Simulation System For Optical Science (SISYFOS) – tutorial

Gunnar Arisholm and Helge Fonnum

Norwegian Defence Research Establishment (FFI)

31 October 2012

FFI-rapport 2012/02042

1178

P: ISBN 978-82-464-2141-4
E: ISBN 978-82-464-2142-1

Keywords

Ulineær optikk

Simulering

Optisk parametrisk forsterker

Approved by

Knut Stenersen	Project manager
Johnny Bardal	Director

English summary

Sisyfos is a program for simulation of optical parametric frequency conversion, lasers, and beam propagation in nonlinear or turbulent media. The main part of Sisyfos is a C++ library with classes corresponding to optical components such as beam sources, lenses, mirrors, and nonlinear crystals. By combining such components it is possible to build simulation programs for a wide range of devices, including amplifiers, oscillators, and harmonic generators. This report gives an introduction to Sisyfos by guiding the reader through a sequence of example programs of gradually increasing complexity. The example programs are an essential part of the tutorial, and a reader cannot expect to learn Sisyfos thoroughly without working with the examples.

Sammendrag

Sisyfos er et program for simulering av optisk parametrisk frekvensomforming, lasere og stråleforplantning i medier med ulineæritet eller turbulens. Den største delen av Sisyfos er et C++ bibliotek med klasser for optiske komponenter som strålekilder, linser, speil og ulineære krystaller. Ved å sette sammen slike komponenter er det mulig å lage simuleringprogrammer for ulike optiske systemer, inkludert forsterkere, oscillatorer og harmoniskgeneratorer. Denne rapporten gir en introduksjon til Sisyfos ved å føre leseren gjennom en rekke eksempelprogrammer med gradvis økende kompleksitet. Eksempelprogrammene er en essensiell del av innføringen, og leseren kan ikke vente fullt utbytte av rapporten uten å arbeide med eksemplene.

Contents

1	**Introduction**	**7**
1.1	Structure of Sisyfos	7
1.2	Representation of beams	8
2	**Installation**	**9**
2.1	System requirements	9
2.2	Installation procedure	10
2.3	Directory structure	11
2.4	Finding documentation	11
3	**Example 1 - A simple OPA**	**11**
3.1	Program structure	12
3.1.1	Parameters	12
3.1.2	Optical path	15
3.1.3	Initialisation and computation	16
3.2	Running Sisyfos from the command line	17
3.3	Retrieving beam data	18
3.4	Exploring the result file	20
3.5	Reading parameters from files	22
3.6	Help	23
4	**Resolution parameters**	**23**
4.1	Spatial parameters	26
4.2	Temporal parameters	27
4.3	Narrow- and wide-band mode	30
5	**Function classes**	**31**
5.1	Func1Any	32
5.2	Func2Any and Func3Any	34
5.3	SellmeierAny	35
6	**Example 2 - OPA with more flexible input beams**	**35**
6.1	Using beam data from simulation files	37
6.2	Changing solver parameters	39

6.3 Test program for AnySource 40

7 Example 3 - Advanced OPA 40

7.1 Overlapping spectral ranges 44

8 Example 4 - OPO 45

8.1 Thermal effects 50

9 Running Sisyfos from a python shell 53

9.1 Struct class 53

9.2 Run-functions 54

9.3 Predefined parameter structures 57

9.4 Multiple threads 58

9.5 Iteration for thermal effects 59

9.6 Running func_test, sell_test and source_test from python 60

Appendix A Introduction to optical frequency conversion 63

Appendix B Parameter syntax 65

B.1 Examples 66

B.2 Help and other special parameters 66

Appendix C Structure of Func1Any 67

References 68

1 Introduction

Sisyfos is a program for simulation of optical parametric frequency conversion, lasers, and beam propagation in nonlinear or turbulent media. The purpose of this tutorial is to introduce you to the basics of Sisyfos, with emphasis on frequency conversion, by means of examples. The target readership is physicists with a good knowledge of lasers and optical frequency conversion, but to make the report at least partly readable to nonspecialists, a brief introduction to nonlinear optical frequency conversion is given in Appendix A. While reading the tutorial you should study the example programs, run them, and inspect the results as shown in the text. You should also consult the html-documentation for at least some of the functions and classes used in the examples.

The rest of this section explains the structure and features of Sisyfos, and Section 2 describes the installation. Section 3 presents the first example program, which simulates a simple optical parametric amplifier (OPA), and shows how to run it and inspect the result files. To get valid results and efficient computation it is important to choose sensible resolution parameters, and Section 4 describes this issue and shows how to explore resolution parameters using the program from Section 3. Sisyfos uses various function objects to represent input data in 1, 2, or 3 dimensions, and these classes are described in Section 5. It is important to be familiar with the function classes because they are used for most input data such as pulse and beam shapes, absorption spectra and temperature distributions. Sections 6 to 8 present further example programs: A generalised version of the OPA from example 1, a broad-band non-collinear OPA, and an optical parametric oscillator (OPO). Section 9 shows how to run Sisyfos from a python shell. This is helpful to manage complex parameter structures and to run simulations in parallel. Sisyfos accepts parameters on the command line or in (possibly nested) text files, and Appendix B describes the syntax of such text parameters. Appendix C describes the design of the class Func1Any and uses this to introduce some important concepts from C++.

This tutorial applies to Sisyfos version 5.1.24, but except for some of the python modules, most of the features described here are expected to be relatively stable.

1.1 Structure of Sisyfos

The Sisyfos package consists of the following parts:

- A C++ library with classes corresponding to components such as lenses, mirrors, beam sources and crystals with nonlinearity or laser gain.
- Matlab functions for reading and analysing result files.
- Matlab functions for phase-matching calculations and other utilities.
- Python functions for reading result files. At this stage very few of the analysis functions have been ported to python.
- Python functions to run Sisyfos programs from a python shell.

The 'configuration file', which defines the device to be simulated, is the C++ main program. This must create objects corresponding to the components of the device and assemble them in a data structure. This is admittedly a rather complex way of describing the device, but on the other hand it is much more flexible and powerful than any ad-hoc configuration 'language' we could have developed with a realistic effort. Thanks to the power of C++, a single main program can be made flexible enough to simulate a variety of devices. Even if you are not going to write your own C++ programs, a certain understanding of these programs is necessary to use them correctly, and example programs are included with this tutorial.

Sisyfos can include most of the physical effects that are relevant for frequency conversion and lasers:

- Diffraction
- Birefringence
- Dispersion (to all orders)
- Linear absorption
- Thermal effects
- Nonparaxial beams
- Broad-band beams
- Degenerate or non-degenerate $\chi^{(2)}$ interactions
- Multiple parametric processes in the same medium
- Nonlinear refractive index (n_2) and two-photon absorption
- Non-collinear beams
- Laser gain (currently 3-level)
- Atmospheric turbulence
- Noise (semi-classical)
- Resonators

1.2 Representation of beams

The real electric field E of a beam is represented by a complex amplitude e,

$$E(x, y, z, t) = e(x, y, z, t) \exp[-i(\omega_0 t - k_{0z} z)] + cc, \tag{1.1}$$

where x, y are the transverse coordinates, z is the coordinate in the main propagation direction, t is time, the centre (angular) frequency ω_0 and wavevector component k_{0z} have been factored out, and cc means complex conjugate. In the program, the beam at a certain z-position is represented as a 3D array of complex numbers, where the dimensions correspond to x, y, and t.

Beam propagation is handled in (spatial and temporal) frequency space, where the beam is represented as a superposition of monochromatic modes

$$e(x, y, z, t) = \sum_{k_x, k_y, \omega} a(k_x, k_y, \omega, z) \exp[-i(\omega t - k_x x - k_y y)], \tag{1.2}$$

where k_x, k_y are transverse wavevector components, ω is an offset from the centre frequency, and a is a complex mode amplitude. The modes are usually plane waves, which are eigenmodes in free space or in birefringent crystals, but in case of cylindrical symmetry they are Bessel beams.

The details of the nonlinear propagation equations are outside the scope of this tutorial, but they will be described in [1]. The main assumptions in the derivation of the equations are

- The medium has no magnetic effects, free currents or charges.
- The field amplitudes can be treated scalars.
- Mode amplitudes change little over the distance of a wavelength. Note that this is much less restrictive than the slowly varying amplitude approximation (SVEA). Pulses can be very short and subject to rapid dispersive evolution, it is only the nonlinear coupling that is restricted.
- The nonlinear response is taken to be instantaneous (except the thermal part).

2 Installation

Users outside FFI are not allowed to use Sisyfos without an agreement with FFI. Please make sure that you or your group leader have such an agreement, and do not distribute Sisyfos to others.

This section covers installation of executable Sisyfos programs (exe-files) and the supporting matlab and python functions on Windows systems. (Users of the Sisyfos library, who can build their own Sisyfos programs, should read the file Installation-FFI.txt in addition to this section). Sisyfos can run on 32-bit or 64-bit Windows. Executables for 32-bit Windows can run on 64-bit Windows, but not vice versa.

2.1 System requirements

- Sisyfos is usually compiled with the Intel C++ compiler, and it requires the redistributable Intel library libiomp5md.dll, which is usually included with the Sisyfos distribution.
- The development is done partly on Linux and partly on Windows, so the text files do not have consistent line endings. An editor which can handle both Linux (LF) and Windows (CR-LF) line endings is required. We recommend Crimson editor or Notepad++, both of which are free.
- Matlab is necessary to use advanced analysis functions.
- For simple post-processing, python with numpy and scipy is sufficient. The python functions in Sisyfos work under version 2.7, they have not yet been ported to python 3. We recommend the distribution python(xy).

The example programs for this tutorial have been compiled with the Visual studio compiler for 32-bit systems and do not need the Intel library. However, they do need the Visual studio 2010 redistributable package, which can be downloaded for free. This is used by many programs, so it

may well have been installed on your computer already.

2.2 Installation procedure

Sisyfos is usually delivered in a zip-file or similar archive.

- Create a directory (typically called "...\Sisyfos5") and unpack the zip-file there. The subdirectories are described below.
- Install libiomp5md.dll. If you are only going to run in "...\Sisyfos\app" you can copy it to that directory. If you are going to run Sisyfos in multiple directories, put the library in a directory which is already in Windows' PATH, or put it in a new directory which you add to the PATH environment variable as shown below.
- Set the environment variable SisMatData to "...\Sisyfos5\MatData", where the path corresponds the Sisyfos directory created above. This directory contains Sellmeier equations and absorption data.
- If you use python, set PYTHONPATH to include "...\Sisyfos5\python".

On Windows 7, you can set and check environment variables in the command window by the commands:

```
set SisMatData=C:\Sisyfos5\MatData
echo %SisMatData%
```

There must not be space between the name and the '=' sign. Setting an environment variable in this way applies only to that single window. To set a variable permanently, go to control panel – system and security – system – advanced system settings – environment variables – user variables. Such changes do not affect windows that are already running, so a new command window must be started after setting or changing environment variables in the control panel.

To add the Sisyfos matlab files to Matlab's path, run the following commands in matlab:

```
sisyfosroot = '..../Sisyfos5/matlab'
addpath(sisyfosroot)
sisyfosstart
```

You can put these commands into Matlab's startup file startup.m to run them automatically when matlab starts.

2.3 Directory structure

Subdirectory	Content
app	Executable program. Can be used as working directory.
doc	Misc. documentation
doc\cpp	html documentation for C++ libraries.
doc\python	html documentation for python modules.
MatData	Material data, i.e. Sellmeier coefficients and absorption data
matlab	Matlab functions. Reading and analysing results, phase-matching calculations, etc.
python	Python modules. Reading and analysing results, running from a python shell.
tutorial	This tutorial and associated example programs.

2.4 Finding documentation

Details of the Sisyfos C++ library tend to change frequently, so the library is documented in html files rather than in a report. The root file of the html documentation is ...\Sisfos5\doc\sisyfos_main.html. To look up a C++ class, go to C++ modules, select the tab ”classes” and then ”class index”. The class documentation shows the methods that are defined by the class itself. Many classes inherit additional methods from base classes or interfaces, and to see these you can follow the link ”List of all members”. To find a function which is not in a class, select the tab ”namespaces”, then ”namespace members” and then the alphabetic tabs. The python modules are documented in the same way as the C++ modules. You should look at the documentation for some of the classes you encounter in the tutorial examples.

The matlab functions are only sparsely documented, and because they are due for revision they are not described in this tutorial. However, the classes for reading result files, ReadSis5 and gfm3, are almost identical to the corresponding python classes, so the python documentation applies except for minor differences in syntax. Some of the matlab functions for phase matching calculations are described in a separate text file in the tutorial directory.

The theoretical basis of Sisyfos will be described in a separate report [1]. Information about C++ can be found at [2] and information about python at [3] or in [4].

3 Example 1 - A simple OPA

The purpose of this section is to give a simple example using the minimal number of Sisyfos’ features. The resulting program is longer and less flexible than it could be, but simplicity is our priority at this stage. A more realistic program, taking advantage of more of Sisyfos’ features, will be shown in Section 6.

The example program for this section is called opa_ex1.cpp, and it is located in the directory ...\Sisfos5\tutorial\ex1. The program simulates a collinear OPA with a single nonlinear crystal and Gaussian input pulses and beams. The device is shown in Fig. 3.1. You should read the

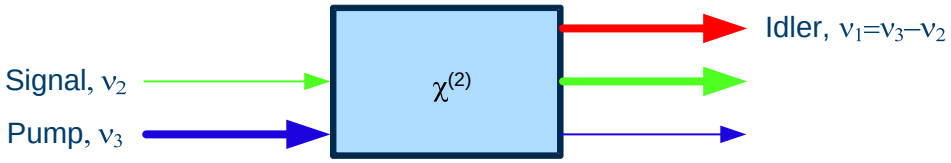

Figure 3.1 Optical parametric amplifier with nonlinear crystal, input beams and output beams. The colours of the beams indicate the relation between the frequencies.

program in parallel with the following subsections, but is not necessary to understand all its details. The most important points are:

- The section which defines the allowed parameters and how this is related to the parameters you can pass to the program.
- The structure of the optical path. Remember that the C++ main program is effectively the configuration file for the simulation.

3.1 Program structure

The program consists of three main parts with distinct tasks:

- Set up a parameter structure, which defines the allowed parameters to the program, their default values and allowed ranges. Like structures in Matlab or C++, parameter structures can be nested. Array values can have limits to the number of dimensions and to the number of elements in each dimension.
- Create an object of the class Path and fill it with objects corresponding to the optical components. You can find the list of components in the html documentation under Namespaces – Sis::Comp.
- Run the simulation.

Each of these parts are treated in the subsections below.

3.1.1 Parameters

The allowed parameters to the program are defined by a data structure built from ParamStruct and its related classes. Specifically, the data structure is a tree where the interior nodes are of class ParamStruct and the leaf nodes can be ParamString, ParamInt and so on for different data types. The tree structure supports structures inside structures, as in C or matlab. It is possible to set default values for all leaf nodes, limits for numeric nodes, and size limits for vectors and arrays.

