user customizations). It is recommended that users inspect these exercises in the order shown below.

- */templates/wrk_010_Test_Linear_Propagation*
- */templates/wrk_020_Supercontinuum_In_Water*
- */templates/wrk_021_Supercontinuum_Microstructured_Fiber*
- */templates/wrk_031_Filament_In_Gas_Radial*
- */templates/wrk_032_Filament_In_Gas_3D*

The second group contains simulation templates (input files) for situations which require user-defined customization. Description of how various user-defined objects and capabilities can be added to the framework can be found in the corresponding folders in the source directory (src). Commented source files implement examples which can be easily adapted to specific needs. Of course, arbitrary number of user-defined add-ons can be combined.

- */templates/wrk_040_User-Addons-Operators-Multiple-Filaments*
- */templates/wrk_041_User-Addons-Medium-Response*
- */templates/wrk_042_User-Addons-Initial-Conditions*
- */templates/wrk_043_User-Addons-Propagators*
- */templates/wrk_044_User-Addons-Operatorsfy2*

gUPPElab simulation example/template content:

- *README.pdf* gives the description of the example. Each lab concentrates on a small number of issues related to the usage of gUPPEcore, and interpretation of results. With the less experienced users in mind, it may also discuss potential pitfalls.

  Together, these descriptions play the role of the user manual. It is therefore best, especially for the first-time users, to visit the example directories in the suggested order.

  Note that discussion of the underlying physics in these descriptions is only rudimentary. While most of the included simulation scenarios are realistic, they only serve as a backdrop for illustration of gUPPEcore capabilities.

- *inputXY*: Input file(s) for the simulation.

- *recXY*: Files that record all simulation outputs. They can be used to re-create the given simulation exactly. They also hold some useful outputs.

- *SIMDIR*: Directory storing most of the simulation outputs. Note that this folder ships empty. Users must execute the simulation to re-create its content (which may require a lot of storage).

- *runthis*: Script(s) to illustrate control of a sequence of simulations. Other scripts may illustrate execution of massively parallel jobs.

- *extract_observables*: Script(s) that illustrate how, and from where to obtain observable quantities.

- *genPlots.py*: (Optional) Python script to generate plots from simulation data. Can be run on its own or called from the runthis script.

- *PLOTS*: (Optional) Directory for storing plots generated from simulation output. Like SIMDIR this folder is initially empty and may require significant disk space to store plots.

The following subsections contain the README files from each of the templates. These documents may be found individually in each template folder.

3.1 Linear Gaussian Beam Propagation

Testing Linear Propagation, Analytic vs. Simulated Solutions & Computational Domain Parameters

gUPPEcore usage illustrated in this example:

- setup for different geometries of the computational domain
- simulation execution
- using observer-generated files for visualization

Computational issues illustrated:

- This package shows a useful way to test linear propagators

Physical context:

- Femtosecond pulse propagating in a linear medium
- Comparison to an analytic Gaussian pulsed beam solution

Input files

All input files initiate simulations with a Gaussian pulse propagating in a linear medium.

- *inputA*: Radially symmetric simulation using DOMAIN_RADIAL with AXIS_RADIAL.
- *inputAcmp*: Comparative run: Performs no propagation. It only saves its initial condition that is the analytic solution "propagated analytically" to the same distance as the *final* distance reached in the simulation starting from *inputA*. Results of the two runs should be very close.
- *inputB*: Radially symmetric simulation, using AXIS_RADIAL. The difference from run A is that there are two field components in run B. Each is a different pulsed Gaussian, propagating independently. This setup thus tests usage and observation of computational domains with multiple (polarization) components.
- *inputBcmp*: Comparative run producing target solution for run B. It works the same way as *inputAcmp*.
- *inputC*: Fully spatially resolved simulation, using DOMAIN_LINEAR_x_LINEAR. This template also propagates two field components with initial parameters that are the same as in run B. We can thus test radial and 3D solutions against each other.
- *inputCcmp*: Comparative run producing target solution for run C.

- *inputD*: Fully spatially resolved simulation, using DOMAIN_LINEAR_x_LINEAR. This is nominally same run as C, but using different temporal domain and resolution. Illustrates fine points of comparison with an analytic target solution, namely that the reference frequency, together with phase and group velocities must match exactly those utilized by the analytic solution.
- *inputDcmp*: Comparative run producing target solution for run D.

Linear propagator test

gUPPEcore implements several initial-condition waveforms. One of them is a pulsed Gaussian beam which can be instantiated at an arbitrary propagation distance. We use this feature here to construct a simple testing procedure for the linear propagator associated with each Spatial Domain.

The idea is to start form an analytic solution Analytic($z = 0$) and save profiles of the numerically propagated waveform Numerical($z = z_{stop}$). Then a run with Analytic($z = z_{stop}$) serving as an initial condition is executed without any propagation. It saves the "result" which can be subsequently compared with the data from the previous simulation.

The difference between the two runs is easiest to see by executing diff on the two input files. For example *diff inputA inputAcmp* produces:

Listing 3.1: Input-file difference: diff inputA inputAcmp

```
1  <    base_name                    SIMDIR/A
2  ---
3  >    base_name                    SIMDIR/Acmp
4
5  <    z_stop                       1.00
6  ---
7  >    z_stop                       0.00
8
9  <          pulse_distance              0
10 ---
11 >          pulse_distance              1.00
```

This listing of differences highlights the differences in parameters that realize the procedure outlined above. Simulation A propagates the initial condition numerically to a final distance of 1 meter, while simulation Acmp applies analytic formula for the initial condition calculated for the distance of 1 meter.

Domain axis objects and spectral transforms

As shown in the input files, AXIS objects "define" the geometry of the computational domain. Besides providing the frame of reference, axes also hold the corresponding Hankel (AXIS_RADIAL) and FFT (AXIS_LINEAR and AXIS_TEMPORAL) spectral transforms. This is why they must be initialized with some care.

The linear spatial domains are defined in sections like this one:

Listing 3.2: X×Y computational domain setup

```
1  Domain_Spatial
```

```
2     domain_type                  DOMAIN_LINEAR_x_LINEAR
3
4     Domain_linear_x_linear
5
6       Domain_axis_x
7         axis_type                AXIS_LINEAR
8         real-space_size          0.015
9         number_of_points         192
10        init_type                2
11
12      Domain_axis_y
13        axis_type                AXIS_LINEAR
14        real-space_size          0.015
15        number_of_points         192
16        init_type                2
```

There are two important parameters above: First, the number of grid points that populate the linear axis must be compatible with FFT, namely it must factorize into small primes. The fastest performance is still obtained with number of points being a power of two.

The second parameter of importance is the *init_type*. It controls how much time will the FFTW library spend on selection of the optimal algorithm for the requested number of points and given hardware. Most of the time we use value of two for it.

The spatial domain for radially symmetric simulations is defined as in:

Listing 3.3: Radial computational domain setup

```
1   Domain_Spatial
2       domain_type              DOMAIN_RADIAL
3
4       Domain_radial
5
6         Domain_axis
7           axis_type            AXIS_RADIAL
8           real-space_size      0.01
9           number_of_points     500
10          init_type            0
```

The only less-then-obvious parameter here is the *init_type*. The value zero represents the order of the underlying Hankel transform (which is used for spectral transform in the linear propagator). While higher-order Hankel transforms can be used to simulate vortex beams, the zero value is required for all radially symmetric simulations without angular momentum. Note that this meaning is different form that of the same-name parameter that appears in the temporal domain definition.

The number of grid points need not be any special value. The Hankel transform is a full-matrix operation with a pre-calculated and stored matrix. Its precise size is therefore not as important as in the FFT case. However, users must keep in mind that the computational complexity scales with the square of the number of points.

Execution of simulations

gUPPEcore is parallelized with MPI, and must therefore be started with the program *mpirun* which comes with the given MPI implementation. In this example, we have included several scripts, e.g. *runthisA*, which show how the simulation set up in this folder may be executed. Typically, standard output produced is stored in "record file" (e.g. *recA*). One reason for doing this is that the top portion of such a record can be subsequently used to reproduce the original simulation. Script *runthisA* contains a line that executes the simulation as (with the path to mpirun held in an auxiliary variable):

Listing 3.4: Simulation execution, command line

```
1 mpirun -n 4 ../../bin/guppeCore.out inputA > recA
```

Note that the number of requested mpi processes should not exceed the number of physical CPU cores available. Users should therefore pay attention to the values specified in the attached scripts, and modify these appropriately.

Using observer-output files

Outputs generated by gUPPEcore are organized into Reports, and these come in four different levels. Level one produces observables that are inexpensive in terms of numerical effort and disk space they require. They are scalar global values directed to the standard output, and will be illustrated in subsequent simulation templates. In this example we concentrate on Report level two. Its output consist of files which hold "one-dimensional" data.

Two kinds of outputs from Report level two are utilized here, namely the temporal and spatial profiles of the optical electric field. From the record files, e.g. *recA* and *recAcmp*, one can see that Report_2 number 10 occurred at $z = 1.0$ which is the same distance Report_2 number zero occurred in the comparative run. Thus, one needs to compare field profiles:

SIMDIR/A_T_c0_10_r_0 vs *SIMDIR/Acmp_T_c0_0_r_0*
for the temporal slice, and

SIMDIR/A_S_c0_10_r_0 vs *SIMDIR/Acmp_S_c0_0_r_0*
for a view along the radius of the domain.

Script runthisA and similar indicate which files hold the data for comparison in each pair of simulations. Users who have installed *xmgrace* grapher can execute these scripts to see the plots. Users utilizing other plotting tools should note that the format of these and similar output files is described in the folder *000_Outputs_Explained*.

Pitfalls

The idea of this example is to set up simulations that demonstrate that the numerically propagated waveform produced by the simulator are in close agreement with the analytic solutions. This indeed happens with the parameters given in these examples.

However, it is rather easy to obtain results that will show a gap between the numerical and analytic solutions. In most cases this will be because the linear propagators used in gUPPEcore are spectral and non-paraxial, and produce essentially exact solutions, while the analytic solution is a paraxial, and monochromatic approximation.

Runs D and Dcmp illustrate yet another possibility. The time-domain dispersion section holds parameters for the reference frequency/wavelength, and displays group and phase refractive index

(for this wavelength). These values must coincide with their counterparts in the Gaussian initial conditions. A mismatch will cause the the frame of reference to "slide" and the resulting solution will appear translated in the local pulse time, thus not matching the analytic one. Experimentation with the input values in *inputD* and *inputDcmp* should illuminate this issue.

In particular, the reference wavelength of the temporal domain must be set to the same value as in the analytic Gaussian solutions. The temporal domain utilizes a grid with the extent and number of points specified in the AXIS sub-section. These in turn determine which angular frequencies appear in the numerical spectrum. While a specific reference frequency is requested in the input file, it will be adjusted to the nearest active frequency given by the domain. This is usually of no practical consequence, because the numerical spectrum is normally quite dense, and the reference frequency is only used to obtain the value of the moving-frame velocity. It therefore has no influence on the simulation results beyond keeping the simulated pulse close to the center of the temporal computational domain. However, in the present case, even a small deviation from the wavelength of the analytic solution would cause walk-off between the analytic and numerical frames of reference. The consequence would be apparent temporal shift of the simulated pulse, and disagreement with the target solution.

The input files in this simulation template were set up such that the appropriate reference wavelength was "read of" from the output of a trial run. The value was then inserted into the initial condition section to specify the wavelength of the analytic solution. Users are invited to experiment with the relevant parameters...

3.2 Femtosecond Supercontinuum Generation in Water

Angularity Resolved Spectra Extraction & Medium Susceptibility Tabulation

gUPPEcore usage illustrated:

- typical femtosecond pulse simulation setup
- extraction of angularly resolved spectra

Illustrated computational issues:

- Medium susceptibility tabulation

Physical effects:

- Femtosecond filament dynamics in water, characterized by dispersion-driven temporal pulse splitting.
- Relation between dynamics and chromatic dispersion on one side and the structure of angularly resolved spectrum on the other.

Input files

- *inputA*: Femtosecond pulse loosely focused on the entrance of a water sample. The central wavelength of the pulse is 800 nm.
- *inputB*: This is a comparative run with 520 nm central wavelength. It also uses an adjusted window of active frequencies (compare temporal axis sections).

Simulation execution

Two scripts, *runthisA* and *runthisB* can be used in a Linux environment to execute the simulations. The scripts set an auxiliary variable holding an absolute path to the appropriate *mpirun* program — user will need to adjust this appropriately. With the *mpirun* selected, each run is started as follows:

Listing 3.5: Simulation execution, command line

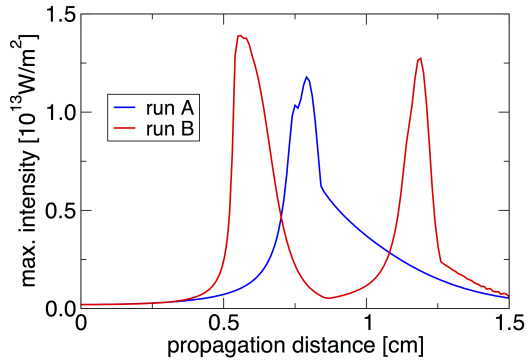
```
1 \${MYmpirun} -n 16 ../../bin/guppeCore.out inputA > recA &
```

Standard output is redirected into a "record file", *recA* in this case, which is subsequently utilized to extract certain global quantities, such as maximal on-axis pulse intensity.

