FFI Norwegian Defence Research Establishment

21/01183

FFI-RAPPORT

Simulation System For Optical Science (SISYFOS)

– tutorial, version 2

Gunnar Arisholm
Helge Fonnum

Simulation System For Optical Science (SISYFOS)

– tutorial, version 2

Gunnar Arisholm
Helge Fonnum

Norwegian Defence Research Establishment (FFI) 14 June 2021

Keywords
Simuleringsverktøy
Ikke-lineær optikk
Lasere

FFI report
21/01183

Project number
1602

Electronic ISBN
978-82-464-3337-0

Approvers
Halvor Bjordal, *Research Manager*
Halvor Ajer, *Director of Research*

The document is electronically approved and therefore has no handwritten signature.

Summary

Sisyfos consists of a C++ library and a set of python modules for simulating optical parametric frequency conversion, lasers, or beam propagation in nonlinear or turbulent media. The Sisyfos library contains 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. A Sisyfos program loosely means an application program making use of the Sisyfos library to simulate some optical device. This report gives an introduction to Sisyfos by guiding the reader through a sequence of example programs of gradually increasing complexity. The example programs form an essential part of the tutorial, and we recommend running the examples while reading the tutorial.

Sammendrag

Sisyfos består av et C++ bibliotek og python-moduler for å simulere optiske parametriske frekvensomformere, lasere eller stråleforplantning i medier med ulineæritet eller turbulens. Sisyfos-biblioteket inneholder 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 generatorer for harmoniske frekvenser. Med Sisyfos-program mener vi et applikasjonsprogram som bruker komponenter fra Sisyfos-biblioteket til å simulere en optisk innretning. 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 vi anbefaler å kjøre programmene etter hvert som man leser.

Contents

1 Introduction

Sisyfos consists of a C++ library and a set of python modules for simulating optical parametric frequency conversion, lasers, and beam propagation in nonlinear or turbulent media. By Sisyfos program we mean an application program making use of the Sisyfos library to simulate some optical device. Most end-users do not use the Sisyfos library directly, but rather get pre-built Sisyfos programs from FFI. Nevertheless, they need some knowledge of the underlying library components to be able to use the Sisyfos programs. 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 and enough programming knowledge to make sense of simple C++ programs. An introduction to nonlinear optics in general can be found in [1], and a more detailed treatment of optical frequency conversion is given in [2]. 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 chapter explains the structure and features of Sisyfos. Installation is covered in a separate "Getting started" guide because this tends to change more frequently than the rest of the tutorial. Chapter 2 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 Chapter 3 describes this issue and shows how to explore resolution parameters using the program from Chapter 2. Sisyfos uses various function objects to represent input data in 1, 2, or 3 dimensions, and these classes are described in Chapter 4. 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. Beam sources are handled in a similar way to functions and are described in Chapter 5. Chapter 6 presents a generalized version of the OPA from example 1 and introduces more of Sisyfos' features. Chapters 7 presents a broad-band non-collinear OPA and discusses how to handle sets of alternative parameters by python strings and string functions. Chapter 8 explains how to run parallel simulations under python. Chapter 9 presents an optical parametric oscillator (OPO) and uses the features introduced in chapter 8 to explore the parameter space. Chapter 10 describes simulation of a simple laser and uses that example to illustrate the use of apertures to aid the simulation.

Sisyfos accepts parameters on the command line or in (possibly nested) text files, and Appendix A describes the syntax of such text parameters. Sisyfos includes python functions for phase-matching calculations, and Appendix B shows some examples of how to use these.

The first version of this tutorial [3] was published in 2012, and although the design has been quite stable, many details of the code have changed since then. The accumulated number of changes to the tutorial examples had now reached a point where a new version was needed. Based on experience and feedback from users, we have also improved and expanded the treatment of some topics and added a new example on laser simulation. This tutorial applies to Sisyfos version 6.0, but most of the features described here are expected to be relatively stable.

1.1 Structure of Sisyfos

Most of the code in Sisyfos is library classes or functions that represent beams, optical components and their operations. The "configuration file", which defines the device to be simulated, is the C++ main program, which creates objects corresponding to the components of the device and assembles 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.

The results from a simulation are stored in a binary file. The format is specific to Sisyfos, but it is fully described in the documentation. Sisyfos includes a set of python functions for reading and analyzing result files, phase-matching calculations, and running Sisyfos programs from a python shell.

1.2 Features

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
- Non-collinear beams
- Degenerate or non-degenerate $\chi^{(2)}$ interactions
- Multiple parametric processes in the same medium
- Nonlinear refractive index (n_2) and two-photon absorption
- Quasi-phase-matching
- Stimulated Raman scattering (SRS)
- Stimulated Brillouin scattering (SBS, only in one dimension, suitable for fibres)
- Laser gain (currently 3-level)
- Atmospheric turbulence
- Noise (semi-classical)
- Resonators

1.3 Beam representation and propagation

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

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

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 wave-vector $\mathbf{k}_0$ have been factored out, and cc means complex conjugate. Sisyfos works in a frame moving with the beam, so the beam at a certain z-position is represented by a 3D array of complex numbers, where the dimensions correspond to x,

y, and $\tau = t - z/v_R$, where v_R is the reference velocity. v_R is usually chosen be the group velocity of the fastest beam. If the system contains different optical media, v_R is set independently in each medium. The transverse components k_{0x} and k_{0y} of the central wave vector are usually zero, but they can be non-zero for non-collinear beams. This allows representation of non-collinear beams without excessively fine spatial resolution.

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

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

where k_x, k_y are transverse wave-vector 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.

In principle, two beams, one for each polarization, would be sufficient to model any optical system with unidirectional propagation along an axis. However, the required temporal resolution in this case would be dictated by the inverse total bandwidth. For a nonlinear interaction of multiple narrow-band beams, it is more efficient to represent the beams individually and use a temporal resolution corresponding to the inverse bandwidth of a single beam.

The details of the nonlinear propagation equations are outside the scope of this tutorial, but they may eventually be described in a separate report on the theoretical background of Sisyfos. 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 as scalars.
- With the exceptions of SRS, SBS and thermal effects, the nonlinear response is taken to be instantaneous.
- Mode amplitudes change little over the distance of a wavelength. This is much less restrictive than the slowly varying envelope approximation (SVEA): Pulses can be very short and subject to rapid dispersive evolution, it is only the nonlinear coupling that is restricted. Nevertheless, this assumption can limit simulation of difference frequency generation to the THz range (also called optical rectification), where the wavelength can become comparable to the length of the nonlinear medium.

The latter restriction is related to the fact that, apart from the special case of SBS, Sisyfos does not handle nonlinear interactions of counter-propagating beams. The limitation of this approximation is discussed in [4].

1.4 Documentation

Details of the Sisyfos C++ library tend to change frequently, so the library is documented in doxygen-generated html-files rather than in a report. The directory structure and where to find the top-level html-file is described in "Getting started". To look up a C++ class, go to C++ modules, select the tab "classes" and then "class index". 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.

2 Example 1 – Optical parametric amplifier (OPA)

The purpose of this chapter 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 Chapter 6. Many details in this chapter are directed at users writing their own Sisyfos application programs, but even if you are only going to use a ready-made program it is worth understanding some of the internal details.

The example program for this chapter is called opa_ex1.cpp and is located in the directory .../Sisyfos6/tutorial/ex1. It simulates a collinear, non-degenerate OPA with a single nonlinear crystal, as shown in Fig. 2.1. You should read the 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.

2.1 Program structure

The program consists of four functions with distinct tasks:

- make_param_struct() sets 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.
- make_optical_path() reads data from the parameter structure, creates an object of the class OpenPath, and fills it with objects corresponding to the optical components.
- run_sim() runs the simulation.
- The main() function calls the other functions and catches exceptions.

Each of these functions is treated in the subsections below.

2.1.1 Parameter structure (make_param_struct())

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

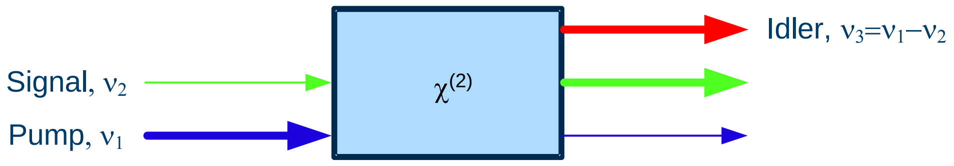


Figure 2.1 Optical parametric amplifier with nonlinear crystal, input beams and output beams. The colours of the beams indicate the relation between the frequencies.

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.

Table 2.1 Simulation parameters (non-physical) for the opa_ex1 program. Parameters are always given in raw SI-units.

ParamStruct and the leaf nodes can be ParamString, ParamInt and so on for different data types. It is possible to set default values for all leaf nodes, value limits for numeric nodes, and size limits for vectors and arrays. The first parameter created in opa_ex1 is an integer called "smode", for spatial mode:

```
ps->add_int("smode", 2, 0, 4);
```

add_int() is a convenience method of ParamStruct which creates a ParamInt object and adds it to the structure. ParamStruct has similar factory methods for other data types. In this example, the ParamInt object has default value 2, limits 0 and 4 and is added to the parent structure under the name "smode". The parameter smode determines the spatial mode of the simulations, where 0 means plane-wave, 1 means cylindrical symmetry, 2 means full 2D, 3 means half-plane, and 4 means a single quadrant.

The next lines are similar except that they define a vector:

```
piv = ps->add_int_vec("points", false, 1, 2, 1, 512);
piv->push_back(32);
```

The arguments to add_int_vec() are the name for the field, a boolean flag which tells if an empty value is allowed, lower and upper limits for the size (if not empty), and lower and upper limits for the values. It returns a pointer to a ParamIntVec object in piv. piv->push_back() adds elements, which serve as default values, to the vector.

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. Table 2.1 summarizes the first few parameters, which are simulation parameters rather than physical parameters. smode, points, scale, nt and dt are described in detail in Chapter 3. Sisyfos has a random-number generator, which 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 ($\beta - BaB_2O_4$ (BBO) in this example) are collected in a substructure with the fields shown in Table 2.2. We use the term Sellmeier equation for functions from wavelength to refractive index,

Parameter	Meaning
se	Name of Sellmeier equation. Refers to a file in ".../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. If theta=0, the program computes the phase matching angle.

Table 2.2 Parameters for the BBO crystal, in the substructure "bbo".

even though Sisyfos allows arbitrary functions and does not restrict them to the Sellmeier form. The angle ϕ is fixed to $\pi/2$ in this example. 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 1.3, 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.

2.1.2 Beam numbers

In this program, the beams are ordered pump, signal and idler. This is an arbitrary choice, and Sisyfos does not make any assumptions about the order of the beams. 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 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++.

In the first version of the tutorial the beams were ordered by ascending frequency, and this convention is still used in some older Sisyfos programs. The motivation for the new convention is that the pump and signal beams can have the same numbers (0 and 1, respectively) in degenerate and non-degenerate OPAs and OPOs, and the idler beam, if present, can be number 2.

2.1.3 main()-function

Although this comes last in the program, it is necessary to describe it before the other remaining functions. It contains a try-catch structure to catch exceptions and display the corresponding error message. After creating the parameter structure, the program runs the lines:

```
ps->read_arg(argc-1, argv+1, "SisyfosConfig.txt");
ps->check();
if (ps->get_help_opt() != 0) return 0;
```

read_arg() first looks for the configuration file "SisyfosConfig.txt" and then parses the arguments on the command line. The main purpose of "SisyfosConfig.txt" is to provide a value for mat_dir, which is usually fixed for a particular computer. Directives on the command line can tell the program to read additional parameters from text files or Sisyfos binary files, and this makes the program flexible in terms of input data. check() checks that all parameters have values within their allowed ranges. If invalid parameters or values are found, check() throws an exception, which will be caught by main().

If the command line contains a help-request (see Section 2.6), read_arg() will display help information. In this case it is not desirable that the program performs an actual simulation, so it checks the method get_help_opt() and returns if help was set.

If the program proceeds, Global::init() initializes factory functions for FFT objects and random-number generators. The function make_optical_path(), which is described in the next section, creates an object of type OpenPath, which is passed to run_sim(). Finally Global::finish() is called to clean up.

2.1.4 Optical path (make_optical_path())

This function makes, and eventually returns, an object of type OpenPath, which is the top-level object. 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. 2.2.

The argument to make_optical_path() is the parameter structure, which was created by make_param_struct() and had values filled in by the read_arg() method called from main(). Before make_optical_path() can fill in the path it reads values out of the structure and into variables in 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 with three elements, corresponding to the three beams. BeamPar includes the centre frequency and also tells if noise should be added to a beam. To keep the noise content consistent, new noise can be added whenever a beam experiences a loss, for example by being reflected by a partially reflecting mirror. In frequency

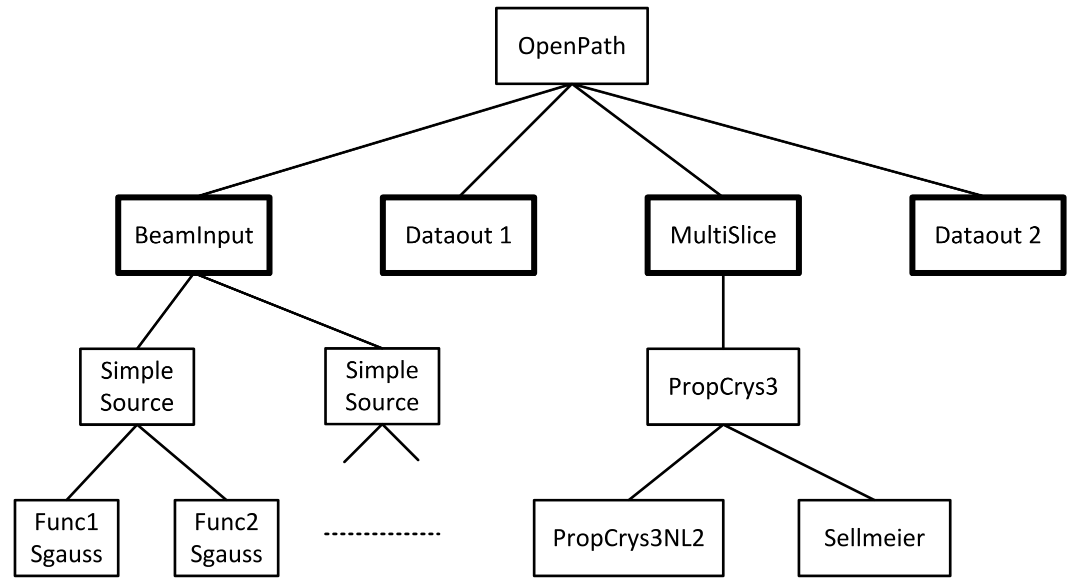


Figure 2.2 Structure of the OPA simulation program. The components in the optical path have thick frames, and they are shown in the same order as in the path. Helper components have light frames. The pump- and seed sources are modelled by SimpleSource-objects. MultiSlice and its subcomponents represent the nonlinear crystal. Dataout are non-physical objects which store beam-data for later analysis.

conversion applications it is usually not necessary to add noise to the pump, but noise in the signal- and idler-beams is essential to simulate spontaneous parametric emission. The BeamPar vector is passed to "path", which passes it on to its subcomponents when they are added.