The first parameter to be created is an integer called ”smode”, for spatial mode. Briefly, 0 means

plane-wave, 1 means cylindrical symmetry, 2 means full 2D, 3 means half-plane, and 4 means a single quadrant. These modes are described in detail in Section 4. The line

```
ps->add_field("smode", new ParamInt(2, 0, 4));
```

creates a ParamInt object with default value 2 and limits 0 and 4 and adds it to the root structure ps under the name ”smode”. The next lines are similar except that they define vectors:

```
ps->add_field("points", new ParamIntVec(false, 1, 2, 1, 512, 32));
ps->add_field("scale", new ParamRealVec(false, 1, 2, 0, 1, 1, 2e-4));
```

The first five arguments to ParamIntVec and ParamRealVec are common: A boolean flag tells if empty vectors are valid parameters, then comes lower and upper limits for the size (if not empty), and then lower and upper limits for the values. ParamRealVec then takes a scale argument (1 in this case) which ParamIntVec does not have, and the last argument is a default value. ParamIntVec and ParamRealVec have constructors for different numbers of default elements, and you should look them up in the html documentation for the classes. Once you have understood these first few parameters you have caught the essence of the parameter structure. You should be able to read limits, size limits and default values from the C++ program, but you do not need to understand these methods in detail.

The table below summarises the first few parameters, which are simulation parameters rather than physical parameters:

Parameter	Meaning
smode	Spatial mode.
points	Number of transverse points. Can have two elements for rectangular matrices.
scale	Spatial resolution, in m. Can have two elements for rectangular elements.
nt	Number of temporal sample points.
dt	Temporal resolution, in s.
mat_dir	Directory where Sisyfos can find material data.
ran	Seed for random number generator.
file	Output file name.

smode, points, scale, nt and dt are described in detail in Section 4. Sisyfos has a random-number generator that is used to generate semi-classical noise as an approximation to the quantum noise in real devices. The ’ran’ parameter is the seed for the random-number generator, and its default value is obtained from the clock so that it is different in every run. It can be overridden if you need to run multiple simulations with the same noise.

The next parameters, ’lamp’ and ’lams’, are the wavelengths of the pump and signal beam. The idler wavelength is calculated from the difference frequency. The parameters for the nonlinear crystal (BBO in this example) are collected in a substructure:

Parameter	Meaning
se	Name of Sellmeier equations. Refers to a file in "...\Sisyfos5\MatData\SE\BBO"
len	Length of crystal, in m.
d22	Nonlinear tensor coefficient d_{22}, in m/V.
theta	Polar angle θ of the beams with respect to the crystal axis, in radians.

The angle ϕ is fixed to $\pi/2$ in this example. The 'bbo' substructure is finally added to the top-level structure like any individual parameter. The next two substructures contain parameters for the pump and signal beams. Both contain the same field names, and this is allowed because the fields are in distinct substructures.

The string parameters 'store1' and 'store2' contain arguments for Dataout objects, which store beam data in the result file so you can read them after the simulation. As explained in Section 1.2, the program represents beams as 3D arrays of complex amplitudes. Users are not always interested in so detailed information, so Dataout has options to reduce the amount of data by computing various properties of the beam. For example, it can compute the intensity and integrate over space or time to compute power or fluence. In the example program we have used the options 'p' to store power (or pulse shape), 'tnf' to store fluence (total near field), and 'tff' to store far-field fluence (total far-field). Each of these keywords is followed by a string with one letter for each beam. 'n' means don't store, 'b' means store in the backward direction (not relevant in this example, where beams only propagate forward), 'f' means store in forward direction, and 'a' means all (both) directions. The equal signs are not required, but they can be inserted to improve readability. For backward compatibility, Sisyfos also accepts numbers instead of the letter codes after the field names, where 0 means 'n', 1 means 'b', 2 means 'f', and 3 means 'a'. The default value for 'store1', 'p=nff tnf=nff', means that Dataout 1 will store power and total near field of beams 1 and 2. Similarly, 'p=fff tnf=fff tff=fff' for 'store2' means that Dataout 2 will store power, total near field, and total far field for all three beams. In this program, Dataout 1 is placed before the nonlinear crystal and Dataout 2 is placed after it. You should look at the html documentation for Dataout to get an overview of which beam properties Sisyfos can store.

Beam numbers

It is conventional in Sisyfos programs to order the beams by ascending frequency, so in this example beams 0, 1, and 2 correspond to idler, signal, and pump. Beam numbers start at 0 in C++ programs, but because a former version of Sisyfos was written in matlab, where indices start at 1, we follow this convention (i.e. beam numbers starting at 1) in the result files and in the (matlab or python) functions for reading them. We admit that this can be confusing, but it is the price of backward compatibility. In this tutorial we follow the numbering convention of the program being described, thus starting at 0 when discussing C++.

After setting up the parameter structure, the program runs the lines:

```
ps->read_arg(argc-1, argv+1);
ps->check_top();
if (ps->get_help_opt()) return 0;
```

read_arg() parses the arguments on the command line. argv[0] is omitted because it just contains the program name, and the actual arguments start in argv[1]. Directives on the command line can tell the program to read additional parameters from text files or binary files, and this makes the program flexible in terms of input data. check_top() checks that all parameters have values within their allowed ranges. The program terminates if invalid parameters or values are found. Finally, get_help_opt() displays help information if the user has set the help option on the command line. In this case, the program stops after displaying the help.

3.1.2 Optical path

After reading parameters from the command line or files into the parameter structure the values can be fetched into variables of the program. For non-programmers, it is important to understand the meaning of lines such as

```
real64 lamp = ps->get_real("lamp");
```

'real64 lamp' defines a variable in the C++ program, and the rest of the line assigns it a value fetched from the parameter structure. For convenience we use the same name for the program variable and the field in the parameter structure, but they do not have to be the same, and there is no automatic relation between them. It is only because of the explicit assignment that the program variable 'lamp' gets the same value as the parameter 'lamp'. As far as the C++ program is concerned, names in the parameter structure are only strings. If you create two fields with the same name in a ParamStruct, the compiler cannot detect it, but ParamStruct will report the error at run-time.

After retrieving 'lamp' and 'lams' the program creates a vector of BeamPar (parameters for each beam, which for example include centre frequency).

The pump source is an object of class SimpleSource, which represents a source that is separable in space and time. It uses a Func1Sgauss object (super-Gaussian function of 1 variable) for the pulse shape and a Func2Sgauss object (super-Gaussian function of 2 variables) for the beam. The seed source is similar. More general beam sources will be described in example 2 in Section 6.

The Path object is the top level of the optical path. A single such object is created, and other objects, which represent components of the device or non-physical simulation objects such as Dataout, are inserted into it. The structure of the optical path in this example is shown in Fig. 3.2. The first object in the path is a Source, which obtains data from the SimpleSource objects and injects them in into the optical path. The second object is a Dataout which stores beam data after the Source and before the nonlinear crystal. The main program assigns an id-number to each Dataout object. These must be unique, nonzero, positive integers.

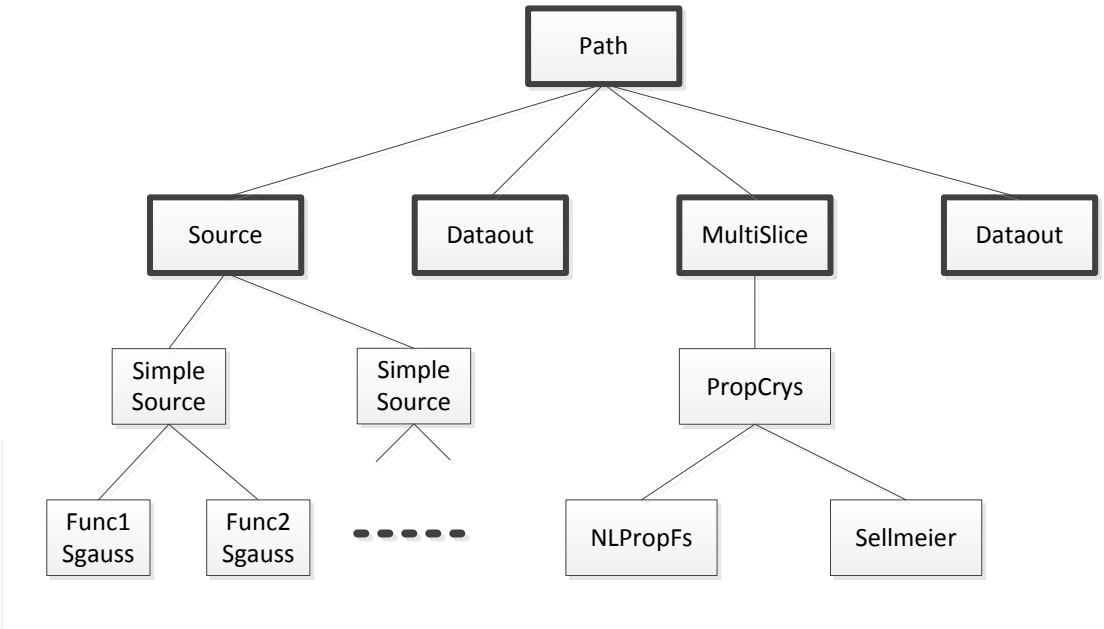

Figure 3.2 Optical path of the OPA. The objects in the Path are shown with thick frames, and some of their helper objects are shown below with light frames.

The nonlinear crystal is represented by a MultiSlice object, which requires several helper objects. This structure may seem excessively complicated for this example, but it allows great flexibility, which is useful in more complex systems.

- First, an object of class NLPropFs is created. This is a solver for the differential equations governing the nonlinear beam propagation.
- A Sellmeier object represents the dispersion of the crystal.
- These two are passed to a PropCrys object, which handles the actual propagation.
- Zero or more nonlinear processes can be associated with the PropCrys object by the method add_chi2_proc(). Its arguments are the numbers of the interacting beams and the effective nonlinear susceptibility.
- The PropCrys object is passed to the MultiSlice object, which is added to the optical path.

A second Dataout is inserted to store beam data after the crystal.

3.1.3 Initialisation and computation

The program creates a new data file with the call

```
WriteSis5File::make()
```

Sisyfos has its own binary file format which supports efficient storage of structured data. A file is

organised as a tree, so the parameter structure can easily be stored in the file, which is done by the line

```
ps->store(store, "par");
```

The program then creates a second substructure called ’par2’ and stores derived parameters, in this case only chieff, there. The names of these structures do not have to be ”par” and ”par2”, but we recommend following this convention.

In order to support different FFT packages and random-number generators, these are represented by objects that are passed to main_init1a. In this example, objects of the classes FftSimpleFac and RandomGen1Fac are used.

For convenience, main_init1a reads ’smode’, ’points’, ’scale’, ’nt’, and ’dt’ directly from the parameter structure. For reasons which will be explained in Section 4, the time resolution is set in a second init function, main_init2a(). The line

```
path->run(0, dt);
```

runs a simulation, where the arguments to run() are the start and stop times. run() always runs a whole number of intervals of length nt*dt, which we call time slices, and it stops when the simulated time becomes greater than or equal to the stop time. Therefore, setting the stop time to dt makes it run a single time slice. The last few lines of the program close the output file and clean up.

3.2 Running Sisyfos from the command line

The program can be run in a command window simply by typing its name, optionally followed by parameters to override the default values. It should be possible to copy commands from the pdf-version of this tutorial and paste them into the command window. On Windows 7 you find the command window under All programs - Accessories - Command prompt, and you can paste into it by clicking the right mouse button (Ctrl-V does not work). Run the commands

```
cd ....\Sisyfos5\tutorial\ex1
opa_ex1md file *t00
```

where the dots ’....’ in the path name must be replaced with the path where you created the Sisyfos directory. In this case, default values will be used for all parameters except the file name. The asterisk in front of the file name tells Sisyfos to overwrite any existing file with the same name. If there is no asterisk and the file already exists, Sisyfos generates a new unique file name by appending a number. To override additional default values, you simply specify more parameters on the command line, e.g.

```
opa_ex1md file *t01 bbo.len 3e-3
```

Note how the value for the field 'len' in the nested structure 'bbo' is specified: bbo.len 3e-3. The syntax with dot before field names corresponds to the structure syntax in matlab or C++. There is also an alternative syntax, which is convenient when you need to set more than one field in a nested structure:

```
opa_ex1md file *t02 bbo [len 3e-3 d22 2e-12]
```

The parameter syntax is described in Appendix B. Try running the program with parameter values outside their ranges or with parameter names which are not defined.

3.3 Retrieving beam data

The results from a simulation are stored in a single file which by default has extension '.sis'. Beam data, which are stored by Dataout, typically make up the bulk of the file, but other objects also store some data. This section explains how to read the beam data, results stored by other objects are described in Section 3.4.

The class gfm3, which exists in both python and matlab versions, is designed to conveniently retrieve data stored by Dataout. We use the python version of gfm3 in the examples, but the matlab class is almost identical apart from differences in syntax. In particular, beam numbers start at 1 in both versions, because that is the convention used in the Sisyfos data files. Thus beam 0 in the C++ program corresponds to beam 1 in gfm3, and so on. Retrieved arrays, on the other hand, must be indexed according to the rules of the language in use, so indices start at 0 in the python examples and you have to increment them by one if you run the examples in matlab. The python functions are documented in html-files in "...\Sisyfos5\doc\python". You should browse the python class gfm3 to get an overview of the available methods, most of which correspond to fields stored by Dataout. If a field was not selected for storage in Dataout, the corresponding retrieval method in gfm3 will fail.

You should open a second command window so that you have one for python and one for running Sisyfos programs. Assuming that you have installed python(xy), start python and load the gfm3 module by:

```
ipython -pylab
from gfm3 import *
```

where the '-pylab' options loads modules with matlab-like behaviour, in particular for plotting. Instead of the import statement you could run the startup-script, which also imports other Sisyfos modules:

```
run .../Sisyfos5/python/startup_ext_vs10
```

Then, assuming you have run opa_ex1md successfully and created the file t00.sis, open it with

```
g = gfm3("t00")
```

The methods of gfm3 correspond to the fields that can be stored by Dataout, e.g.

```
g.w(1,2)
```

gives the energy of beam 2 at the position of Dataout 1. For simplicity, Dataout n is often referred to as ”position” n in the optical path. The corresponding commands in matlab are

```
g = gfm3('t00')
w(g, 1, 2)
```

The spectrum of beam 1 at position 2 can be plotted by

```
clf()
plot(g.ts(2,1)*1e16)
```

and the signal pulse (instantaneous power) at position 2 by

```
clf()
plot(g.p(2,2)/1e6)
```

where the scale factors were chosen to avoid very small or large values on the vertical axes, and the horizontal axes simply show sample numbers. The resulting plots are shown in Fig. 3.3.