Extracting global quantities from the record file

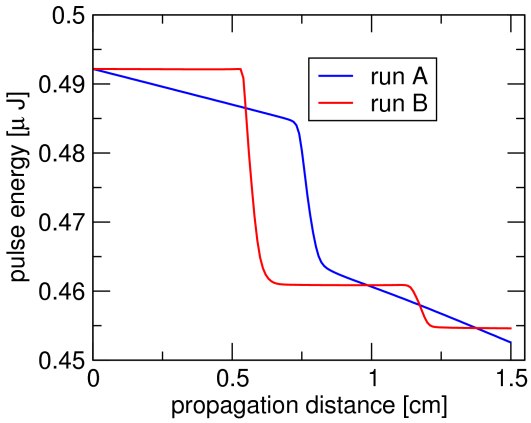
Script *extract_observables* shows how to obtain values of on-axis intensity, pulse energy, integration step-size, maximal plasma density, and linear plasma density as functions of the propagation

distance. The resulting files, such as *InteA.dat* are subsequently used to produce figures like this one:



Intensity maxima in femtosecond filamentation in water: Each is generated by self-focusing collapse that is arrested by temporal pulse splitting. Chromatic dispersion plays an important role in water and condensed media in general. This figure illustrates how different wavelength result in quantitatively different dynamics.

Another global quantity we want to show here is the pulse energy evolution with the propagation distance:



Pulse energy evolution reveals the increased role of water absorption at longer wavelength (run A). The steep energy decrease sections in these plots reveal losses due to ionization — they only occur at highest intensities. The gradual energy decrease in the longer-wavelength run is caused by wavelength-dependent linear absorption.

Since gUPPEcore is a spectral solver in the time dimension, it has the capability to account for arbitrary wavelength-dependent dispersion and linear losses. Both are incorporated through the complex-valued susceptibility tabulation invoked in the input file, in this case it is *waterchi.tab*. This is an example of tabulation that the user needs to prepare for working with a new medium.

The tabulation must hold the susceptibility as a function of angular frequency (first column). Both real (second column) and imaginary (third column) parts of the susceptibility must be present, even when the imaginary part is zero. It is important that the table is dense, because it is interpolated inside the simulator. Also, when working with experimental data to obtain a susceptibility table, one must ensure that the tabulation is sufficiently smooth. One way to test if the noise in the table is acceptable is to plot the group-velocity dispersion obtained by a finite-difference approximation from the corresponding propagation constant.

Extracting angular-spectrum arrays from multiple files

The script *extract_observables* also illustrates one way to extract data needed to plot angularly resolved spectra. gUPPEcore stores two-dimensional arrays separated into ranks. Rank labels depend on the number of parallel MPI processes used in the given simulation, which was 16 in the

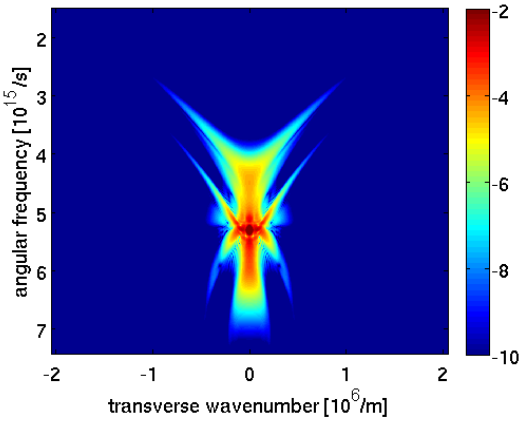
present example. Most of the time, only a subset of these "partial data sets" are used, and this is why we keep them separated. But here we want to show how to join the whole set together if needed. For instance:

Listing 3.6: Concatenating data files for angularly resolved spectra

```
1 cp SIMDIR/A_KO_c0_8_r_* .
2 rename r_ r_0 A_KO_c0_8_r_?
3 cat A_KO_c0_8_r_* > A_K_Omega_spc
4 rm -f A_KO_*
```

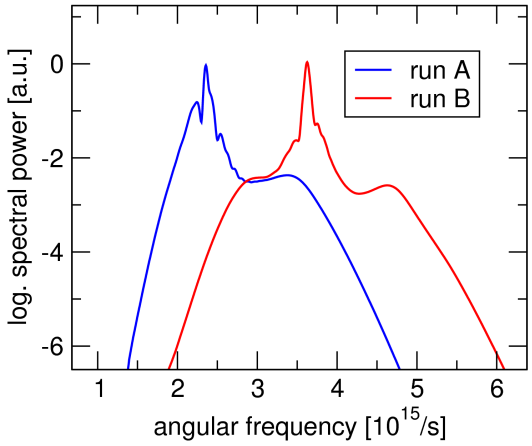
In the first line, angularly resolved spectra created in the Report labeled with number 8 are copied into the working directory. At this point they are separated into files, one for each individual MPI parallel process, and we need to join them in order to create one large rectangular array for plotting. But first, we add zero in the names of those files named *A_KO_c0_8_r_?* where ? stands for a wild-card representing one character. This occurs in line two, and ensures that the files are naturally "ordered" by their file-names. The next line simply concatenates them into a single larger file. This is where the user could choose to use only certain "slices" of the computational domain, for example to reduce the size of the array in case that the "action" only occurs in the central portion of the domain. For simplicity, we use slices from all MPI process ranks. The result is an array stored in *A_K_Omega_spc* which we use to plot the spectrum shown in this report. Run B is processed in the same way...

The resulting files *A_K_Omega_spc* and *B_K_Omega_spc* then can be used to visualize the distribution of the pulse energy in the angularly resolved spectrum. Here we used a simple Matlab script *PlotKOspc.m* to produce the following picture:



Angularly resolved supercontinuum spectrum (run B). This figure illustrates the "native" representation of the optical waveform in gUPPEcore. One half of this map represents the array that holds the complex-valued spectral amplitudes. Here we show the spectral-power map in log scale. This kind of plot is useful to check that the whole spectrum is safely contained within the computational box. The X-like features in this spectrum reflect pulse-splitting dynamics: Each extended feature is created by one of the "daughter" pulses.

An easier, and more practical way to inspect spectral properties of the evolving pulse waveform is with the angularly integrated spectra. These are generated by the gUPPEcore built-in observer (user must switch their output selector "on" in the input file) in the Report-level 2. The next picture compares the spectra obtained at propagation distances after the first pulse splitting event in each case:



Angle-integrated supercontinuum spectra. Pulse with the central wavelength of 520 nm (run B), generates a more symmetric spectrum, because the chromatic dispersion "landscape" is more symmetric than that around 800 nm wavelength used in simulation A. The wings of these spectra correspond to the extended features evident in the angularly resolved spectrum (previous figure).

Here we have plotted data stored in files *SIMDIR/A_FFS_c0_10_r_0* and *SIMDIR/B_FFS_c0_8_r_0*. String "FFS" indicates (see folder "000_Outputs_Explained") that these files hold both on-axis (second column) and angularly integrated (third column) log-scale spectra as functions of angular frequency (first column). The report index (10 in case of run A) can be associated with the propagation distance by inspecting the record file. For example *recA* shows a line with *REP2: 10 0.0100184* which means that the Report-level 2 was executed at the propagation distance of 0.0100184 meters. From *recB*, the propagation distance at which the run B spectrum was recorded was 0.00801558 meters.

3.3 Femtosecond Supercontinuum Generation in a Microstructure Fiber

Fiber Geometry, Optical Field Normalization & Imported Medium Properties

gUPPEcore usage illustrated:

- Simulation setup for a fiber-like geometry
- Import of tabulated chromatic medium properties

Computational issues illustrated:

- Different fields in ultra-short pulse propagation simulation prefer different optical-field normalizations. In particular, normalization to peak power is often used for simulations of fibers and waveguides. The convention adopted by gUPPEcore is the same as in bulk media, because the fiber geometry is treated as a special case of a "bulk medium" with chromatic dispersion equivalent to that if the simulated fiber.

Physical background:

- Femtosecond supercontinuum dynamics in fibers.
- Soliton fission mechanism, and soliton self-frequency-red-shift.

Input files

- *inputA*: 800 nm wavelength, 100 fs duration pulse propagating in a silica strand with radius of 400 nm.
- *inputB*: The same as run A, but with observer report period(s) set up for frequent outputs in order to allow smoother visualization of various quantities versus propagation distance. To ensure that the propagation distance sampling is equidistant, the *step_ type* is set to *fixed* in the ODE_Driver section. With this input, the ODE solver performs a fixed-step integration.
- *inputC*: Physically equivalent to run A, but realized with a radial spatial domain. This serves as a comparative simulation run.

Simulation template setup

1. Spatial computational domain for fiber-like geometry

Fiber geometry can be simulated with gUPPEcore capabilities in two equivalent ways. One is using the radial spatial domain with a single radial-grid sampling point, the other utilizes an $X \times Y$ domain with a single point in each transverse dimension.

The following is partial listing of *inputA*, showing the section that defines the spatial computational domain:

Listing 3.7: 1. Spatial computational domain mimicking a fiber

```
1  Domain_Spatial
2
3   domain_type                    DOMAIN_LINEAR_x_LINEAR
4
5       Domain_linear_x_linear
6
7         Domain_axis_x
8           axis_type                AXIS_LINEAR
9           real-space_size          2.0e-06
10          number_of_points         1
11          init_type                0
12
13        Domain_axis_y
14          axis_type                AXIS_LINEAR
15          real-space_size          2.0e-06
16          number_of_points         1
17          init_type                0
```

Here we have set two linear axes, each with a single grid point. The real-space size should be selected such that the resulting area represent the effective fiber area. The value requested for *init_ type* is irrelevant, because the spectral transform along each spatial dimension reduces to an identity operation.

In the next listing, it is the radial domain that "simulates" the fiber. This is part of the *inputC* file:

Listing 3.8: 2. Spatial computational domain mimicking a fiber

```
1  Domain_Spatial
2
3   domain_type                    DOMAIN_RADIAL
4
5       Domain_radial
6
7         Domain_axis
8           axis_type                AXIS_RADIAL
9           real-space_size          2.0e+06
10          number_of_points         1
11          init_type                0
```

Note that in this case the *init_ type* must be set to zero.

Simulations with both of the above geometries are essentially equivalent, as they both realize a 0×1-dimensional domain. Users can check this by comparing the results of the two "equivalent" runs defined by input files *inputA* and *inputC*.

2. Dispersion properties associated with the temporal domain

gUPPEcore associates the dispersion properties of a medium with the temporal computational domain. The following listing shows the corresponding section of the input:

Listing 3.9: Dispersion properties

```
1       T_Domain_Dispersion
2           lambda_reference            800.0e-09
3           omega_reference             2.34572e+15
4           omega_min                   1.3e+15
5           omega_max                   8.5e+15
6           index_phase                 -1.32107
7           index_group                 -1.34471
8           gvd                         -9.8345e-26
9           susceptibility_file_0        silicastrand_chi_0.4
```

The relevant named parameter here is in the last line, and specifies the name of a file that holds tabulation of the susceptibility for the medium we want to simulate. Note that the user must specify one table for each field component carried by the computational domain (single component in this case), even if all field components experience the same chromatic dispersion.

The susceptibility file must be a human-readable, consisting of three columns. The first column is the angular frequency, followed by the real and imaginary parts of susceptibility in the second and third columns. The tabulation must be sufficiently smooth, with typical number of sampling points from a few hundred to two thousand. Most often a susceptibility table is generated from a Sellmeyer formula. When the temporal axis is instantiated, its dispersion properties are initialized, and this is when the susceptibility table is read and interpolated onto the grid of active frequencies.

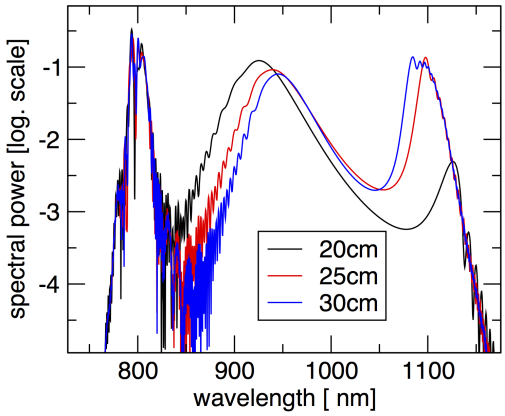
Note that when one wants to simulate pulse propagation in a fiber, the susceptibility table should contain the *effective* susceptibility that accounts for both the material properties and geometry of the fiber. In this example, we have calculated the propagation constant for the fundamental mode of of silica strand (in air) as function of angular frequency, $\beta(\omega)$ from which the effective susceptibility is straightforward to obtain and tabulate.

Generated spectrum illustration

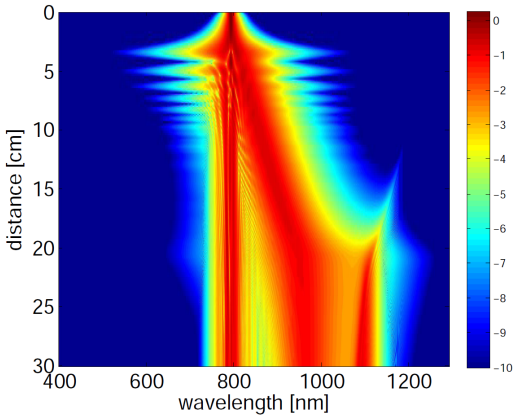
In this simulation template, we have directed all output files generated by the built-in gUPPEcore observer into folder *SIMDIR*. These files are not shipped with the the gUPPElab, as the whole lab would take much longer to download. Thus, users must re-run most of the simulations, and re-create the files in *SIMDIR*.

