WrightTools

Central package for Wright Group data processing and presentation.

Contents

1 Introduction

WrightTools is a python package for processing and viewing data. WrightTools is designed to be used in python scripts. For the purpose of this document, we will assume that WrightTools is imported with the alias `wt`: `import WrightTools as wt`.

As a living software package with evolving capabilities, detailed documentation of WrightTools here is neither easy nor useful. For specific usage notes look to the docstrings in WrightTools itself. This document is meant to provide a coarse-grained description of the entire package. It also describes design philosophies and best-practices when using WrightTools.

2 Background

2.1 Arrays

Arrays are always regular—that is, each axis position is defined over all other axes. In the two-dimensional case this means that each row contains a value for the same set of column positions (and *vice versa*). Positions in an array may be not-a-number, but they cannot be empty or undefined.

One-dimensional arrays are trivially regular.

In general, data is *not* regular. One of the central challenges that WrightTools addresses is to enforce regularity in data—a process called griding.

2.2 Interpolation

2.2.1 Delaunay

Attempts to maximize the minimal angle of all triangles—avoid 'sliver' triangles as much as possible.

Possible that both angles identical (as in grid)—not sure how choice is made in this case?—Essentially random?

See http://www.qhull.org/

2.3 Fitting

3 Fit

The fit module provides tools for fitting data. It contains a series of `Function` objects, which handle the actual minimization routines given a dataset. These can be used outside of WrightTools. For example, given `xi` and `yi` as 1D arrays:

The fit module also contains a class made to send `data` through fitting operations: `wt.fit.Fitter`. Like in WrightTools artists, `Fitter` will take high dimensionality data and iterate over the dimensions higher than the ones being addressed.

4 Data

The Data class is an incredibly general class capable of representing any rectangular dataset. It fills two important roles within WrightTools:

1. Data defines a single format to represent the results almost any measurement or simulation.
2. Data provides a suite of methods for data manipulation.

Almost every capability of WrightTools relies upon Data to some extent. Data can be thought of as the glue that holds the entire Wright group software stack together.

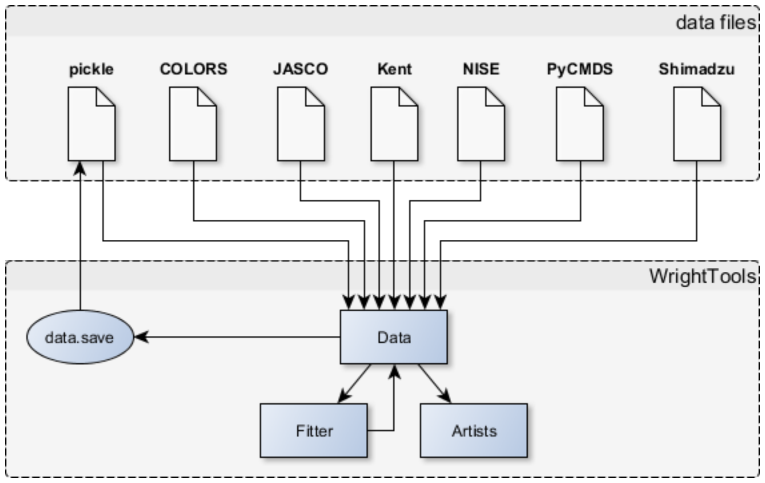

Figure 1: Data is the gateway to WrightTools.

Figure 1 represents WrightTools accepting data from a variety of sources and formats. Data objects are typically not created directly through Data.__init__ like most python classes. Instead, Data objects are created from files through a series of custom 'from_file' methods found in the data.py module. By creating separate methods, WrightTools can allow for easy Data creation from many different file formats. More details on Data creation can be found in Section 4.2. Once created, regardless of origin, the Data object is ready to be used by other classes in WrightTools, such as artists and Fitter.

4.1 Structure & Properties

The heart of Data are the numpy.ndarrays that contain the numbers themselves. Everything else is, in some sense, *decoration*. The arrays are not attributes of Data itself, rather they are attributes of the closely related wt.data.Axis and wt.data.Channel classes. Data contains lists of Axises and Channels. To understand the basic structure of Data, then, one must first understand the structure of Axis and Channel.

Axes are the *coordinates* of points in a dataset. Since all datasets are rectangular, axes are always one-dimensional. wt.data.Axis is the class that contains these coordinates and their properties. Table 1 highlights the most important attributes and methods of Axis. Axes may be regular or differential.