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 Chapter 5 and used in example 2 in Chapter 6.

The first object in the optical path is a BeamInput, which obtains data from the SimpleSource objects (seed and pump are handled by the same BeamInput-object) 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, non-zero, positive integers. Each Dataout object is also assigned a descriptive text ("Input" and "Output" in this example), which does not have to be unique. This is not used by Sisyfos, but it can be read back from the result file for as information to the user.

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.

- MultiSlice needs a propagator, which can be linear or nonlinear. In this example, the nonlinear propagator PropCrys3 is used.

- The propagator needs a Sellmeier object to represent the dispersion of the crystal. This is made by the factory function SellmeierAny::make().
- The nonlinear propagator needs a solver for the nonlinear equations. PropCrys3 has an associated solver class, PropCrys3NL2, an instance of which is made by PropCrys3NL2::make(). The motivation for having the solver in a separate object is to support flexibility with alternative solvers.
- Zero or more nonlinear processes can be associated with the PropCrys3 object by the method add_nl_term(). Chi2Factory::nondegen() is a factory function for a class representing non-degenerate second-order interactions. There are similar factory functions for other second- and third-order interactions.
- 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.

2.1.5 run_sim()-function

The output file is created by the calls

```
string fname = ps->get_string("file");
unique_ptr<StoreGroupIfc> store(WriteSis5File::make(fname));
```

where the pointer to the file-object is stored in a C++ smart pointer so that the file is automatically closed when the program terminates. Sisyfos has its own binary file format which supports efficient storage of structured data. A file is organized as a tree, so the input parameter structure can easily be stored in the file for future reference, which is done by the line

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

The name of this structure does not have to be "par", but we recommend following this convention.

SimDomainFactory() is a factory function for SimDomain objects, which encapsulate all dependence on spatial mode (see Chapter 3). For convenience, SimDomainFactory() reads 'smode', 'points', 'scale', and 'nt' directly from the parameter structure. main_init1() prepares the OpenPath object to run a simulation with the specified SimDomain object and output file. For reasons which will be explained in Chapter 3, the time resolution is set in a second init function, main_init2(). 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.

2.2 Running Sisyfos

The program can be run in a command window simply by typing its name, optionally followed by parameters to override the default values. However, we find it more convenient to run Sisyfos from an ipython command window, which offers powerful methods to handle parameter strings, and where the Sisyfos functions for reading and analyzing the simulation results can be run in the same window. For information on python, see [5]. The examples have been tested with ipython. They should also work under Spyder, but Spyder handles output from external programs (such as Sisyfos programs) differently from ipython (see Section 8.1), and the behaviour depends on the Spyder version. If you run under Spyder you should set sr.Silent as in the example below, and if you still get problems with Spyder we recommend using ipython for the tutorial examples.

Most of the example commands can be copied from the tutorial pdf-file and pasted into the ipython console. The exception is multi-line commands with indentation. Therefore, the example commands have also been collected in the file example_commands.py in the tutorial directory. Having the commands in a text file also makes it simple to modify them, which we encourage you to try.

If you have installed Anaconda python, run the following in an Anaconda prompt:

```
cd ..../tutorial/ex1
ipython --matplotlib
from pylab import *    # Imports matplotlib, numpy and scipy
import SisRun as sr    # Usually done in the python startup-file
# sr.Silent = 1        # Uncomment this if you run under Spyder
sr.run("opa_ex1", "file *t00")
```

where the dots '....' in the path name must be replaced with the Sisyfos directory. The '- -matplotlib' option sets up the plotting package matplotlib, but it does not import anything. SisRun contains the functions for running Sisyfos applications programs under python. The function sr.run takes an arbitrary number of strings as arguments and joins them to form a command line, which it runs. In this example, 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.

```
sr.run("opa_ex1", "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:

```
sr.run("opa_ex1", "file *t02 bbo [len 3e-3 d22 2e-12]")
```

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

Error messages from programs running under python

There is one exception where a program should be run from the command line instead of under python: If the Sisyfos application fails to start, python may display an error code which is not helpful. In such cases, try to run the program directly in the command window, which sometimes gives better diagnostics. sr.run() displays the actual command string, so you can easily copy and paste it into a command window. One type of error where python does not give good diagnostics is if a dll-file is missing.

2.3 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 2.4.

The python class gfm3 (get filedata module) is designed to conveniently retrieve data stored by Dataout. Note that beam numbers start at 1, so beam 0 in the C++ program corresponds to beam 1 in gfm3, and so on. The python functions are documented in html-files in ".../autodoc/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. Because gfm3 has been designed for interactive use, most of the method names are short rather than self-explaining. If a field was not selected for storage in Dataout, the corresponding retrieval method in gfm3 will fail.

Assuming you have run opa_ex1 successfully and created the file t00.sis, open it with

```
from gfm3 import *  # Usually done in the python startup-file
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 spectrum of beam 3 (idler) at position 2 can be plotted by

```
clf()
plot(g.ts(2,3)*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. ts() returns the spectrum as energy density per unit frequency. The resulting plots are shown in Fig. 2.3.

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

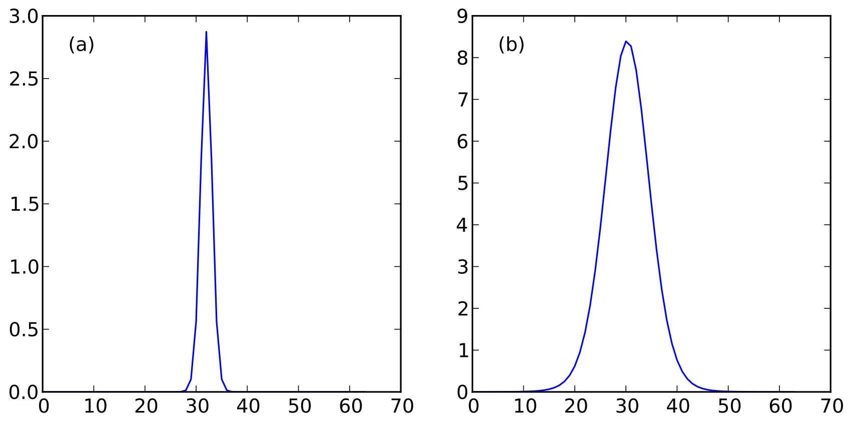


Figure 2.3 (a) Idler spectrum and (b) signal pulse at the output of the OPA.

```
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()
subplot(2,1,1)
plot(u.la*1e9, u.m_la)
xlabel("Wavelength (nm)")
subplot(2,1,2)
plot(u.nu*1e-12, u.m)
xlabel("Frequency (THz)")
```

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). tsr() returns the spectrum both as density per Hz (.m) and per meter (.m_la), and the plot shows both versions. m_la is added by the tsr() method, Sisyfos uses frequency-density internally. 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)")
```

2.4 Exploring the result file

The purpose of this section is to explain the structure of the result file and retrieve data not only from Dataout but also from other objects. 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 recognize 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.

- 'sis' contains information about version and build time.
- 'top' contains information about run time and the optical path.
- 'PropCrys3' contains data from the propagator. Since there can be multiple such objects, there are substructures named '1', '2', etc., where only '1' is present in this example.
- 'MultiSlice' contains data from the MultiSlice object. It has substructures like PropCrys3.
- Other classes, but not all, also store data in substructures with the same name as the class.

Try the commands

```
g.fields("PropCrys3")   # Displays the field names in the substructure
u = g.get("PropCrys3")
u.fields()
```

u is now a data structure in python. Note that the substructure called 'PropCrys3.1' in the file has been named 'PropCrys3.a1' in python because python does not allow names starting with digits. If the device had multiple crystals, the corresponding data structures would be called 'MultiSlice.2', 'PropCrys3.2' and so on. Data stored by PropCrys3 include the refractive index and group index of each beam:

```
print(u.a1.n)
print(u.a1.ng)
```

The command

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

prints a list of the components in the optical path. For more complicated programs, where the path can depend on the input parameters, this can be useful to check that the program has created the intended structure.

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(squeeze(v.a2.p.m))’ (squeeze is necessary to rearrange the shape of p.m).
- ’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.

2.5 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.

```
sr.run("opa_ex1", "-tf par1 file *t03")
```

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

```
sr.run("opa_ex1", "pump -tf pump2 file *t04")
```

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

```
sr.run("opa_ex1", "pump -bf t03 par.pump file *t05")
```

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

2.6 Help

Try typing:

```
sr.run("opa_ex1", "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

```
sr.run("opa_ex1", "help . hlevels 1")
```

or just

```
sr.run("opa_ex1", "help .")
```

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

```
sr.run("opa_ex1", "help pump")
```

3 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 results to verify that the resolution parameters were appropriate. Insufficient resolution, too short time window or too small beam matrix yield invalid results. On the other hand, too fine resolution or too many points will lead to longer run time than necessary.

This chapter 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.

3.1 Spatial parameters

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. Table 3.1 shows the meaning of the smode parameter and restrictions on nx and ny in each case. Note that the size of the FFTs which Sisyfos use are not always equal to nx and ny: In smode 3 or 4 the FFT-size in the symmetric direction(s) is one less than the number of points. nx and ny should be chosen such that FFT sizes are good values for the FFT package being used. Advanced FFT packages, such as the one in Intel MKL (Maths kernel library) can handle an arbitrary number of points, but they may be more efficient when the number is a power of 2 or at least has only small prime factors.

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

Table 3.1 Spatial modes. The modes that take advantage of symmetry (1, 3, and 4) have an effective area that is greater than the actual matrix. The column 'effective size' gives the corresponding number of points in smode 2.

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 main program should place Dataout objects at critical points along the path, and the user must 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 fold 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. In some devices, particularly in resonators, light propagating out of the beam path can be a real loss mechanism. When simulating such devices it is not always practical to use matrices that are large enough to contain the "lost" beam components, so it may be necessary to insert non-physical apertures to block them before they fold around the matrix. This is discussed further in Chapter 10 on laser simulation.

A coarse estimate of the required number of points in each transverse direction can be made as follows (we consider the x-direction for definiteness): Suppose the beam has diameter D_N in the near-field and D_F (with units rad/m) in the far-field. The specific width measure is not important for the general argument, but for definiteness let $D_{N,F} = 4\sigma_{N,F}$, where $\sigma_{N,F}$ is the second moment of the power distribution in the respective domain. Assume that the matrix size has to be at least K times the beam diameter, for some K, in both domains:

$$\mathrm{nx} \cdot \mathrm{sx} \geq K D_N \tag{3.1}$$
$$\mathrm{nx} \cdot \mathrm{dkx} \geq K D_F, \tag{3.2}$$

where nx, sx and dkx are defined in Table 3.2. The resolutions are related by $\mathrm{dkx} = 2\pi/(\mathrm{nx} \cdot \mathrm{sx})$, so the second inequality can be rewritten as

$$\mathrm{sx} \leq \frac{2\pi}{K D_F} \tag{3.3}$$

Setting the inequalities equal and solving for nx yields

$$\mathrm{nx} = \frac{K^2 D_N D_F}{2\pi}. \tag{3.4}$$

If D_N is measured at the waist, the product $D_N D_F = 16\,\sigma_{N,\mathrm{waist}}\,\sigma_F = 8M_x^2$, where M_x^2 is the beam quality measure [6] for the x-direction. If we take $K = 3$ we obtain $\mathrm{nx} \approx 12M_x^2$, so for a high quality beam near the waist, a small number of points is enough. A greater number of points is required farther away from the waist because D_N increases while D_F is fixed in free propagation.

We stress that this derivation only gives an indication of the number of points required. An M^2-value substantially greater than unity can correspond to a variety of beams that are very different in detail, so the relation $\mathrm{nx} \propto M_x^2$ must not be taken literally. The essential point is that the inequalities (3.1) and (3.3) are satisfied, but the value of K and even the appropriate measure of beam diameter can depend on the situation. If the near- and far-field shapes are different it may not be optimal to use the same K for both.

Apart from increasing the run-time, using larger matrices than necessary can lead to errors when M^2 is calculated from simulation data. M^2 is sensitive to noise because energy far from the beam centre is weighted heavily in the second moments, and although the noise in simulations is typically below the noise of real sensors, it cannot be neglected if the beam fills only a small part of the matrix.

Table 3.2 summarizes the spatial parameters. 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.

Parameter	Meaning
nx	Number of transverse sample points
sx	Spatial resolution
k_{0x}	Offset of k_x (for tilted beam)
nx*sx	Size of beam matrix
dkx = 2π/(nx*sx)	Resolution in k-space
nx*dkx = 2π/sx	Range in k-space

Table 3.2 Spatial parameters. nx and sx are the free parameters, the other quantities follow as shown. For brevity, only the parameters for the x direction are shown.

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 (start of) slices
tsf	Time slice factor, tsf = tsi / (nt * dt)
df = 1/(nt*dt)	Frequency resolution
nt*df = 1/dt	Bandwidth

Table 3.3 Temporal parameters.

3.2 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. The second level of temporal sampling, which is necessary in devices with feedback, uses multiple time slices. Table 3.3 summarizes the temporal parameters.

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 moving with the pulses, where the time slice only needs to be long enough for the pulses, their relative temporal walk-off, and dispersive broadening. 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 3.1 shows two pulses and the moving time slice versus absolute 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. In addition, the temporal resolution (dt) should be so small that the simulation spectrum contains the spectrum of the pulse. This is analogous to the requirement that the spatial resolution is small enough to support the width of the far-field. As in the spatial domain, it is the user's responsibility to check that these conditions hold. The required number of sample points nt scales with the time-bandwidth product, analogously to the number of spatial sample points in Eq. (3.4).

In a resonant device, such as an OPO, the resonant signal is fed back with a delay equal to

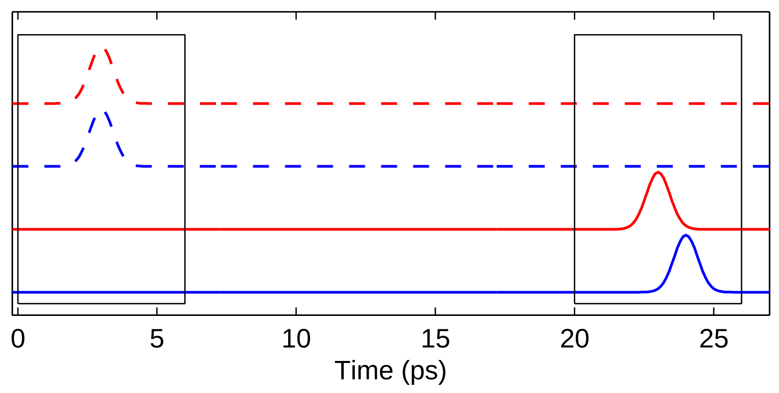


Figure 3.1 Reference pulse (red) and another pulse (blue) in absolute time before a crystal (dashed) and after it (solid). The transit time is 20 ps for the reference pulse. The black frames show the time slice, which moves with the reference pulse. The required length of the time slice is determined by the pulse length and the temporal walk-off, and it is much shorter than the transit time.

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.

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 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 3.2 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. Interaction of adjacent time slices can be handled by using overlapping slices and selecting the valid parts, and this will be implemented if the need arises.

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

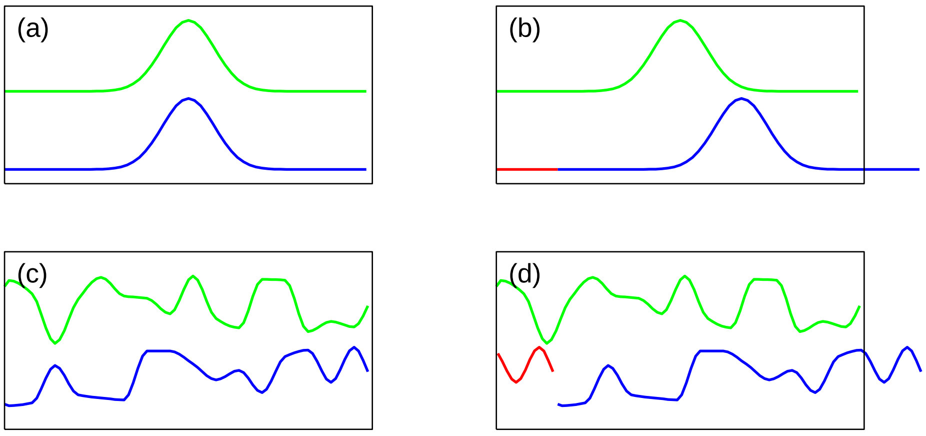


Figure 3.2 *(a, b) Time-slices with short pulses before (a) and after (b) the optical path. The green curve represents the reference beam and the blue curve represents a beam with temporal walk-off. The black frame shows the time slice. Because of the temporal walk-off, the blue pulse slides out of the 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 (non-zero) trailing part of the blue pulse is folded into it.*

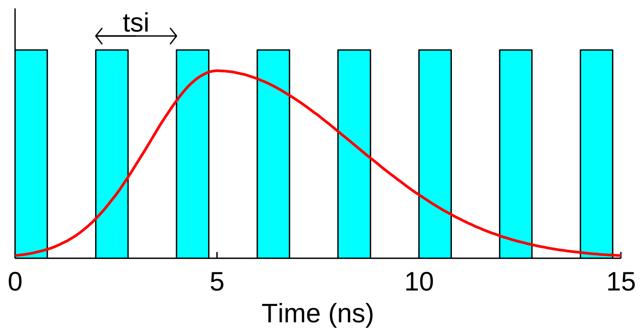


Figure 3.3 *Time slices (light blue) that do not cover the time axis completely. Sisyfos assumes that the average power between the slices is the same as in the slices and scales up the energy correspondingly.*

illustrated in Fig. 3.3. 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 Resonator classes have methods to specify the parameters in either way. The temporal parameters are set in two stages by the methods main_init1 and main_init2. 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.

3.3 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

$$k_z(\nu, k_x, k_y) \approx k_z(\nu, k_{x0}, k_{y0}) + k_z(\nu_0, k_x, k_y) - k_z(\nu_0, k_{x0}, k_{y0}), \tag{3.5}$$

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. Wide-band mode is selected by default, and we recommend using it unless you have verified that the narrow-band approximation is good. The advantage of the narrow-band mode is that it can be somewhat faster and needs less memory.

3.4 Examples

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:

```
sr.run("opa_ex1", "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 explained in Chapter 3.1.

Compare the results from the simulations with smode 2 and 3:

```
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"), g1.get("top.t_run")]
```

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

```
clf()
imshow(g1.tnf(1,1))
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:

```
sr.run("opa_ex1", "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. The resulting plots are shown in Fig. 3.4.

```
g1 = gfm3("t10")
g2 = gfm3("t11")
[g1.w(2,2), g2.w(2,2)]
[g1.get("top.t_run"), g2.get("top.t_run")]
u1 = g1.tnfr(2,2)
u2 = g2.tnfr(2,2)
v1 = g1.tffr(2,2)
v2 = g2.tffr(2,2)

clf()
subplot(1,2,1)
plot(u1.x*1e3, u1.m[0,:],"-o")
plot(u2.x*1e3, u2.m[0,:],"-o")
xlabel("Position (mm)")
text(-3,35,"(a)")
legend("t10 t11".split())

subplot(1,2,2)
plot(v1.x/1e3, v1.m[0,:],"-o")
plot(v2.x/1e3, v2.m[0,:],"-o")
xlabel("$k_x$ (mm$^{-1}$)")
text(-15,8,"(b)")
```

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:

```
sr.run("opa_ex1", "file *t20 smode 3 points [16 9] scale 4e-4 nt 16 dt 2e-12")
```

Compare the results:

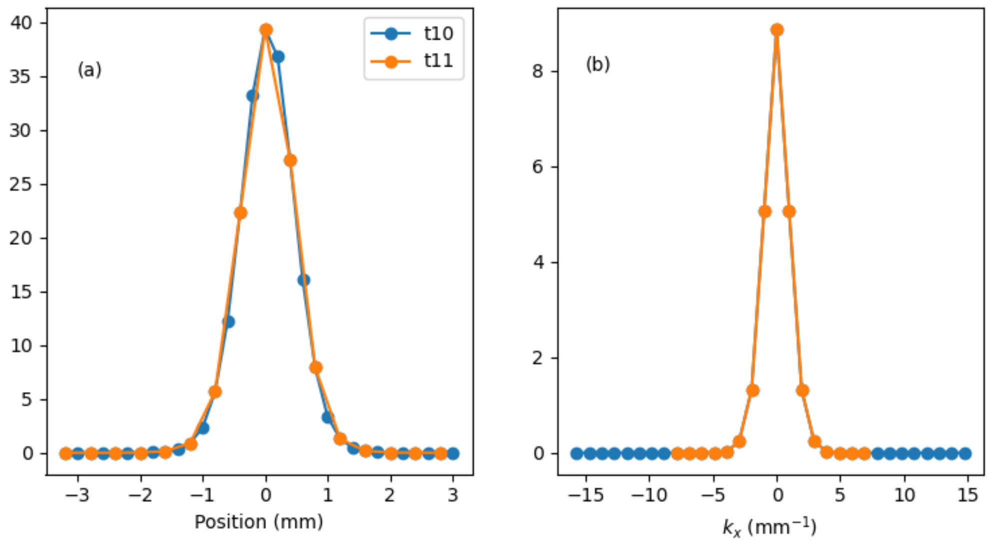


Figure 3.4 (a) Near-field signal profiles for files t10 and t11. (b) Corresponding far-field profiles.

```
g3 = gfm3("t20")
[g2.w(2,2), g3.w(2,2)]
[g2.get("top.t_run"), g3.get("top.t_run")]
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 put them in python strings:

```
sim_par_1 = """
 smode 3
 points [16 9]
 scale 4e-4
 nt 16
 dt 4e-13
"""
```

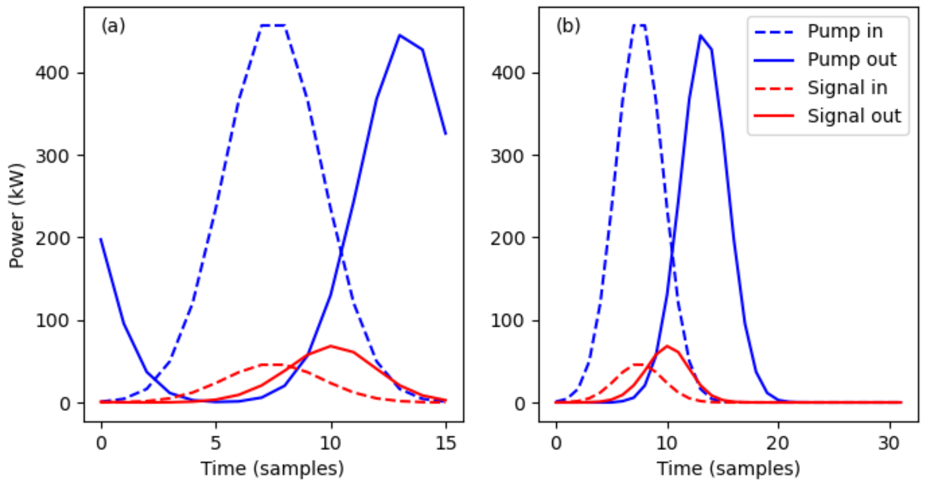


Figure 3.5 *(a) Input and output pulses from the file t30.sis. The time slice is too short compared to the temporal walk-off. (b) Same simulation with twice as many time samples (file t31.sis).*

```
opa_par_1 = "bbo.len 0.02"
pump_par_2 = """
 pump [
  tw 1e-12
  t0 3e-12
  w 1e-6
 ]
"""

seed_par_2 = """
 seed [
  tw 1e-12
  t0 3e-12
  w 1e-7
 ]
"""
sr.run("opa_ex1", opa_par_1, pump_par_2, seed_par_2, sim_par_1, "file *t30")
```

and inspect the results:

```
g = gfm3("t30")
clf()
plot(g.p(1,1)/1e3,"b--")
plot(g.p(2,1)/1e3,"b")
plot(g.p(1,2)/1e3,"r--")
plot(g.p(2,2)/1e3,"r")
ylabel("Power (kW)")
xlabel("Time (samples)")
legend("Pump in;Pump out;Signal in;Signal out".split(";"))
```

The plot is shown in Fig. 3.5 (a). 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.:

```
sr.run("opa_ex1", opa_par_1, pump_par_2, seed_par_2, sim_par_1, "nt 32 file *t31")
```

The results, plotted in Fig.3.5 (b), show that the pulses now stay within the time window. Check the group indices as shown in Section 2.4:

```
u = g.get("PropCrys3.1.ng")
```

The '.1' 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 Functions and their test programs

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 main program. 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. Sellmeier equations is a special type of function (of frequency and temperature) which is represented by SellmeierAny.

The purpose of this chapter 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.

4.1 Func1Any

Table 4.1 shows some of the functions that Func1Any can represent, with some of their parameters. See the html-documentation for a full list of functions and parameters. In addition to the parameters for specific functions, Func1Any takes the parameters v0, vs, wt, t0 and argf, which modify the underlying function f to

$$g(t) = \mathrm{v0} + \mathrm{vs} \cdot f((\mathrm{argf}(t) - \mathrm{t0})/\mathrm{wt}). \tag{4.1}$$

The argument transformation argf() is replaced by an identity if it is not specified. wt can have two elements, where wt[0] is used for $t <$ t0 and wt[1] for $t >=$ t0.

Name	Parameters	Meaning
lin	k	$f(t) = kt$
inv	k	$f(t) = k/t$
sgauss	ord, level	$\exp(-t^{\mathrm{ord}})$
wrap	period, t0p	Makes argument periodic
prod	f1, f2	Multiplies f1 and f2
comp	f1, f2	Composes functions, f1(f2(t))
tab	...	Interpolates in table

Table 4.1 Some of the functions that can be created with Func1Any.

Here is an example of the parameter syntax for an asymmetric, super-gaussian pulse:

```
pulse [sgauss [level 0.5 ord 4] wt [1e-9 2e-9] t0 5e-9]
```

'level 0.5' means that wt corresponds to width at the half-maximum points. The best way to learn the functions is to run them and plot the results, so Sisyfos includes a test program for this purpose. That is the topic of the next section.

4.2 Function test program

The example program func_test in the directory .../tutorial/ft can test Func1Any, Func2Any, and Func3Any. It evaluates a function on one or more points on a grid and writes the results to a file or to the display. The top-level parameters for func_test are shown in Table 4.2.

type	Selects the type of function to test. Should be 1 (for Func1Any), 2, or 3.
f1_p	Parameters for Func1Any (see Func1Any::param_struct()). Must be set if 'type' is 1
f2_p	Similar for Func2Any
f3_p	Similar for Func3Any
file	Name of the output file. If no file name is given the results are printed to the display.
points	The size of the grid where the functions is evaluated. 1–3 elements. Default 1.
x0	Start value for each axis of the grid. 1–3 elements. Default 0.
dx	Point spacing for each axis of the grid. 1–3 elements. Default 1.
index	Index or list of indices for which to evaluate vector-valued functions. The default is 0
complex	Selects real (0) or complex (1) values.

Table 4.2 Parameters for the func_test program.

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 documentation of Func1Any we often use t instead of x. An example command to test Func1Any is:

```
sr.run("func_test", "type 1 f1_p [lin [k 2.5] v0 3 vs -2] points 3 x0 1 dx 1")
```

which computes the function

$$f(t) = 3 + (-2) \cdot (2.5 \cdot t). \tag{4.2}$$

A slightly more advanced example, where the results are saved in the file 'test.sis', is

```
sr.run("func_test", "type 1 f1_p [sechsq [] wt 4] points 10 x0 -10 dx 2 file *test")
```

To plot the results in python you have to open the file by ReadSis5, which is the low-level class for reading any sis-file. (gfm3 is a specialization of ReadSis5 with additional methods for retrieving data stored by Dataout-objects, and it works only for files that contain such data. gfm3 inherits all the methods of ReadSis5, but not vice versa.)

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

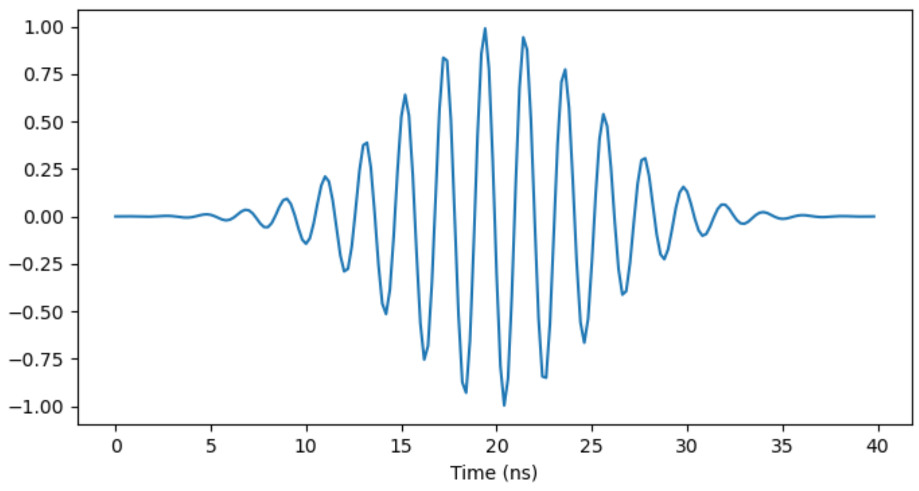


Figure 4.1 Modulated pulse.

To simplify running func_test and plotting results, func_test is complemented by a python class FuncTest, which runs func_test and returns a structure with results, ready for plotting.