Here we have used outputs stored in files *SIMDIR/A_FFS_c0_XX_r_0* with $XX = 20, 25, 30$ corresponding roughly to propagation distances of 200, 250, and 300 mm. Note that when we simulate propagation in a fiber there is no difference between the on-axis and angularly integrated spectra stored in such files.

The following figures illustrate the evolution of supercontinua generated from the red-shifted solitons which split-up from the input pulse. The initial pulse undergoes a series of "oscillations" during which a solition-like pulse is "emitted" from it. This soliton subsequently experiences two effects that contribute to the resulting shape of the supercontinuum spectrum. First is the self-frequency red shift, which is caused by the Raman effect. The second is emission of dispersive waves which appear in these figures as the long-wavelength feature in the vicinity of 1100 nm wavelength. It is caused by the reaction of the to-be-soliton to its propagation medium which is not exactly NLS-like.



Supercontinuum generation in a nanometer-sized silica strand. The three main peaks in these spectra correspond to: Input pulse wavelength (800 nm), temporal soliton (red-shifted upon propagation), and dispersive wave radiation emitted from the soliton.



Supercontinuum (log-scale spectral power shown) evolution with the propagation distance. The "diagonal" feature is the self red-shifted soliton spectral peak.

The physical mechanism underlying this supercontinuum dynamics is has been described in detail in PRA 82 (4), 045802 (2010) for a micro-structure fiber with properties closely resembling those of the nano-meter size silica strand used for the present example.

3.4 Femtosecond Filament in Gas: Radial Coordinates
Time Domain & Extraction of Global Quantity

gUPPEcore usage illustrated:

- extraction of global quantities from the simulation record file

Computational issues illustrated:

- Usage and potential pitfalls of time-domain boundary guard

Physical background:

- Femtosecond filament dynamics in gases (air in this case)
- Standard optical filamentation model

Input files

- *inputA*: 800 nm wavelength, 30 fs Gaussian pulse with $\sim$ 2mJ energy, loosely focused to 2.5 m distance. Temporal domain guard is applied after every 0.1 m propagation (this is specified in the third line of section *Domain_Temporal*. The role of the guard is to eliminate radiation that reaches the edge of the computational domain (in the temporal dimension).
- *inputB*: The same as above, but the temporal domain size is increased from 400 fs in run A to 600 fs in run B. At the same time, resolution has been increased from 2048 to 4096 temporal grid points.

Simulation template setup

There are two pre-defined simulation runs in this simulation template. Run A is using what is a rather minimal computational domain, while run B utilizes somewhat larger temporal domain. Files *inputA* and *inputB* hold the respective inputs.

The goal here is to appreciate the role of the domain boundary. When optical filamentation occurs, it is accompanied by creation of new spectral components. This light subsequently propagates with different group velocities and gradually walks off from the main "parent" pulse. Eventually, the supercontinuum radiation reaches the temporal domain boundary.

When it happens, there are two options, from the computing pount of view. First, one can stop the simulation at this point to avoid potential artifacts. This should be the preferred approach. However, sometimes one really needs to propagate further. For those "emergencies," gUPPEcore implements a boundary guard that can be invoked periodically, and which eliminates radiation in the vicinity of the domain edge.

The two runs in this package illustrate the associated effects.

Global observables

gUPPEcore simulation run is typically executed such that a full standard output is stored in "record file" (*recA* and/or *recB* in this case). One reason for doing this is that the top of such a record can be subsequently used to reproduce the original simulation. Script *runthisA* contains a line that executes the simulation as:

Listing 3.10: Simulation execution, command line

```
1 ${MYmpirun} -n 16 ../../bin/guppeCore.out inputA > recA &
```

where the variable *MYmpirun* stores the path pointing to the appropriate mpirun. Lines of *recA* that start with REP1 carry a listing of global quantities measured as functions of propagation distance. For example this line from *recA*

Listing 3.11: Extracting global observables from a record file

```
1 REP1: 0.116 0.025 16466 5.49915e+15 0.00184568 1.3412e+09 1841.13
```

reports global quantities measured at $z = 0.116$m. In the left to right order they are: propagation distance, proposed integration step, grid-index of the temporal observation point, maximal intensity, pulse energy, maximal plasma number density, and total linear plasma density.

Script *extract_observables* shows how the values can be extracted for subsequent visualization and analysis:

Listing 3.12: Extracting global observables from a record file

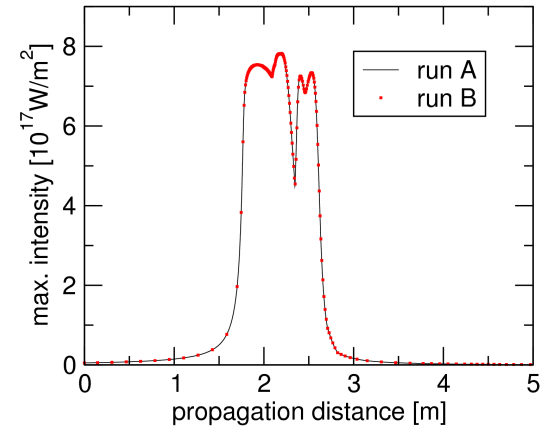
```
1 #!/bin/bash
2
3 grep REP1 recA | cut -d" " -f2,5 > InteA.dat
4 grep REP1 recA | cut -d" " -f2,6 > EnerA.dat
5 grep REP1 recA | cut -d" " -f2,7 > DensA.dat
6 grep REP1 recA | cut -d" " -f2,8 > PlasA.dat
```

and similarly for run B. For user less familiar with Linux utilities, let's just note that the first command, grep, extracts all lines that contain "header" REP1. The result is then fed (through pipe |) into command cut which is instructed to extract specified fields or columns (separated by space).

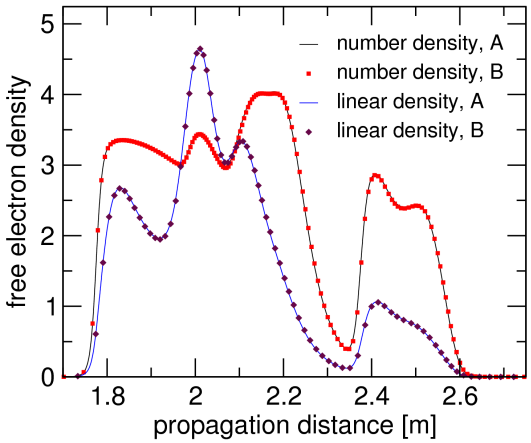
As a result, we extract a set of two-column files that hold the propagation distance in the first column (second field from left in the output of the grep command). Meaning of the second column in these output files is as follows: maximal intensity (in units specified in the observer section of *inputA*), total pulse energy estimate, maximal plasma density (in m^{-3}), and linear plasma density (in 1/m).

The XXXX.dat files are used in the following figures to visualize the results of this simulation. The first two figures compare maximal light intensity, and two measures of free electron generation, respectively. The good agreement between data from runs A and B suggests that the temporal domain size is sufficient. However, the third figure reveals that the temporal domain guard extracts measurable energy from the computational domain. This is less significant in run B because the domain is larger and it thus takes longer for high- and low-frequency light to walk off from the main pulse and reach the computational domain edge (where it gets "annihilated" by the domain guard). This illustrates that one has to be careful when applying boundary guard. Situations with broad spectra may require a guard to keep the domain reasonably small, but caution must be exercised to use only data that has not been affected by the action of the guard. For example,

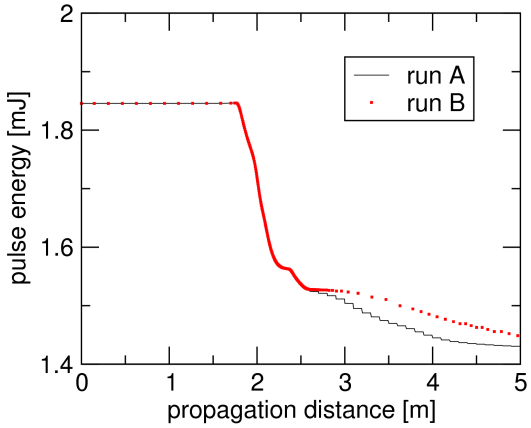
in the present case, the proper way to deal with radiation spreading away from the main pulse is to simply increase the temporal domain size until the guard becomes unnecessary.



Maximal intensity versus propagation distance. Re-focusing typical of optical femtosecond filaments in gases manifests as the second "peak." The intensity is clamped by the simultaneous effects of plasma-induced de-focusing, diffraction, and ionization losses all competing with self-focusing nonlinearity due to optical Kerr effect. Two runs (A,B) with different temporal domain parameters are compared in this plot.



Free electron ("plasma") generation in the femtosecond filament. Maximal number density is shown in units of 10^{22} per cubic meter, and the linear density is in units of 10^{14} freed electrons per meter of propagation distance. The latter quantity is a measure of total electron yield. The ratio of the two densities measures the effective area of the plasma column. The agreement between runs A and B is obviously good also for the electron densities.



Pulse energy recorded in two "nominally identical" simulation runs. The sharp drop around $z \approx 2$m is due to multi-photon ionization losses. The more gradual energy decrease that follows is due to temporal spread and "leakage" of energy outside of the temporal domain. Steps in the curve correspond to point at which the guard was invoked. Run B was executed with a wider domain in order to lessen this issue.

3.5 Femtosecond Filament in Gas: 3D Spatial Coordinates
Large-scale Simulations & Coordinate System Comparison

gUPPEcore usage illustrated:

- execution of large-scale simulations (more than a billion of degrees of freedom)
- parallel scaling

Computational issues illustrated:

- Observation-focus issues, comparison of radial and 3+1D simulations

Physical background:

- Femtosecond filament dynamics in gases
- Standard optical filamentation model

Input files

- WARNING: This simulation template contains a large-scale simulation which should not be attempted with more MPI processes that there are physical CPU cores on the given computer.
- *inputXY*: 800 nm wavelength, 30 fs Gaussian pulse with $\sim$ 2mJ energy, loosely focused to 2.5 m distance. Temporal domain guard is applied after every 0.1 m propagation (this is specified in the third line of section *Domain_Temporal*. The role of the guard is to eliminate radiation that reaches the edge of the computational domain (in the temporal dimension).
- Note: This set up is nominally the same as in the example with the radially symmetric solution.

Simulation template setup

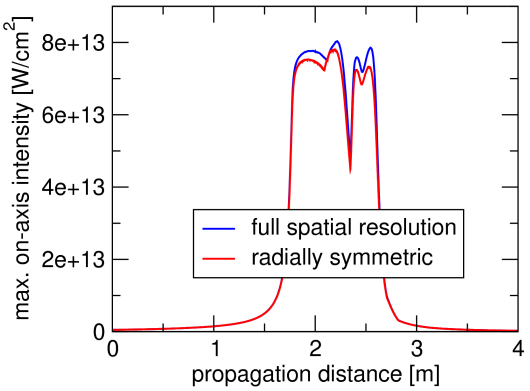
The purpose of this example is two-fold. Because the physical parameters are chosen equal to those in the previous, radially symmetric simulation example, users have the opportunity to compare the corresponding inputs and verify the proper way to setup "equivalent" simulations for radially symmetric situation on one side and a fully spatially resolved simulation on the other.

The second purpose is to illustrate the efficiency of parallel scaling in gUPPEcore. The same simulation is executed with an increasing number of parallel processes, and we show that the parallel speed-up is close to optimal.

Moreover, this template provides an example of a job-submission script that will be typically required when using gUPPEcore on a large cluster with a queuing system.

Global observables, comparison with radially-symmetric simulation

Script *extract_observables* is utilized to obtain on-axis intensity versus propagation distance. It works the same way as in the previous example(s). Because the simulation is nominally equivalent to that in the radially symmetric case (see *wrk_031_Filament_In_Gas_Radial*), one should expect equivalent results. This is indeed the case, but users may be surprised that plots of the on-axis intensity reveal a small difference between the two runs. This is illustrated in the following figure:



Femtosecond filament in air. The apparent difference between the on-axis intensities produced by fully resolved and radially symmetric simulations is due to the fact that the built-in gUPPEcore observer has a different observation focus in the two cases. In the radial case, it produces the data for the grid point closest to, but not on the axis, while the focus is in the exact center for the 3+1D run.

The issue to keep in mind is that the actual points of observation are different. While the 3+1D simulation does have a grid point exactly located on the axis, the radial simulation does not. This is because the spectral solver utilizes the Hankel transform, and with it the radial grid which can't place a grid point at $r = 0$. Thus, the gap between the two curves is nothing but a variation of the solution between two close-to-axis points of observation.

Observation of the radial grid point closest to the axis is sufficient for most practical purposes. Should users require accurate on-axis observables, they can be obtained by extrapolation to $r \to 0$. For example, one could use first two radial points, together with the requirement that the on-axis value of the radial derivative must vanish.