Channels are the values in a dataset. A Data instance may contain many channels, since each coordinate may have multiple recorded values e.g. signal, pyro1, and pyro2. wt.data.Channel is the class that contains these values and their properties. Table 2 highlights the most important attributes and methods of Channel. The

`points`	`numpy.ndarray` of axis coordinates
`name`	the 'python-friendly' name
`units`	the current units of the axis
`convert`	method to convert axis units
`label_seed`	list of subscript strings for label
`get_label`	method to get formated label string

Table 1: The most important attributes and methods of the `wt.data.Axis` class.

`values`	`numpy.ndarray` of dataset values
`name`	the 'python-friendly' name
`invert`	method to invert values
`zmin`	maximum
`zmax`	minimum
`znull`	null value
`signed`	`bool` flag for signed data
`normalize`	method to enforce scaling such that znull to zmax spans 0-1
`clip`	method to remove values outside of a certain range

Table 2: The most important attributes and methods of the `wt.data.Channel` class.

`signed` flag of `Channel` changes how other parts of WrightTools interact with the dataset. For example, an artist may choose to use a different colorbar if a dataset is signed. Allowing the 'null value' of a dataset to be non-zero is an important feature of WrightTools. When possible, developers should anticipate non-zero `znulls`. Also note that `Channel.values` may contain `NaNs`. When manipulating the `values` array directly, it is best practices to use `Channel._update()` to force the flags and attributes of the object to correspond to the new values.

The `Data` object has two primary attributes: `axes`, a list of `Axis` objects, and `channels`, a list of `Channel` objects. `axes` contains exactly one `Axis` object for each dimension of the dataset. The order of `axes` matters, so that the nth member of `axes` contains the coordinates along the nth dimension of the dataset. The order of `channels` is arbitrary. It is typical for code in WrightTools to assume that the most important channel is first. In addition to members of lists, `Data` dynamically sets the `Axis` and `Channel` objects it contains as attributes of itself according to the names of those objects. This means that users have two choices when accessing axes and channels. For example, consider an axis `w1` located in dimension `0`. To access this axis' points, one may go via the list: `data.axes[0].points` or directly `data.w1.points`. A consequence of this feature is that `Axis` and `Channel` object names must be unique to the `Data` container. Beyond the two main lists, `Data` is primarily a long series of methods that operate on the dataset it contains. Table 3 highlights the 'structural' attributes and methods of `Data`. There are many other attributes and methods of `Data`, most of which will be talked about in specific context elsewhere in this document.

4.2 Instantiation

The creation of a `Data` object from a raw file or other source may be a very different process depending on the dataset's origin. It is not reasonable to handle all details for all datastreams directly in `wt.data.Data.__init__`. Instead, `Data` objects are typically created using a `wt.data.from_X` method. These methods accept any relevant parameters and return an instantiated `Data` object. `wt.data.from_X` methods have been written for all of the commonly-used file formats in the Wright Group.

`axes`	list of the contained `Axis` objects
`channels`	list of the contained `Channel` objects
`name`	the 'python-friendly' name
`chop`	powerful method for accessing lower-dimensional slice(s)
`split`	method for splitting dataset at axis coordinate(s)
`transpose`	method to change order of dimensions in dataset
`copy`	method for getting deep copy
`flip`	method for flipping order of coordinates within an axis
`map_axis`	method to map dataset onto new coordinates using interpolation
`remove_channel`	method to remove a channel
`bring_to_front`	method to bring a channel to the front of `channels`

Table 3: The most important attributes and methods of the `wt.data.Data` class.

The details of importing from each source are unique, but there is a common theme:

1. Recognize axes of scan
2. Grid raw data onto scan axes
3. Assemble `Data` instance

4.2.1 Direct

Direct instantiation of `Data` is easy, if you start with your dataset as regular arrays in python.

4.2.2 PyCMDS

PyCMDS was developed from the start to work well with WrightTools. One of the most important products of that design is the rich system of headers that PyCMDS `.data` files posses, making the step of scan axis recognition trivial...