```
import FuncTest as ft
ft1 = ft.FuncTest()
```

Now you can repeat the example above by

```
q = ft1.func(1, "sechsq [] wt 4", x0=-10, dx=2, points=10)
plot(q.x, q.v)
```

Here is a list of other examples:

```
q = ft1.func(1, "sgauss [ord 4 level 0.135] t0 1 wt 0.8 v0 -1 vs 3 ",
  x0=-2, dx=0.1, points=60)
q = ft1.func(1, "sgauss [] t0 1 wt 0.8 v0 -1 vs 3 ", x0=-2, dx=0.1, points=60)
```

A pulse train

```
q = ft1.func(1, "sgauss [argf [ wrap [period 10e-9]]] wt 1e-9 t0 5e-9 ",
   x0=-30e-9, dx=2e-10, points=300)
```

A modulated pulse, as shown in Fig. 4.1.

```
q = ft1.func(1, "prod [f1 [sgauss [] wt 6e-9 t0 20e-9] f2 [sin [k 3e9]]]",
  x0=0, dx=2e-10, points=200)
clf()
plot(q.x * 1e9, q.v)
xlabel("Time (ns)")
```

Sisyfos uses frequency, not wavelength, internally. If you have, say, an absorption spectrum given as a function of wavelength it is convenient to be able to convert it to a function of frequency so Sisyfos can use it. Here is an example:

```
q = ft1.func(1, "sgauss [argf [inv [k 2.997925e8]]] t0 1e-6 wt 0.1e-6 v0 3 vs 4 ",
  x0=200e12, dx=1e12, points=200)
plot(q.x/1e12, q.v)
xlabel("Frequency (THz)")
```

In practice the absorption would probably be given by a table, but here we have used a sgauss function for simplicity. The function is evaluated at 200 THz, 201 THz etc, whereas t0 and wt apply to the wavelength axis of the raw sgauss function. The resulting function is of course skewed when it is plotted against frequency. Because the conversion between wavelength and frequency is a common operation, the inv-function has a short-cut to avoid having to write out the value of c:

```
q = ft1.func(1, "sgauss [argf [inv [kc 1]]] t0 1e-6 wt 0.1e-6 v0 3 vs 4 ",
  x0=200e12, dx=1e12, points=200)
```

This could of course also be written as a composition instead of using the argf() feature:

```
q = ft1.func(1, "comp [f1 [sgauss [] t0 1e-6 wt 0.1e-6 v0 3 vs 4] f2 [inv [kc 1]]] ",
  x0=200e12, dx=1e12, points=200)
```

Func1Any can also handle interpolation in tables. Interpolation can be linear or spline-based:

```
q1 = ft1.func(1, "tab [x0 -5 dx 1 y [25 16 9 4 1 0 1 4 9 16 25] v_out 0 ]",
  x0=-6, dx=0.1, points=120)
q2 = ft1.func(1, "tab [x0 -5 dx 1 y [25 16 9 4 1 0 1 4 9 16 25] v_out 0 mode spline]",
  x0=-6, dx=0.1, points=120)
```

Plotting these together illustrates the smoothing effect of the splines. v_out defines a value to use outside the table. If v_out is not specified, the program fails if lookup outside the table is attempted. It possible to specify a list of x-values explicitly instead of by x0 and dx, see html-documentation and further examples i the tutorial/ft directory.

The edge function, Func1Edge, changes from 0 to 1 over the interval $[-1, 1]$, with a selection of different shapes. Note that t0 is set to 10 for some of the functions below make the graph clear. The resulting plot is shown in Fig. 4.2.

```
q0 = ft1.func(1, "wt 10 edge [ shape step]", x0=-20, dx=0.1, points=500)
q1 = ft1.func(1, "wt 10 edge [ shape power]", x0=-20, dx=0.1, points=500)
q2 = ft1.func(1, "wt 10 edge [ shape power order 0.5]", x0=-20, dx=0.1, points=500)
q3 = ft1.func(1, "wt 10 edge [ shape power order 2]", x0=-20, dx=0.1, points=500)
q4 = ft1.func(1, "wt 10 edge [ shape sin] t0 10", x0=-20, dx=0.1, points=500)
q5 = ft1.func(1, "wt 3 edge [ shape atan] t0 10", x0=-20, dx=0.1, points=500)
q6 = ft1.func(1, "wt 10 edge [ shape sgauss order 2] t0 10", x0=-20, dx=0.1, points=500)

clf()
plot(q0.x, q0.v)
plot(q1.x, q1.v)
plot(q2.x, q2.v)
plot(q3.x, q3.v)
plot(q4.x, q4.v)
plot(q5.x, q5.v)
plot(q6.x, q6.v)
grid()
legend("step linear sqrt square sin atan sgauss".split())
```

Func1Hat resembles Func1Edge, but it is two-sided. Its basic form is a unit-valued hat on the interval $[-1, 1]$, but it can be modified with soft edges, which extend the interval of non-zero values. Note that the width of the transition region can be different for the rising and falling edge and that it can be arbitrarily wide. The ”ramp” parameter applies to the basic function, so it is scaled by wt:

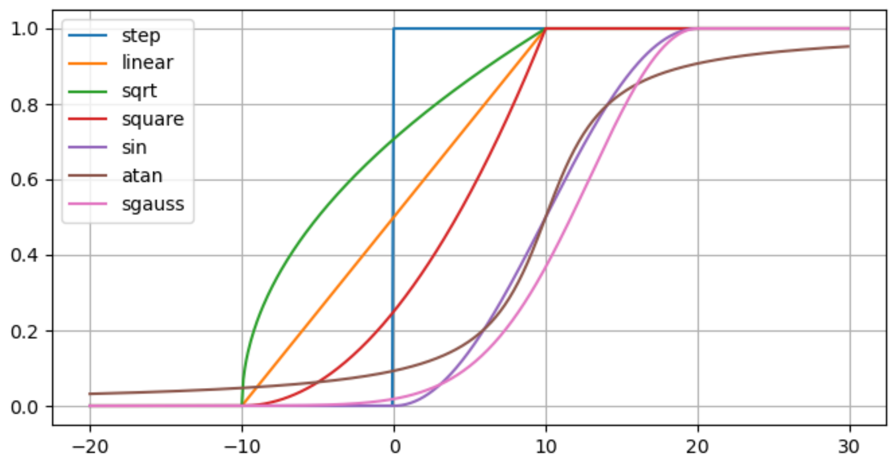


Figure 4.2 Different shapes for Func1Edge.

```
q0 = ft1.func(1, "wt 10 hat [ shape power order 0.5 ramp 0.3]", x0=-20, dx=0.1, points=500)
q1 = ft1.func(1, "wt 10 hat [ shape sin ramp [0.2 0.7]]", x0=-20, dx=0.1, points=500)
q2 = ft1.func(1, "wt 4 hat [ shape power ramp [3 1]]", x0=-20, dx=0.1, points=500)
clf()
plot(q0.x, q0.v)
plot(q1.x, q1.v)
plot(q2.x, q2.v)
grid()
```

As an example of a complex function, consider a chirped pulse:

```
q = ft1.func(1, "prod [f1 [sgauss [] wt 3e-13] f2 [chirp [k 1e26]]] t0 1e-12 ",
  x0=0, dx=2e-15, points=1000, cmplx=1)
clf()
plot(q.x*1e12, real(q.v),’b’)
plot(q.x*1e12, imag(q.v),’r’)
plot(q.x*1e12, abs(q.v),’g’)
```

Note that the t0 parameter is set for the prod-function instead of individually for each factor. If ft1.func() were called without ’cmplx=1’ it would simply return the real part.

Func1SpecFilter can apply amplitude scaling and phase in the spectral domain. It takes an input function, Fourier transforms it, applies the amplitude and phase factors, and transforms back. It creates a table internally, and the user must specify the number of points nt and temporal resolution dt for this table. These parameters are local to Func1SpecFilter and independent of the simulation parameters with the same names. In the example below, the string p1 defines the input pulse and the parameters for the table, and filter_f and phase_f represent transmission and phase as functions of frequency. Func1SpecFilter has more options, which can be found in the html-documentation.

```
p1 = "specfilt [pulse_f [sgauss [] wt 1e-14] nt 2048 dt 1e-15] t0 1e-12 "
q0 = ft1.func(1, p1, x0=0, dx=2e-15, points=1000)                    # No modification
q1 = ft1.func(1, p1 + 'specfilt.filter_f [sgauss [] wt 6e11]', x0=0, dx=2e-15, points=1000)
q2 = ft1.func(1, p1 + 'specfilt.phase_f [poly [k [0 0 3e-26 0]]]',
  x0=0, dx=2e-15, points=1000, cmplx=1)
q3 = ft1.func(1, p1 + 'specfilt.phase_f [poly [k [0 0 3e-26 6e-40]]]',
  x0=0, dx=2e-15, points=1000, cmplx=1)
clf()
plot(q0.x*1e12, q0.v)
plot(q1.x*1e12, q1.v)
plot(q2.x*1e12, abs(q2.v))
plot(q3.x*1e12, abs(q3.v))
legend('Input; Amplitude filter; 2. order phase; 2. and 3. order phase'.split(';'))
```

In q0, neither filter_f nor phase_f is given, so the input function is not modified. In q1, the bandwidth is reduced and the pulse becomes longer. The pulse in q2 is chirped, and therefore longer, by the second order phase. The pulse in q3 is subjected to third order dispersion in addition to the chirp.

As the examples show, 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. There are more examples in the file .../tutorial/ft/func_test_examples.py.

4.3 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 v0 and vs. Func2Any has parameters 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 – Hyper-elliptic 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 generalized radius is

$$r = ((x')^{\text{e\_order}} + (y')^{\text{e\_order}})^{1/\text{e\_order}} \tag{4.3}$$

e_order = 2 gives a circular or elliptic shape, and e_order > 2 makes the shape more rectangular. e_order should be an even integer.

Func2Any can be tested with func_test in a similar way to Func1Any. Here are some examples of rectangular apertures with soft edges. The options for edge shapes are the same as for Func1Edge and Func1Hat:

```
q0 =ft1.func(2, "ap_rect [] wxy [0.1 0.15]",
  x0=-0.3, dx=0.01, points=[50,60])
q1 =ft1.func(2, "ap_rect [soft_w 0.7 shape power] wxy [0.1 0.15]",
  x0=-0.3, dx=0.01, points=[50,60])
q2 =ft1.func(2, "ap_rect [soft_w 0.7 shape power order 0.5] wxy [0.1 0.15]",
  x0=-0.3, dx=0.01, points=[50,60])
q3 =ft1.func(2, "ap_rect [soft_w 0.5 shape sgauss] wxy [0.1 0.15]",
  x0=-0.3, dx=0.01, points=[50,60])

clf()
imshow(q1.v)
clf()
plot(q0.x, q0.v[30,:])
plot(q1.x, q1.v[30,:])
plot(q2.x, q2.v[30,:])
plot(q3.x, q3.v[30,:])
legend("step linear sqrt sgauss".split())
```

Func3Any has parameters for both the t-transformation (where t corresponds to z) and the xy-transformation. Only a few 3D functions have been implemented, and table lookup is probably the most important one in practice. For a simple example, the "move" function, which has been made for that purpose, is more convenient. It takes a 2D function (fxy) as one of its arguments and moves it in the xy-plane as specified by the functions xf(z) and yf(z):

$$g(x, y, z) = f(x - \mathrm{xf}(z), y - \mathrm{yf}(z)) \tag{4.4}$$

```
f3 = "move [fxy [sgauss [] wxy 3e-4 pos0 [-7e-4 0]] xf [lin.k 6e4] yf [sin.k 1e9 vs 2e-4]]"
q=ft1.func(3, f3, x0=[-2e-3, -2e-3, 0], dx=[1e-4, 1e-4, 1e-9], points=[40,40,30])
clf()
imshow(squeeze(q.v[0,:,:]))
imshow(squeeze(q.v[:,20,:]))
imshow(squeeze(q.v[:,:,20]))
```

Func3Move can also scale the width with wf(z) and the value by sf(z):

```
f3a = "move [fxy [sgauss [] wxy 2e-4 ] wf [lin.k 1e8 v0 1] sf [cos.k 1e8]]"
q=ft1.func(3, f3a, x0=[-2e-3, -2e-3, 0], dx=[1e-4, 1e-4, 1e-9], points=[40,40,30])
```

4.4 SellmeierAny

Sellmeier equations depend on frequency and temperature. They could have been represented by Func2AIfc and Func2Any, but because they have some specialized methods they are represented by SellmeierIfc and SellmeierAny. SellmeierAny was used in example 1, but like functions, it can have its own parameter structure to give greater flexibility at runtime. Sisyfos includes many Sellmeier equations in the directory .../MatData/SE/BBO, and users can add new Sellmeier files if they need to. There is a program sell_test that corresponds to func_test, and it can be run through the same python class FuncTest:

```
q = ft1.sell("mat KTP name Kato2002", la0=2e-6, dnu=1e13, 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])
```

This displays the principal refractive indices from $\nu_1 = c/(2\,\mu\text{m})$ to $\nu_2 = \nu_1 + 19 \cdot 10^{13}$ Hz. The dielectric tensor is stored in the field q.v.

Sellmeier equations are only valid within a limited range of frequencies, but that range is not always clearly stated. Therefore, Sisyfos leaves the user to judge the range of validity instead of enforcing limits. Another reason for this choice is that the frequency range covered by a simulation must be greater that the width of the actual spectrum, and errors in the Sellmeier equation near the end of the spectral range may be harmless. However, Sisyfos does throw an exception if a Sellmeier equation yields a refractive index < 1. This can be suppressed, and the invalid index replaced by 1, by setting the 'relax' option, which you can try in the above example by extending the wavelength range too far into the UV (try turning off relax):

```
q = ft1.sell("mat KTP name Kato2002 relax 1", la0=2e-6, dnu=1e13, points=150)
```

Here is an example that uses a temperature-dependent Sellmeier equation:

```
q1 = ft1.sell("mat LNO name Gayer2008_mgo_cln", nu0=100e12, dnu=10e12, points=30, temp=293)
q2 = ft1.sell("mat LNO name Gayer2008_mgo_cln", nu0=100e12, dnu=10e12, points=30, temp=450)
clf()
plot(q1.la*1e6, q1.n[:,0])
plot(q2.la*1e6, q2.n[:,0])
```

Most Sellmeier equation do not include temperature dependence, so Sisyfos has an option to augment them with thermo-optic coefficients:

```
se1 = "mat LNO name Zelmon1997 toc [1e-5 1e-5 2e-5] ref_temp 293"
q1 = ft1.sell(se1, nu0=100e12, dnu=10e12, points=30, temp=293)
q2 = ft1.sell(se1, nu0=100e12, dnu=10e12, points=30, temp=450)
clf()
plot(q1.la*1e6, q1.n[:,0])
plot(q2.la*1e6, q2.n[:,0])
```

SellmeierAny has many additional features that you can find in the html documentation:

- Second order thermo-optic coefficients.
- Thermo-optic coefficients that are functions of wavelength.
- Combine Sellmeier equations that are valid in different frequency ranges.
- Define a Sellmeier equation by dispersion coefficients
- Gas mixtures
- Use a function object as Sellmeier equation. This can be useful for table lookup.

5 Beam sources and their test programs

Using Func1Any and Func2Any with SimpleSource gives great flexibility, but the source is still 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, SourceIfc, 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 func_test 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. It can be run under python with FuncTest. Unlike the methods for testing functions, FuncTest.source() returns a gfm3-object, not a data structure:

```
s1 = """
 simple [
  pulse [sgauss [level 0.135] t0 20e-9 wt 10e-9 ]
  beam [sgauss [ord 6] wxy 1e-3 ]
 ]
 i0 1e10"""
g = ft1.source(s1, smode=2, points=32, scale=1e-4,  nt=1, dt=1e-10, tsi=1e-9, tlen=30e-9)
clf()
imshow(g.tnf(1,1))
```

The next examples show how to use Func3Source with the move-function:

```
s2 = """
 func3 [
  pulse [move [
   fxy [sgauss [] wxy 2e-4 ]
   wf [lin.k 5e7 v0 1]
  ]]
 ]
 i0 1e10"""
g = ft1.source(s2, smode=2, points=32, scale=1e-4,  nt=32, dt=1e-9, tlen=1e-9)
u = squeeze(abs(g.e(1,1,1)))
clf()
imshow(u[0,:,:])
imshow(squeeze(u[:,:,16]))
imshow(g.tnf(1,1))
```

Rename the output file from the above example, 'tmp.sis', to 'pump1.sis' so it can used in an example with FileSource without overwriting it:

```
s3 = "file [  fname pump1.sis pos 1 beam 1 mag 0.6]"
g = ft1.source(s3, smode=2, points=32, scale=1e-4,  nt=32, dt=1e-9, tlen=1e-9)
```

The results can be shown with the same commands as in the previous example. Because of the magnification by 0.6, the beam is smaller in the new example.

5.1 Beam source with arbitrary input data

The table functions facilitate the use of measured data in beam sources. Arrays with measured data can be processed in python or matlab and stored in a file that Sisyfos functions can read. In this example the input tables are computed, but replacing them with measured data would be simple. First, make a table for the beam shape (the difference of two Gaussians in this example). To illustrate the general case we make the matrix rectangular with different resolution in x and y. The function numpy.ix_() makes row and column vectors for the x and y axes, and the broadcasting mechanism of numpy yields a matrix r2 when they are added:

```
nx = 150
ny = 140
x = linspace(-3e-3,3e-3,nx)
y = linspace(-3.6e-3,3.6e-3,ny)
y1, x1 = ix_(y, x)
r2 = x1**2 + y1**2
beam = exp(-r2/(1.1e-3**2)) - 0.6 * exp(-r2/(0.35e-3**2))
plot(beam[int(ny/2),:])
```

Then make a table for a complex spectrum from spectral power and phase (a chirped and modulated Gaussian in this case):

```
nt = 2000
nu0=300e12
nu = nu0 + linspace(-100e12,100e12,nt)
spec_power = exp(-(nu - nu0)**2 / (30e12**2)) * (2 + sin(2e-13*nu))
spec_phase = (nu - nu0)**2 * 1e-25
spec = sqrt(spec_power) * exp(1j * spec_phase)
```

This can be transformed to get the complex pulse envelope. The inner fftshift changes the spectrum to FFT-order, and the outer one shifts the pulse to the centre of the time domain. The sign convention in Sisyfos is such that fft in numpy must be used to transform from frequency to time. Always double-check the phase convention for measured data to avoid ending up with a time-reversed pulse.

```
pulse = fft.fftshift( fft.fft( fft.fftshift(spec) ) )
d_nu = diff(nu)[0]
t = linspace(0, 1/d_nu, nt)
plot(t*1e12, abs(pulse)**2)
plot(t*1e12, real(pulse))
plot(t*1e12, unwrap(angle(pulse)))
```

The beam- and pulse-data can now be assembled in Sisyfos structures and saved in a sis-file. The field names have to correspond to the parameters of Func1Table and Func2Table, therefore the time data is placed in pulse_s.x, for example.

```
beam_s = su.Struct()
beam_s.smode = 2
beam_s.scl = array([diff(x)[0], diff(y)[0]])  # Note order: x, y
beam_s.mr = beam
pulse_s = su.Struct()
pulse_s.x = t
pulse_s.y = pulse
root_s = su.Struct()
root_s.beam = beam_s
root_s.pulse = pulse_s
# import WriteSis5 as ws # Usually in startup file
ws.write_struct("seed_1.sis", "seed", root_s)
```

write_struct() creates the file "seed_1.sis" and stores the members of root_s in the field "seed". The file "seed_1.sis" can now be used in a simulation. Note that "tabc" is used for the complex pulse and "tab" for the real beam. The pulse is shifted closer to $t = 0$ by setting "t0 -2.5e-12". "pamp 1" tells that the pulse function represents (complex) amplitude instead of power, which is default. By default, Sisyfos takes the (0,0) position to correspond to element $(\lfloor N/2 \rfloor, \lfloor M/2 \rfloor)$ of an $N \times M$ matrix, so for the 140×150 matrix in this example the centre position would correspond to (70, 75). This is not consistent with the array returned by linspace, as can be seen be inspecting x[75] and y[70]. Therefore, the actual position of element (70,75) is set by the "pos0"-option. This step would have been unnecessary if we had chosen an odd number of points in the matrix, but the use of "pos0" is worth showing because measured data are rarely perfectly centered. The pulse- and beam-data could of course have been stored in separate files.

```
s4 = """
 simple [
  pulse [tabc [-bf seed_1 seed.pulse mode spline ] t0 -2.5e-12]
  pamp 1
  beam [tab [-bf seed_1 seed.beam v_out 0] pos0 [2e-5 2.6e-5] ]
 ]
 i0 1e8"""
g = ft1.source(s4, smode=2, points=64, scale=1e-4,  nt=1024, dt=5e-15, tlen=1e-12)
```

When using complicated source parameters, it is advisable to check the results:

```
u = g.tnfr(1,1)
clf()
plot(1e3*u.x, u.m[32,:], "b")
plot(1e3*u.y, u.m[:,32], "g--")
plot(1e3*x, beam[int(ny/2),:]*1.43, "r--")
plot(1e3*y, beam[:,int(nx/2)]*1.43, "c--")
xlabel("Position (mm)")

u1 = g.tsr(1,1)
clf()
plot(1e-12*u1.nu, u1.m)
plot(1e-12*nu, spec_power*4.9e-20, "r--")
xlabel("Frequency (THz)")

u2 = g.pr(1,1)
clf()
plot(1e12*u2.t1, u2.m)
plot(t*1e12 - 2.5, abs(pulse)**2*490, "r--")
xlabel("Time (ps)")
```

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 Sisyfos programs tend to have many parameters in common, so Sisyfos offers some predefined substructures to simplify the program. The purpose of this chapter 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 together with this chapter, is similar to opa_ex1, except that it makes use of predefined parameter structures and a few other Sisyfos features.

6.1 Program structure

make_param_struct() has been simplified by using setup_std_param4(), which 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 html-documentation for the list of allowed fields (C++ modules, namespaces, namespace members, select the tab for 's' to get to setup_std_param4). make_param_struct() also uses built-in parameter structures for AnySource and PropCrys3.

make_optical_path() calls factory functions for AnySource and PropCrys3, which take the corresponding parameter structures as arguments.

6.2 Running

The input beams in example 1 were restricted to have Gaussian spatial distribution and pulse shape. Example 2 uses AnySource for greater flexibility, for both pump and seed beams. A parameter file (called beam2.txt in the example directory) for pump can be:

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

To test example 2, run a simulation with this file as pump:

```
sr.run("opa_ex2", "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(1,1),"b--")
plot(g0.p(2,1),"b")
plot(g0.p(2,2),"r")
plot(g1.p(1,1),"g--")
plot(g1.p(2,1),"g")
plot(g1.p(2,2),"k")

clf()
plot(g0.tnf(1,1)[16,:])
plot(g1.tnf(1,1)[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. Try to replace the hard aperture function in the pump parameters by a soft one (see Section 4.3) or by a super-Gaussian function of order > 2.

6.3 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):

```
sr.run("opa_ex2", "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:

```
sr.run("opa_ex2", "file *t51 seed [-tf beam3] pump [-tf beam1 w 5e-3] bbo.len 1e-3")
```

where beam3.txt reads

```
file [
 % 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, then 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:

```
sr.run("opa_ex2", "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

```
tab [
 % Parameters for Func2TableR
 smode 2
 mr -bf t50.sis do.2.f.2.tnf.m
 scl 2.5e-4
 v_out 0
]
```

’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, 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 (this does not affect the spatial resolution of the new simulation). 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 factor 2.5 in the last line was chosen just to make the graphs overlap.

6.4 Changing solver parameters

The program opa_ex2 includes a substructure 'rk2' with parameters for PropCrys3NL2, the differential equation solver. The default values for PropCrys3NL2 usually work, but it is wise to run some simulations with other tolerances to check that results are consistent. Try varying the relative tolerance rtol

```
sr.run("opa_ex2", "file *t41 seed [-tf beam1] pump [-tf beam2] crys.rk2.rtol 1e-3")
sr.run("opa_ex2", "file *t42 seed [-tf beam1] pump [-tf beam2] crys.rk2.rtol 1e-2")
sr.run("opa_ex2", "file *t43 seed [-tf beam1] pump [-tf beam2] crys.rk2.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 barely become noticeable even for the very high relative tolerance of 0.1.

It is also possible to choose between three alternative steppers in the Runge-Kutta solver. In order of increasing complexity they are 5th order Fehlberg, 5th order Dormand-Prince, and 8th order Dormand-Prince [7, 8]. The more advanced steppers can potentially give greater accuracy, or need fewer steps for the same accuracy, but the optimal stepper depends on the application. Here is an example with the 8th order stepper:

```
sr.run("opa_ex2", "file *t44 seed [-tf beam1] pump [-tf beam2] crys.rk2.stepper 2")
```

See the html documentation for PropCrys3NL2 for details of the tolerance calculations.

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 chapter 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).
- 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 make_optical_path():

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

Most Sisyfos programs include the options for multiple slices in the nonlinear or active medium and for Dataout objects between these slices. The numbers of the Dataouts do not necessarily increase along the path, and in general there can be gaps between the numbers. In this example the numbers of the first and last Dataout are fixed, so position 2 corresponds to the output regardless of the number of slices in the crystal.

Strings with parameters are stored in param1.py. You should inspect this file, run it and then run a simulation with a wide-band, chirped seed pulse, but without without n_2 and absorption:

```
from param1 import *
sr.run("opa_ex3", pump1, seed1, dp1, sp1, "file *t60 ")
```

The string seed1 includes the section

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

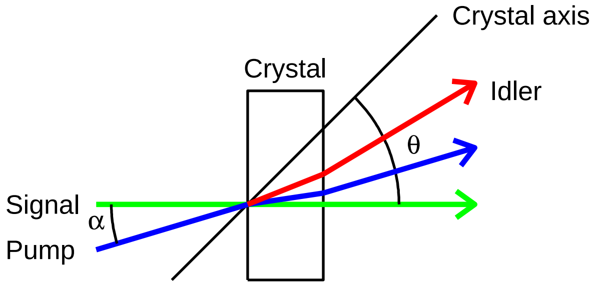


Figure 7.1 OPA with non-collinear beams. The signal beam is taken to be perpendicular to the crystal face.

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 $\exp(i\ \mathrm{kr}\ (t - t_0)^2)$. 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 non-zero.

Plot the input and output signal spectra:

```
g = gfm3("t60")
u = g.tsr(1,2)
clf()
plot(u.nu*1e-12, u.m * 200)    # Note scaling
plot(u.nu*1e-12, g.ts(2,2))
```

The spatially resolved spectrum is stored in tnfs, and when it is plotted for the centre of the beam and three adjacent positions we can see signs of back conversion in the centre (the result is shown in Fig 7.2):

```
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])
xlabel("Wavelength (nm)")
legend("Centre 1 2 3".split())
```

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]          # Beam centre
s1 = fftshift(ifft(e1))
phi = unwrap(angle(s1))
clf()
plot(u.la*1e9, phi)
```

Plot the pump and signal pulses

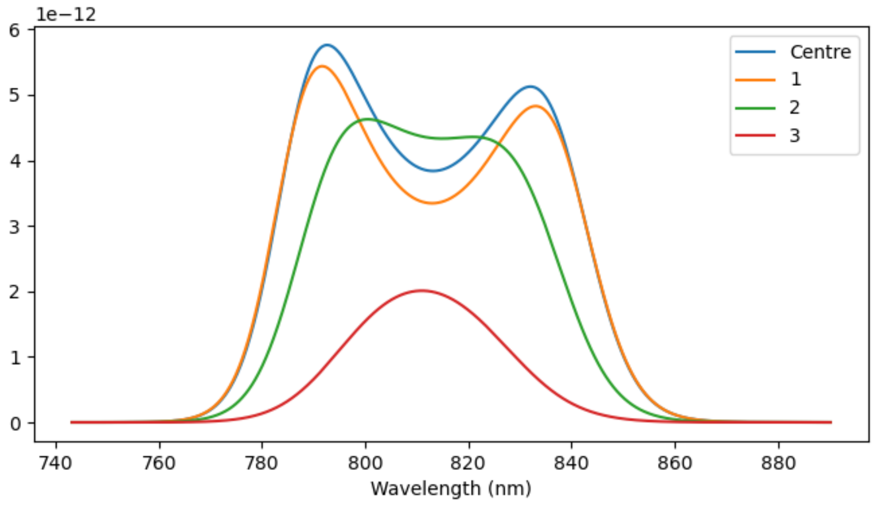


Figure 7.2 Spectra for the centre and three adjacent positions.

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

The format of the n_2-matrix (n21 in param1.py) 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:

```
sr.run("opa_ex3", pump1, seed1, dp1, n21, sp1, "file *t61")
```

Compare the results

```
g0 = gfm3("t60")
g1 = gfm3("t61")
clf()
plot(g0.ts(2,1)[730:810])
plot(g1.ts(2,1)[730:810])
clf()
plot(g0.ts(2,2))
plot(g1.ts(2,2))
clf()
plot(g0.tff(2,1)[0,:])
plot(g1.tff(2,1)[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 also been broadened outside the matrix, so the simulation should be repeated with higher spatial resolution if correct results were required.