It is left to the user to inspect other global quantities, such as the pulse energy, linear plasma density (integrated over the transverse cross-section), and the maximal on-axis plasma density. The first two show the expected agreement between the two simulations, while to on-axis plasma density value exhibits the same issue as the on-axis intensity.

All in all this example confirms that the spectral solvers used by gUPPEcore for radially symmetric and full 3+1D geometries are both reliable.

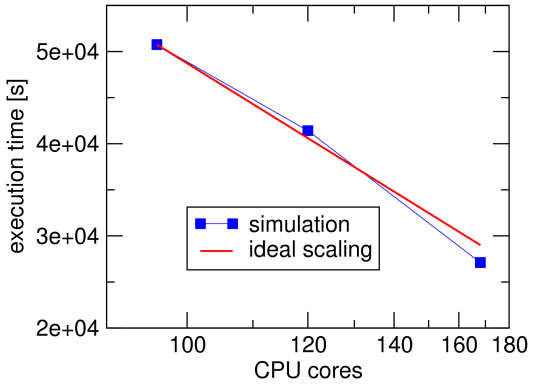
There are other simulation templates included in gUPPElab that utilize spatially resolved transverse computational domain. In particular *wrk_042_User-Addons-Initial-Conditions* is a closely related example in which the radial symmetry imposed in the present case is broken by the perturbation applied onto the initial pulsed beam. In *wrk_041_User-Addons-Medium-Response*, which models the Anderson localization of light, it is the weak disorder of the medium that requires a fully resolved approach.

Parallel performance

Of course, it makes no practical sense to simulate a radially symmetric solution with a full spatial resolution as we do it in this example. It takes more than a billion variables to describe the pulse waveform in the spectral space (and much more in the real space!) with full resolution. Consequently, the numerical effort required to solve the problem is incomparable. However, this is an example meant to showcase the parallel performance in gUPPEcore, while providing, at the

same time, a comparison test for the usage of different spatial domains. Moreover, the setup of this exercise can be easily adopted to situations without symmetry.

The parallelization model built-in in the gUPPEcore is based on the domain decomposition and Message Passing Interface. Its implementation is relatively simple, and aims mainly at smaller and medium-scale applications of this simulation framework. Nevertheless, the performance is very good as is illustrated in the following figure.



Parallel scaling in simulations performed with 96, 120, and 168 CPU cores and on UV2000 from Silicon Graphics with the total of 196 cores. The red line indicates the ideal behavior.

Our computational experience with parallel gUPPEcore jobs indicates that the parallel efficiency remains very good with up to 250 processes. We have not tested this simulation framework on clusters larger than that.

3.6 User Addons: Multiple Filamentation in High-Power Pulses
Pertubation Operator Addon, Checkpoints & Domain Size Issues

gUPPEcore usage illustrated in this example:

- using checkpoints to store a simulated wave-form snapshot
- using checkpoints to restart simulations
- extending gUPPEcore by adding a simple perturbation operator, used to modify the initial pulse
- this is a 3+1D simulation (relatively large-scale, with 27 million degrees of freedom) run on a personal workstation

Computational issues illustrated:

- domain-size issues, and related artifacts at long propagation distances

Physical context:

- Multi-filamentation regime

Input files

- *inputA*: The first simulation in a series of three. Simulates the first ten meters of propagation, and stores a checkpoint-file to be used by the subsequent run (inputB) as its initial condition. The initial pulsed beam is Gaussian, with additional perturbation, which is implemented in this folder. The perturbation represents an imperfect beam, and induces a multi-filamentation regime.
- *inputB*: The only difference from the previous input file (execute: diff inputA inputB) is that it specifies that a checkpoint file(s) produced by the previous simulation is to serve as its initial condition. Note that only the base of the checkpoint file-name(s) must be given in the input, followed by the index (number) which specifies which of the checkpoints to use. In this case, there is only one, labeled zero.
- *inputC*: This simulation starts from the checkpoint generated by inputB run. The sequence A,B,C thus "models" a piece-wise approach to a larger simulation. Having stored checkpoints at certain distances allows to adjust simulation parameters (for example those that control observation and how frequently is it done) between different segments.

Checkpoint usage

Checkpoint files serve to record the exact state of the simulated waveform. Note that there is no practical way to create a checkpoint outside of the simulator. First, we need to instruct gUPPEcore to produce a checkpoint:

Listing 3.13: Directing a simulation to produce checkpoints

```
In_report_4:
    Report4_period              5e+24
    Check_point_out[y/n]        y
```

Note that the second line simply states "yes" but does not specify any file names. This is because the names of files that collectively constitute a checkpoint are derived from the base-name of the given simulation run. Each MPI process stores its own slice of the total computational domain. *This means that the checkpoints can be only re-used with the same number of parallel processes.* Needless to say, the size of the computational domain also must not change between the different runs.

The first parameter in the above listing specifies how often is Report level four, which is nothing but saving checkpoint files, executed. Here we specify an effectively infinite distance which means that no checkpoints will be produced *during* the simulation. Only a single one, at the very end of the simulation run, will be saved.

To utilize it in the subsequent simulation, one needs to include the following in the input file:

Listing 3.14: Directing a simulation to start from a checkpoint

```
Initial_condInitial_conditions
    CKPT_in                     SIMDIR/A_CKPT_0
```

The name consists of the string equal to the base-name of the previous simulation run, followed by an index (here it is 0) which specifies which checkpoint set is to be read.

Note that the above directive is followed by what seems to be, and in fact is, an full initial condition specification. But it is unimportant, because the content of the checkpoint, if one is indeed read, takes precedence and replaces the initial condition. This is an example of having parameters in the input file which are ignored in the end. This may be confusing for new users, but on the other hand it makes different files easier to compare and in general to work with.

An important feature of checkpoints in gUPPEcore is that they do not store any "auxiliary" information, for example current integration step or propagation distance. All quantities other than the optical field snapshot are retained in the record files produced by the simulation.

Execution of simulations

gUPPEcore is parallelized with MPI, and must therefore be started with the program *mpirun* which comes with the given MPI implementation. In this example, we have included script *runthis*, which shows how the simulation set up in this folder should be executed:

Listing 3.15: Simulation execution, from a script

```
#!/bin/bash

#edit to set appropriate mpirun
MYmpirun=/home/kolesik/icpc-openmpi/bin/mpirun

#clean the simulation directory
rm -f SIMDIR/A_*
rm -f SIMDIR/B_*
rm -f SIMDIR/C_*
```

```
10
11 ${MYmpirun} -n 16 guppeA.out inputA  > recA
12
13 ${MYmpirun} -n 16 guppeA.out inputB  > recB
14
15 ${MYmpirun} -n 16 guppeA.out inputC  > recC
```

Usually, the standard output produced is stored in "record file" (e.g. *recA*). One reason for doing this is that the top portion of such a record can be subsequently used to reproduce the original simulation. Each of the above commands must finish before the next starts automatically. In practice, the three would be started manually, after inspection of the results produced by the preceding run.

Invoking user-defined capabilities

There are several classes of objects that can be implemented by the user and plug into the simulator. In this worked-out example, a simple perturbation is imposed "on top" of the initial condition. It is an example of an operator. Operators can be invoked in two stages of the simulation, either before or after the main simulation loop:

Listing 3.16: Input section: Invoking a user-defined operator

```
1  UPPE_Operator_Prolog
2     num._of_items              1
3
4     type_id_string             Perturbation
5       Perturbation
6       rng_seed                 345
7       number                   100
8       amplitude                0.5
9       box_x                    0.012
10      box_y                    0.012
11      corr_length              0.010
12
13   UPPE_Operator_Epilog
14      num._of_items             0
```

As indicated in this input snippet, there can be an arbitrary number of operators applied in the epilogue and/or prologue. Their implementation follows the same scheme, which is illustrated in this folder:

Implementing user-defined capabilities

All user-defined additions to gUPPEcore must behave as any other object that the simulator knows. They are required to be derived from the objects that carry named parameters, i.e. NamedParameterList class (users should inspect the *parameter_input_output.h* in the include folder), and must implement certain functions depending on their purpose.

There are two components to all user-defined additions. The first is the source included in *user_addons.cc*. Its purpose is essentially to provide a list of used-defined classes, and assign to each a name which will identify the class in the input. Users have to compile this, and link it

with the gUPPEcore library to obtain a modified executable. There are four areas which allow additions: Initial conditions, Operators, Medium-response plug-ins, and Linear propagators. All user additions must appear in *user_addons.cc*.

The second component is the implementation of the add-on. In this example it is implemented in *user_operator_screen.h*, which file is included in *user_addons.cc* for compilation.

We recommend that users interested in adding capabilities to gUPPEcore first read the commented sources provided in the various add-on directories within the source folder. The examples worked-out in those folders only contain the very minimum in terms of files, and should be easier to study than the present example which must carry a number of files that are not directly related to the add-on implementation.

Simulation in 3+1 dimensions

The setup of the present simulation shows an example of a fully spatially resolved case. 3+1D simulations are always large scale. However, many interesting cases can be handled on personal workstation type hardware as this example illustrates.

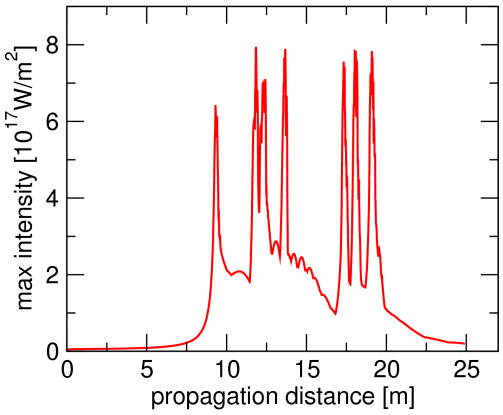
With large-scale simulations, one should be careful not to overwhelm the hardware used. After the program starts, it creates an informative report like this one:

Listing 3.17: Standard output section: Grid size and parallel information

```
1   # Report: gUPPE_Domain_Temporal
2   # active frequencies: 208
3   # thread chunk:       13 (16 threads, 1 components)
4   # minimal index:      24
5   # maximal index:      231
6   # reference index:    112
7   # reference lambda:   8.03016e-07
8   # reference omega:    2.34572e+15
9   # omega resolution:   2.0944e+13
10
11  # Report: gUPPE_Domain_Spatial
12  # active grid points: 65536
13  # total grid points:  65536
14  # thread chunk:       4096
15
16  # Report: gUPPE_Domain
17  # total (real) eqns:    27262976
18  # data view_1 geometry: 208 x 4096
19  # data view_2 geometry: 13 x 65536
```

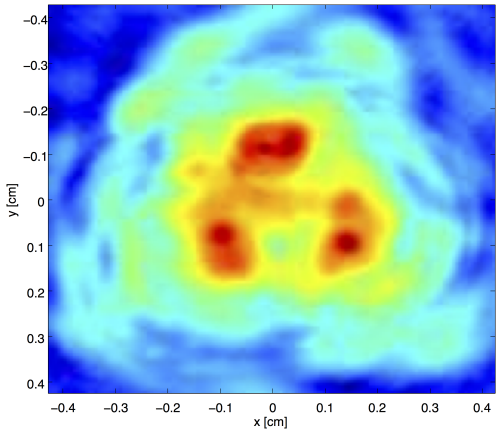
These comments included in the standard output list several quantities that give the user an idea about "how big" the given simulation actually is. In particular, the total number of equations, or degrees of freedom characterizes the (distributed) array needed to hold a single snapshot of the waveform. Depending on the ODE solver method, several auxiliary arrays of this size will be also allocated. This constitutes the major component of the memory usage in gUPPEcore. Users should inspect *inputA*, especially the temporal and spatial computational domain sections, and correlate it with the size and running times (in the bottoms of the record files) of this example simulations.

The perturbation imposed on the initial beam in this example initiates the modulation instability which in turn seeds multiple filaments, appearing seemingly randomly at different points in space and time. The following figure illustrates the evolution of the maximal intensity observed in such a multi-filament waveform along the propagation distance:



Maximal intensity versus propagation distance. Multiple re-focusing events, generally located at different point in the transverse cross-section of the beam manifest here as peaks in the maximal intensity.

This simulation models a pulsed beam confined in a computational domain 1cm×1cm. Hot spots appear "randomly" on top of the wider beam profile, and achieve significant intensities. To appreciate the overall structure of the waveform, which spans a large dynamic range, the following figure utilizes logarithmic scale to depict the transverse profile of fluence (total energy per unit area):



Log-scale map of the transverse spatial profile of fluence in a multiple-filament beam. Different hot spots correspond to filaments that attain their maximal intensities at different propagation distances, evident in the previous figure.

Important for re-curring filaments, the low-intensity background visible in the above picture stores a major part of the beam energy. It acts as a reservoir with which the hot spots exchange their energy. This kind of dynamics makes it possible to "by-pass" losses of energy which would need to be expended for continuous, static filaments.

Domain-size related issues

The fluence map shown above was recorded toward the end of the simulation run, and shows that there are parts of the waveform that are about to reach the vicinity of the transverse domain boundary. Should this simulation need to continue even further, the user would either employ larger domain (which in turn may require more grid points), or apply a soft artificial aperture to eliminate the weak outgoing component of the waveform in the vicinity of the computational grid

edge. In general, great care must be exercised in order to stop simulations before finite-size-domain artifacts set in.