4.2.3 COLORS and Kent

4.3 Manipulation

chop, slice, join etc

4.4 Scaling & Normalization

4.4.1 dOD

Transient absorption data is most usefully viewed in dOD units, but it is naturally recorded in dI and I. The `dOD` method of `Data` does the necessary transformation. Conveniently, absolute OD cancels out such that dI

and I are all that is needed to calculate dOD.

$$\begin{aligned}
d\text{OD} &= -\log_{10}\left(\frac{I+dI}{I_0}\right)+\log_{10}\left(\frac{I}{I_0}\right) && (1)\\
&= -\left(\log_{10}(I+d\text{I})-\log_{10}(I_0)\right)+\left(\log_{10}(I)-\log_{10}(I_0)\right) && (2)\\
&= \log_{10}(I)-\log_{10}(I+dI) && (3)\\
&= -\log_{10}\left(\frac{I+dI}{I}\right) && (4)
\end{aligned}$$

Here I_0 is the probe intensity before the sample, I is the probe intensity after the sample without pump, and dI is the change in probe intensity caused by the pump.

There are a few instrumental considerations when measuring I and dI:

1. Typically it is most convenient to measure I as the average probe intensity for all shots (with and without pump). This is a good approximation as long as $dI << I$.
2. dI is defined as the total change in intensity. Boxcar averagers in active background subtraction mode return *one half* of the total change. The `dOD` method has a `kwarg` '`method`' that can be used to appropriately take this factor of two into account. PyCMDS takes this factor into account in shots processing.

5 Artists

The artists module contains a variety of tools, all related to visualizing data. It contains two workhorse classes: `wt.artists.mpl_2D` and `wt.artists.mpl_1D`, which generate general visualizations of 2D and 1D slices of data objects. It also contains a series of specialized artists.

5.1 Overview

5.2 Colormaps

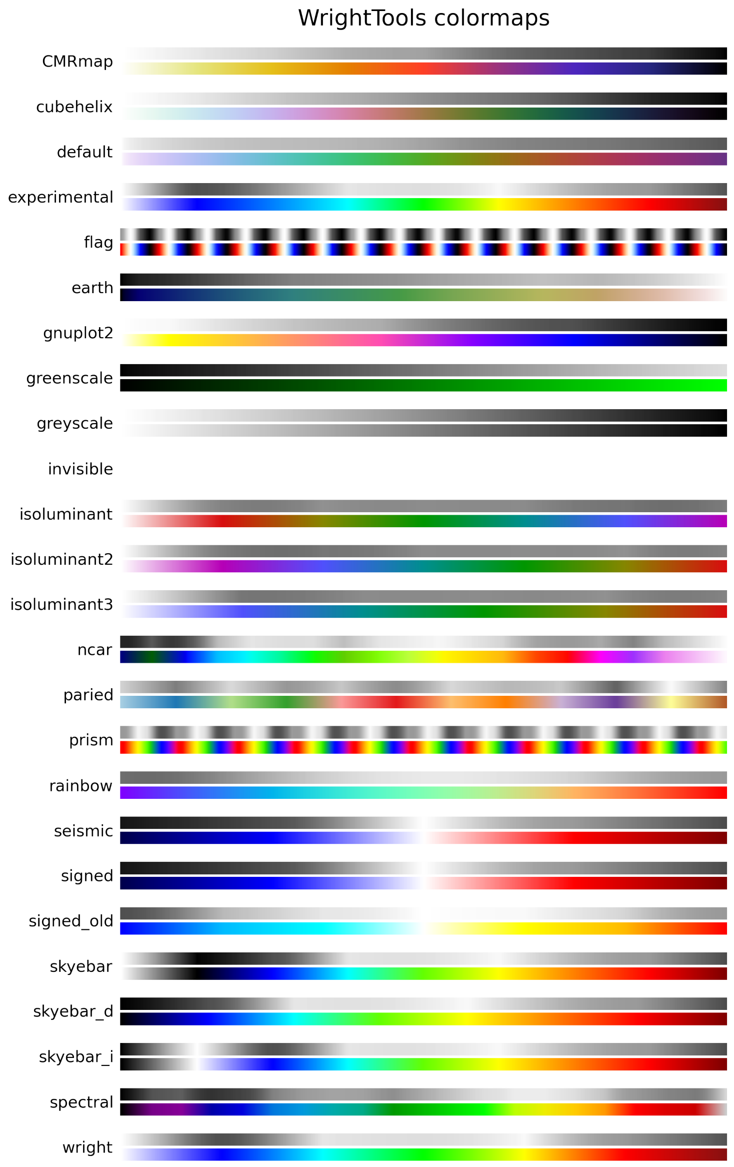

Figure 2: WrightTools colormaps as of version 1.5.0.