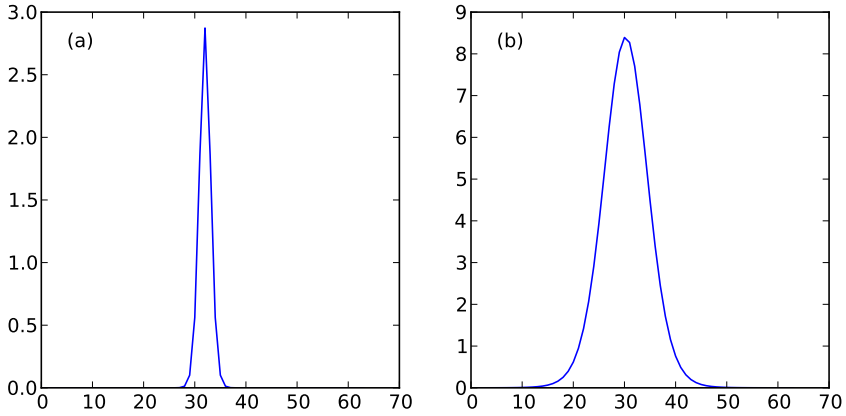

Figure 3.3 (a) Idler spectrum and (b) signal pulse at the output of the OPA.

tnf() (for ’total near field’) retrieves the fluence, that is, intensity integrated over the duration of the pulse.

```
clf()
imshow(g.tnf(2,2))
```

It is often convenient to get meta-data for the axes of a plot, and for this reason most of the methods in gfm3 have variants that also return meta-data. These variants have suffix 'r', and they return structures where the raw data are stored in a field called 'm' and the meta-data depend on the method. Try

```
u = g.tsr(2,2)
print u
clf()
plot(u.la*1e9, u.m)
xlabel("Wavelength (nm)")
```

tsr corresponds to ts and returns a spectrum. The meta-data in this case are wavelength (la), frequency (nu) and frequency offset from centre (f). Similarly, tnfr returns tables of the transverse coordinates x and y, and pr returns power with temporal sample points:

```
u = g.tnfr(2,2)
clf()
plot(u.x*1e3, u.m[17,:])
xlabel("Position (mm)")

v = g.pr(2,2)
clf()
plot(v.t1*1e12, v.m)
xlabel("Time (ps)")
```

3.4 Exploring the result file

The purpose of this section is to explain the structure of the result file and to present the class ReadSis5, which is used to access the file at a lower level than gfm3. Data from other objects than Dataout must be retrieved by the methods of ReadSis5. gfm3 is derived from ReadSis5, which means that a gfm3 object is also a ReadSis5 object (but not vice versa), so you can use all the methods of ReadSis5 directly on an object created by gfm3.

As already mentioned, Sisyfos files have a tree structure. To see the fields on the top level, type

```
g = gfm3("t00")
g.fields()
```

The 'par' structure contains the input parameters, so if you type

```
print g.get("par")
```

you can recognise the parameter structure from the C++ program and the values which were changed on the command line. Explore some of the other fields by similar print statements.

- 'par2' contains derived parameters, in this case 'chieff', which you also can recognise from the C++ program.
- 'sis' contains information about version and build time.
- 'top' contains information about run time and some parameters.
- 'PropCrys' contains data from the propagator. Since there can be multiple such objects, there are substructures named '0', '1', etc., where only '0' is present in this example.
- 'MultiSlice' contains data from the MultiSlice object. It has substructures like PropCrys.
- Other classes, but not all, also store data in substructures with the same name as the class.

Try the commands

```
g.fields("PropCrys")   # Displays the field names in the substructure
u = g.get("PropCrys")
u.fields()
```

u is now a data structure in python (or similarly in matlab). Note that the substructure called 'PropCrys.0' in the file has been named 'PropCrys.a0' in python or matlab because these languages do not allow names starting with digits. Data stored by PropCrys include the refractive index and group index of each beam:

```
print u.a0.n
print u.a0.ng
```

Data from the Dataout objects are stored in the substructure 'do' (for backward compatibility it has this name instead of 'Dataout'). This has substructures corresponding to the id-numbers given to the Dataout objects, typically '1', '2', etc. (but gaps in the sequence are allowed). Each of these has substructures 'f' and 'b' for data from the forward and backward directions, and 'f' and 'b' have substructures '1', '2', etc. for the different beams. It is simpler to access these data through the methods of gfm3, but for the sake of getting familiar with the structure of the file, try typing

```
clf()
plot(g.get("do.2.f.1.ts"))
```

This is equivalent to 'plot(g.ts(2,1))'. If you type

```
print g.get("do.2.f.1")
```

you will see the data and meta-data stored for Dataout object (or 'position') 2, forward direction, beam 1. Other ways to display the same information would be

```
u = g.get("do")
print u.a2.f.a1
```

or

```
v = g.get("do.2.f")
print v.a1
```

The fields present depend on the options passed to Dataout, and the possible fields are described in its documentation. To to give a flavour:

- 'w' is the energy.
- 'p' is the power versus time. Try 'plot(v.a2.p)'
- 'ts' is the spectrum. Try to plot this too.
- 'tnf' is total near field. This is itself a structure with field 'm' for the actual fluence and 'x' and 'y' for the meta-data.

It is possible to read the full file into a structure by

```
u = g.get("")
```

but this may take a lot of memory.

3.5 Reading parameters from files

Setting many parameters on the command line is inconvenient, so Sisyfos allows parameters to be read from text files. On the command line, you can write '-tf <filename>', to include the contents from a text file, e.g.

```
opa_ex1md -tf par1 file *t03
```

You can also use a text file for a specific field, e.g.

```
opa_ex1md -tf par1  pump -tf pump2 file *t04
```

It is also possible to read parameters from a field in a binary file:

```
opa_ex1md pump -bf t03 par.pump bbo.len 2e-3 file *t05
```

This will get the pump parameters from the field 'par.pump' in the file t03. See Appendix B for a detailed description of the input format.

3.6 Help

Try typing:

```
opa_ex1md help . hlevels 0
```

This displays a list of the fields on the top level of the parameter structure together with their types, default values, allowed ranges and sizes (for arrays). If you type

```
opa_ex1md help . hlevels 1
```

or just

```
opa_ex1md help .
```

the substructures will also be expanded to the first level. You can get help for a specific field by typing e.g.

```
opa_ex1md help pump
```

4 Resolution parameters

Choosing suitable parameters for spatial resolution, matrix size, temporal resolution, and time slice length is essential to get correct results. The choice involves judgement which is hard to implement in a program, so Sisyfos does not check these parameters itself. Therefore it is important that the user understands how to set them and inspects the result file to verify that the resolution parameters were appropriate. Insufficient resolution, too short time window or too small beam matrix give invalid results. On the other hand, too fine resolution or too many points will lead to longer run time than necessary.

This section is probably the most complicated in the tutorial, but it is also the most important, so if you don't understand it after the first reading you should come back to it later. It begins by running the program from example 1 with different parameters and comparing the results. Then the resolution parameters are described in more detail.

The examples so far have used spatial mode 2 (full 2D), 32×32 transverse points, 0.2 mm transverse resolution, 64 temporal sample points, and 0.5 ps time resolution. Since the OPA has symmetry about the critical (walk-off) plane and the input beams have even higher symmetry, we can use smode 3 instead of smode 2:

```
opa_ex1md file *t10 smode 3 points [32 17]
```

Because smode 3 uses a half-plane, the beam matrix was now chosen to have approximately half the number of points in the symmetric direction (the direction normal to the symmetry plane), 32×17 points. The number of points in the symmetric direction must be odd because of the way the cosine transform is implemented. The FFT package can restrict the number of points further, as will be explained in Section 4.1.

Compare the results from the simulations with smode 2 and 3 (if you use matlab instead of python, remember that indices in matlab start at 1):

```
g0 = gfm3("t00")
g1 = gfm3("t10")
g0.w(2,2)
g1.w(2,2)
g0.tnf(2,2).shape
g1.tnf(2,2).shape
clf()
plot(g0.tnf(2,2)[16,:])
plot(g1.tnf(2,2)[0,:],"r--")
# Compare runtime
g0.get("top.t_run_cpu")
g1.get("top.t_run_cpu")
```

There should be good agreement between the two. Inspect some of the near- and far fields:

```
clf()
imshow(g1.tnf(1,3))
clf()
imshow(g1.tff(2,2))
```

There is plenty of room in the tff matrix, so we can try with double element size and half as many points in each direction:

```
opa_ex1md file *t11 smode 3 points [16 9] scale 4e-4
```

Compare the results again. Because the axes are different, it is now necessary to use the meta-data to plot and compare near and far-fields.

```
g2 = gfm3("t11")
g1.w(2,2)
g2.w(2,2)
g1.get("top.t_run_cpu")
g2.get("top.t_run_cpu")
u1 = g1.tnfr(2,2)
u2 = g2.tnfr(2,2)
clf()
```

```
plot(u1.x, u1.m[0,:],"-o")
plot(u2.x, u2.m[0,:],"-o")
v1 = g1.tffr(2,2)
v2 = g2.tffr(2,2)
clf()
plot(v1.x, v1.m[0,:],"-o")
plot(v2.x, v2.m[0,:],"-o")
```

The narrow peak in the far-field indicates that even coarser resolution may be sufficient. Inspection of the spectra indicates that we can also reduce the temporal resolution:

```
opa_ex1md file *t20 smode 3 points [16 9] scale 4e-4 nt 16 dt 2e-12
```

Compare the results:

```
g3 = gfm3("t20")
g2.w(2,2)
g3.w(2,2)
g2.get("top.t_run_cpu")
g3.get("top.t_run_cpu")
u1 = g2.pr(2,2)
u2 = g3.pr(2,2)
clf()
plot(u1.t1, u1.m,"-o")
plot(u2.t1, u2.m,"-o")
v1 = g2.tsr(2,2)
v2 = g3.tsr(2,2)
clf()
plot(v1.nu, v1.m,"-o")
plot(v2.nu, v2.m,"-o")
```

The energy still agrees well, but there is a noticeable difference in pulse shape with the low time resolution. Note that the spectral range in t20 is much narrower than in the other files, and it cannot be reduced much further without cutting the tails of the spectrum.

Finally, try a longer crystal, shorter pulses, and lower pump energy to clearly see the effect of temporal walk-off. The number of parameters to set becomes quite large, so we have put them in the file par2.txt. Run

```
opa_ex1md file *t30 -tf par2
```

and inspect the results:

```
g = gfm3("t30")
clf()
```

```
plot(g.p(1,3))
plot(g.p(2,3))
plot(g.p(1,2))
plot(g.p(2,2))
```

The pump pulse propagates slower than the signal pulse, and it eventually slides outside the time window and aliases into the other end. It is necessary to use more temporal sample points (or a greater dt if the spectra allow it), e.g.:

```
opa_ex1md file *t31 -tf par2 nt 32
```

Inspect the results again and check that the pulses stay within the time window. Check the group indices as shown in Section 3.4:

```
u = g.get("PropCrys.0.ng")
```

The '.0' in this example indicates the first (and in this case only) PropCrys object. Sisyfos uses the fastest beam as default reference frame, so the group indices are consistent with the observed lag of the pump pulse.

4.1 Spatial parameters

Sisyfos uses x and y for the transverse coordinates and z along the beam. The spatial parameters are smode (spatial mode), the number of transverse points (nx, ny), and the transverse scale. The matrix can be rectangular, and the scale can be different in the x and y directions. The following table shows the meaning of the smode parameter and restrictions or recommendations on nx and ny in each case.

smode	Meaning	Restrictions	FFT sizes
0	Plane wave	nx=ny=1	
1	Cylindrical symmetry, transverse samples along a radius.	ny=1	
2	Full 2D		nx, ny
3	Symmetry about the x-axis, half-plane with $y \geq 0$	ny odd	nx, ny-1
4	Symmetry about x and y axes, quadrant with $x \geq 0, y \geq 0$	nx and ny odd	nx-1, ny-1

Note that the sizes of the FFTs which Sisyfos use are not always equal to nx and ny. nx and ny should be chosen such that FFT sizes are good values for the FFT package being used. The package FftSimple, which is used in the example programs, supports only powers of 2. FftIntel and Fftw support arbitrary transform sizes, but they are most efficient when the number of points has only small prime factors.

The matrices must be large enough to contain the beam everywhere in the path, and the resolution must be fine enough to contain the far field. Sisyfos does not check these conditions, so the user

must place Dataout objects at critical points along the path and check that both the near-field and far-field matrices contain their respective beams. If the beam matrix is too small, light will spill outside and alias into the other side of the matrix. Similarly, if the resolution is too coarse, the matrix in k-space is too small, and high angular components will be aliased. When simulating devices where some light really diffracts out of the system it is necessary to insert apertures to absorb the lost light before it aliases to the opposite side of the matrix. The apertures should be soft to avoid strong diffraction from the edges. The following table summarises the spatial parameters. For brevity, only the parameters for the x direction are shown.

Parameter	Meaning
nx	Number of transverse sample points
sx	Spatial resolution
nx*sx	Size of beam matrix
dkx = 2π/(nx*sx)	Resolution in k-space
nx*dkx = 2π/sx	Range in k-space

In smode 2 and for the x-direction in smode 3, it is possible to specify transverse k-vector offsets individually for each beam. This can be used to represent tilted beams, e.g. in a non-collinear OPA, with low spatial resolution. Without factoring out the offsets, tilted beams would require very high transverse resolution to represent their k-vectors correctly. This is analogous to factoring out the centre frequencies in the time domain.

smode 1 is efficient for devices with cylindrical symmetry, but in our experience the small area of the centre element makes it susceptible to numerical errors and very sensitive to light spilling out of the simulation domain. smode 1 must be used with care, especially when beams have sharp peaks in the centre.

4.2 Temporal parameters

Sisyfos can use two levels of temporal sampling. At the lowest level (finest resolution), a time slice consists of nt sample points with interval dt. This gives a spectral bandwidth of 1/dt and a spectral resolution of 1/(nt * dt). A time slices is treated as a unit when it propagates through the optical path.

If the reference frame were fixed, the time slice would have to be at least as long as the transit time of the optical path, which is often much longer than the duration of the pulses. Therefore, Sisyfos works in a frame that moves with the pulses, where the time slice only needs to be long enough for the pulses and the temporal walk-off. The group velocity of the frame follows a reference beam, which is usually chosen to be the fastest beam. In general, all beams except the reference beam will experience temporal walk-off with respect to the frame. Equivalently, for a set of passively propagating pulses, the pulse of the reference beam will stay at a fixed position within the time slice whereas the other pulses will move. Figure 4.1 shows two pulses and the moving time slice versus absolute time.

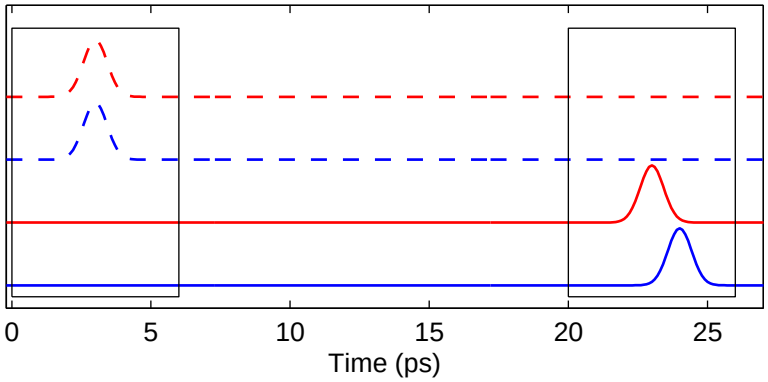

Figure 4.1 *Reference pulse (red) and another pulse (blue) in absolute time before a crystal (dashed) and after it (solid). The transit time is 20 ps for the reference pulse. The black frames show the time slice, which moves with the reference pulse. The required length of the time slice is determined by the pulse length and the temporal walk-off, and it is much shorter than the transit time.*

Devices without feedback can, at least in principle, be simulated using only the first level of sampling by choosing the time slice long enough to contain the signals of interest. This is analogous to the requirement for the near-field matrix in the spatial domain. In addition, the temporal resolution (dt) should be so small that the simulation spectrum contains the spectrum of the pulse, which is analogous to the requirement for spatial resolution. As in the spatial domain, it is the user's responsibility to check that these conditions hold.