3.7 User Addons: Anderson Localization of Light

Medium Response Addon & 2D Cross-section from Observer

gUPPEcore usage illustrated in this example:

- application of a medium-response plugin
- instructing the built-in gUPPEcore observer to produce two-dimensional transverse crossections of pulsed beam fluence

Physical context:

- Anderson localization of light

Input files

- *inputA*: This input represents a narrow, collimated pulsed beam entering a medium with random refractive index inclusions. The disorder leads to Anderson localization which "arrests" the diffraction.

User-defined medium response

In order to make it easier to utilize files in this example as a start-point for other user-defined plug-ins, the implementation has been moved into the *src/addons_medium_response*. The headers and source in the folder are meant to serve as a template.

The input section specific for this simulation template is in the following:

Listing 3.18: Input section: Parameters specifying randomly placed index inclusions

```
1  Medium_response
2     num._of_items                1
3
4    type_id_string                WeakDisorder
5
6       Index_Perturbation
7       rng_seed                   54123
8       number                     4800
9       delta_chi                  0.0001
10      radius                     0.000025
11      box_x                      0.0056
12      box_y                      0.0056
13      map_file                   map.dat
```

In this case, there is only one item comprising the list of medium response functions. The named parameters that appear above are defined in the plugin header file (*src/addon_medium_response/user_medium_weakdisorder.h*), and constitute an example of the mechanism gUPPEcore utilizes to "connect" the users input with the objects that execute various tasks in the simulation.

This medium-response plugin represents 4800 refractive index inclusion with a small, 0.0001, variation of local medium susceptibility. The size of each randomly placed inclusion is 25 micron. For debugging purposes, this object saves map of the disorder in the file specified.

The propagation of the pulse is purely linear, despite the add-on being part of the "nonlinear" polarization response. However, nonlinearity, for example the optical Kerr effect, can be easily included by simply adding mode medium response objects.

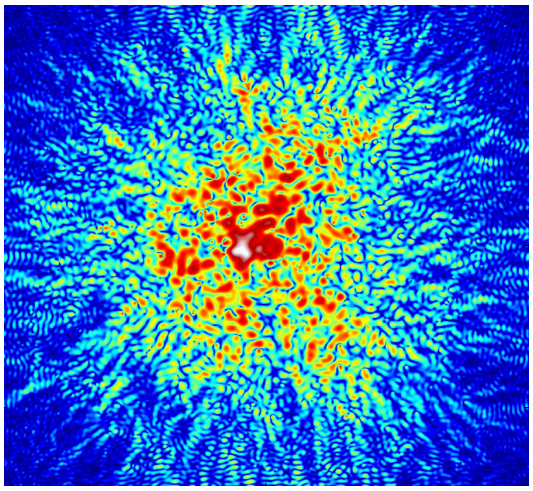
Simulation in 3+1 dimensions

The setup of the present simulation shows an example of a fully spatially resolved case. The record file of the run, *job-A_10x6.log*, indicates that the calculation was executed with sixty parallel processes, collectively utilizing 500 million degrees of freedom to represent the optical pulse.

Listing 3.19: Standard output section: Grid size and parallel information

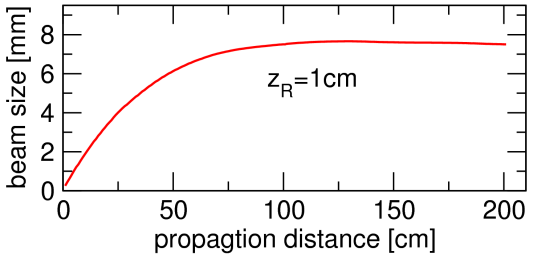
```
1   # Report: gUPPE_Domain_Temporal
2   # active frequencies: 60
3   # thread chunk:       1 (60 threads, 1 components)
4   # minimal index:      72
5   # maximal index:      131
6   # reference index:    112
7   # reference lambda:   8.03016e-07
8   # reference omega:    2.34572e+15
9   # omega resolution:   2.0944e+13
10
11  # Report: gUPPE_Domain_Spatial
12  # active grid points: 4194304
13  # total grid points:  4194360
14  # thread chunk:       69906
15
16  # Report: gUPPE_Domain
17  # total (real) eqns:    503323200
18  # data view_1 geometry: 60 x 69906
19  # data view_2 geometry: 1 x 4194360
```

The transverse localization of light is illustrated in the following figures. The first shows the beam profile as it evolves due to diffraction in the disordered medium. We use the logarithmic scale in which the characteristic signature of beam localization is the linear decrease of the light intensity away from the center of localization.



Anderson localization of light. This is a log-scale map of the transverse profile of fluence in a pulsed beam propagating through a medium with weak refractive-index disorder. The area shown is 6×6 mm.

A more quantitative way to visualize the transverse dynamics of the beam is to look at a measure of its diameter. Thus is shown in the following figure. The saturation in the curve is a sign that the localized propagation mode has been formed.



Beam-size as a function of propagation distance. Initially fast diffraction, due to a small waist size, slows down and is eventually arrested by the interaction with the random refractive index profile.

Of course, this illustration has not even scratched the surface of transverse localization of light. However, the intent here is to show an example of adopting gUPPEcore to a particular subject or situation.

3.8 User Addons: Airy Pulse Propagation

User-defined Initial Conditions

Airy pulse implementation

The */templates/wrk_042_User-Addons-Initial-Conditions* folder contains the working directory for theuser-defined initial condition (Airy pulse) implemented in the file */src/addons_initial_conditions*.

The corresponding executable */src/addons_initial_conditions/guppeAddInitialCondition.out* should be copied to the working folder.

To check the initial condition, users will usually execute a mock simulation, and interrupt it after the first Report level 2 is executed. Then the following files:

SIMDIR/A_T_c0_0_r_0

SIMDIR/A_S_c0_0_r_0

will contain, respectively, the (T)emporal and radial (i.e. Spatial) profiles showing intensity, real, and imaginary parts of the pulse analytic signal.

The file *SIMDIR/A_FFS_c0_0_r_0* contains both the on-axis and and over-angle-integrated (i.e. total) spectrum shown on log scale.

The above files should be used to inspect the properties of the initial condition, and that it fits in the computational domain. Note that the present example, namely the Airy pulse, requires better grid resolution that experienced user may expect to be sufficient for the conventional Gaussian pulses.

3.9 Femtosecond Supercontinuum Generation on a Chip

Beam Propagation Addon, User-defined Geometry & Propagation

gUPPEcore usage illustrated:

- application to femtosecond nonlinear optics in waveguides

Illustrated computational issues:

- gUPPEcore augmented with a suitable Beam-Propagation-Method plug-in in the role of the linear propagator makes fully resolved (in space and time) simulations feasible with a modest computational effort

Physical effects:

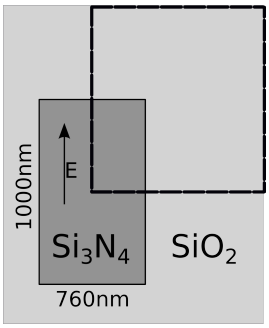
- Supercontinuum generation on chip

Simulation example

Fully resolved simulation of femtosecond-scale nonlinear optics in waveguide structures are computationally demanding. The purpose of this example is to showcase the gUPPEcore capabilities in this area.

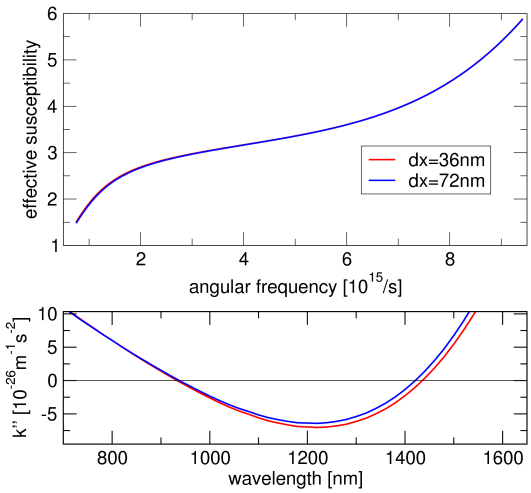
This kind of application requires implementation of the linear propagator, using one of the many available Beam Propagation Methods (BPM). In general, BPM methods, especially when applied to waveguide structures with strong contrast between constituent materials, are much less robust than for example spectral-propagation methods we applied in other worked-out examples. Accordingly, working with them in the extremely nonlinear regimes targeted by the gUPPEcore framework requires care and some experience. We therefore encourage interested users to contact authors to discuss possible propagator implementations and related practical simulation issues.

In this example we utilize a semi-vectorial alternate direction implicit method, second-order accurate in the propagation step, with fourth-order accurate discretization in the transverse computational domain.



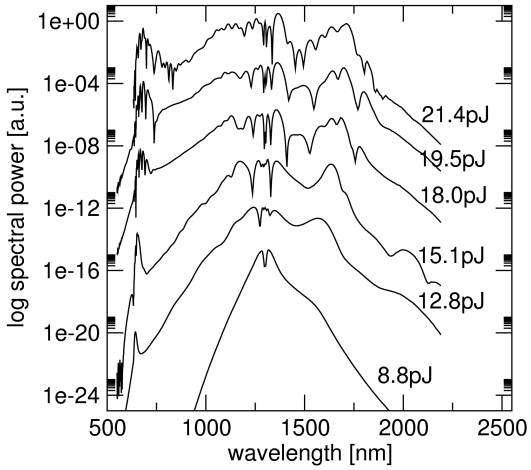
Simulated waveguide. Symmetric solution is assumed, so that the computational domain can be restricted to the rectangle indicated in the picture.

The properties of the simulated waveguide are first characterized by the same method which is to be used in the user-defined linear propagator. This is important for eliminating additional numerical dispersion which could affect the model dispersion properties.



Chromatic dispersion properties of the simulated waveguide. The top plot illustrates the tabulation of the effective waveguide susceptibility required for the linear propagator. The bottom panel curves represent group-velocity dispersion of the fundamental mode. Different colors show data obtained for spatial resolutions indicated in the top plot.

The most interesting observable in this numerical experiment is the supercontinuum spectrum, partly because it is simple to compare with experimental results. An example is shown in the following figure for a range of pulse energies:



On-chip generated supercontinuum for several output pulse energies.

The physics that governs the evolution of the spectrum is quite well understood (see also the gUPPElab example dealing with supercontinuum generation in microstructure waveguides). The characteristic feature seen in the above plots are the broad central peak, with two "satellite" structures, one in each wing of the spectrum. The latter features correspond to dispersive waves radiated by the central "soliton-like" pulse.

The simulation record file included in this folder should give the user an idea about the parameters of the simulation, especially grid sampling and resolutions, and the required execution time. Section specific to this example is that of the user-defined linear propagator (i.e. Spatial Domain).

3.10 User Addons: Multiple Filaments II

Complex Pertubation Operator & Periodic Boundary Conditions

gUPPEcore usage illustrated in this example:

- extending gUPPEcore by adding a complex perturbation operator, used to modify the initial pulse
- this is a more sophisticated version of another example (*wrk_040_User-Addons-Operators-Multiple-Filaments*) — it is meant to illustrate the extension technique for situations that require a bit more than a few lines in a header file.

Computational aspect(s):

- Very similar technique can be used to "teach gUPPEcore" how to simulate propagation through a turbulent medium.
- Usage of periodic boundary conditions in spatial domains that support a spectral transform.

Physical context:

- Multi-filamentation regime

Input files

- *inputR*: Models a collimated pulsed beam that passes through a random phase "screen." It imposes a perturbation on the phase-fronts of the initial waveform, which in turn results in creation of multiple filaments. This simulation "pretends" to simulate a small patch within a beam with very large (infinite) transverse dimensions. The idea is to eliminate effects related to the finite Rayleigh range of a laser beam, and concentrate on the mutual interaction dynamics of multiple filaments.
- Input file *inputR* also shows how a 2D fluence profile can be requested from the built-in gUPPEcore observer:

  S-spatial-profile[y/n] 2DF

  The built-in gUPPEcore observer creates output files that hold two-dimensional arrays with the fluence profile (over the whole transverse cross-section of the computational domain).
- The last section in the input file (Operator Prolog/Epilog) shows the input parameters encoded in the user-defined perturbation operator.

User-defined capabilities

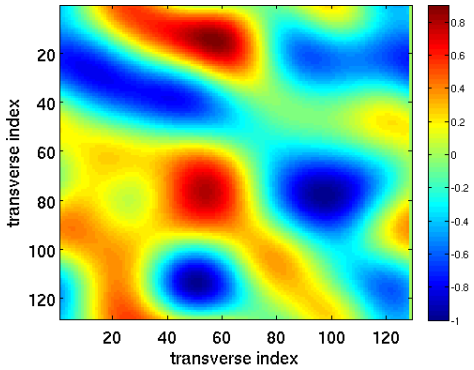
This simulation template is similar to that in *wrk_040_User-Addons-Operators-Multiple- Filaments*. However, here we have a better, more sophisticated implementation of the added perturbation operator. It is meant to be used with periodic boundary conditions (PBC) applied to the

transverse computational domain, so it creates a "phase screen" with PBC. This in turn requires slightly more code, so it is more convenient to split off the implementation into a separate compilation unit. Material in the folder */src/addons_operators* shows how this can be organized, and it should be straightforward to keep adding more user-defined capabilities in the same fashion.