Figure 3 shows the RGB components of WrightTools' default colormap for single-signed data. This colormap is a custom implementation of the 'cubehelix' color scheme developed by Green [1]. The parameters

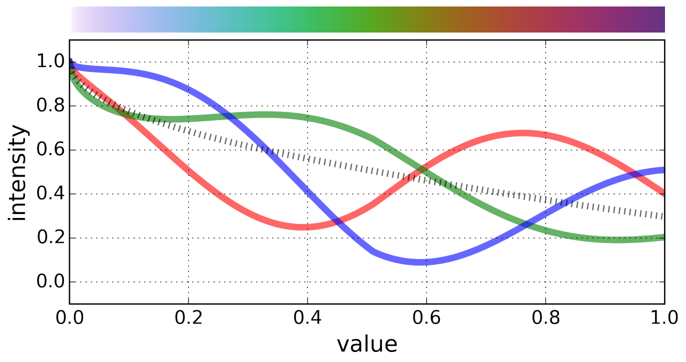

Figure 3: WrightTools default colormap RGB components.

were chosen to keep the colormap perceptual while closely matching the traditional Wright group colormap (`wt.artists.colormaps['wright']`). The colormap is slightly distorted in luminosity for aesthetic purposes.

5.3 Generic Artists

5.4 Speific Artists

5.5 Artist Helper Methods

6 Calibration

The `Calibration` class, defined in `calibration.py`, is designed to work with multidimensional calibration data. Like other classes in WrightTools, `Calibration` is units-aware and interpolation-enabled. The `Calibration` class is closely associated with the `.calibration` file format, which WrightTools defines.

In WrightTools a calibration is a simple proportionality defined over a set of experimental axes. The calibration system in WrightTools is designed primarily to help account for color-dependent measurement details in an experiment. Classic examples include

→ filter transparency.
→ detector sensitivity curve.
→ monochromator throughput.

For each color, the calibration defines a proportionality constant c, such that

$$\text{output} = c \times \text{input}. \tag{5}$$

output and input are generic signals—they could be intensities, amplitudes, voltages, or any combination thereof. `Calibration` does not try to be intelligent about signal units.

`Calibration` is multidimensional because some proportionalities depend on continuous axes other than color. For example, wheels with variable optical density will have a specific attenuation for each input color and each angle of rotation. `Calibration` supports unstructured (irregular) data, so that it is easy to add and replace regions of the calibration without having to redefine the entire calibration space. `Calibration` offers simple methods like `Calibration.append` to allow for easy manipulation of the calibration space.

In anticipation of irregular data, `Calibration` is designed to store proportionalities as one long list, with the coordinates for each value defined separately. For a `Calibration` instance `calibration` containing `n` proportionality constants defined in a `d`-dimensional space, `calibration.values` is a one dimensional numpy array with shape `(n,)`, and `calibration.points` is a two-dimensional numpy array with shape `(d, n)`. Importantly `calibration.points` is still two-dimensional even when the dimensionality of the calibration space is `1`—its shape will be `(1, n)` in that special case.

The `Calibration` class uses linear interpolation to determine the proportionality for intermediate coordinates. This means that it is *not* appropriate to define calibration axes for discrete things, such as grating angle or rotation of a set of discrete filters. It does not make sense to define the calibration for grating 0.25 as a quarter of the way between grating 0 and 1. In such cases, simply use multiple `Calibration` objects.

Figure 4 displays the interpolated two-dimensional calibration space for a very sparsely defined example optic. Note that the value of the calibration is undefined outside of the convex hull of the dataset. The contents of the associated `calibration` file is also shown.