In a resonant device, such as an OPO, the resonant signal is fed back with a delay equal to the round-trip time of the resonator, t_R. In such cases, Sisyfos works iteratively and uses multiple time slices. If a time slice corresponds to input signals in the interval $[t_1, t_2]$ (in absolute time), the resulting output data after propagation through the optical path will correspond to the interval $[t_1 + t_R, t_2 + t_R]$. Therefore, the interval between the start of two consecutive time slices, which is called the time slice interval or tsi, cannot exceed t_R. The following table summarises the temporal parameters:

Parameter	Meaning
nt	Number of sample points per time slice, must be compatible with FFT
dt	Time interval between sample points
tsi	Time interval between slices (from the start of one slice to the start of the next)
tsf	Time slice factor, tsf = tsi / (nt * dt)
df = 1/(nt*dt)	Frequency resolution
nt*df = 1/dt	Bandwidth

If the signal consists of isolated pulses within each round trip time, as in a mode-locked laser or a sync-pumped OPO, the time slice can be chosen to contain a single pulse, and it can be much shorter than t_R. tsi should equal the period (which is determined by the synchronous pump or

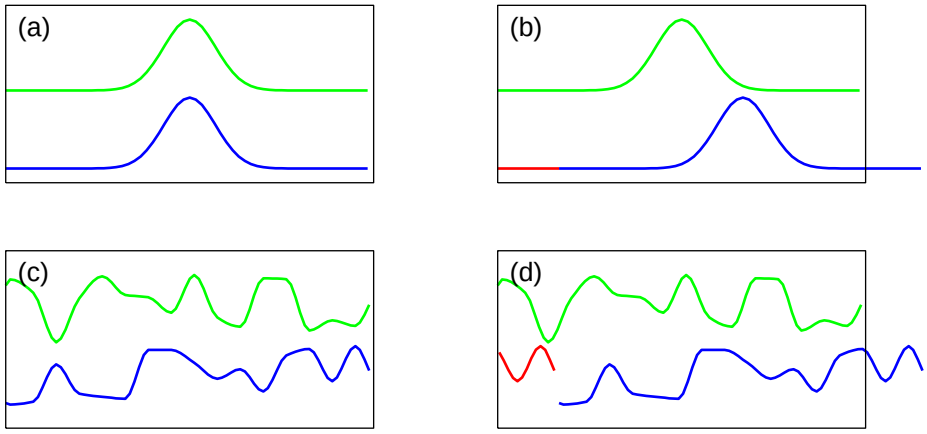

Figure 4.2 *(a, b) Time-slices with short pulses before (a) and after (b) the optical path. The green curve represents the reference beam and the blue curve represents a beam with temporal walk-off. The black frame shows the time slice. Because of the temporal walk-off, the blue pulse slides out of the time slice, and its tail is folded into the leading part of the slice, as indicated by the red curve. This part of the time slices becomes invalid, but it does not matter because the signals are nearly zero there. (c, d) Corresponding time slices where the signal durations are longer than the time slice. In (d), the leading part of the slices becomes invalid because the (nonero) trailing part of the blue pulse is folded into it.*

other mode-locking element in an actively mode-locked system). Since the pulses in adjacent round trips do not interact, splitting the signal into time slices does not introduce errors. By using time slices shorter than t_R the simulation program saves time by not covering the time axis completely.

If the pulse is longer than the round trip time, which is typical for OPOs pumped by nanosecond pulses, the situation is more complicated. Because at least some beams have temporal walk-off, adjacent time slices are not fully independent. The current version of Sisyfos makes an approximation by treating time slices as isolated units, where temporal walk-off becomes cyclic within each slice because the Fourier-based propagation algorithms implicitly assume that the signal is periodic. Figure 4.2 shows time slices for short and long pulses.

Experience shows that the approximation with isolated time slices and cyclic temporal walk-off gives good results for energy, pulse shape, beam quality and spectral width of an OPO. However, it does not give correct results for fine spectral details, which cannot be resolved in the limited length of one time slice. The error caused by neglecting the interaction between adjacent time slices is related to the ratio of the temporal walk-off to the time slice length, i.e. the fraction of the time slice that is affected by the (incorrect) cyclic temporal walk-off. A future version of Sisyfos is planned to include interaction of adjacent time slices by using overlapping slices and selecting the valid parts.

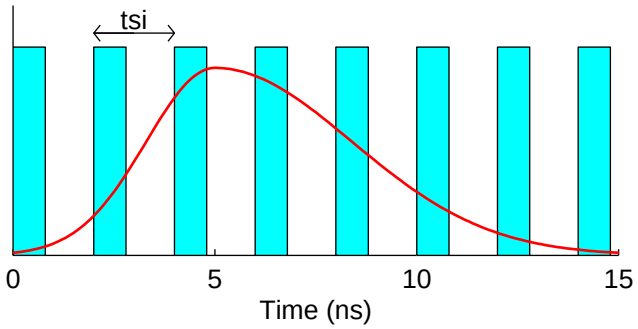

Figure 4.3 *Time slices (light blue) which do not cover the time axis completely. Sisyfos assumes that the average power between the slices is the same as in the slices and scales up the energy correspondingly.*

Nanosecond OPOs can be simulated with time slice length and time slice interval both equal to t_R. However, when the interaction between adjacent time slices is neglected, it is possible to use shorter time slices and gaps between them, as in the case with sync-pumped OPOs. This is illustrated in Fig. 4.3. The difference from the sync-pumped case is that the power between the time slices is not really zero. Sisyfos assumes that the short time slices are representative for the signal and scales up the energy by the time slice factor tsf = tsi / (nt*dt). This approximation can be useful to save time when the bandwidth dictates a small dt so that nt would have to be large to make the time slices cover the time axis densely. However, pulse to pulse fluctuations become exaggerated because properties which really depend on an average over the whole pulse, are computed based on only part of the pulse. The ratio of temporal walk-off to time slice length obviously increases with tsf, so large tsf must be used with care.

In single-pass devices, it is usually convenient to specify nt and dt. In resonators it may be more convenient to take tsi equal to the round trip time, specify tsf and nt and let Sisyfos calculate dt from these. The class Path has methods to specify the parameters in either way. The temporal parameters are set in two stages by the methods main_init1 or main_init1s in the first stage and main_init2 or and main_init2_res in the second. The reason for splitting main_init in two stages is to give the main program the possibility to retrieve and modify the round-trip time of a resonator before setting the time resolution, and the initial round-trip time can only be computed after some of the parameters have been set by main_init1 or one of its variants.

4.3 Narrow- and wide-band mode

Propagation calculations in Sisyfos are based on the longitudinal wave vector for each plane-wave component in the beam, $k_z(\nu, k_x, k_y)$, where ν is the frequency and k_x, k_y are the transverse wave vector components. In wide-band mode this is computed exactly, but in narrow-band mode it is approximated by

$$k_z(\nu, k_x, k_y) \approx k_z(\nu, k_{x0}, k_{y0}) + k_z(\nu_0, k_x, k_y) - k_z(\nu_0, k_{x0}, k_{y0}),$$

where ν_0, k_{x0}, and k_{y0} refer to the centre component. The approximation is good for narrow spectra, but wide-band mode must be used when wide-band beams interact or when it is important to account for the different diffraction of different frequency components within a beam.

From version 5.1.24 wide-band mode is selected by default, and we recommend using it unless you have verified that the narrow-band approximation is good.

5 Function classes

In example 1 you saw how the functions Func1Sgauss and Func2Sgauss were used for pulse and beam shapes in SimpleSource. Sisyfos has many such functions, so you could get other pulse or beam shapes by replacing the SGauss functions with other types. However, a user should be able to choose pulse and beam shapes with parameters, without changing and recompiling the program. The main program could include parameters for all possible functions in its parameter structure and have a large switch statement to create the right one, but this would be cumbersome. Therefore, Sisyfos has general function classes called Func1Any (for functions of one variable), Func2Any and Func3Any (for two or three variables), which let the user choose functions at run-time with minimal overhead in the main program. There is also a general class for Sellmeier equations (SellmeierAny).

The purpose of this section is to present Func1Any thoroughly. Func2Any, Func3Any and SellmeierAny are similar and can be understood by analogy. It is important to master the function classes because Sisyfos uses them for a lot of different purposes in addition to pulse and beam shapes. For example, functions of one variable are used for spectra and for quantities which vary with temperature. Functions of two variables are used for lens shapes and apertures, and functions of three variables are used for temperature distributions and initial population distribution in a laser rod.

The example program func_test in the directory .../Sisyfos5/tutorial/ft can test Func1Any, Func2Any, and Func3Any. It evaluates a function on one or more points in a grid and writes the results to a file or to the display. At the top level func_test takes the parameters

- 'type' selects the type of functions to test. Should be 1 (for Func1Any), 2, or 3.
- 'f1_p' is a parameter structure for Func1Any (see Func1Any::param_struct()). Must be set if 'type' is 'f1'
- 'f2_p' and 'f3_p' are similar parameter structures for the other function types.
- 'file' is the name of the output file. If no file name is given the results are printed to the display.
- 'points' is the size of the grid where the functions is evaluated. Can have 1–3 elements. Default 1. In the following, we also denote the number of points on each axis by nx, ny, and nz.
- 'x0' is the start value for each axis of the grid. 1–3 elements. Default 0.

- 'dx' is the point spacing for each axis of the grid. 1–3 elements. Default 1.
- 'index' is an index or list of indices for which to evaluate vector-valued functions. The default is 0, and you can forget this parameter for most functions.
- 'complex' selects real (0) or complex (1) values.

The output file (if any) contains

- par - Structure with input parameters
- x(nx), y(ny), z(nz) - Arrays with coordinates of the grid points (where nx=points[0], etc.)
- index - Copy of the index parameter
- v[nx, ny, nz, n_index] - Array with real or complex function values

func_test uses x, y, and z for the three independent variables, but in the context of Func1Any we often use t instead of x.

5.1 Func1Any

An example command to test Func1Any is:

```
func_testmd type 1 f1_p [type linear kr 2.5 v_base 3 v_scale -2]
```

This computes the function

$$f(t) = 3 + (-2) \cdot (2.5 \cdot t),$$

where the numeric coefficients were chosen to be unique so you can easily associate them with the command line. A slightly more advanced example, where the function is evaluated at multiple points and the results are saved in the file 'test.sis', is

```
func_testmd type 1 f1_p [type sechsq wt 4] points 10 x0 -10 dx 2 file *test
```

Run the command and plot the results in python. You have to open the file by ReadSis5 instead of gfm3 because this program uses no Dataout objects:

```
g = rs.ReadSis5("test")
x = g.get("x")
v = g.get("v")
clf()
plot(x,v)
```

In more general, Func1Any takes the parameters

- v_base – Added constant.

- v_scale – Scale factor.
- wt - Scale factor for t. Can have two elements, where the first is used for $t < 0$ and the second for $t > 0$.
- t0 - Offset for t.
- invert_t – Constant to invert t, e.g. to convert between wavelength and frequency.
- t0i - Offset after inversion, only used with invert_t.
- rep_f – Repetition frequency, to make the function periodic.
- t0p – Start point for period.

and computes

$$f(t) = \text{v\_base} + \text{v\_scale} \cdot f(t'),$$

where f is the function selected by 'type' and t' is obtained from t by

```
if (rep_f > 0) t = (t - t0p) mod (1 / rep_f)
t1 = t - t0
if (invert_t != 0) t1 = invert_t / t1 - t0i
if (t1 < 0) t' = t1 / wt[0]
else        t' = t1 / wt[1]
```

In addition to these parameters, which can be used for all functions, Func1Any takes a number of parameters that are used by only one or some of the possible functions. For example, the linear function demonstrated above used the parameter 'kr'. The 'level' parameter is used by pulse-like functions to specify the level (relative to the peak value) at which the half-width of the underlying function is 1. The actual half-width is then set with the wt parameter. See the html documentation for Func1Any for a list of all the possible functions and their parameters. Some of the more complicated functions, for example for table interpolation, require substructures with their own specific parameters.

The parameters for transformation of the t-argument are useful for creating trains of pulses, and the two independent elements of wt can be used to make the pulses asymmetric. In the following example, the width of an asymmetric super-Gaussian pulse is specified on the $\exp(-2) \approx 0.135$ level:

```
func_testmd type 1
 f1_p [type sgauss wt [1 3] t0 5 rep_f 0.01 t0p 10 level 0.135]
 points 500 x0 -100 file *test
```

If you plot the results you will see that the function has peak value (1) for t=15, 115 and so on.

Here is an example where a table of some quantity versus wavelength is used in an interpolation function and the argument is inverted so that the resulting function takes frequency as argument:

```
func_testmd type 1
 f1_p [type tab tab_p [x0 1e-6 dx 2e-7 y [1 2 3 4 5]] invert_t 3e8]
 points 13 x0 1.7e14 dx 1e13
```

tab_p contains data for Func1TableR, and it has fields 'x0' and 'dx' which must not be confused with the 'x0' and 'dx' parameters to func_test. x0 means the start value for x (the independent variable), dx the increment between adjacent points in the table, and y represents the actual table. Since it has 5 elements the valid x-values run from 1e-6 to 1.8e-6 i this case. If you run the command you should get sensible values for x between 1.667e14 and 3e14 and errors for other values. Func1TableR has many other options: It is of course possible to have uneven spacing of the x-values, and the v_out option can set a fixed value for lookup operation outside the table (instead of the error message. Try it!). See the html documentation for details. Additional examples can be found in the file .../tutorial/ft/run.txt.

You should look at the func_test.cpp program to see how Func1Any is used. Func1Any::param_struct() returns a parameter structure with the parameters described above. The predefined structure saves a lot of typing in the main program, but the disadvantage is that you cannot control the range and default values of each parameter. Func1Any::make() creates a function object of the type chosen in the parameter structure. Since a random function is one of the options, make() needs a random number generator as its second argument. A detailed understanding of Func1Any requires some knowledge of C++, and some of the concepts are described in more detail in Appendix C.

The parameters to Func1Any and similar classes can become complicated, so func_test is useful to check that a parameter set works as expected before using it in a simulation. Section 9 shows how func_test can be run from a python shell.

5.2 Func2Any and Func3Any

These classes are logically similar to Func1Any, and they have some parameters in common with Func1Any or each other. In particular, all three classes have v_base and v_scale. Func2Any has parameters for for transformation of x and y:

- wxy – Width in x and y, 2 elements
- pos0 – Centre position, 2 elements
- ang – Rotation angle in the xy-plane (radians)
- e_order – Hyperelliptic order, used for underlying functions that depend only on the radius.