Simulation in 3+1 dimensions

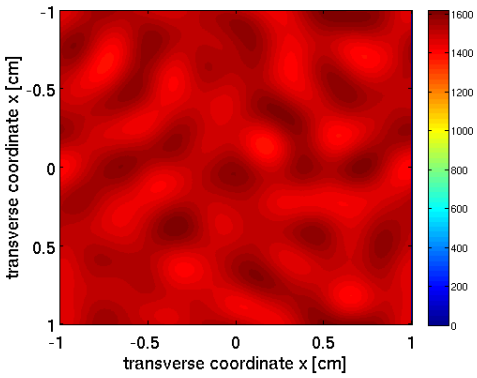
The setup of the present simulation shows an example of a fully spatially resolved case which should only be attempted on a medium-sized cluster. Here we provide examples of results.

Here is an example of the perturbation profile we use to seed multiple filamentation:



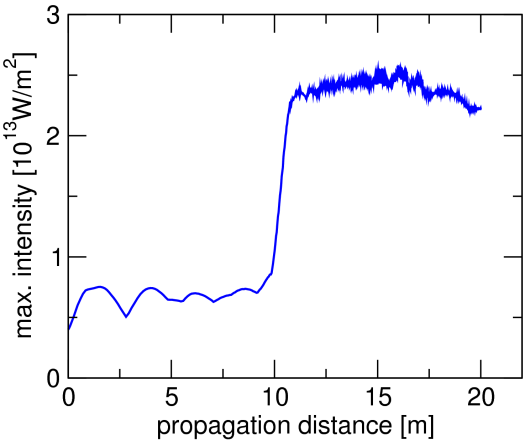
Normalized profile of the perturbation imposed on the initial phase front.

The initial condition proper, (i.e. the one that exists in the simulator *before* the operators listed in the prologue input section are applied to it) is flat an periodic. The user-added operator breaks this symmetry. After a short propagation the beam develops modulation instability driven by the optical Kerr effect. Here we show the fluence after one meter:



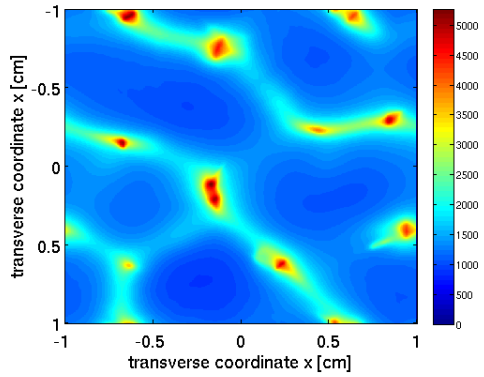
Fluence profile of the beam shortly after passing through a random phase-screen.

The above irregularities in the pulse are amplified by nonlinearity and eventually give rise to multiple hot spots, or filaments. The following Figure shows the maximal intensity along the propagation distance of the first 20 meters. Around z=10m, multiple filaments set in and persist over next 10m (and beyond but this is not shown here). Note that what we show here is a mid-IR regime (six micron wavelength) in which the filamentation mechanism is qualitatively different from that of typical 800 nm wavelength filaments.



Maximal intensity (taken across the transverse cross-section of the computational domain and temporal extension of the pulse) versus propagation distance.

To visualize the dynamics across the beam, the next Figure shows an example of the fluence profile after 15 meters of propagation.



Fluence profile of the beam after multiple filaments develop and "attract" energy from their vicinity.

The hot spots in this picture are filaments that persist for distances significantly longer than those in the better studied, well-known near-infrared filaments. The reason for this behavior is a qualitatively different mechanism that arrests the self-focusing collapse initiated by the Kerr effect.

A Visualization

The UPPE Observer report text files are structured for ease of analysis. Their structure should be compatible with almost any data analysis or visualization tool that the user is comfortable with. For those who do not have a preferred data visualization tool we present two options here.

A.1 (xm)Grace Plotting

Grace is a well established tool for creating publication quality graphics. Grace can be configured using command-line options, scripts or though its WYSIWYG GUI interface. In-depth use of Grace is beyond the scope of this guide. There are many on-line resources for both simple and advanced usages of Grace. More information about Grace can be found on the Grace website.

In the worked-out examples of this guide we do use Grace to generate plots. Two of these examples are shown here as a very basic introduction to Grace usage,

Basic example from wrk_010:

```
$ xmgrace -nxy SIMDIR/A_T_c0_10_r_0 -nxy SIMDIR/Acmp_T_c0_0_r_0
```

In this example -nxy tells xmgrace to expect 3 column data from the file that follows. This example also shows that you can stack "-nxy file1 -nxy file2" to plot multiple files on the same plot.

Advanced example from wrk_031:

```
$ xmgrace -nxy PlasA.dat -nxy DensB.dat -pexec 's0.y = s0.y/10E14'
  -pexec 's1.y = s1.y/10E22' -pexec 'autoscale'
```

In this example we scale the data from each file so that we can observe their profiles on the same plot. This is done with the "-pexec 'yData = yData/scale'" commands. The -pexec gives command line access to some of the GUI features of xmgrace. Once this level of data control is desired it begins to make more sense to either get into writing plotting scripts or using the GUI to make the plot appear as desired.

A.2 Python genPlots.py

For quick plotting of all of the UPPE Observer reports the included genPlots.py code can be used. The code will produce a plot for every polarization state and z-distance that is reported by the UPPE observer (see section B.2 for working with the observer).

This code is not meant to generate publication quality graphics. It is simply meant to quickly plot all data for an initial look at the results. Once this initial look is achieved, the user can determine how to plot the data for further analysis. If one is inclined to code in Python then the relevant sections of genPlot.py can be copied into the users code for further plotting.

A.2.1 Requirements

genPlot.py has been tested to work with both Python 2.7 and Python 3. It also requires the numpy and matplotlib packages associated with the Python version. Additional packages may be required based on your machine configuration and Python setup.

A.2.2 Configuration file

There is an associated (optional) configuration file named genPlots.cfg. The configuration file is assumed to be in the same directory as the genPlots.py script. If the configuration file is not found then the code attempts to generate the plots based on a set of default values. If the file is found then the code loads the following values from the file:

1. **plotList:** This allows the user to specify which plots are to be generated. For example, if the next variable (KOcdt) is changed, the user can only re-plot the KO plots instead of having to generate all the plots. Default: plotList = T,S,KO,FFS,TS
2. **KOcdt:** This variable allows the user to truncate some of the KO plot data. See the KO plot section below for more information. Default: KOcdt = none
3. **dataPath:** This is the relative path and directory name of the report files. An absolute or relative path may be specified here. Default: dataPath = ./SIMDIR/
4. **plotPath:** This is the relative path to a directory that will be created to house the plots that are generated. Default: plotPath = ./PLOTS/

A.2.3 Plot Generation Flow

The code is run by executing 'python genPlot.py', most likely from within the simulation directory. The code will first concatenate data from all of the Parallel CPU ranks (see C.2) into their associated propagation distance. Then it will calculate how many plots need to be created. If it is going to create over 100 plots it will ask the user if it should continue. The user should be sure they have plenty of disk space before agreeing to create a large number of plots. An overestimate of 250 kB for each plot should be used. If the user decides to continue then the code will generate plots for each propagation distance for each type of plot specified in the input file. The method in which the plots are generated are discussed in the next section.

A.2.4 Plots Generated

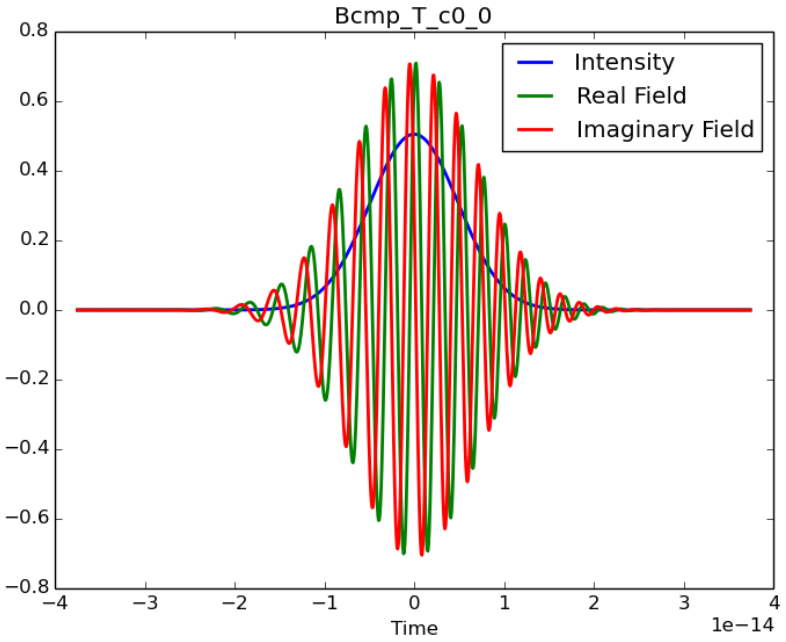


Figure A.1: Temporal Plot

Temporal plots are simple xy plot with the 3 y-axis functions shown. The x-axis is time while the y-axis is normalized field values. To get the non-normalized values the y-axis value should be multiplied by the square-root of the Intensity_unit in the UPPE observer section (see B.2) of the input file. Note that in the example above all 3 plot-lines are clearly visible. For longer pulses the oscillations may be so compact that all of the lines are not visible. In that case one may wish to change the Time axis to get finer resolution. See section A.2.5 for how to do that.

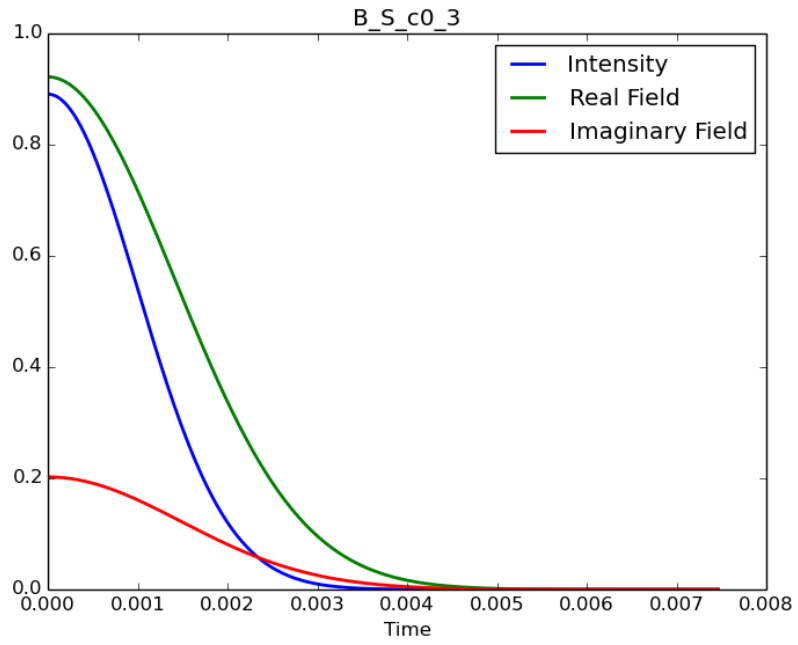


Figure A.2: Spatial Plot

Spatial plots are also simple xy plot with the 3 y-axis functions shown. The x-axis is time while the y-axis is normalized field values. Actual y-axis values can be found in the same way as in temporal plots above.

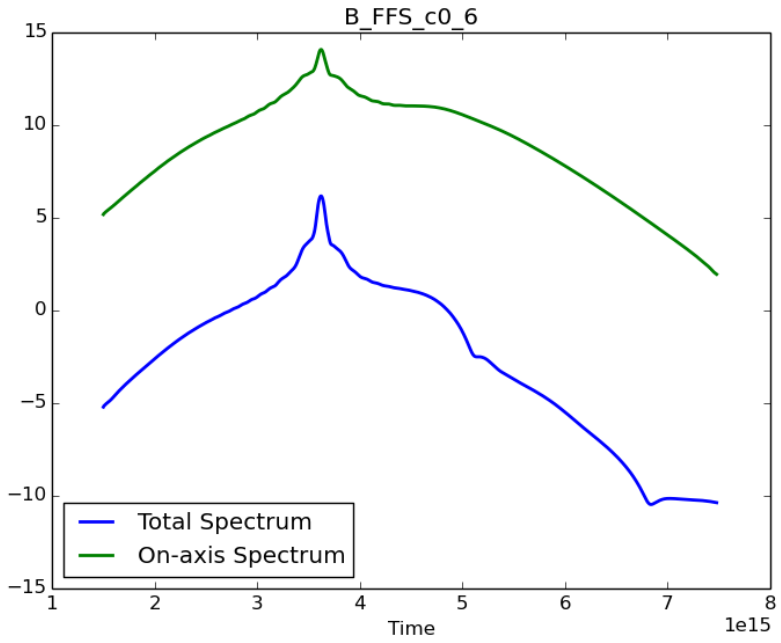


Figure A.3: Far-Field Spectrum Plot

Far-Field Spectrum plots are also simple xy plot with the 2 y-axis functions shown. Note that the y-axis in in arbitrary logarithmic units. They are non-physical. This is done so that once can compare spectra at different

distances.