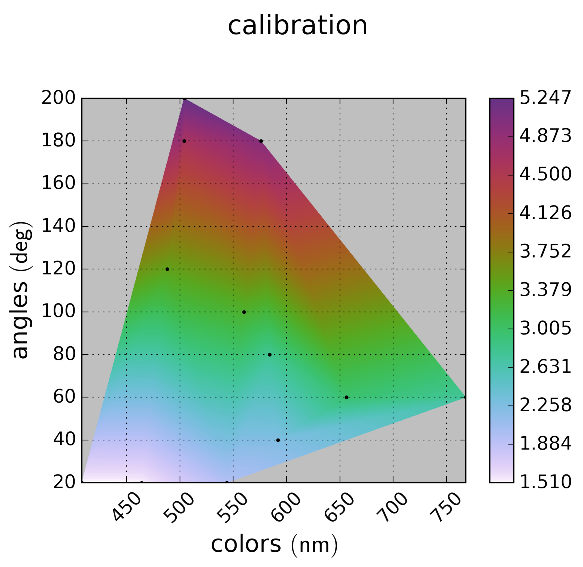

Figure 4: CAPTION TODO

```
# name: 'calibration'
# file created: '2016-12-09T17:05:34.463393+06:00'
# axis names: ['colors''angles']
# axis units: ['nm''deg']
# note: ''
4.080000e+02 2.000000e+01 1.510237e+00
4.640000e+02 2.000000e+01 1.518641e+00
4.880000e+02 1.200000e+02 3.467706e+00
5.040000e+02 1.800000e+02 4.630266e+00
5.040000e+02 2.000000e+02 5.246919e+00
5.440000e+02 2.000000e+01 1.959195e+00
5.600000e+02 1.000000e+02 3.304983e+00
5.760000e+02 1.800000e+02 4.987872e+00
5.840000e+02 8.000000e+01 2.792777e+00
5.920000e+02 4.000000e+01 2.245909e+00
6.560000e+02 6.000000e+01 2.967671e+00
7.680000e+02 6.000000e+01 2.750956e+00
```

`Calibration` objects are primarily intended for usage in data workup/post-processing. `Data.calibrate`, given a calibration object and a mapping of the calibration axes to the experimental axes, uses interpolation to transparently apply a calibration to a `channel`. Users may call `Data.calibrate` many times, accounting for each of the non-innocent optical elements used during the experiment.

7 Tuning

The tuning module provides tools for interacting with for optical parametric amplifier (OPA) tuning curves. It defines a main class, `wt.tuning.curve.Curve`, which is capable of representing tuning curves from multiple kinds of OPA. The tuning module also contains a series of modules and methods that accomplish certain specialized data processing tasks related to OPA tuning.

7.1 OPA Tuning Curves

OPA tuning curves are arrays of motor positions corresponding to parametric output colors. WrightTools offers an object-type to represent and manipulate OPA tuning curve arrays.

7.1.1 The Curve Class

The `Curve` class is made to contain OPA tuning curve data for any kind of OPA. By representing curves from different OPA hardware in the same format, WrightTools consolidates common tuning curve manipulations. `Curve` is a lot like the `wt.data.Data` class:

1. `Curve` is a container for other objects - in this case instances of the `wt.tuning.curve.Motor` class.
2. `Curve` objects are not meant to be instantiated directly. Instead they are returned by the 'from' methods of the `wt.tuning.curve` module.

OPA tuning curves are often defined in 'stages' where each stage accomplishes an additional parametric conversion on the light supplied by the previous stage. For example, a 'base' OPA may supply pump and signal to a mixing crystal to be added together again to sum frequency signal. The sum frequency may then go on again to be doubled in another mixing process and so on. Each stage has its own tuning curve. To accommodate this structure `Curve` objects may be 'nested' inside of themselves, such that each curve refers to its own attribute `curve.subcurve` to get the motor positions for the previous stage.

Curve objects have some useful methods for visualizing and working with tuning curve arrays.

Curve objects also serve as a convenient backend for OPA hardware in PyCMDS...

`curve.get_motor_positions` returns a numpy array with shape (motor count, color count)...

7.2 TOPAS-C Motortune Processing

Traveling-wave Optical Parametric Amplifier of White-light Continuum (TOPAS-C) is an automated OPA made by Light Conversion. In TOPAS-C, four motors are used in the creation of signal and idler. These can be divided into subcategories of 'preamp' and 'poweramp', each containing two motors. After signal and idler are generated, additional motorized crystals, 'mixers', can be used to upconvert or downconvert the base colors. This allows TOPAS-C to achieve a huge dynamic range of output colors, from ultraviolet to mid-infrared.z

WrightTools contains a processing method for each parametric process used in the Wright group TOPAS-C. Each of these methods is described in this section.