The next example includes absorption. The parameter string "abso1" 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:

```
sr.run("opa_ex3", pump1, seed1, dp1, abso1, sp1, "bbo.slices 2 stheat.st 1 file *t62")
```

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 (amplitude-) absorption spectrum for each beam can be retrieved and plotted against frequency by

```
u = g2.get("PropCrys3.1.bd")
v = g2.tsr(1,1)
clf()
plot(v.nu/1e12, u.a0.absa)
```

where ’bd’ means beam data and contains substructures for each beam. The tsr() method is used to get the frequency table, which is not stored in ’bd’. Now inspect the distribution of absorbed power in the crystal:

```
u = g2.get("MultiSlice.1.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:

```
sr.run("opa_ex3", pump1, seed1, dp1, sp1, "wb 0 file *t65 ")
g0 = gfm3("t60")
g1 = gfm3("t65")
clf()
plot(g0.ts(2,2))
plot(g1.ts(2,2))
```

The spectrum in narrow-band 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.

By using multiple slices in the crystal and storing intermediate results, we can search for the optimal crystal length without running simulations for each length separately. As noted above, the Dataouts are not numbered consecutively along the optical path. However, each Dataout stores its z-position, which can be retrieved by gfm3 and used for plotting. Note that we plot points to avoid criss-crossing lines when the Dataouts are not ordered:

```
sr.run("opa_ex3", pump1, seed1, dp1, sp1, "bbo [len 5e-3 slices 10] file *t66")
g = gfm3("t66")
clf()
for i in range(1, g.npos() + 1): plot(1e3 * g.z(i), 1e3 * g.w(i,2), "bo")
xlabel("Length (mm)")
ylabel("Signal energy (mJ)")
```

This simple example showed how to plot the energy, a true optimization would also need to take beam quality, spectrum and the compressibility of the pulse into account.

7.1 Managing parameters with python strings or functions

In examples 1 and 2, parameters were read from text files by the '-tf' option. It is simple to override a few parameters in the command itself, e.g. by "pump.w 5e-3", but this becomes inconvenient if you need to vary many parameters. One approach is to have many small files with alternative pump- and seed sources, simulation parameters etc. and combine them in different ways, but we find it more convenient to use short python strings, as in param1.py. String can be made more flexible by use of the string formatting features of python, e.g. 'dp2 = "bbo [len %f slices %d]"', which is defined in param2.py:

```
from param2 import *
sr.run("opa_ex3", pump1, seed1, dp2 % (5e-3, 10), sp1, "file *t70")
```

This is simpler than "bbo [len 5e-3 slices 10]" from the example above. Functions that return strings can be even more flexible because they can have default values for some parameters. param2.py also defines

```
def dp_f(len=3e-3, slices=1):
  s = f"bbo [ len {len:.5f} slices {slices:d}]"
  return s
```

This function uses a python f-string for formatting. If you are not familiar with f-strings, it's worth looking it up on the web. The function dp_f can be used as

```
sr.run("opa_ex3", pump1, seed1, dp_f(5e-3, 10), sp1, "file *t71")
sr.run("opa_ex3", pump1, seed1, dp_f(slices=8), sp1, "file *t72")
```

Note how the keyword argument "slices=8" can be used to override a single parameter without having to list values for all the other, and it also makes the meaning very clear.

Functions can also assist by making sure that related parameters are set consistently. The function pulse_f(), which is also defined in param2.py, modifies the widths of both the seed and pump pulses and sets t0 consistently for both pulses and the chirp. The function chirp_f makes it simple to override the chirp without the lengthy "seed.simple.mod.chirp.k 1e25".

```
sr.run("opa_ex3", pump1, seed1, pulse_f(7e-12,3e-12,2e-12),
 chirp_f(1e25), sp1, "file *t73")

g = gfm3("t73")
clf()
u = g.pr(1,1)
t = u.t1 * 1e12
plot(t, u.m,"b--")
plot(t, g.p(2,1),"b")
plot(t, g.p(1,2) * 200,"g--")   # Note scaling
plot(t, g.p(2,2),"g")
grid()
xlabel("Time (ps)")
```

When assembling commands from strings it is useful to be able to inspect the result without running the command. The function SisRun.pprint (for pretty-print) can be used for this. It prints the command with indentation that reflects the structure of the parameters:

```
sr.pprint("opa_ex3", pump1, seed1, pulse_f(7e-12,3e-12,2e-12),
 chirp_f(1e25), sp1, "file *t73")
```

As you see, short structures are displayed on single lines whereas long structures are split.

These examples have given an indication of how parameter sets can be handled by python strings or string functions. The details have to be adapted to the project, so Sisyfos does not offer any general functions for parameter handling.

7.2 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

```
sr.run("opa_ex3", pump1, seed1, dp1, sp2, "file *t75 ")
```

you get the warning message 'Overlapping spectra for beams: 1, 2'. This means that the spectral ranges of two beams with the same polarization overlap. You can see the spectral ranges of the beams by typing

```
g = gfm3("t75")
s1 = g.tsr(2,2)
s2 = g.tsr(2,3)
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 they would not interfere. In general, if two beams with the same polarization have components with the same frequency and direction they should be treated as a single beam. Otherwise, interactions involving the overlapping components will be incorrect. 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.

7.3 Noise

Sisyfos can add semi-classical noise to approximate the effect of quantum noise in a real device. Run simulations with and without noise in an OPA without seed input and compare the results

```
sr.run("opa_ex3", pump1, dp1, sp2, "pump.i0 2e14 file *t77 ")
sr.run("opa_ex3", pump1, dp1, sp2, "pump.i0 1 noise 1 file *t78 ")
sr.run("opa_ex3", pump1, dp1, sp2, "pump.i0 2e14 noise 1 file *t79 ")
g0 = gfm3("t77")
g1 = gfm3("t78")
g2 = gfm3("t79")
clf()
plot(g0.ts(2,2))
plot(g1.ts(2,2))
plot(g2.ts(2,2)/2e3)  # Scale to show in same graph
legend("Without noise; Weak pump; Strong pump".split(";"))
```

The example with weak pump (file t78) gives negligible amplification, so the graph shows the semi-classical noise that is added to the signal. It rises with frequency, corresponding to the photon energy. With strong pump the noise is amplified (note the different scale), and the spectrum is shaped by the phase matching bandwidth. The noise is essential in optical parametric generators (OPG) and can be important in OPAs with high gain.

8 SisRun – advanced features

8.1 Controlling the output of SisRun

The function SisRun.run() has already been used in most of the examples. SisRun contains a class variable "Silent" that controls how output from the application program is displayed:

- If Silent=0, output is displayed directly in the console.
- If Silent=1, output is collected and displayed when the program terminates.
- If Silent=2, output is collected, but it is displayed only if the program terminates with an error.

This feature is useful to avoid messy output when running parallel simulations, which is the next topic. You can set it by e.g.

```
import SisRun as sr   # Usually in the startup file
sr.Silent = 1
```

Silent=0 does not work in the Spyder console, at least not in Spyder 4.1. Setting Silent to 1 or 2 is a good solution for commands that execute rapidly, such as the examples in this tutorial. If you want to see progress information during a simulation you can run it in a separate command window. This is described in the "Getting started" guide because the details are likely to change.

8.2 Parallel simulations

SisRun.run() always 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 (in windows, or equivalent programs in Linux) is useful for monitoring the memory usage. The number of threads for Sisyfos should not exceed one per core, or maybe two per core with hyper-threading. Here is an example using the program from Chapter 6:

```
seed1 = """
seed [
 simple [
  pulse [sgauss [] wt 5e-12 t0 15e-12]
  beam [sgauss [] wxy 1e-3]
 ]
 ta 0
 tb 30e-12
 w 2e-6
]
"""
```

```
pump1 = """
pump [
 simple [
  beam [ ap_ellip[] wxy 1.2e-3 ]
  pulse [ sechsq [] wt 4e-12 t0 15e-12 ]
 ]
 ta 0 tb 30e-12 w %g
]
"""

# Run in .../tutorial/ex2
sr.Silent = 1
com = "opa_ex2"
sr.runt(com, seed1, pump1 % 4e-3, "seed.w 1e-6 file *t45a")
sr.runt(com, seed1, pump1 % 6e-3, "seed.w 1e-6 file *t45b")
sr.runt(com, seed1, pump1 % 7e-3, "seed.w 1e-6 file *t45c")
```

sr.Silent was set to avoid interleaving output from the different threads. If you wish to watch output as the simulation progresses, it may be better to use two or more python shells – one for interactive work and others for running simulations.

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 parallel jobs, and addcom() takes the same kinds of arguments as run() and runt(). The following example will run 5 simulations using two threads. The last line waits until all jobs have completed.

```
q = sr.JobQueue(2)
q.addcom(com, seed1, pump1 % 4e-3, "file *t46a")
q.addcom(com, seed1, pump1 % 5e-3, "file *t46b")
q.addcom(com, seed1, pump1 % 6e-3, "file *t46c")
q.addcom(com, seed1, pump1 % 7e-3, "file *t46d")
q.addcom(com, seed1, pump1 % 8e-3, "file *t46e")
q.wait()
```

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. An example of this is shown in Section 9.2.2.

9 Example 4 – Optical parametric oscillator (OPO)

The examples so far have been OPAs, which are non-resonant devices. This example shows how to add a resonator, use quasi-phase-matching (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 9.1 shows the OPO.

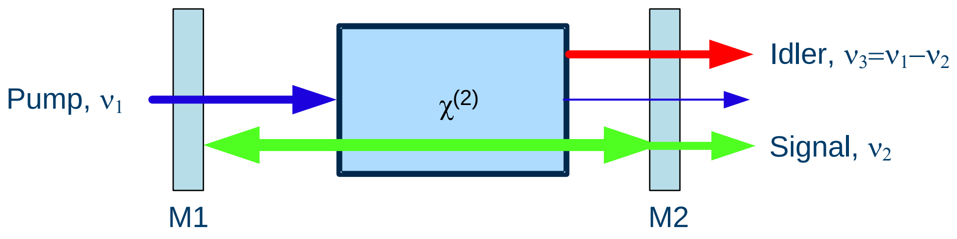


Figure 9.1 Optical parametric oscillator with resonator formed by the mirrors M1 and M2. In this example, only the signal beam is resonant, and the pump beam makes a single pass through the crystal.

As explained in Chapter 3, 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 dt = $t_R/(\text{tsf} \cdot \text{nt})$. ’tlen’ is the length of the simulation interval. The number of round trips simulated is $\lceil \text{tlen}/t_R \rceil$.

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 through a Func3Table object. ’ref_temp’ is the reference temperature for thermo-optic calculations.

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 that would diffract out of a real resonator.

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

```
ResoDirectProp *path = ResoDirectProp::make_linear(2);
```

The argument 2 tells that beams make two passes (forward and backward) in the resonator. Figure 9.2 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.

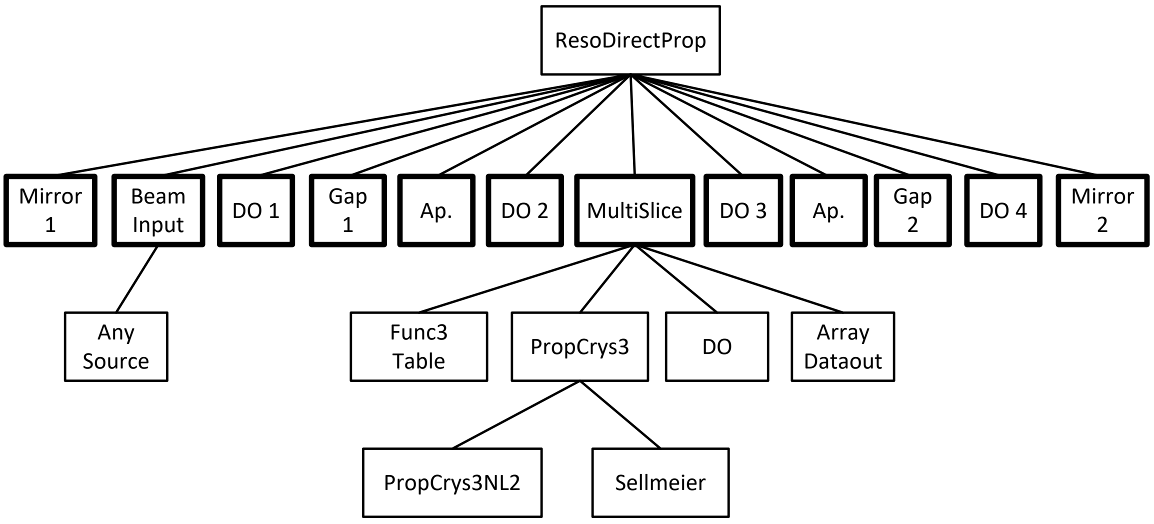


Figure 9.2 Structure of the OPO simulation program. The components in the optical path have thick frames, and they are shown in the same order as in the path. Helper components have light frames. "DO" means Dataout and "Ap." means apertures (optional).

Resonators and mirrors

As far as Sisyfos is concerned, optical paths are straight. Mirror components act only by their curvature and/or loss, they do not change the direction of the beam, and they have nothing to do with the optical layout. A Resonator-object is a resonator even if it contains no Mirrors. By definition, the beam reflected by a Mirror remains in the optical path, and the transmitted beam is coupled out. For example, in a strictly singly resonant OPO the non-resonant beams must be coupled out through Mirrors with zero reflection.
Since Sisyfos does not know the geometry of the optical path, 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 curved 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 BeamInput object is placed after mirror 1, and it too operates in the forward direction. The BeamInput object normally *replaces* [1] the content of the beam(s) for which it has sources, so it

[1] It has a mode where the input amplitude is added to the beam. This can be used for example to insert a short pulse in the first round trip, and if the pulse then drops to zero it does not matter that it is added to subsequent round trips.

does not support a partially resonant pump beam. Such a device could be simulated by adding the BeamInput to the input port of the Mirror object instead of directly to the optical path. Gaps and optional apertures are inserted between the mirrors and the crystal. The soft edge of the aperture is added to the clear width, and its width is specified as a fraction of the clear width. The air gaps, apertures and crystal are set to work in both directions.

Dataout objects are inserted next to the mirrors and before and after the crystal. The Dataouts near the crystal are applied in both directions, whereas the others are only applied in the forward direction because when the mirrors are plane and spectrally flat, only the power of the beam will change between the forward and backward pass. With curved mirrors, it might be necessary to apply all Dataouts in both directions to monitor 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

```
 chi2_proc->set_mismatch(0);
```

See the documentation of PropCrys3 and Chi2Factory for details. A factor of $2/\pi$ is included in chieff to account for the effect of QPM [9]. The default Sellmeier equation in the example program includes temperature dependence. run_sim() passes 'tsf' instead of 'dt' to main_init2a(). The argument is interpreted as tsf when it is ≥ 1.

Run the example program by typing

```
sr.run("opo_ex4", "-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(1,1)
t = u.t*1e9
clf()
plot(t, u.m)
plot(t, g.ap(4,1))
plot(t, g.ap(4,2))
plot(t, g.ap(4,3))
```

'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 selected for beam 2 in Dataout 2, you can also retrieve the power at full time resolution:

```
u = g.p(4,2)
```

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, "k--")
```

gives the same result as 'plot(t, g.ap(4,2))'. 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(4,2)[0,:])
plot(g.tnf(-2,2)[0,:])

clf()
plot(g.tff(1,2)[0,:])
plot(g.tff(4,2)[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(4,2)
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(4,2)
v = g.s(4,2)
nu = u.nu*1e-12
clf()
plot(nu, u.m)

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

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

```
sr.run("opo_ex4", "-tf opopar1 file *y010 ap 0")
```

and compare the results:

```
g0 = gfm3("y000")
g1 = gfm3("y010")
clf()
plot(g0.tnf(4,2)[0,:])
plot(g1.tnf(4,2)[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.