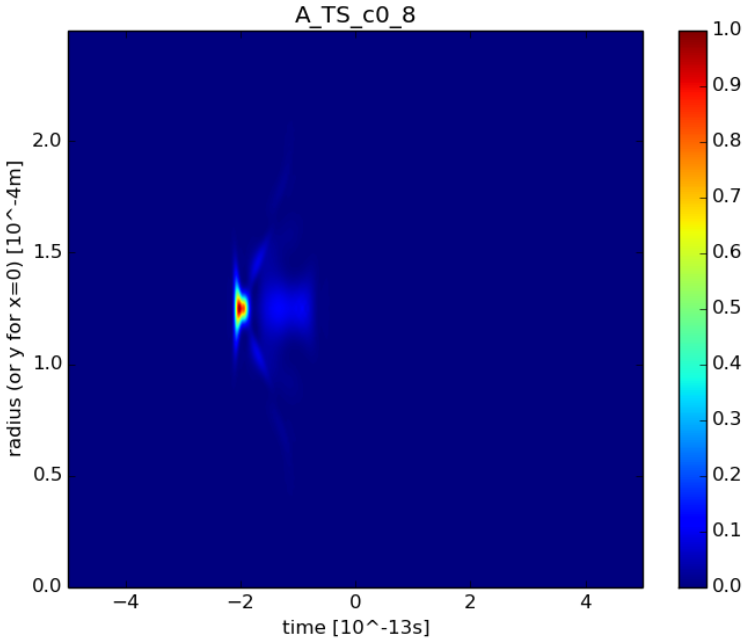


Figure A.4: Spatial-Temporal Plot

Spatial-Temporal plots are 2D plots with a colorbar providing the 3rd dimension. The axis are as labeled. The colorbar is normalized field values. Actual y-axis values can be found in the same way as in the spatial and temporal plots above.

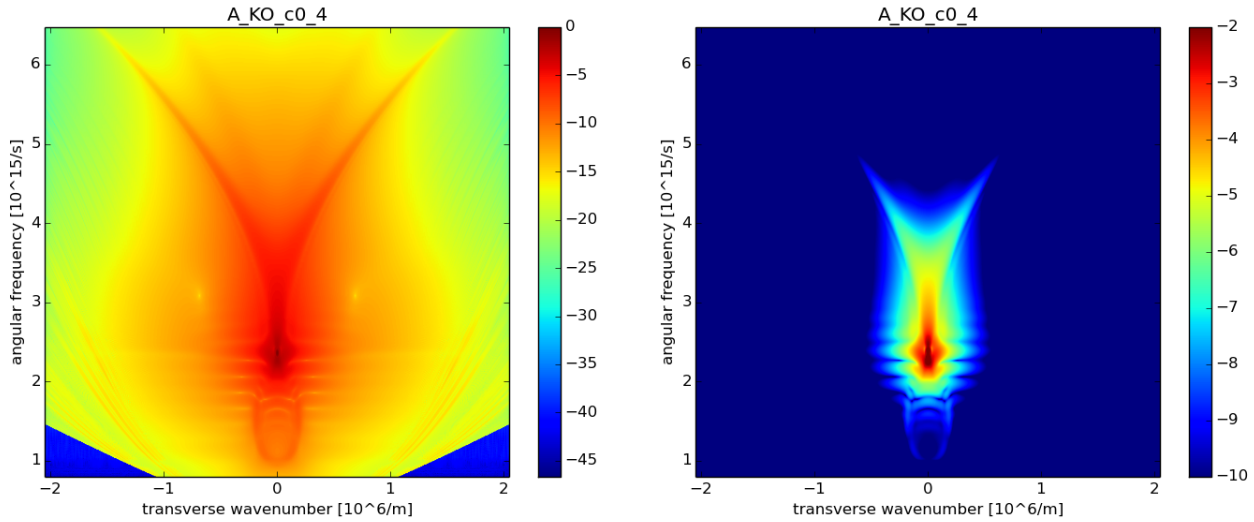


Figure A.5: Left: K-Omega Plot without truncation. Right: K-Omega plot with truncation at -10 and -2.

K-Omega plots are linear 2D plots of logarithmic data with a colorbar providing the 3rd dimension. The axis are as labeled. The colorbar range can be adjusted in the configuration file. The image above shows a situation where you would want to truncate some of the data for plotting. The image on the left shows that the full dataset

spans about 48 orders of magnitude. The yellow color that dominates the image is over 15 orders of magnitude smaller that the maximum signal. The image on the right shows where we have truncated the data above -2 and below -10. This is to mimic a lab situation where the intensity peaks are saturated so that data within the detectors response range (in this case 8 orders of magnitude) is resolvable. The orders of magnitude plotted is controlled by the KOcdt variable in the *genPlot.cfg* configuration file.

A.2.5 Editing the Plot code

The configuration file included with the code only allows for a few modifications to the plotting routine. If further plot customization is desired then one can edit the plotting code itself. The plotting code has been written in such a way that basic modifications such as titles, axis labels, x-axis data range and y-axis data range can easily be modified by someone with minimal Python coding experience. All of the plotting functions in genplot.py are at the beginning of the file. Each plot type has a function name beginning with plot. For example, FFS plots have a function plotFFS. Variables have been set-up in the various functions to allow easy modifications to the plots appearance. These will be discussed in the following subsections. More substantial plot edits will be left o the user. There are many online resources for Python plotting with matplotlib.

T, S & FFS plot editing

These plots are all plotted in the same manner. The plotting functions give the user 4 editable variables as shown below.

```
1 def plotS(title, dataFile):
2     title = title
3     labels = ['Time', 'Norm Field', 'Intensity', 'Real Field', 'Imaginary Field']
4     xLimits = []
5     yLimits = []
6     plotSimple(title, dataFile, '3', os.path.join(plotPath,'S/'), labels, xLimits, yLimits)
```

Variable:	How to edit
title	By default title is set to the filename that contains the data being plotted. For example: A_S_c0_0. To set a custom title change line 2 to **title = "Your Title in Quotes"**.
labels	Name of header file that user addon will be written in. The first 2 items in the list are the x-axis labe & the y-axis label. The remaining items are the plotted data labels that will show up in the plot legend.
(xy)Limits	By default these are empty lists. This will cause all of the data to be plotted. If you want to limit the data simply put 2 comma-separated numbers inside the square brackets. For example: **xLimits = [0.1, 4.4]**.

The plotSimple line in this function should not be edited. However, if one desires to completely hide labels or remove the legend then one can comment out (Python comments begin with #) the appropriate lines in the plotSimple function.

KO & TS plot editing

Editing these plots is a little more involved than the first plots. The actual line of plotting code looks like this:

```
1     im = ax.imshow(d1,
2         cmap=plt.get_cmap(),
3         extent=[tMin, tMax, rMin, rMax],
4         vmin = minV, vmax = maxV,
5         aspect='auto')
```

For these plot types there is no legend enabled. To change the text associated with the axes or the plot title look for the relevant lines in the plot code. They will be labeled plt.title, plt.xlabel, plt.ylabel. Simply change these to the desired values. The x & y limits are set by the extent variable on line 3. The order of the variables in the list is [xMin, xMax, yMin, yMax]. By default their values are set to the maximum extent of the data. To change this simply replace the values shown by the values desired. For example: extent = [0, 4, 0, 2]. Similarly, the vmin, vmax on line 4 control the limits of the colorbar. To change simply set vmin and vmax to the values desired. for the KO plots this can also be achieved by editing the configuration file instead of editing the code.

B Input Files

An input file is used to detail the simulation run to gUPPEcore. gUPPEcore always uses input from a single rigidly structured file. If the rigid structure is not followed gUPPE will let the user know what input is expected. Theoretically, a complete input file could be generated by passing gUUPEcore a blank input file and then adding parameters to it as the program complains. This would require a significant patience by the user. The following section shows compulsory section/subsection headings along with relevant parameters. Some parameters may vary depending on simulation configuration. Compulsory section/subsection headings must be in the input file, even if "empty." Note that the parameters listed in the input file are in MKS units unless otherwise specified.

B.1 gUPPE Parameters

All input files must start with the following line. This allows gUPPEcore recognize the input file.

```
gUPPE_Parameters
```

ODE Driver

The first section of an input file is the ODE driver section. The driver section specifies how the ODE solver should behave, propagation distance to simulate, simulation run base name etc.

```
ODE_driver
  base_name      A            # Output file base-name.
  z_step         1.0e-03      # [m] Initial integration step
  z_stop         0.1e-00      # [m] Final propagation distance
  tol_abs        1e+04        # [m] Integration step control: Note 1
  tol_rel        0            # [m] Integration step control: relative error
  max_step       0.05         # [m] Maximal z-step integrator can use
  method         rkf45        # ODE method: Note 2
  step_type      adaptive     # Adaptive or fixed
```

Note 1: tol_abs is the setting for integration step control absolute error. This parameters value will depend on the field normalization in the SPECTRAL representation. It is best to treat these values as purely "experimental" because it is difficult to predict what a reasonable parameter is in any given case. Absolute tolerances may seem rather high, but this is because electric field unit used is V/m, and we are often dealing with high intensities.

Note 2: The method is the ODE method to use. NEVER try to use an (implicit) ODE method that requires Jacobian! This is because the underlying ODE systems are HUGE, and such methods simply do not work for systems that may contain many millions (sometimes more than a billion!) variables. (see uppe_solver.h header for implemented ODE methods)

B.2 UPPE Observer

The next section is the UPPE_observer section. The observer section controls several "levels" of outputs, from inexpensive at level 1 to full check-point files at level 4. This section has six compulsory sub-sections.

```
UPPE_observer
  Intensity_unit              1e17    # [W/m^2] Intensity unit for output files
  In_report_1:
    Report1_period            0.01    # [m] How often report is written
  In_report_2:
    Report2_period            0.01    # [m] How often report is written
```

```
    T-temporal-profile[y/n]      y        # Time, intensity, real part, imag part
    S-spatial-profile[y/n]       y        # At observation focus: Note 1
    FFS-Far-field_spc[y/n]       y        # FF & angle-integrated (total) spectrum
    M-medium_response[y/n]       n        # Medium response: Note 2
  In_report_3:
    Report3_period               0.1      # [m] How often report is written
      KO-spectrum[y/n]           n        # Angle resolved spectrum: Note 3
      Spatial-Temporal-Map[y/n]  n
      Spatial-Omega-Map[y/n]     n
  In_report_4:
    Report4_period               0.0      # [m] Zero here means report never written
      Check_point_out[y/n]       n        # Note 4
  Medium_focus                   1        # Note 5
```

Note 1: The spatial profile is taken at the temporal position corresponding to the observation focus.
Note 2: The medium response can include, but is not limited to, the optical Kerr effect, plamsa...
Note 3: Angularly resolved spectrum where K is the transverse wavenumber and O is ω, the angular frequency
Note 4: A 'y' here will save a final checkpoint file. See example 3.6 for checkpoint file usage.
Note 5: This variable selects a medium response to output in Report-level-1. The available medium responses are defined in the UPPE_medium_response section of the input file. In that section, if num._of_items is set to 2 then the valid values for this Medium_focus variables are 0 & 1. With 0 being the first medium response defined in UPPE_medium_response and 1 being the second. Set to -1 to not include any of the medium responses.

See Appendix C for a complete description of each report type.

B.3 gUPPE Domain Parameters

This section specifies the "computational box" and number of field components that it holds. It is broken down into subsections for time & spatial dimensions.

```
gUPPE_Domain_Parameters
  Domain_Temporal
    components                   1                # Number of polarization components
    guard_frequency_[m]          1e25             # [m] Boundary guard: Note 1
    Axis_Parameters
      axis_type                  AXIS_TEMPORAL    # Note 2
      real-space_size            750.0e-15        # [seconds] temporal domain length
      number_of_points           4096             # Sampling points: Note 3
      init_type                  2                # Note 4
    T_Domain_Dispersion                           # Subsection for type-TEMPORAL axis only
      lambda_reference           2000.0e-09       # [m] Reference wavelength
      omega_reference            2.356194e+15     # This is calculated from lambda_ref.
      omega_min                  0.500000e+14     # Minimal active angular frequency
      omega_max                  5.500000e+15     # Maximal active angular frequency
      index_phase                1.375161e+00     # Note 5
      index_group                1.499803e+00     # Note 5
      gvd                        2.315337e-26     # [s^2/m] Note 5
      susceptibility_file_0      airchi.tab       # Susceptibility file-name: Note 6
    Domain_Spatial
      domain_type                DOMAIN_RADIAL    # Note 7
      Domain_radial
        Domain_axis
          axis_type              AXIS_RADIAL      # RADIAL or LINEAR
          real-space_size        2.5e-03          # [m]
```

```
    number_of_points        512             # Sampling points: Note 8
    init_type               0               # For RADIAL, sets Bessel order
```