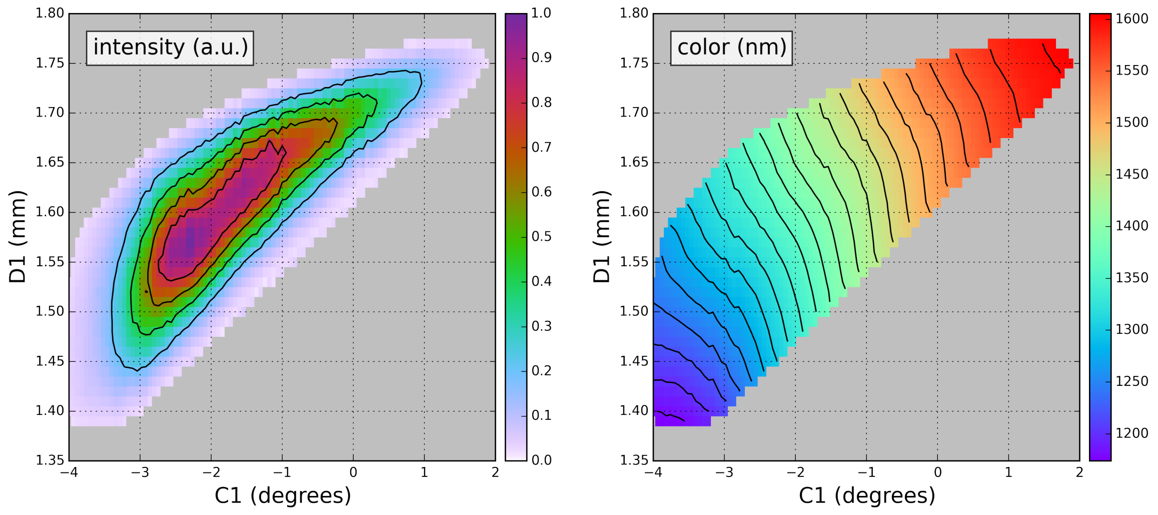

Figure 5: OPA output intensity and color as a function of C1 and D1. Measured on OPA1 (serial number 10743) 2016/01/13. Plotted pixels correspond directly to actual measured OPA outputs. The OPA output was measured using an array detector and fit to a single Gaussian lineshape. Grey pixels gave bad fits and were therefore filtered out. A small number of fits within the body of good data did fail and were filled in using bi-cubic interpolation. Data taken with PyCMDS MOTORTUNE module.

7.2.1 Signal Preamp

In the TOPAS-C preamp white light is mixed non-colinearly with a small amount of pump to generate signal and idler. The signal output is angularly isolated and sent on to seed parametric output in a second crystal (the poweramp).

Two motors are used to control preamp output: Crystal 1 (C1) controls the angle of the mixing crystal and Delay 1 (D1) controls the delay between white light and pump in the mixing crystal. White light in TOPAS-C is intentionally chirped, so that D1 influences color by choosing a portion of the white light spectrum to overlap with pump. Since phase matching controlled by C1 angle also influences preamp output color, both motors influence color and intensity of preamp output. The measured dependence of output intensity and color on C1 and D1 positions is shown in Figure 5.

The correlation between motor position, color, and intensity present special challenges to tuning TOPAS-C preamp...

To address this, WrightTools groups contours of constant color together and fits each contour to a Gaussian. This is shown in Figure 6.

In the final step, WrightTools passes the resulting tune points through a univariate spline to ensure smoothness and attempt to extend the curve outwards. In principle a polynomial fit could be used, but in practice splines are more robust to poor behavior at the edges of the tuning curve, where polynomials may oscillate wildly (Runge's phenomenon).

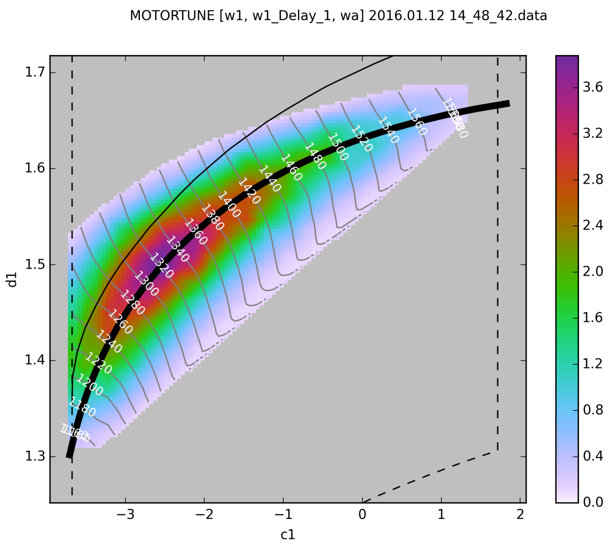

Figure 6:

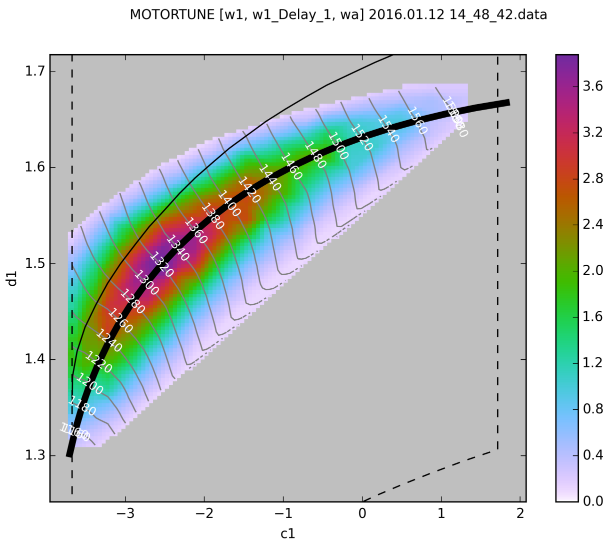

Figure 7:

7.2.2 Signal Poweramp

7.3 OPA800 Motortune Processing

OPA800

7.3.1 Signal & Idler

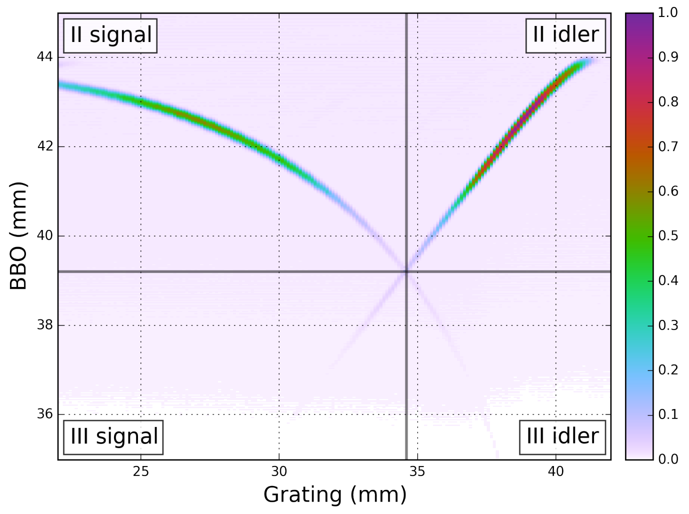

Figure 8: OPA800 (OPA2) signal and idler as measured by pyro. Data recorded 2015/10/15. Four quadrants are labeled with type and seed identification. Data taken with PyCMDS MOTORTUNE module.

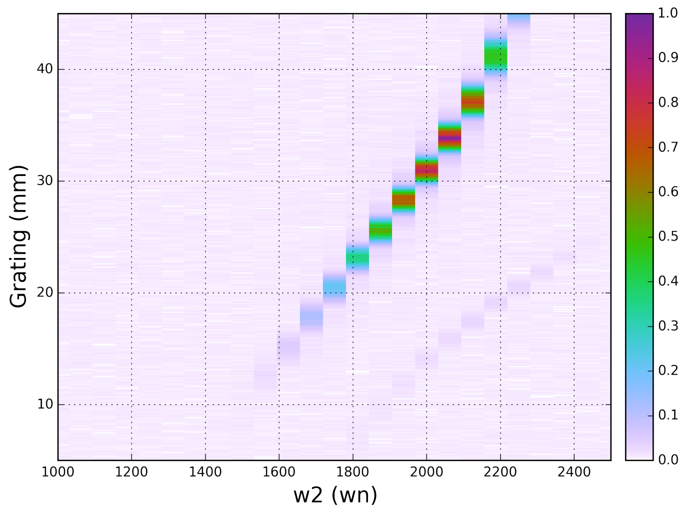

Figure 9: OPA800 (OPA2) DFG as measured by pyro. Data recorded 2015/10/16. Data taken with PyCMDS MOTORTUNE module.

7.4 CoSet

8 Diagrams

The diagrams module provides specialized artists and tools for creating diagrams commonly used to describe CMDS experiments.

References

[1] D.A. Green. "A colour scheme for the display of astronomical intensity images". In: *Bulletin of the Astronomical Society of India* 39.2 (2011), pp. 289–295.