The transformation of an input position (x, y) to an output position (x', y') is

```
xa = x - pos0.x
ya = y - pos0.y
x' = ( cos(ang)*xa + sin(ang)*ya) / wxy.x;
y' = (-sin(ang)*xa + cos(ang)*ya) / wxy.y;
```

For functions that depend only on r, the generalised radius is

$$r = ((x')^{\mathrm{e\_order}} + (y')^{\mathrm{e\_order}})^{1/\mathrm{e\_order}}$$

$\mathrm{e\_order} = 2$ gives a circular or elliptic shape, and $\mathrm{e\_order} > 2$ makes the shape more rectangular. e_order should be an even integer. Like Func1Any, Func2Any has several additional parameters which are used by one or more of the possible functions.

Func3Any has parameters for both the t-transformation (where t corresponds to z) and the xy-transformation.

Func2Any and Func3Any can be tested with func_test in a similar way to Func1Any, e.g.:

```
func_testmd type 2 f2_p [type sgauss wxy 2 ord 4 ] points [5 5]
```

This will display values for 5×5 points in the first quadrant.

5.3 SellmeierAny

Sellmeier equations depend on frequency (or wavelength) and temperature. They could have been represented by IFunc2A and Func2Any, but because they have some specialised methods they are represented by the class SellmeierAny. This is conceptually similar to the function classes described above, and it can be tested by the program sell_test in the ft directory, e.g.:

```
sell_testmd se_p [mat_dir ../../MatData mat KTP name Kato2002] la0 5e-7 points 10
```

This displays the principal refractive indices for wavelengths 500 nm, 600 nm, ... 1400 nm. If you store the results in a file, the program also stores the dielectric tensor. See the comments in sell_test and the html documentation for SellmeierAny and ISellmeier for details.

6 Example 2 - OPA with more flexible input beams

In example 1, the parameter structure was built from scratch to make its structure clearly visible. In practice, different main programs tend to have many parameters in common, so Sisyfos offers some predefined substructures to simplify the main program. The purpose of this section is to show how these features can be used to make the program shorter and more flexible. The program opa_ex2, which should be read in conjunction with this section, is similar to opa_ex1, except that it makes use of predefined parameter structures and a few other Sisyfos features.

First, the function setup_std_param4() creates an initial parameter structure with fields and default values selected by the string argument. The syntax is the same as for arguments on the command line. See the documentation for the list of allowed fields (C++ modules, namespaces, namespace members, select the tab for 's' to get to setup_std_param4).

The input beams in example 1 were restricted to have Gaussian spatial distributions and pulse shape. Specifically, the pump and seed sources were represented by SimpleSource-objects with Func1Sgauss for the pulse shape and Func2Sgauss for the beam. The program could have been generalised by using Func1Any for the pulse shape and Func2Any for the beam, but the sources would still be restricted to be separable in space and time. For this reason, Sisyfos has two additional source-classes: Func3Source, which represents a (non-separable) function of (x, y, t), and FileSource, which takes data from the output file of a previous simulation. Like the various types of functions, these source-classes share an interface, IBeamSource, so that they can be used interchangeably. Following the pattern from the function classes, there is a class AnySource with methods param_struct() and make() which makes it possible to choose the type of source with parameters at run-time. The program in example 2 uses this class for the pump and seed beams, and that makes the program shorter and much more flexible than example 1:

```
ParamStruct *ps_pump = AnySource:param_struct());
ps->add_field("pump", ps_pump);
.....
IBeamSource *bs_pump = AnySource::make(ps_pump);
```

A parameter file (called beam2.txt in the example directory) for pump can be:

```
% Parameters for AnySource
type simple
simple_p [
 % Parameters for SimpleSource
 pulse [
  % Parameters for Func1Any
  type sechsq wt 4e-12 t0 15e-12
 ]
 beam [
  % Parameters for Func2Any
  type ap_ellip wxy 1.2e-3
 ]
]
w 5e-3          % Energy
ta 0 tb 30e-12  % Integration limits for energy.
```

'type' must be 'simple', 'file', or 'func'. Depending on 'type', you must include one of the following substructures, where the correspondence should be obvious from the names:

- 'simple_p' with parameters for SimpleSource (see SimpleSource::param_struct())
- 'func_p' with parameters for Func3Source.
- 'file_p' with parameters for FileSource.

Inside 'simple_p', which was used in this example, you can recognise parameters for Func1Any in

the 'pulse' field and Func2Any in the 'beam' field. To test example 2, run a simulation with this file as pump:

```
opa_ex2md file *t40 seed [-tf beam1] pump [-tf beam2]
```

Compare the results to the base-line case in the file t00 from example 1:

```
g0 = gfm3("../ex1/t00")
g1 = gfm3("t40")
g0.w(2,2)
g1.w(2,2)
clf()
plot(g0.p(1,3),"b--")
plot(g0.p(2,3),"b")
plot(g0.p(2,2),"r")
plot(g1.p(1,3),"g--")
plot(g1.p(2,3),"g")
plot(g1.p(2,2),"k")
clf()
plot(g0.tnf(1,3)[16,:])
plot(g1.tnf(1,3)[16,:])
clf()
plot(g0.tnf(2,2)[16,:])
plot(g1.tnf(2,2)[16,:])
clf()
semilogy(g0.tff(2,2)[16,:])
semilogy(g1.tff(2,2)[16,:])
```

The signal energy is higher because beam2.txt specified a flat-top pump beam, but the signal beam becomes ugly because of the sharp edges. This is also seen in the greater tails of the far-field.

6.1 Using beam data from simulation files

The output from one OPA stage can be used as input to a next stage. To demonstrate this, first simulate stage 1 and store the signal field (complex amplitude) at the output by adding 'store2 +[e=nfn]' (the '+' here means that '[e nfn]' is appended to the default string given in the C++ program instead of replacing it):

```
opa_ex2md file *t50 seed [-tf beam1] pump [-tf beam1 w 5e-3] bbo.len 1e-3 store2 +[e=nfn]
```

Then simulate stage 2 with the signal from stage 1 as input:

```
opa_ex2md file *t51 seed [-tf beam3] pump [-tf beam1 w 5e-3] bbo.len 1e-3
```

where beam3.txt reads

```
type file
file_p [
 % Parameters for FileSource
 fname t50.sis
 pos 2
 beam 2
]
```

This means that data for FileSource are taken from the file 't50.sis', position 2, beam 2 (beam numbers start at 1 in this context). Note that FileSource does not work unless the full complex amplitude for the selected beam and position was stored in the file, as selected by the 'e nfn' option above. FileSource has options for modifying the beam in various ways, for example by magnification or position shift. See the html documentation for details. As with other beam sources, you can specify the peak intensity, the peak power, or the energy within a time interval. If you want to specify the energy the power must also be stored for the selected beam and position, as selected by 'p fff' in the default parameter.

To check that FileSource worked as expected, you can compare data from the selected position and beam in the input file with the seed beam stored in the new file:

```
g0 = gfm3("t50")
g1 = gfm3("t51")
g0.w(2,2)
g1.w(1,2)
clf()
plot(g0.p(2,2))
plot(g1.p(1,2))
clf()
plot(g0.tnf(2,2)[16,:])
plot(g1.tnf(1,2)[16,:])
```

The two stages have the same total crystal length as in example 1. The reason that the signal energy is lower is that the idler is removed between the stages, which reduces the gain in the second stage.

If you want to use only the beam shape from a former simulation and combine it with a different pulse shape you can use SimpleSource:

```
opa_ex2md file *t52 seed [-tf beam4 w 2e-6] pump [-tf beam1 w 5e-3] bbo.len 1e-3
```

where the beam part of beam4.txt reads

```
type tab
tab_p [
 % Parameters for Func2TableR
```

```
 smode 2
 mr -bf t50.sis do.2.f.2.tnf.m
 scl 2.5e-4
 v_out 0
]
```

'type tab' selects a function of the class Func2TableR, which performs table lookup and interpolation with real data (there is a corresponding Func2TableC for complex data). Func2TableR has its own parameters in the substructure 'tab_p', where 'smode' specifies the spatial mode of the input matrix, and 'mr' the actual matrix. The '-bf' directive tells Sisyfos to obtain the matrix from the field 'do.2.f.2.tnf.m' in the file t50.sis. Setting the scale 'scl' to 2.5e-4 instead of 2e-4, which was used in t50, effectively expands the beam. The v_out option tells Func2TableR to use 0 if it tries to look up a value outside the matrix. If v_out is not used, an attempt to get a value outside the matrix causes an error. Check again that the input beam worked:

```
g0 = gfm3("t50")
g1 = gfm3("t52")
u0 = g0.tnfr(2,2)
u1 = g1.tnfr(1,2)
clf()
plot(u0.x,u0.m[16,:])
plot(u1.x/1.25,u1.m[16,:]*2.5)
```

The scale factor 2.5 was chosen just to make the beams overlap.

6.2 Changing solver parameters

The program opa_ex2 includes a substructure 'rk' with parameters for NLPropFs, the differential equation solver. The default values for NLPropFs are usually ok, but it is wise to run some simulations with other tolerances to check that results have converged. Try varying the relative tolerance rtol

```
opa_ex2md file *t41 seed [-tf beam1] pump [-tf beam2] rk.rtol 1e-3
opa_ex2md file *t42 seed [-tf beam1] pump [-tf beam2] rk.rtol 1e-2
opa_ex2md file *t43 seed [-tf beam1] pump [-tf beam2] rk.rtol 1e-1
```

and compare the results by

```
clf()
g0 = gfm3("t40")
plot(g0.p(2,2))
g1 = gfm3("t41")
plot(g1.p(2,2))
```

and so on. In this example, deviations become noticeable only for the very high relative tolerance of 0.1. See the html documentation for NLPropFs for details of the tolerance calculations.

6.3 Test program for AnySource

The ft directory contains a program source_test for testing AnySource, analogous to func_test for testing Func1Any. The program runs a minimal 'simulation' where the optical path consists of only a Source object and a Dataout object and there is one single beam. Try for example

```
source_testmd -tf data/srcpar1.txt file *test2
```

The result file will contain data from a single Dataout object with a single beam. See Section 9.6 for more examples with source_test.

7 Example 3 - Advanced OPA

Example 2 made use of some of Sisyfos' features for parameter handling and beam sources, but the actual OPA was still very simple. The purpose of this section is to introduce some of Sisyfos' advanced features by means of an OPA with non-collinear beams, nonlinear refractive index (n_2), frequency-dependent absorption in the crystal, and chirped signal pulses. Furthermore, the crystal is divided into multiple slices to be able to resolve the longitudinal distribution of absorbed power. The changes in the parameter structure compared to opa_ex2 are

- The standard parameters from setup_std_param4() include "wb" to allow wide-band mode (on by default). In this mode, Sisyfos accounts for different diffraction of different frequency components within a beam.
- A "noise" parameter is added to turn noise on signal and idler beams on or off.
- The "bbo.abso" parameter can define an absorption function (of frequency)
- The "bbo.n2" parameter can define n_2. To include cross modulation this is a 3×3 matrix.
- "bbo.slices" specifies the number of slices in the crystal.
- The "pump" parameters include an optional lens (which only affects the pump beam) to study the effect of an imperfectly collimated pump.
- The "storec" parameter selects which beam data to store between crystal slices.
- The "stheat" substructure contains parameters to store the energy which is absorbed in the crystal, see ArrayDataout.

There are also changes in the main part of the program

- Wavelengths and tuning angles are fixed in this example.
- To handle non-collinear beams efficiently, a k_x-offset corresponding to the non-collinear angle is included in beams 0 and 2 (idler and pump). Figure 7.1 shows the geometry with the non-collinear beams.
- There is a section to insert the optional pump lens.
- The absorption function and n_2 are added to the PropCrys object.

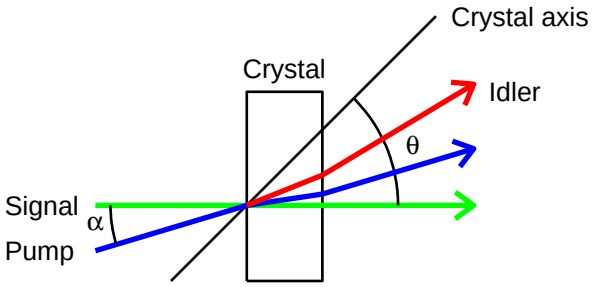

Figure 7.1 OPA with non-collinear beams. The signal beam is taken to be perpendicular to the crystal face.

- Multiple slices are added to the MultiSlice object, and Dataout objects are inserted between them.

The parameters are stored in the following files: Pump beam in pump1.txt, seed beam in seed1.txt, device parameters in dp1.txt, resolution parameters in sp1.txt, nonlinear index in n2.txt, and absorption function in absof1.txt. Run a simulation with a wide-band, chirped seed pulse, without n_2 and absorption:

```
opa_ex3md file *t60 -tf pump1 -tf seed1 -tf dp1 -tf sp1
```

The file seed1.txt now includes the section

```
mod [
 type chirp
 t0 6e-12
 kr 3e25
]
```

where 'mod' means a modulation function, which is multiplied into the amplitude. 'chirp' selects a function of the class Func1Chirp, which is a complex exponential $\exp(i\ \mathrm{kr}\ (t-t_0)^2)$. 'kr' is a parameter in Func1Any, and it represents a real constant which is used in various ways by several of the possible function. Note that t0 for the 'mod' function is set to the same value as for the 'pulse' function (6e-12 s). If they were different the centre of the spectrum would be shifted because the instantaneous modulation frequency at the centre of the pulse would be nonzero.

Plot the input and output signal spectra:

```
g = gfm3("t60")
u = g.tsr(1,2)
clf()
plot(u.la*1e9, u.m*200)
plot(u.la*1e9, g.ts(2,2))
```

The spatially resolved spectrum is stored in tnfs, and when it is plotted for the centre element and three adjacent elements we can see signs of back conversion in the centre:

```
v = g.tnfs(2,2)
clf()
plot(u.la*1e9, v[:,0,8])
plot(u.la*1e9, v[:,0,9])
plot(u.la*1e9, v[:,0,10])
plot(u.la*1e9, v[:,0,11])
```

The spectral phase is not stored directly in the file, but it can be computed from the complex amplitude (the 'e' field). The sign conventions for the fields in Sisyfos is such that ifft() must be used to go from time to frequency:

```
e = g.e(2,2)
e1 = e[0,:,0,8]
s1 = fftshift(ifft(e1))
phi = unwrap(angle(s1))
clf()
plot(u.la*1e9, phi)
```

Plot the pump and signal pulses

```
clf()
plot(g.p(1,3),"b--")
plot(g.p(2,3),"b")
plot(g.p(1,2)*200,"g--")
plot(g.p(2,2),"g")
```

The format of the n_2-matrix in the file n2.txt is that the first line specifies the number of dimensions (2) and the size of each dimension (3, 3), and the values follow on the subsequent lines. The values are artificially high to see an effect. To run a simulation with n_2, type:

```
opa_ex3md file *t61 -tf pump1 -tf seed1 -tf dp1 -tf sp1 -tf n2
```

Compare the results

```
g0 = gfm3("t60")
g1 = gfm3("t61")
clf()
plot(g0.ts(2,3)[482:542])
plot(g1.ts(2,3)[482:542])
clf()
plot(g0.ts(2,2))
```

```
plot(g1.ts(2,2))
clf()
plot(g0.tff(2,3)[0,:])
plot(g1.tff(2,3)[0,:])
```

The transmitted pump spectrum has been broadened by self-phase modulation, and the signal gain spectrum is shifted. The far-field of the transmitted pump has been broadened outside the matrix, so the simulation should be repeated with higher spatial resolution if correct results were required.