Note 1: Boundary guard in real space applied periodically after this distance. This should only be used RARELY! In this example, case the parameter effectively eliminates application of the guard...
Note 2: Valid axis types: AXIS_LINEAR, AXIS_RADIAL or AXIS_TEMPORAL
Note 3: Number of sampling points: This must be an "FFT-nice" number! An FFT-nice number is generally a number that is the product of small primes.
Note 4: 0→default choice of FFT, 1→better, 2→better yet (best choice), 3→do not use, takes too long
Note 5: These three values would be used to calculate NLS-like dispersion ONLY IF susceptibility table file not specified. Normally we set these numbers negative, and they are re-calculated.
Note 6: File-name in which the susceptibility tabulation is stored, includes one line for each component.
Note 7: Type can be DOMAIN_RADIAL or DOMAIN_LINEAR_x_LINEAR. For DOMAIN_LINEAR_x_LINEAR both an x & y Domain_axis must be defined. See example following this section for DOMAIN_LINEAR_x_LINEAR usage.
Note 8: See worked out example 3.3 for case where a small number of sample points will require the use of n=1 (number of threads/processors) in the simulation

```
# DOMAIN_LINEAR_x_LINEAR example
gUPPE_Domain_Parameters
  # Sections not shown identical to above example
    Domain_Spatial
      domain_type               DOMAIN_LINEAR_x_LINEAR
      Domain_linear_x_linear
        Domain_axis_x
          axis_type             AXIS_LINEAR
          real-space_size       0.015   # [m]
          number_of_points      1024    # Sampling points: "FFT-nice"
          init_type             2       # For LINEAR same as Note 4 above.
        Domain_axis_y
          axis_type             AXIS_LINEAR
          real-space_size       0.015   # [m]
          number_of_points      1024    # Sampling points: "FFT-nice"
          init_type             2       # For LINEAR same as Note 4 above.
```

B.4 Initial Conditions

This section passes initial condition parameters to gUPPEcore. This is generally the input pulse. In the worked out examples there are examples for Gaussian_x_Gaussian 3.4, Secanthyp_x_Gaussian 3.3 and for implementing a user-defined initial condition 3.8.

```
Initial_conditions
  CKPT_in                   none                # Filename of check-point file: Note 1
  num._of_items             1                   # Note 2
  type_id_string            Gaussian_x_Gaussian     # Note 3
    Initial_secanthyp_pulse                     # Note 4
      field_component       0                   # Component of optical field to be ADDED to
      space_[R/S]           R                   # Real or spectral space
      input_intensity       60.0e+13            # [W/m^2]
      wavelength            8.0e-07             # [m]
      pulse_distance        0                   # [m] Distance to propagate before evaluating
      time_shift            2e-12               # [s] Temporal domain shift
      pulse_duration        100.0e-15           # Tau in the Gaussian exponential
      pulse_waist           0.0                 # [m] Waist parameter w
```

```
  temporal_s_order   1              # Supergaussian order: Note 5
  radial_s_order     1              # Same in radius
  pulse_focus        0              # [m] 0 for infinite focal length
  carrier_phase      0              # Carrier-envelope phase
  pulse_chirp        0              # Chirp parameter
  pulse_nb           1.0            # Note 6
  pulse_ng           1.0            # Note 6
  pulse_gvd          -5.0e-26       # Note 6
```

Note 1: File name of a check-point file to read. It will override any initial condition specified in the input file. 'none' means that no ckp file will be read. See example 3.6 for checkpoint file usage.

Note 2: More than 1 item can be specified! Each item will have it's own subsection. Items may be of different types. They are either predefined in gUPPEcore standard initial condition set, or users can define their own initial condition objects. Initial condition objects have two purposes: to hold parameters & to define and implement the initial condition function.

Note 3: This identifies the class to be instantiated at this point in reading the input. It is an example of an initial-condition object included in gUPPEcore. This case uses direct product of temporal and spatial Gaussians.

Note 4: This is just a name defined in the corresponding header file.

Note 5: 1 = Normal Gaussian, 2 = Gaussian^2, etc...

Note 6: These three values are used ONLY if pulse_distance is not zero. They are used to parameterize the linear propagation formula.

B.5 Medium Response

Several objects can be included in the section for medium-response models. Each item in this section is a named-parameter list which has a function that computes the nonlinear response to the given temporal profile of the optical field. User-defined medium-response objects can be included in this section. This example shows the "Standard Filamentation Model" medium-response. The two objects listed here are included in the gUPPEcore, but it is intended that users normally define their own medium-response objects.

```
UPPE_medium_response
  num._of_items          2          # Note 1
  type_id_string         Kerr_effect      # Type ID - Class ID
  Kerr_effect                       # Item name
    on/off               on         # Use off to switch off
    n2                   1e-23      # [m^2/W] Nonlinear index value
    nb                   1          # 'background' refractive index
    carr_res_[y/n]       y          # Note 2
  type_id_string         MPI_and_Plasma
  Plasma
    on/off               on_plasma    # Plasma on or off
    background_density   0            # Plasma density in background
    collision_time       3.5e-13      # [s] Controls damping in the Drude current model
    E_g_[eV]             11           # [eV] Used to estimate MPI losses
    source_1_on/off      on_source_1  # Two MPI sources:
    neutral_density_1    5e+24        # Number density per cubic meter
    mpi_cross-section_1  8.85e-105    # [m^2] Both sources are fitted with an
    mpi_order_1          6.5          # effective power law
    source_2_on/off      on_source_2  # ... same for source two
    neutral_density_2    2e+25
    mpi_cross-section_2  7.9e-124
    mpi_order_2          7.5
    response_mode        j_mode       # Preferred: driven by current
    ref_omega            2.35e+15     # Only used in sigma-mode
    mpi_loss_on/off      on_mpi_loss  # Loss due to MPI
```

```
  avalanche_on/off      off          # Not used in very short pulses
```

Note 1: num._of_items = how many components of medium response are specified in this input file
Note 2: Values y/n select if the nonlinear response is driven by cycle-averaged intensity (n) or by electric field with a resolved carrier wave (y). NOTE: use 'y' to include harmonic generation.

B.6 Operator Prolog

Operators can be applied to the initial field. They are executed after the "raw" initial condition has been set. UPPEcore runs user-specified operators acting on the optical waveform, this is done twice:

1. in Operator_Prolog
2. in Operator_Epilog

Operator_Prolog is a series of "transformations" applied AFTER initial conditions are introduced in the computational domain, and BEFORE the main evolution solver loop is entered The idea is to implement experimental optical elements such as apertures, polarizers, lenses, axicons, filters, etc. gUPPEcore implements only a few basic operators, and user-defined transformations can be added. Consult the simulation template dealing with user-defined operators to see how this can be done and what operators are already defined in the gUPPEcore.

```
UPPE_Operator_Prolog
  num._of_items          0       # In this case there is nothing to do in Prolog
```

B.7 Operator Epilog

Operators can also be applied after the final requested propagation distance has been reached. One can think of this section as a model for measurement "apparatus." For example, one can apply color filters or apertures before final measurement is done. Operator_Epilog is a series of "transformations" applied AFTER the main evolution loop has been executed. The set of applicable operators is the same as in Prolog. The idea of Epilog is to implement the "detection" part of an experiment.

```
UPPE_Operator_Epilog
  num._of_items 4                                  # Four elements from this input
    type_id_string              Operator_Lin-Prop  # Linear propagation
      UPPE_Operator_Lin-Prop
        distance                0.7                # [m] ...for this length
    type_id_string              Operator_BandPass  # Optical filter
      UPPE_Operator_BandPass
        omega_min               4.25e+15
        omega_max               5.00e+15
        component(s)            0                  # Polarization Component: Note 1
    type_id_string      Operator_Report            # Observer report
      UPPE_Operator_report
        report_type[1,2,3,4]    2                  # Note 2
     type_id_string     Operator_Polarizer         # Polarizer operator
      UPPE_Operator_Polarizer
        component_1             0                  # Specifies two polarization axes
        component_2             1
        angle_[deg]             +45.0              # ... and orientation
```

Note 1: Select polarization component to apply to, choose -1 to apply to all.
Note 2: In this case, Report_2 saves samples of the optical waveform this may be useful for inspection between optical elements.

C Output Files

gUPPEcore produces output files form 2 sources. The first source is the standard output stream. gUPPEcore directs lots of useful information into the standard output stream. This is usually captured in a *rec* file as shown in section 2.3. The top of the recA output is an exact re-statement of the input, and as such it can be used to re-create an identical simulation. The record file is thus a convenient mechanism to store initial properties of the simulation together with some global observables it calculated.

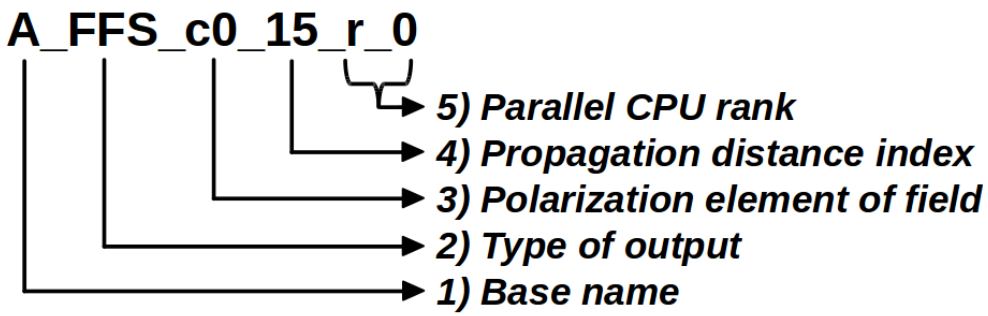
C.1 UPPE Observer Reports

The second kind of output are files written by the UPPE_observer as specified in the input file. These files usually contain the human-readable results of the simulation. How often these files are written is determined by the UPPE_observer section of the input file (as shown in B.2). They are organized into columns or hold a matrix (rectangular array). The first column is always the independent variable, while the following columns hold different quantities. Complex-valued quantities (field amplitudes) are usually stored in the following order: modulus squared, real part of amplitude, imaginary part of amplitude.

Abr	Name	Level	Description
DBG	Debug Output	-	Debug information. Generally not used.
T	Temporal Profile	2	On-axis temporal profile of the field. Report file columns: Time, Intensity, Real Field, Imaginary Field.
S	Spatial Profile	2	Radial or spatial profile of the field. Field taken at global intensity maximum corresponding to observation focus. Report file columns: Time, Intensity, Real Field, Imaginary Field.
FFS	Far-Field Spectrum	2	On-axis and angle-integrated (total) spectrum. Report file columns: Time, Total Spectrum, On-axis Spectrum.
M	Medium Response	2	Polarization and current medium response.
KO	Angularly Resolved Spectrum	3	Angularly resolved spectrum for RADIAL spatial domain only. Report file: 2D Matrix
ST	Spatio-Temporal Map	3	2D matrix slice spanning time & spatial axis. 2D case: time & radius. 3D case: time & y for x = 0. Report file: 2D Matrix
CKP	Checkpoint File	4	Contains complete field snapshot in binary. Used as a re-start point for simulations. Also used for user-defined observation. Since they hold complete information at a given propagation distance they allow the most general definition of observables.

C.2 Report Filename Convention

The observer reports have a very specific file naming convention. The convention is intended to give a description of what type of data the file contains. See the legend below for convention & description:



1) Output files start with base-name as specified in the ODE_driver section of the input file.
2) Type of report data in file. See table below for report data description.
3) c0, c1, ... labeling the (polarization) component of the field
4) integer labeling the Report - this maps to the propagation distance the mapping can be determined by extracting header REP2 from a record file - these lines specify the report label and their corresponding propagation distances.
5) r_integer specifies the parallel CPU rank. Most of the outputs will have this equal to "_r_0" because the rank-zero process creates consolidated files that collect information from all parallel mpi processes. However, two-dimensional outputs (rect. arrays) are kept split into their rank-portions. If needed different files XXX_r_yy can be concatenated into a single file for visualization. Often only the ranks that hold the central portion of the computational domain are needed for visualization, so it is efficient to save them rank-separated.

D Advanced Topics in User-Addons

A deeper understanding of the gUPPE user-addon system may be necessary for more complex situations. Topics related to these advanced topics are presented in this appendix.

D.1 C++ GNU Scientific Library

gUUPEcore includes the GNU Scientific Library, a.k.a GSL, in the *include/gsl* folder. GSL mathematical routines can be used when writing user addons by including the relevant header file in the addon code. More information about the GSL can be found on the GSL website.

D.2 From MATLAB to C++

Those who primarily use MATLAB codes may find some adjustment is needed to create mathematical C++ codes. Some examples that one may run into when writing addons are:

- The syntax for things like variable declaration and function calling differ.
- MATLAB "dot-operators", like .* and .^, are not used in C++.
- In C++, indexes (subscripts) are 0 based instead of 1 based.
- C++ functions only return single values while MATLAB functions can return many values.

There are other differences which are beyond the scope of this guide. There are many online resources, including the MATLAB official documentation, that discuss the differences between writing C++ code at MATLAB code.

D.3 The user-addon Operate function

In the Operate function shown in section 2.5.2 calculates a phase contribution for every frequency and polarization state. It then stores that phase in the following structure:

```
target[o*numC + c] *= phs
```

Where:

- target is the array on which the operator acts.
- o is the frequency integer index.
- c is the polarization integer index.
- numC is the number of polarization states, either 1 or 2.
- phs is the calculated phase.

This structure applies the results of each phase calculation into a uniquely indexed array. If there is only 1 polarization state (numC = 1) then the phase results are stored in sequential indexes. If there are 2 polarization states (numC = 2) then the first polarization state (polarization 0) is stored in even values while the other polarization state (polarization 1) is stored in odd values. This is the structure of the target array so the results of the operator must be presented in this same manner. If not then the operator will operate on unintended frequencies.