9.1 Optimization

It is often necessary to search the parameter space for a good set of parameters, and in that case it is convenient to use loops to run multiple simulations. Here is an example that scans a range of output couplings and crystal lengths. To make the dependence on output coupling more interesting, the reflectance of mirror 1 is set to 0.9 to represent losses in the resonator. The example uses JobQueue to manage parallel simulations.

```
q = sr.JobQueue(4)
len_range = [10,20,30]
oc_range = list(range(20,91,10))
for len_mm in len_range:
  for oc in oc_range:
    fname = "a_%d_%d" % (len_mm, oc)
    p = f"lno.len {len_mm/1e3:.4g} m1.ref [0 0.9 0] m2.ref [0 {1 - oc/100:.3f} 0]"
    q.addcom("opo_ex4", "-tf opopar1", p, " file *%s" % fname)
q.wait()  # Wait before plotting results
clf()
for len_mm in len_range:
  w = []
  for oc in oc_range:
    fname = "a_%d_%d" % (len_mm, oc)
    g = gfm3(fname)
    w.append(g.w(4,2) * oc/100)
  plot(oc_range, array(w)*1e3)
legend(["clen %d" % x for x in len_range])
xlabel("OC (%)")
ylabel("Output signal energy (mJ)")
```

As expected, the optimal output coupling is greater for greater crystal length.

9.2 Thermal effects

Heating of the crystal by absorption can affect the beams by thermal lensing and thermal phase mismatch. This example shows how to compute the energy absorbed in the crystal, use it to compute the steady state temperature distribution (assuming a high pulse rate), and use the temperature distribution in a new OPO simulation.

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

```
sr.run("opo_ex4", "-tf opopar1 file *y020 lno [abso [1 1 10] 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 per unit length in each slice. The fields in the stored structure correspond to these parameter names:

```
g = gfm3("y020")
v = g.get("MultiSlice.1.heat")
v.fields()
```

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

```
u = v.st
clf()
imshow(u.mr[2,:,:])
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,:])
legend(["Slice %d" % i for i in range(5)])
xlabel("x-position (sample points)")
```

The last plot is shown in Fig. 9.3. 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(1,1)
w_out = g.w(4,1) + g.w(4,3) + g.w(4,2)*oc
w_loss = w_in - w_out
v = g.get("MultiSlice.1.heat")
w_abso = sum(v.slice_tab*v.srt)
[w_loss, w_abso, w_loss - w_abso]
```

The small difference between w_loss and w_abso can be attributed to the apertures.

Sisyfos includes a program called find_temp2, in the StdApp directory, which can compute the steady-state temperature distribution in a crystal from the distribution of absorbed power and the boundary conditions. find_temp2 can only handle simple geometries with rectangular crystals and cooling through the side faces. Cooling through the end faces is neglected.

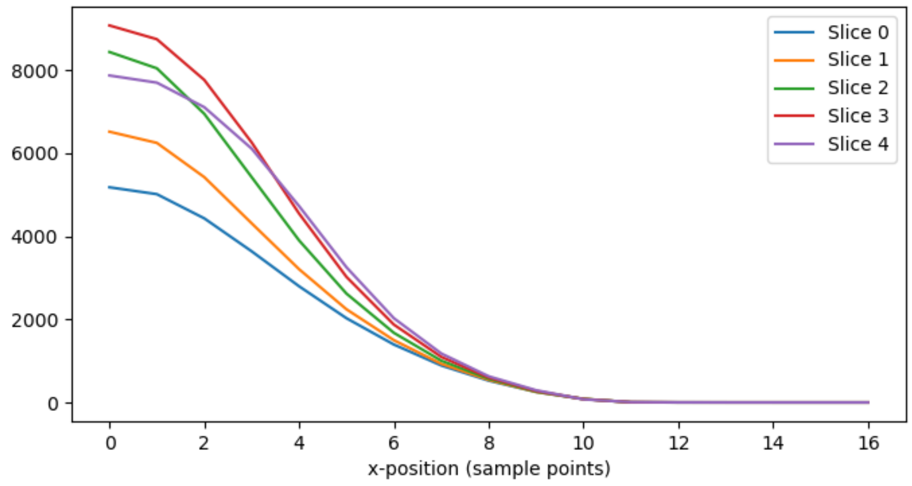


Figure 9.3 Distribution of absorbed power in each slice.

```
ft_com = "../../StdApp/FindTemp/find_temp2"
sr.run(ft_com, "-tf temp_par pt -bf y020.sis MultiSlice.1.heat.st file *y020_t")
```

"pt" is a parameter structure for a Func3Table object that represents the distribution of absorbed power. In this case the data for the table are fetched from the specified field in the file y020.sis. 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_temp2 – 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 if 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 (10 points) and wide in the y-direction (30 points), with cooling through the wide faces (perpendicular to -x and +x). 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, see Section 3). 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.

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:

```
sr.run("opo_ex4", "-tf opopar1 file *y021 lno [abso [1 1 10] slices 5",
  "temp [-bf y020_t temp v_out 300]]")
```

and compare the results:

```
g0 = gfm3("y020")
g1 = gfm3("y021")
clf()
subplot(2,2,1)
plot(g0.ap(4,2)/1e3)
plot(g1.ap(4,2)/1e3)
xlabel("Time (samples)")
ylabel("Power (kW)")
text(0,50,"(a)")
legend(["Without temp.", "With temp."])

subplot(2,2,2)
plot(g0.ts(4,2)*1e16)
plot(g1.ts(4,2)*1e16)
xlabel("Frequency (samples)")
ylabel("Spectral energy (a.u.)")
text(0,3,"(b)")

subplot(2,2,3)
plot(g0.tnf(4,2)[0,:]/1e4)
plot(g1.tnf(4,2)[0,:]/1e4)
xlabel("Position (samples)")
ylabel("Fluence (J/cm$^2$)")
text(2,0.9,"(c)")

subplot(2,2,4)
plot(g0.tff(4,2)[0,:])
plot(g1.tff(4,2)[0,:])
xlabel("Far-field position (samples)")
ylabel("Far-field fluence (a.u.)")
text(2,0.72,"(d)")
```

Figure 9.4 shows the results. 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 Func3Table objects, which are used to input absorbed power density to find_temp2 or temperature to opo_ex4, interpolate if necessary, so the resolutions for the OPO simulation and the

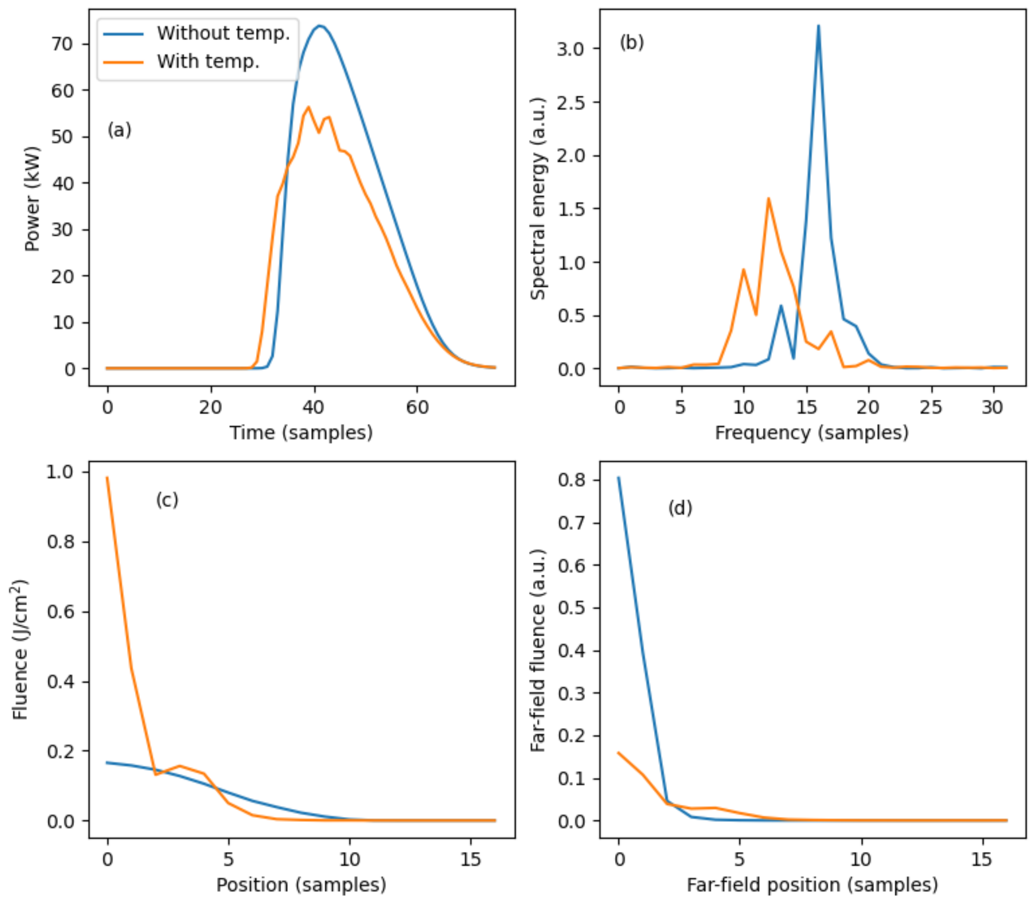


Figure 9.4 Comparison of simulation without thermal effects (file y020) and with thermal effects (file y021). (a) Signal power. (b) Signal spectrum. (c) Near-field profile. (d) Far-field profile.

temperature calculation do not have to be equal. The 'pt.v_out 0' option to find_temp2 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_ex4 allows the beam matrix to be wider than the temperature matrix and sets the temperature to 300 K for points outside it.

9.2.1 Script for iteration

Open the file iterate_script.py in an editor and read it. It contains a simple loop that repeats temperature calculation and OPO simulation until the signal energy changes less than the specified tolerance or the maximum number of iterations is reached. 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 when running the example. Run the file with:

```
run iterate_script
```

In this example, the signal energy converged to the specified tolerance after 3 iterations. In practice, it may be necessary to check for convergence of other quantities too. The following code reads the files and plots the signal energy versus iteration:

```
v = []
for i in range(3):
  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.

9.2.2 Python class for iteration

The script in the preceding section has to run sequentially because each iteration depends on the result from the one before. If you need to run iterations for different sets of parameters it would be useful if the different iteration loops could run in parallel. SisRun and JobQueue cannot handle arbitrary scripts, but JobQueue can handle classes derived from SisJob.

The file iterate.py defines such a class, TempJob, which can iterate OPO-simulations and temperature calculations. TempJob inherits SisRun.PythonJob, which in turn is derived from SisRun.SisJob. The method is similar to the script above, but the class is more flexible. JobQueue will call the run2() method of the TempJob object. Other jobs can be run in parallel by adding them to the JobQueue.

```
import iterate as it
com = "opo_ex4"
ft_com = "../../StdApp/FindTemp/find_temp2"
q = sr.JobQueue(2)
y = it.TempJob(com, "-tf opopar2", ft_com, "-tf temp_par", 10, 0.01, "tsim2")
q.addjob(y)
```

The results can be plotted in the same ways as for the script. This program is not a general solution, but it is an example which users can modify according to their needs.

10 Example 5 – Laser simulation and apertures

This chapter presents simulation of a pulsed laser and uses it as an example of how to add apertures to obtain correct results without excessively big matrices. The laser has been simplified in several ways: First, the laser medium is modelled as an ideal four-level system. Second, the pump mechanism is omitted, so the laser medium is simply taken to have a prescribed distribution of population inversion at the start of the simulation. Third, the beam is represented by a single temporal sample point per round trip (nt=1, and dt equals the round trip time), so spectral properties and fast dynamics are omitted. The effects that are included are multiple transverse modes and apertures.

The laser is shown in Fig. 10.1. It consists of a laser medium, two mirrors, and a number of apertures. The radius and reflectance of the mirrors, the length of the resonator, and the length, position and initial population distribution of the laser rod can all be varied, as can the size and number of apertures. The file acbd_reso_calc.txt in the example directory shows how Sisyfos' python functions can be used to analyze the resonator by ABCD-matrix calculations.

The laser starts with a certain population inversion. There is no explicit representation of a Q-switch, but it is effectively Q-switched at the moment the simulation starts. Try first with a narrow distribution of population inversion, similar to what could be achieved by end-pumping with a high-quality beam:

```
pop_1 = "pop0 [func2 [sgauss wxy 0.4e-3] vs 3e19]"
# Parameters for resonator
reso_1 = """
 m1 [rc 1]
 m2 [rc 0 ref 0.9]
 gaps [0.1 0.1 0.01]
 rod_len 0.02
"""
sp_2 = "smode 2 points 64 scale 70e-6"
sr.run("laser_ex5", reso_1, pop_1, sp_2, "tlen 2e-7 file *b000 store1 +[ e f]")
```

"pop0" expects a 3D function, and "pop0 [func2 [sgauss...." means that the 2D function sgauss is used for (x,y) and that there is no z-dependence. Due to the strong gain guiding, the laser produces a nice beam even without apertures. Repeated simulations show small fluctuations from pulse to pulse. The M^2 beam quality can be estimated by the python class PulseAna.BeamTransformer:

```
g  = gfm3("b000")
plot(g.ap(5,1))
# Makes BeamTransformer using data from position 5, beam 1, round-trips 22-59
q = pa.make_beam_transformer(g, 5, 1, list(range(22,60)))
q.m2xy()
```

To save time in BeamTransformer, the pulse is plotted and only the samples with substantial power

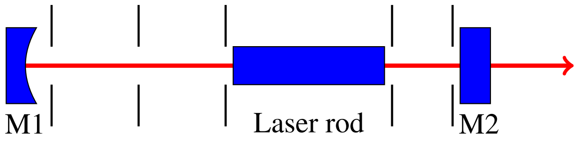


Figure 10.1 Laser with apertures. The figure indicates layout and is not to scale. Gaps and mirror-radii can be configured.

are included. Note that the complex field ("e") must have been stored for the position and beam used by BeamTransformer. BeamTransformer includes methods to collimate the beam, but since the reported M^2 value in this example is about 1.05, there is not much to gain. It is consistent with theory that the beam has a waist at the plane mirror. For a demonstration of the collimation method, try the same for the beam at the curved mirror (position 1):

```
q = pa.make_beam_transformer(g, 1, 1, list(range(22,60)))
q.m2xy()
q.collimate(use_lp=True)
```

The M^2 estimate before collimation is about 1.2. The collimation method reports an optimal lens power of about $-1\ \mathrm{m}^{-1}$, which is consistent with the mirror curvature, and the M^2 estimate after collimation is again about 1.05. See documentation of BeamTransformer for details on its methods.

Now try a much wider distribution of population inversion, which will support multiple transverse modes:

```
sp_3 = "smode 2 points 128 scale 45e-6"
pop_2 = "pop0 [func2 [sgauss wxy 3e-3] vs 3e19]"
sr.run("laser_ex5", reso_1, pop_2, sp_3, "tlen 2e-7 file *b100")
g = gfm3("b100")
imshow(g.tnf(1,1))
plot(g.tnf(1,1)[64,:])
plot(g.tff(1,1)[64,:])
```

In this case the beam is very poor, and both the near-field and the far-field are too wide for the respective matrices in the example. Instead of repeating the simulation with greater resolution and greater matrix (and hence a much greater number of points), we introduce a physically realistic 2 mm radius for the laser rod:

```
sr.run("laser_ex5", reso_1, pop_2, sp_3, "tlen 2e-7 file *b110 ap [0 0 2e-3 0]")
```

Now the near-field is just contained in the matrix, the far-field is well within, and $M^2 \approx 20$. Try to make the rod even narrower (or equivalently, add limiting apertures at each end):

```
sr.run("laser_ex5", reso_1, pop_2, sp_3, "tlen 2e-7 file *b120 ap [0 0 .5e-3 0] ")
sr.run("laser_ex5", reso_1, pop_2, sp_3, "tlen 2e-7 file *b130 ap [0 0 .4e-3 0] ")
```

These yield M^2 of 1.5 and 2.1, respectively. Comparison of the far-fields show that the narrow aperture gives rise to a pedestal, which can be ascribed to diffraction from the edges. Simulation with soft apertures improve M^2 of 1.15 and 1.33, but it is clear that the narrow aperture is smaller than optimal:

```
sr.run("laser_ex5", reso_1, pop_2, sp_3, "tlen 2e-7 file *b140 ap [0 0 .5e-3 0] soft 0.2")
sr.run("laser_ex5", reso_1, pop_2, sp_3, "tlen 2e-7 file *b150 ap [0 0 .4e-3 0] soft 0.2")
```

Note that the soft region of the apertures come in addition to the clear aperture, which is the same as in the examples above. Figure 10.2 shows some of these beam profiles, plotted with the commands:

```
g0 = gfm3("b100")
g1 = gfm3("b110")
g2 = gfm3("b120")
clf()
plot(g0.tnf(1,1)[64,:],'b')
plot(g1.tnf(1,1)[64,:],'r')
plot(g2.tnf(1,1)[64,:],'g')
legend('b100 b110 b120'.split())
```

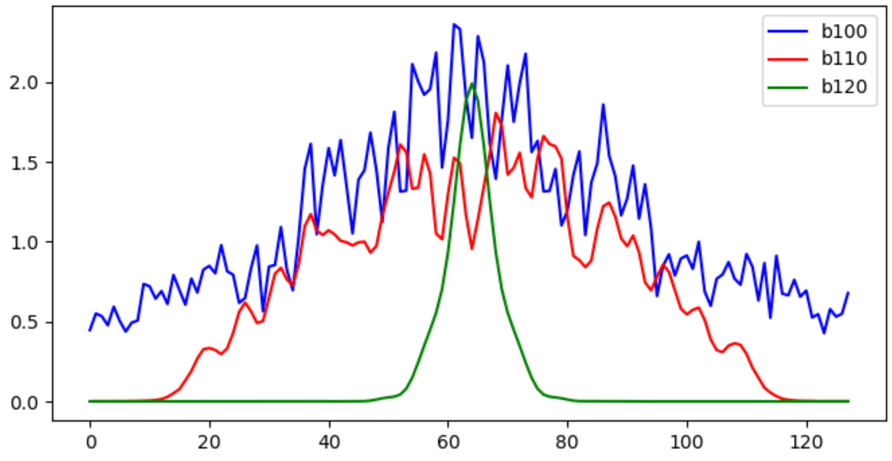


Figure 10.2 Beam profiles without aperture (file b100, blue) and with two different apertures (file b110 red and b120 green)

A reliable simulation should give the same result if the resolution is increased, and this can be a useful consistency check:

```
sp_4 = "smode 2 points 256 scale 20e-6"
sr.run("laser_ex5", reso_1, pop_2, sp_4, "tlen 1e-7 file *b200 ap [0 0 2e-3 0]")
```

In this case, the result is very different, with extremely poor beam quality. The far-field reveals strong components with angles around 24 mrad:

```
g = gfm3("b200")
u = g.tffr(1,1)
plot(u.ax*1e3, u.m[128,:])
```

Their sideways propagation in the 0.2 m gap is about 5 mm, which corresponds to the size of the matrix, so they can wrap around. The near-field at the mirror looks plausible, but plotting the near-field in the middle of the long gap (position 2) also reveals the problem. It can be solved by using a greater matrix:

```
sp_5 = "smode 2 points 512 scale 20e-6"
sr.run("laser_ex5", reso_1, pop_2, sp_5, "tlen 1e-7 file *b210 ap [0 0 2e-3 0]")
```

This gives $M^2 \approx 21$, consistent with the simulation with the same aperture at lower resolution (file b110), but at the cost of much longer run-time. Since the system limits the angular range of the beam to approximately $r_{\text{rod}}/(2L_{\text{gap}}) = 2\,\text{mm}/400\,\text{mm} = 5\,\text{mrad}$, it is legitimate to introduce this limit by a non-physical aperture in the middle of the gap and use a smaller matrix:

```
sp_6 = "smode 2 points 320 scale 20e-6"
sr.run("laser_ex5", reso_1, pop_2, sp_6, "tlen 1e-7 file *b220 ap [0 2.5e-3 2e-3 0] ")
```

Adding non-physical apertures requires careful thought by the user, and it is essential to avoid unrealistic clipping that falsely "improves" the results. The validity of the results should be checked by repeating simulations with different apertures and matrix sizes, and soft apertures may be necessary to avoid edge effects. In long propagation paths it may be necessary to insert multiple apertures. A large margin between the aperture and the edge of the matrix reduces the risk that a beam component that just passes the aperture, can wrap back into the clear area before the next aperture. Thus there is a trade-off between the matrix size and the density of apertures. Apertures are cheap operations, but propagation between apertures is more time consuming. On the other hand, increasing the matrix size is costly for all operations. The maximum angle of a beam component is approximately $\alpha_{\mathrm{max}} = \lambda/(2\Delta x)$, where Δx is the spatial resolution. This can be used to estimate the maximum distance between apertures for a given margin between aperture and matrix size. As the example showed, a smaller Δx, which supports a greater range of angles, can require a greater matrix or a smaller spacing between apertures.

A 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.

A '%' starts a comment that extends to the end of the line. A path preceded by '/' is ignored, even if it extends over multiple lines. Undefined paths normally raise exceptions, but if a path is preceded by '*' it is simply ignored if it is not defined in the parameter structure of the program.

It must be noted the meaning of a parameter string depends on the structure that is used to read it. For example, "a [x 1 y 2]" would be valid input to a structure "a" with numeric fields "x" and "y" or to a string "a", which would get the value "x 1 y 2". With hindsight, the "parameter language" should perhaps have been designed to be unambiguous, but brevity and simple syntax was considered more important at the time.

A.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
- lams 1e-6 bbo.len 0.005 *garbage 4 bbo.d22 2e-12
- lams 1e-6 bbo [len 0.005 /junk 5 d22 2e-12]
- -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. "junk 5" is commented out, and "garbage 4" is ignored because "garbage" is not defined in the parameter structure of the program.

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

```
opa_ex1 -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.

A.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.

Some parameter structures, such as for Func1Any, are recursive: For example, it includes Func1Compose as an alternative, and this in turn takes two other instances of Func1Any as its parameters. Recursive structures can obviously not be created in advance, they have to be expanded as the need for new nodes arises while parsing the parameter text. help shows information only for the part of the parameter structure that is created at start-up and not for nodes that *may* be created at runtime.

B Phase-matching calculations

Sisyfos includes python functions for calculating propagation modes and phase mismatch in birefringent crystals. Dispersion properties of a crystal are represented by Sellmeier equations. The Sellmeier classes in python correspond closely to those in C++, and they can read coefficients from the same text files. The following example shows how to make a Sellmeier object and use it for computing the principal refractive indices. "ZGP" refers to a directory under mat_dir, and the text file which defines the equations is "Zelmon2001.txt".

```
# Assumes that mat_dir has been in the startup file
# mat_dir = ".../Sisyfos6/code/MatData"
import Sellmeier2 as se              # Usually in the startup file
import CrysProp as cp
import Chi2Util as c2u
c1 = se.make_from_file("ZGP", "Zelmon2001", mat_dir)
c1.index_la(2e-6)
```

To find the refractive index for an arbitrary propagation direction you define a PropDir object and pass it to the index function:

```
dir1 = cp.PropDir(theta=0.8, phi=1.1)  # Angles in radians
cp.index_nu(dir1, 0, c1, 150e12)       # 0 for slow polarization
cp.index_nu(dir1, 1, c1, 150e12)       # 1 for fast
```

You can also obtain full information on the eigenmode (plane wave) corresponding to the slow or fast polarization. The following function returns a structure with the unit direction vectors for the magnetic field (b), displacement (d), electric field (e), Poynting vector (s), and wave vector (u). The structure also contains the refractive index and the walk-off angle.

```
s = cp.eigenmode_cam(dir1, 0, c1, 150e12)
print(s)
```

The group index can be found by numerical differentiation. The step size used in the differentiation can be controlled by the parameter dnu. If accuracy is important, you should try a few different step size and check if they give consistent results.

```
cp.group_index_nu(dir1, 0, c1, 150e12, dnu=1e9)
```

Here is an example to calculate the phase mismatch of a 2nd order interaction. Most of the functions take frequency as argument, so the conversion function lam2nu is used.

```
nu_pump = c2u.lam2nu(2400e-9)
nu_sig = c2u.lam2nu(3350e-9)
nu_tab = c2u.nutab(-1, nu_sig, nu_pump)
# c2u.nutab() Computes difference frequency and inserts it in first element
pola1 = [0,0,1]               # slow, slow, fast
dir2 = cp.PropDir(theta=0.9)  # phi=0 is default
c2u.mismatch_c1(dir2, pola1, c1, nu_tab)  # In inverse meters, without factor 2*pi
```

You can find the phase matching angle for a collinear interaction by

```
theta_pm = c2u.find_theta_c1(0, pola1, c1, nu_tab)
c2u.mismatch_c1(cp.PropDir(theta_pm), pola1, c1, nu_tab)  # Check
```

C Abbreviations

Abbreviation	Meaning
BBO	Beta-Barium Borate, $\beta - \mathrm{BaB_2O_4}$
FFT	Fast Fourier transform
FWexp(-2)M	Full width at $\exp(-2)$ times maximum
FWHM	Full width at half-maximum
HR	High reflectance
LNO	Lithium Niobate, $\mathrm{LiNbO_3}$
OPA	Optical parametric amplifier
OPO	Optical parametric oscillator
PPLN	Periodically poled Lithium Niobate
QPM	Quasi-phase matching
SBS	Stimulated Brillouin scattering
SRS	Stimulated Raman scattering

References

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

[2] Arlee V. Smith. *Crystal nonlinear optics*. AS-Photonics, Albuquerque, USA, 2015.

[3] Gunnar Arsholm and Helge Fonnum. *Simulation System For Optical Science (SISYFOS) - Tutorial*. FFI-rapport 2012/02042.

[4] M. Kolesik, P. Jakobsen, J. V. Moloney. Quantifying the limits of unidirectional ultrashort optical pulse propagation. *Phys. rev. A*, 86:035801, 2012.

[5] www.python.org

[6] 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.

[7] E. Hairer, S. P. Nørsett, G. Wanner. *Solving Ordinary Differential Equations I.* Springer, 2nd edition, 1993.

[8] W. H Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery. *Numerical recipes*. Cambridge university press, 3rd edition, 2007.

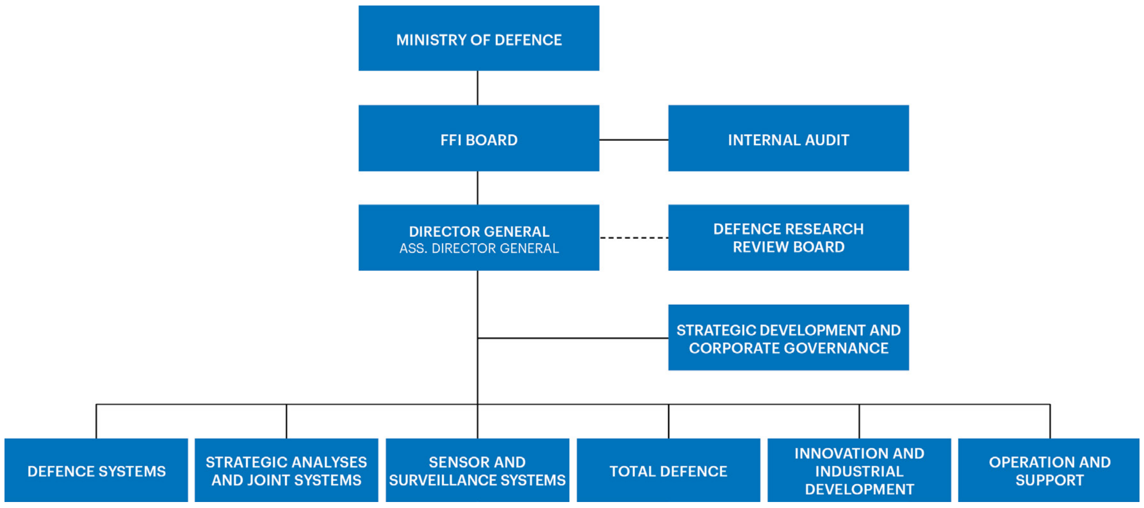
[9] A. E. Siegman. New developments in laser resonators. *Proc. SPIE*, 1224:2-14, 1990.

About FFI
The Norwegian Defence Research Establishment (FFI) was founded 11th of April 1946. It is organised as an administrative agency subordinate to the Ministry of Defence.

FFI's mission
FFI is the prime institution responsible for defence related research in Norway. Its principal mission is to carry out research and development to meet the requirements of the Armed Forces. FFI has the role of chief adviser to the political and military leadership. In particular, the institute shall focus on aspects of the development in science and technology that can influence our security policy or defence planning.

FFI's vision
FFI turns knowledge and ideas into an efficient defence.

FFI's characteristics
Creative, daring, broad-minded and responsible.


MINISTRY OF DEFENCE
FFI BOARD
INTERNAL AUDIT
DIRECTOR GENERAL
ASS. DIRECTOR GENERAL
DEFENCE RESEARCH REVIEW BOARD
STRATEGIC DEVELOPMENT AND CORPORATE GOVERNANCE
DEFENCE SYSTEMS
STRATEGIC ANALYSES AND JOINT SYSTEMS
SENSOR AND SURVEILLANCE SYSTEMS
TOTAL DEFENCE
INNOVATION AND INDUSTRIAL DEVELOPMENT
OPERATION AND SUPPORT

Forsvarets forskningsinstitutt
Postboks 25
2027 Kjeller

Besøksadresse:
Instituttveien 20
2007 Kjeller

Telefon: 63 80 70 00
Telefaks: 63 80 71 15
Epost: post@ffi.no

Norwegian Defence Research Establishment (FFI)
P.O. Box 25
NO-2027 Kjeller

Office address:
Instituttveien 20
N-2007 Kjeller

Telephone: +47 63 80 70 00
Telefax: +47 63 80 71 15
Email: post@ffi.no