The file absof1.txt includes data for an absorption table (absorption in m^{-1} vs frequency in Hz). The values are artificially high to show a clear effect. The option 'stheat.st 1' turns on storage of absorbed energy, see ArrayDataout. Run a simulation with absorption:

```
opa_ex3md file *t62 -tf pump1 -tf seed1 -tf dp1 -tf sp1 -tf absof1
bbo.slices 2 stheat.st 1
```

and compare the results:

```
g2 = gfm3("t62")
clf()
plot(g0.ts(2,2))
plot(g2.ts(2,2))
```

The energy is only slightly reduced. The actual absorption spectrum for each beam can be retrieved by

```
u = g2.get("PropCrys.0.bd")
v = g2.tsr(1,1)
clf()
plot(v.la*1e9, u.a0.absi)
```

where 'bd' means beam data and contains substructures for each beam. The tsr() method is used to get the wavelength table, which is not stored in 'bd'.

Now inspect the distribution of absorbed power in the crystal:

```
u = g2.get("MultiSlice.0.heat")
clf()
imshow(u.st.mr[0,:,:])
imshow(u.st.mr[1,:,:])
```

Type 'print u' to inspect the whole data structure stored by ArrayDataout. The fields depend on the options passed to ArrayDataout.

Run a simulation with wide-band mode turned off and compare the spectra:

```
opa_ex3md file *t65 -tf pump1 -tf seed1 -tf dp1 -tf sp1 wb 0
.....
g0 = gfm3("t60")
g1 = gfm3("t65")
clf()
plot(g0.ts(2,2))
plot(g1.ts(2,2))
```

The spectrum in standard mode is much narrower (and incorrect). The wide phase-matching bandwidth of this non-collinear interaction depends on details which are only represented correctly in wide-band mode.

Finally, run simulations with and without noise in an OPA without seed input

```
opa_ex3md file *t63 -tf pump1 -tf dp1 -tf sp1
opa_ex3md file *t64 -tf pump1 -tf dp1 -tf sp1 noise 1
```

and compare the results

```
g0 = gfm3("t63")
g1 = gfm3("t64")
clf()
plot(g0.ts(2,2))
plot(g1.ts(2,2))
```

The noise is essential in optical parametric generators (OPG) and can be important in OPAs with high gain.

7.1 Overlapping spectral ranges

If you increase the temporal resolution the spectral ranges of the beams in the simulation get wider and may eventually overlap. For example, if you run

```
opa_ex3md file *t68 -tf pump1 -tf seed1 -tf dp1 -tf sp1 nt 2048 dt 6e-15
```

you get the warning message 'Overlapping spectra for beams: 0, 1'. This means that the spectral ranges of two beams with the same polarisation overlap. You can see the spectral ranges of the beams by typing

```
g = gfm3("t68")
s1 = g.tsr(2,1)
s2 = g.tsr(2,2)
clf()
plot(s1.nu/1e12)
plot(s2.nu/1e12)
grid()
```

The actual spectra do not necessarily overlap, as you can check by

```
clf()
plot(s1.nu/1e12, s1.m)
plot(s2.nu/1e12, s2.m)
```

Thus, the warning does not necessarily indicate a problem, but the user should check. In this example the beams are non-degenerate because they are non-collinear, so even if the spectra did overlap it would be ok. In general, if two beams with the same polarisation have components with the same frequency and direction they should be treated as a single beam, and if they interact the interaction should be treated as degenerate. Sisyfos checks only the spectral overlap automatically, so when the warning appears the user should check the angular overlap and find out if the beams are degenerate.

8 Example 4 - OPO

The examples so far have been OPAs, which are non-resonant devices. This example shows how to add a resonator, use quasi-phasematching (QPM), compute the temperature in the crystal from the absorbed power, and run a new simulation which includes thermal effects. It also introduces some additional features of Dataout and gfm3. Figure 8.1 shows the OPO.

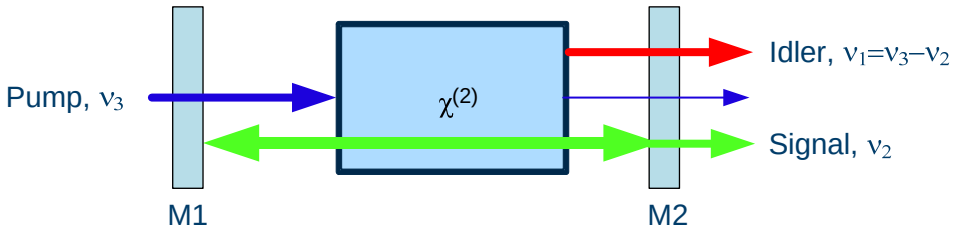

Figure 8.1 *Optical parametric oscillator with resonator formed by the mirrors M1 and M2. In this example, only the signal beam is resonant, and the pump beam makes a single pass through the crystal.*

As explained in Section 4, when simulating a resonator it is convenient to specify tsf (the time slice factor, i.e. the ratio between the round-trip time and the time slice length) and nt instead of specifying dt directly. Therefore, the parameter section of the program opo_ex4 defines 'tsf' in the argument to setup_std_param4(). Sisyfos computes the time resolution from $\mathrm{dt} = t_R/(\mathrm{tsf} \cdot \mathrm{nt})$. 'tlen' is the length of the simulation interval. The number of round trips simulated is $\lceil \mathrm{tlen}/t_R \rceil$.

The example OPO is based on periodically poled $LiNbO_3$ (PPLN), so the 'bbo' substructure from the OPA examples has been replaced by 'lno'. Most of its fields are the same as for BBO, but to show an alternative, the absorption is given by a vector with an element for each beam instead of a function of frequency. In addition, a 'temp' field is defined to input a 3D temperature distribution for the crystal. 'ref_temp' is the reference temperature for thermo-optic calculations.

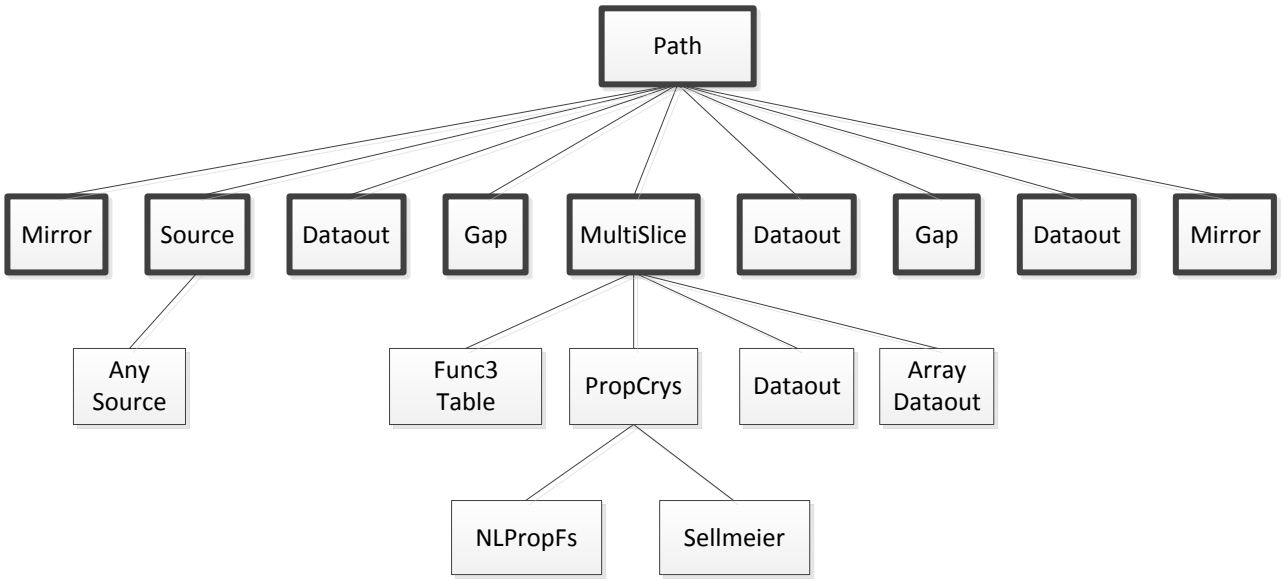

Figure 8.2 Optical path of the OPO. The optional apertures (represented by Mirror objects) on each side of the MultiSlice object have been omitted.

Parameter structures with standard parameters for mirrors are created by Mirror::param_struct(nb), where 'nb' is the number of beams. These structures contain fields for reflectance (ref, default all 1), radius of curvature (rc, default 0, which means plane) and more advanced options which can be found in the documentation for the class Mirror. Other changes in the parameter section are the 2-element vector 'gaps', which holds the length of the air gaps between the ends of the crystal and the mirrors, and the field 'ap' for the aperture radius. Apertures can be added to catch light which would diffract out of a real resonator. If this light was not caught by an aperture it could spill outside the matrix and alias into the other side.

The definition of the optical path differs from the OPA examples by the calls

```
path->set_closed(true);
path->set_two_pass(true);
```

The first tells Path that it represents a resonator, and the second that the beams propagate in both directions. These methods are independent: A ring resonator would be closed but not two-pass, and an amplifier might be two-pass but not closed. Figure 8.2 shows the structure of the optical path.

The mirrors are inserted one at each end of the path. Since an end-mirror should operate on the beam only once in each round trip, they are set to work in the forward direction, but they could equally well have worked in the backward direction.

Resonators and mirrors

A Path can be open, two-pass (which implies a reflector at one end), or closed (i.e. a resonator, which implies reflectors at both ends). Mirror components act only as losses and input/output couplers, and they have nothing to do with the definition of resonators and the optical layout. A closed path is a resonator even if it contains no Mirrors. Without Mirrors all beams remain in the Path, and the Mirrors are necessary to let some of the light out. By definition, the beam reflected by a Mirror remains in the Path, and the transmitted beam is coupled out. For example, a singly resonant OPO must be defined with a closed Path, and the non-resonant beams must be coupled out through Mirrors with zero (or low) reflection.

Sisyfos does not know the geometry of the optical path, so any beam transformations caused by the geometry must be added explicitly: The angle of incidence on a mirror is not specified, so in the case of non-normal incidence on a spherical mirror the mirror must be specified as astigmatic. If a beam is flipped by reflection from an odd number of mirrors, this must be modelled by inserting a FlipRot object in the path. Similarly, if a beam is rotated in a non-planar resonator, the rotation must be specified explicitly.

The Source object is placed after mirror 1, and it too operates in the forward direction. The Source object *replaces* the content of the beam(s) for which it has sources, so it does not support a partially resonant pump beam. Such a device could be simulated by adding the Source to the input port of the Mirror object instead of directly to Path. A gap and an optional aperture are inserted between the first Dataout and the crystal. The air gaps, apertures and crystal are set to work in both directions.

Dataout objects are inserted next to the mirrors and at the end of the crystal. The one at the crystal is applied in both directions, whereas the others are only applied in the forward direction because when the mirrors are plane, the beam will only change power between the forward and backward pass. With curved mirrors, it might be necessary to apply all Dataouts in both direction to monitor the changes in the far-field.

The QPM crystal is modelled in the bulk-approximation, and for this reason the phase mismatch is forced to be zero for the centre frequencies by the call

```
prop->add_chi2_proc(0,1,2, chieff, 0, 0);
```

where the two zeros at the end control the phase mismatch. See the documentation of PropCrys for details. A factor of $2/\pi$ is included in chieff to account for the effect of QPM [5]. The default Sellmeier equation in the example program includes temperature dependence. If an ordinary Sellmeier equation is used, temperature dependence can be added by putting the Sellmeier object inside a ThermoOptic object. The thermo-optic coefficients (TOCs) can be constants or functions of wavelength or frequency. The example program makes a ThermoOptic object if a function is specified in the parameter field 'lno.toc'. Since all the beams are polarised in the same direction,

only a single TOC-function is used. Near the end of the program, 'tsf' is passed to main_init2a() instead of 'dt'. The argument is interpreted as tsf when it is ≥ 1.

Run the example program by typing

```
opo_ex4md -tf opopar1 file *y000
```

and plot the pulse shapes of the input pump and the output of all beams:

```
g = gfm3("y000")
u = g.apr(1,3)
t = u.t*1e9
clf()
plot(t, u.m)
plot(t, g.ap(2,3))
plot(t, g.ap(2,2))
plot(t, g.ap(2,1))
```

'ap' is the power averaged over each round trip, so the length of the time axis corresponds to the 'tlen' parameter. Since the 'p' option was for beam 2 in Dataout 2, you can also retrieve the power at full time resolution:

```
u = g.p(2,2)
```

This is now a 32×76 array, where 32 is the number of sample points per time slice (nt) and 76 is the number of round trips.

```
v = u.mean(0)
plot(t, v)
```

gives the same result as 'plot(t, g.ap(2,2))'. Plot the rapidly varying power for two round trips

```
clf()
plot(u[:,32])
plot(u[:,60])
```

and notice how the character of the fluctuations change from the leading edge of the pulse to the saturated part on the trailing edge. You should check that the near-field and far-field are contained in their matrices, especially for the resonant beam. To get data for the backward direction, use a negative position number, e.g. g.tnf(-2,1) is the total near-field for beam 1 at position 2 in the backward direction.

```
clf()
plot(g.tnf(1,2)[0,:])
plot(g.tnf(2,2)[0,:])
plot(g.tnf(3,2)[0,:])
plot(g.tnf(-2,2)[0,:])

clf()
plot(g.tff(1,2)[0,:])
plot(g.tff(2,2)[0,:])
```

'tnf' is the fluence integrated over the whole pulse. 'nf' stores mean intensity over each time slice, and it shows how the beam profile evolves during the pulse. It is a 3D array with dimensions (round-trips, ny, nx).

```
u = g.nf(2,2)
clf()
plot(u[32,0,:])
plot(u[40,0,:])
plot(u[50,0,:])
plot(u[60,0,:])
```

Similarly, 'ts' is the spectrum integrated over the pulse and 's' is the spectrum in each time slice.

```
u = g.tsr(2,2)
v = g.s(2,2)
la = u.la*1e9
clf()
plot(la, u.m)

clf()
plot(la, v[32,:])
plot(la, v[40,:])
plot(la, v[50,:])
plot(la, v[60,:])
```

For comparison, run a simulation without apertures (ap=0 turns the aperture off)

```
opo_ex4md -tf opopar1 file *y010 ap 0
```

and compare the results:

```
g0 = gfm3("y000")
g1 = gfm3("y010")
clf()
plot(g0.tnf(2,2)[0,:])
plot(g1.tnf(2,2)[0,:])
```

You can see that the near-field extends outside the matrix if the aperture is omitted. It is of course a matter of judgement how to set the aperture. The default value in this program is 0.5 mm, which is much greater than the 0.2 mm exp(-2) radius of the pump beam.

Aborting a simulation

A simulation can be aborted by Ctrl-C. Sisyfos will complete propagation through the top-level Path object, so the effect is not immediate. All the components will finish properly, and the result file will be valid, but it is not possible to restart the simulation from the point where it was stopped. Don't type Ctrl-C more than once if you want a valid result file.

8.1 Thermal effects

Heating of the crystal by absorption can affect the beams by thermal lensing and thermal phase mismatch. Sisyfos includes a program called find_temp1, which can compute the steady-state temperature distribution in a crystal from the distribution of absorbed power and the boundary conditions. find_temp1 can only handle simple geometries with rectangular crystals and cooling through the side faces. Cooling through the end faces is neglected.

First, run a simulation with absorption (the absorption coefficients are exaggerated to show the effect) and 5 slices along the crystal:

```
opo_ex4md -tf opopar1 file *y020 lno [abso [10 1 1] slices 5]
```

The file opopar1.txt includes the line 'stheat [st 1 srt 1]' with options for the ArrayDataout object, which stores absorbed energy. 'st' means sum over time, that is, a 3D array where the absorbed energy in each spatial cell is stored. Similarly 'srt' means sum over transverse coordinates and time, i.e. the total absorbed energy in each slice. The fields in the stored structure correspond to these parameter names:

```
g = gfm3("y020")
v = g.get("MultiSlice.0.heat")
v.fields()
```

'v.st.mr' contains energy density in J/m^3.

```
u = v.st
clf()
imshow(u.mr[2,:,:])
clf()
imshow(u.mr[:,0,:])
clf()
plot(u.mr[0,0,:])
```

```
plot(u.mr[1,0,:])
plot(u.mr[2,0,:])
plot(u.mr[3,0,:])
plot(u.mr[4,0,:])
```

The absorbed power increases along the crystal (except in the last slice) because it is mainly the idler that is absorbed. ArrayDataout has several additional options. See the documentation for ArrayDataout, play with the store options, and look at the resulting data if you want to understand the details.

The following calculation checks the energy balance, assuming that the pump and idler are fully transmitted by mirror 2 and that mirror 1 is HR for the signal. 'srt' contains energy density in J/m.

```
m2_ref = g.get("par.m2.ref")
oc = 1 - m2_ref[1]
w_in = g.w(1,3)
w_out = g.w(2,3) + g.w(2,1) + g.w(2,2)*oc
w_loss = w_in - w_out
v = g.get("MultiSlice.0.heat")
w_abso = sum(v.slice_tab*v.srt)
```

The small difference between w_loss and w_abso can be attributed to the apertures. The steady-state temperature can now be found with find_temp1.

```
find_temp1md -tf temp_par pt -bf y020.sis MultiSlice.0.heat.st file *y020_t
```

The parameter file temp_par.txt reads

```
f_rep 1e4            % Pulse rate
kappa 4.4            % Thermal conductivity, W/(m K)
t_sink 300           % Heat sink temperature
c_sink [0 1e4 0 1e4] % Conductivity from faces to sink, W/(m^2 K).
                     % Cooling through +x and -x faces
pt.v_out 0           % Use 0 outside the table (instead of failing)
smode 0              % 0 means use value from input file
points [10 30]       % Rectangular crystal
```

A temperature calculation with a single matrix element would be meaningless, so smode 0 has a special meaning in find_temp1md – it makes the program use the same smode as in the input file. The element size (scale) will also be the same as in the input file because no other value is given. The pulse rate is simply a scaling factor for the power. The PPLN crystal is taken to be narrow in the x-direction and wide in the y-direction, with cooling through the wide faces (normal to -x and -x). Inspect the resulting temperature:

```
q = rs.ReadSis5("y020_t")
te = q.get("temp")
clf()
imshow(te.mr[0,:,:])
clf()
plot(te.mr[0,0,:])
plot(te.mr[0,:,0])
```

Because the beam is narrow compared to the crystal, the temperature distribution is almost round even though the cooling is only through the x-faces. Run a new simulation with temperature:

```
opo_ex4md -tf opopar1 file *y021 lno [abso [10 1 1] slices 5
temp [-bf y020_t temp v_out 300]]
```

Compare results:

```
g0 = gfm3("y020")
g1 = gfm3("y021")
clf()
plot(g0.ap(2,2))
plot(g1.ap(2,2))
clf()
plot(g0.ts(2,2))
plot(g1.ts(2,2))
clf()
plot(g0.tnf(2,2)[0,:])
plot(g1.tnf(2,2)[0,:])
clf()
plot(g0.tff(2,2)[0,:])
plot(g1.tff(2,2)[0,:])
```

The signal energy is reduced, the pulse shape is changed, the spectrum has been shifted by temperature tuning, the near-field is strongly focused by thermal lensing, and the far-field is correspondingly broader. Temperature calculation and OPO simulation should now be iterated until the results converge. This is convenient to do with a python script, which is the topic of the next section.

The size of the matrix in the temperature calculation should correspond to the physical size of the crystal. In this example it was 0.95 mm by 2.95 mm (the crystal size in smode 4 is (2 nx - 1) by (2 ny - 1) elements). The matrix in the OPO simulation can be different because it only needs to contain the beams and has nothing to do with the physical crystal. In this example, the beam matrix is wider than the crystal in the x-direction and smaller in the y-direction.

The Func3Table objects, which are used to input absorbed power density to find_temp1 or temperature to opo_ex4md, interpolate if necessary, so the resolutions for the OPO simulation and

the temperature calculation can also be different. The 'pt.v_out 0' option to find_temp1 allows the temperature matrix to be wider than the beam matrix and sets the absorbed energy density outside the beam matrix to 0. On the other hand, the 'temp.v_out 300' option passed to opo_ex4md allows the beam matrix to be wider than the temperature matrix and sets the temperature to 300 K for points outside it.

Because most available Sellmeier equations do not have temperature dependence, we also demonstrate the use of thermo-optic coefficients. In this example the single thermo-optic coefficient is simply taken to be a linear function of wavelength, $10^{-5} + \lambda/(1\,\mathrm{m})$. In reality a table of measured data would be more likely.

```
opo_ex4md -tf opopar1 file *y022 lno [abso [10 1 1] slices 5
temp [-bf y020_t temp v_out 300]] lno.toc [type linear kr 1 v_base 1e-5]
```

9 Running Sisyfos from a python shell

If you need to run a lot of simulations and change many parameters, it becomes cumbersome and error-prone to maintain many parameter files or override parameters on the command line. In such cases it can be convenient to run Sisyfos from a python shell and take advantage of python's powerful features for processing strings with parameters. Sisyfos also has python classes to let you run simulations in parallel threads. For documentation of python, see [3] or [4].

9.1 Struct class

One of the features for parameter handling is the Struct class in the module SisUtil. This resembles structures in matlab, and it has methods to convert the content to a parameter string that can be passed to Sisyfos.

```
from SisUtil import *  # Usually done in the startup file
u = Struct()
u.a = 4
u.b = [3,5,7]
u.c = Struct()
u.c.p = 5
u.c.q = 6
print u
print u.text()
print u.textf()
print u.textf("abc")
u.disp()
```

'print u' invokes the __str__() method on u, which produces a format which is human-readable but not readable by Sisyfos. Long vectors or arrays are not displayed in full. 'text()' returns a single-line format that can be read by Sisyfos. 'textf()' returns a similar format, but with multiple lines

and indentation to show the structure. This is human-readable and can also be read by Sisyfos. Both text() and textf() omit field names that begin or end with underscore, so such fields can be used internally by classes derived from Struct(). The final call to textf(), which includes a field name as argument, puts the content of the Struct inside brackets for this field name. It is also possible to call text() in this way. 'u.disp()' is a short-hand for 'print u.textf()'.

It is not allowed to create multi-level structures directly as in the line below – the field 'd' would have to be defined first:

```
u.d.x = 7            # ERROR - u.d not defined yet
```

Instead you can use the set() method to create multi-level structures:

```
u.set("d.x", 9)
u.set("e.f.g.x", 11)
u.e.f.h = 12         # Works since u.e.f was created above
print u
```

A structure can also be set from a python dictionary:

```
d = {"a":3, "b":{"c":4, "d":[5,6]}}  # Dictionary
u = Struct(d)
u.set("p.q", {"x":8, "y":9})
print u
```

9.2 Run-functions

The module SisRun contains a function 'run', which runs a command line formed by joining multiple strings. This can be illustrated by rerunning some simulations from example 2.

```
import SisRun as sr
com = "opa_ex2md"

s = Struct()
s.type = "simple"
s.set("simple_p.pulse", {"type": "sgauss", "wt":5e-12, "t0":15e-12})
s.simple_p.beam = Struct({"type": "sgauss", "wxy":1e-3})
s.ta = 0
s.tb = 30e-12
s.w = 2e-6

pump1 = """
pump [
 type simple
 simple_p [
```

```
  beam [ type ap_ellip wxy 1.2e-3 ]
  pulse [ type sechsq wt 4e-12 t0 15e-12 ]
 ]
 ta 0 tb 30e-12 w %g
]
"""

sp = Struct()
sp.smode = 2
sp.points = 32
sp.nt = 64

sr.run(com, s.text("seed"), pump1 % 5e-3, sp, "seed.w 1e-6 file *t40b")
```

The setting of 's' illustrates several ways to set a Struct. The name 's' is used only at the python level, and it could have been anything.

'pump1' is simply a string variable in python. The triple quotes allows the string to be split over multiple lines. Note the '%g' in the final line – this allows a value to be inserted, with format specifiers similar to printf in C. The substitution is done by the fragment 'pump1 % 5e-3' in the argument to run().

The run() function can take any number of arguments, which must be strings or Structs. Strings are concatenated directly, Structs are first converted to strings by the text() method. The text() method must be called explicitly for s in order to supply the field name 'seed' and wrap the contents in brackets. In contrast, sp can be converted automatically because its content belong on the root level and not inside another field. Note also how seed.w can be overridden in the last argument string, just as before.

Without using python, command lines could be built by combining many small text files. Even this approach is more convenient with python because each file can be replaced by a python string, and many strings can be defined in the same file. Moreover, strings offer the formatting mechanism for inserting values.

Structures have the additional advantage of making it simple to add or modify individual fields inside it. But the full power and flexibility of an interactive script language like python lies in allowing the user to define functions or classes to create parameter structures (or strings, for that matter). Such functions can return parameter structures based on computation instead of just specifying the value of each field directly. Here is a simple example where the input data is total matrix size instead of resolution, and where some meta-data is included in the file name. It is defined in the file 'simpar1.py' in example 2.

```
def simpar1(fname, smode=2, points=32, matsize=3e-3):
  s = Struct()
  s.file = fname + "_%d_%d" % (smode, points)
  s.smode = smode
```

```
  s.points = points
  s.scale = matsize / points
  return s
```

A very useful feature of python is that it allows default values for arguments, as shown above, and that functions can be called by specifying selected arguments by keywords, as in

```
from simpar1 import *
u = simpar1("test", matsize=6e-4, smode=4)
```

Thus, you can define a very general function with lots of arguments without having to specify all of them or remember their order when it is called.

ReadSis5 and gfm3 return Structs when you use the get() method to retrieve a node that is not a leaf:

```
g = gfm3("t40")
u = g.get("par")
u.disp()
```

u is now a structure with all the parameters from the run t40, and you can modify some of them and use it in a new simulation:

```
u.file = "t40c"
u.pump.w = 4e-3
sr.run(com, u)
```

Error messages from programs running under python

If the Sisyfos application fails to start, for example because a dll-file is missing, python displays an error code which is not very informative. If you get incomprehensible error codes from sr.run() you should try to run the program directly in the command window without python. This usually gives better diagnostics.

File name separators in python

The python functions in Sisyfos translate file name separators to the format used by the system ('\' on Windows and '/' on Linux), so you can use either '\' or '/' in their arguments. Note, however, that '\' is used as an escape character in python strings (so that for example '\n' means newline), so a '\' in a string should be written as '\\'. In environment variables you can use single '\'s because these strings are passed directly from windows to python without going through the python parser.

9.3 Predefined parameter structures

The python module SisParam contains classes to make some of the more common parameter structures. It is not essential to use these classes, but they can save typing. For example, the class SisParam.Func1AnyPar represents a parameter structure for Func1Any, and the constructor has the form

```
Func1AnyPar(ftype, v_base=0, v_scale=1, wt=1, t0=0, invert_t=0, rep_f=0, t0p=0,
    level=0.5, ord=2, kr=1, kc=1, ramp=0.1, edge_type='step', dir=1)
```

It handles most of the parameters which Func1Any can accept, and since they have default values you only need to specify the ones you want to override. Func1AnyPar is derived from the SisUtil.Struct class, so it inherits the methods text(), disp() etc. For example

```
import SisParam as sip
u = sip.Func1AnyPar("sgauss", t0=1e-8)
u.disp()
```

Regardless of which fields you set in the constructor, you can also set individual fields as in other Structs, e.g.:

```
u.wt = 3e-9
u.v_base = 5
```

There are some cases which Func1AnyPar does not handle, and they are related to composite functions such as Func1Sum, Func1Prod and Func1Table. Since Func1Any parameters can in principle form an arbitrarily large tree, it is not possible to write a simple function which can create any such tree. Func1AnyPar has to stop at a level where it can offer convenience without excessive complication, and more complicated structures have to be build up gradually by creating substructures and assigning them to higher-level structures.

There is a similar class Func2AnyPar with constructor of the form

```
Func2AnyPar(ftype, v_base=0, v_scale=1, wxy=1, pos0=0, ang=0, e_order=2,
    level=0.5, ord=2, ramp=0.1, shape=0, mode=0)
```

The parameter classes for AnySource are somewhat more complicated. First, there is a base class AnySourcePar with methods

```
setw(w, ta, tb)  # Sets energy in interval [ta, tb]
setf(f0, ta, tb) # Sets peak fluence in interval [ta, tb]
setp(p0)         # Sets peak power
seti(i0)         # Sets peak intensity
set_periodic(self, rep_f, t0p=0, jitter_f=None)  # Sets periodicity and jitter
```

which correspond roughly to the methods of the C++ interface IBeamSource. The constructor for AnySourcePar is not meant to be used by itself, instead there are derived classes for the three types of source:

```
SimpleSourcePar(pulse=None, pamp=0, beam=None, bamp=0,
    mod=None, pmod=None, modspec=None, rep_f=0)
FileSourcePar(fname, pos=1, beam=1, t0=0, tscale=1, mag=1, pos0=0,
    pscale=1, rem_phase=0, interp=0)
Func3SourcePar(pulse=None, pamp=0, mod=None)
```

See the html documentation of the corresponding C++ classes for description of the parameters. An example how parameter classes can be used together is

```
u = sip.SimpleSourcePar(
  pulse = sip.Func1SgaussPar(wt=5e-10, t0=2e-9),
  beam = sip.Func2SgaussPar(wxy=[5e-4, 1e-3], ang=-1, e_order=4))
u.setw(w=1e-3, ta=0, tb=1e-8)
```

Make this structure and inspect the result by u.disp(). As before, it is possible to modify individual fields after creating the structure:

```
u.simple_p.pulse.t0 = 3e-9
u.simple_p.beam.ord = 4
```

SisParam is under development, so the classes may change and new classes will probably be added.

9.4 Multiple threads

SisRun.run() waits until the called program has finished. SisRun.runt() is called like run(), but it starts the program in a new thread and returns immediately. The computer will run inefficiently if too many threads are started or if the total memory requirements of the running programs exceed the physical memory. It is the user's own responsibility to avoid these pitfalls. The task manager is useful for monitoring the memory usage. The number of threads for Sisyfos should not exceed one per core, or two per core with hyper-threading. It is advisable to leave one core for interactive and system tasks. Here is an example:

```
sr.runt(com, s.text("seed"), pump1 % 4e-3, sp, "seed.w 1e-6 file *t45a")
sr.runt(com, s.text("seed"), pump1 % 6e-3, sp, "seed.w 1e-6 file *t45b")
sr.runt(com, s.text("seed"), pump1 % 7e-3, sp, "seed.w 1e-6 file *t45c")
```

Messages from the different threads will be interleaved in the window, and they can also appear in the middle of interactive typing. This nuisance can be minimised by using two python shells – one for running simulations and one for interactive work.

If the computer has enough memory, it is usually more efficient to run multiple simulations in parallel than to use multiple threads for a single simulation. When running parallel simulations, each of them should use only one thread (which is the default).

The class SisRun.JobQueue can manage a queue and run a limited number of jobs in parallel. Jobs can be added to JobQueue at any time, and it remains and can be used again after all the jobs have finished. The argument to JobQueue is the maximum number of threads, and addcom() takes the same kinds of arguments as run() and runt(). The following example will run 5 simulations using two threads:

```
q = sr.JobQueue(2)
q.addcom(com, s.text("seed"), pump1 % 4e-3, sp, "file *t46a")
q.addcom(com, s.text("seed"), pump1 % 5e-3, sp, "file *t46b")
q.addcom(com, s.text("seed"), pump1 % 6e-3, sp, "file *t46c")
q.addcom(com, s.text("seed"), pump1 % 7e-3, sp, "file *t46d")
q.addcom(com, s.text("seed"), pump1 % 8e-3, sp, "file *t46e")
```

9.5 Iteration for thermal effects

A job for JobQueue is not restricted to be a simple command, it can also be an object of a class derived from SisRun.PythonJob. This is useful for iterative calculations, and the program 'iterate.py' in example 4 demonstrates this use. This program is not a general solution, but it is an example which users can modify according to their needs. For example, it may be necessary to check for convergence of other quantities than just the signal energy. In the example below we have used some of the parameter files which already existed for example 4, but they could of course have been replaced by structures. The file opopar2.txt is similar to opopar1.txt except that the seed for the random number generator (ran option) is fixed to avoid random fluctuations and that a lower time resolution is used to save time.

```
os.chdir("../ex4")
import iterate as it
com = "opo_ex4md"
tcom = "find_temp1md"
q = sr.JobQueue(2)
y = it.TempJob(com, "-tf opopar2", tcom, "-tf temp_par", 10, 0.01, "tsim")
q.addjob(y)
```

JobQueue will call the run2() method of the TempJob object. TempJob runs in a single thread because each iteration depends on the previous. Other jobs could be run in parallel by adding them to the JobQueue.

In this example, the signal energy converged to the specified tolerance after 4 iterations. A small python script can read the files and plot the signal energy versus iteration:

```
v = []
```

```
for i in range(4):
  g = gfm3("tsim_%d" % i)
  v.append(g.w(2,2))

clf()
plot(v)
```

Sometimes the solution does not converge, and in such cases it can help to introduce damping by using the average of the two (or more) most recent temperature distributions instead of just the last one.

9.6 Running func_test, sell_test and source_test from python

The test programs from Section 5 can be run by python functions which automatically read the result files and return the data as python structures, which is convenient for plotting. The python file FuncTest.py is located in the ft directory.

```
import FuncTest as ft
import SisParam as sip
u = ft.FuncTest()
# Run func_test
q = u.func(1, "type sechsq", x0=-5, points=100, dx=0.1)
print q
clf()
plot(q.x, q.v)

# Run sell_test
q = u.sell("mat_dir ../../MatData mat KTP name Kato2002", la0=8e-7, dla=1e-7, points=20)
q.n.shape
clf()
plot(q.la*1e6, q.n[:,0])
plot(q.la*1e6, q.n[:,1])
plot(q.la*1e6, q.n[:,2])
```

The last lines plot the principal refractive indices for each axis.

The python program for source_test returns a gfm3 object instead of a structure:

```
p3 = sip.SimpleSourcePar(
  pulse = sip.Func1SgaussPar(wt=5e-10, t0=2e-9),
  beam = sip.Func2SgaussPar(wxy=[5e-4, 1e-3], ang=-1, e_order=4))
p3.setw(w=1e-3, ta=0, tb=1e-8)

g = u.source(p3)
clf()
imshow(g.tnf(1,1))
```

Let us conclude this section by an advanced source-example, where the pulse is determined by a

spectral intensity given as a function of wavelength and a spectral phase which is a function of frequency. Func1Spec computes the complex spectrum and transforms it to a time series. It returns a Func1TableC object, which handles interpolation in the time series. Here is the parameter string for the pulse function:

```
p1 = """
type spec
spec_p [
 spec_f [
  type sgauss
  t0i 1e-6   wt 4e-8
  t0 -300e12
  invert_t 2.99792458e8
 ]
 phase_f [
  type poly
  pk [0 0 5e-26]
 ]
 nt 2048
 dt 5e-15
 v_out 0
]
t0 1e-12
"""
```

spec_p contains the parameters for Func1Spec, and spec_f and phase_f define the functions for spectral intensity and phase, respectively. spec_f.t0i and spec_f.wt are the centre wavelength and half-width of the super-gaussian spectral intesity. spec_f.invert_t converts to a function of frequency. Because the pulse function should represent complex amplitude with respect to a centre frequency, this frequency is added by spec_f.t0 before conversion to wavelength. See Section 5.1 for an explanation of the transformation of the argument. The user must make sure that the centre frequency used in Func1Spec is the same as the centre frequency of the corresponding beam in the simulation program. The user must also set the number of points and time resolution for the intermediate time series to sensible values. The spectral intensity in the example is a super-gaussian, but in practice it might be a table of measured data.

Run func_test and plot the pulse intensity:

```
q = u.func(1, p1, x0=0, dx=1e-14, points=300, cmplx=1)
clf()
plot(q.x*1e12, msq(q.v))
```

Now use the pulse in source_test so you can check that the spectrum is also reproduced correctly. Since the spectrum was super-gaussian in wavelength, is is asymmetric as a function of frequency:

```
p3 = sip.SimpleSourcePar(
```

```
  pulse=p1,
  pamp=1,
  beam=sip.Func2SgaussPar(wxy=1e-3))
p3.seti(1e8)

g = u.source(p3, nt=512, dt=1e-14, tlen=3e-12)
v = g.tsr(1,1)

clf()
plot(v.la*1e9, v.m)
grid()

clf()
plot(v.nu*1e-12, v.m)
grid()
```

Appendix A Introduction to optical frequency conversion

This appendix gives a very brief introduction to nonlinear optical frequency conversion. See e.g. [6] for a thorough presentation.

It is a general phenomenon that nonlinear processes can give rise to new frequency components in signals. This can be an unwanted effect, e.g. distortion in audio systems, or it can be useful, e.g frequency mixing in radio frequency (RF) electronics. The latter phenomenon is used extensively to shift signals into frequency ranges suitable for signal processing. Optical frequency conversion can occur in media where the polarization is a nonlinear function of the electric field. It is conceptually similar to electronic frequency mixing, but there are also some important differences:

- Optical frequency conversion works with propagating electromagnetic waves instead of electric signals.
- The nonlinear optical medium is typically large compared to the wavelength, whereas nonlinear electronic components are typically small compared to the wavelengths they work with.
- Optical media usually have weak nonlinearity where only 2nd and 3rd order terms are significant. Electronic components such as simple diodes can have very strong nonlinearity.

Optical nonlinearity is hardly observable with ordinary light sources, but with intense laser beams it is possible to excite a nonlinear response in certain materials. Nonlinear optical devices have applications in generating coherent radiation at frequencies that cannot be generated directly by lasers. Harmonic generation and sum frequency generation (SFG) can produce higher frequencies and difference frequency generation (DFG) can produce lower. Optical frequency conversion is usually based on second order nonlinearity because this is the strongest term. A second order nonlinearity couples three beams, and we denote their frequencies (in increasing order) ν_1, ν_2 and ν_3. In a DFG process beam 3 and either beam 1 or 2 are input, and the remaining beam at $\nu_1 = \nu_3 - \nu_2$ or $\nu_2 = \nu_3 - \nu_1$ is generated. The DFG process also amplifies the lower frequency input beam, and if the desired output is the amplified beam rather than the difference frequency the device is called an optical parametric amplifier (OPA), which is shown in Fig. 3.1 on page 12. The difference frequency generated in a OPA is usually called the idler.

SFG and OPA/DFG devices can be described well by a set of coupled equations for the slowly varying amplitudes of each beam. The basic equation for beam 1 has the form

$$\frac{\partial e_1}{\partial z} = \frac{i}{2k_1}\nabla_T^2 e_1 - \rho_1\frac{\partial e_1}{\partial x} - \delta_1\frac{\partial e_1}{\partial t} + i\omega_1\gamma e_3 e_2^* \exp(i\Delta k z),$$

where $e_j = e_j(x, y, z, t)$ is the slowly varying amplitude of beam j, z is the coordinate in the beam direction, x and y are the transverse coordinates, k_j is the magnitude of the wave vector of beam j, ρ_j is the birefringent walk-off angle, δ_j is the temporal walk-off rate (with respect to a reference frame which is usually one of the beams), ω_j is the angular frequency of beam j, γ is a nonlinear coupling coefficient, and Δk is the so-called phase mismatch. For collinear beams,

$\Delta k = k_3 - k_2 - k_1$. Sisyfos uses more general equations [1], but this form is suitable for an introduction. The terms on the right-hand side correspond to diffraction, spatial walk-off, temporal walk-off, and nonlinear coupling. The equations for beams 2 and 3 are similar. The combination of beam propagation and nonlinearity can lead to complex and sometimes non-intuitive behaviour, and the coupled equations can be solved analytically only in very restricted cases. Numerical simulation is therefore important both to understand the physics and to design practical devices, and Sisyfos is an attempt to address this need.

Appendix B Parameter syntax

This appendix describes the syntax for command-line parameters (and text files) in detail. Data for one or more fields in the structure are given as

```
<path1> <value1> <path2> <value2> etc.,
```

where <path> is a sequence of names separated by dots, e.g. ”bbo.d22”, <value> is the value assigned to the field, and white space is used as as separator. Field names can contain alphanumeric characters and underscores. <value> for leaf fields are given as:

- Int and Real values are given in the obvious way.
- String values containing white space must be enclosed in [...], and string values without white space can be given directly. Strings cannot contain the bracket characters [and]. (The reason for using brackets instead of quotes is that the command-line handlers in Linux and windows treat them differently). Data can be appended to a string by writing <path> +<value>, with space before the + and not after it, e.g. store1 +[ef=nfn].
- A vector value is enclosed in [...]. The brackets can be omitted for single element vectors.
- An array value is given as [m n_1 n_2 ... n_m elements...], where m is the number of dimensions and n_i is the size of the i’th dimension. The elements are listed in C-order, that is, with the last index varying most rapidly. For example, [2 2 3 10 11 12 13 14 15] corresponds to the 2×3 matrix [10 11 12; 13 14 15].
- <value> for a substructure must be enclosed in brackets, [<path> <value>...].

Data for a substructure, vector or array can be read from a text file by placing a -tf <filename> directive in the <value> position. The file should not contain the outer [...] brackets.

Data for a substructure, vector or array can be read from a binary Sisyfos file by placing a -bf <filename> <path> directive in the <value> position. <path> must be the name of a field in the structured file, e.g. ”do.2.f.1.tnf.m”. The type of the field in the file must be compatible with the field in the ParamStruct.

Text files and binary files are handled differently. A text file is not allowed to contain field names that are not defined in the parameter structure. A binary file may contain additional fields, which will be ignored.

The -tf and -bf directives can also appear in the position of <name>, in which case data from the file are added to the current structure. Again, text files and binary files are treated differently with respect to superfluous fields.

If the value for a field is given more than once, e.g. first in a file and then on the command line, the last specification will be used. In this way, it is simple to override some of the data in a file without editing it.

B.1 Examples

With the parameter structure from opa_ex1, the following command line arguments would be valid and equivalent:

- lams 1e-6 bbo [len 0.005 d22 2e-12]
- lams 1e-6 bbo.len 0.005 bbo.d22 2e-12
- lams 1e-6 bbo -tf pfile1
- lams 1e-6 -tf pfile2
- -tf pfile3

where pfile1.txt reads ”len 0.005 d22 2e-12”, pfile2 reads ”bbo [len 0.005 d22 2e-12]”, and pfile3 contains the same text as line 1 or 2 above.

The -bf option is useful to retrieve parameters from a previous simulation and possibly override some of them, e.g.

```
opa_ex1md -bf oldsim par  bbo.len 0.004 file test2
```

would retrieve parameters from the file oldsim.sis. The -bf option is also useful when data from a simulation is used as input to another stage.

B.2 Help and other special parameters

The fields ”help”, ”hlevels” and ”verbose” are treated specially if they appear in the root structure.

- help <path> makes the program display help information for the substructure indicated by <path>. <path>=’.’ means the root of the structure.
- hlevels N tells how many levels of the structure to display.
- If ”verbose” has an integer value greater than 0, the program will display the final parameter structure before continuing with the actual simulation.

Help displays the data type for each field. For structures it displays (struct) and possibly a class name if the structure corresponds to a standard parameter structure for a class. For example, -help pump 1 may display

```
pump (Struct): AnySource
```

In this case, you can find details about the parameter structure in the documentation for AnySource, and similarly for other classes.

Appendix C Structure of Func1Any

In the documentation for Func1Any, you will see that it has only two methods, param_struct() and make(), both of which are static. 'static' in this context means that the methods are not tied to objects of the class, they are only ordinary functions which are put in a class instead of giving them long and unique names. Func1Any::make() does not return instances of Func1Any but of the interface IFunc1A. An interface defines a set of methods which other classes, in this case the various function classes (Func1Linear, Func1Sgauss and so on) implement. The program using these objects does not need to know exactly what type of object it uses, only that it conforms to the interface.

If you follow the link to IFunc1A you find a class diagram which shows that it inherits three other interfaces and that it is implemented by the various function classes which Func1Any::make() can return. If you follow the link "List of all members" on the page for IFunc1A you see a list of its methods and the interfaces from which they are inherited. IFunc1 contains various get-methods which are used to obtain function values. These are used internally by Sisyfos, but usually not in the main program. IFuncMod0 contains methods for setting v_base and v_scale. IFuncMod1 has methods for setting the parameters of the t-transformation. Similarly, IFuncMod2, which is inherited by IFunc2A and IFunc3A, has methods for setting the parameters for the xy-transformation.

References

[1] Gunnar Arisholm and Helge Fonnum. Simulation System For Optical Science (SISYFOS) - Theoretical background. To be published as FFI Report, 2013.

[2] www.cplusplus.com

[3] www.python.org

[4] Hans Petter Langtangen. *Python scripting for computational science*. Springer, Heidelberg, 3rd edition, 2009.

[5] L.E. Myers *et al.* Quasi-phase-matched optical parametric oscillators in bulk periodically poled $LiNbO_3$. *J. Opt. Soc. Am. B*, 12:2102–2116, 1995.

[6] Robert W. Boyd. *Nonlinear optics*. Academic Press, San Diego, Calif., 2nd edition, 2003.