


# WrightTools documentation

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Data</b>	<b>4</b>
2.1	Structure & Properties . . . . .	4
2.2	Instantiation . . . . .	4
2.3	Manipulation . . . . .	4
2.4	Scaling & Normalization . . . . .	5
2.4.1	dOD . . . . .	5
<b>3</b>	<b>Artists</b>	<b>6</b>
3.1	Colormaps . . . . .	6
<b>4</b>	<b>Fit</b>	<b>8</b>
<b>5</b>	<b>Tuning</b>	<b>9</b>
5.1	The Curve Class . . . . .	9
5.2	TOPAS-C Motortune Processing . . . . .	9
5.2.1	Signal Preamp . . . . .	9
5.2.2	Signal Poweramp . . . . .	10
5.3	OPA800 Motortune Processing . . . . .	10
5.3.1	Signal & Idler . . . . .	13
<b>6</b>	<b>Diagrams</b>	<b>14</b>

## Todo list

 include overview of package structure . . . . .	3
---	---

# 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`.

include overview of package structure

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 Data

The **Data** class 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.

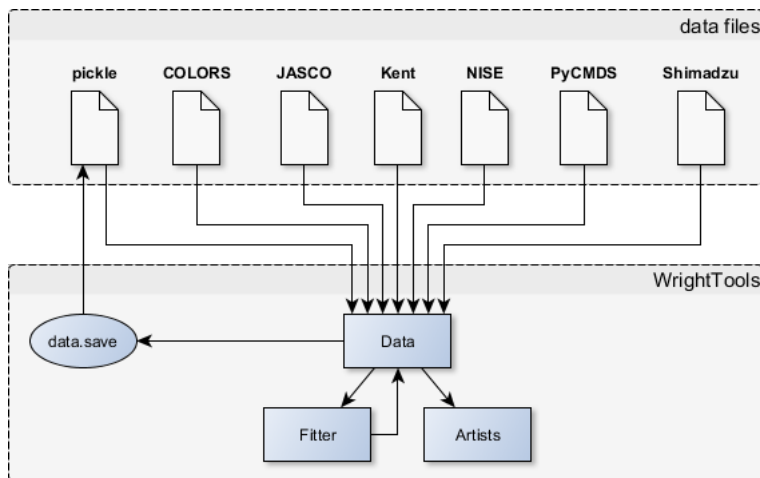


Figure 1: **Data** is the gateway to WrightTools.

Figure 2 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 2.2. Once created, regardless of origin, the **Data** object is ready to be used by other classes in WrightTools, such as artists and **Fitter**.

### 2.1 Structure & Properties

### 2.2 Instantiation

### 2.3 Manipulation

chop, slice, join etc

## 2.4 Scaling & Normalization

### 2.4.1 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  $dOD$ .

$$dOD = -\log_{10}\left(\frac{I + dI}{I_0}\right) + \log_{10}\left(\frac{I}{I_0}\right) \quad (1)$$

$$= -(\log_{10}(I + dI) - \log_{10}(I_0)) + (\log_{10}(I) - \log_{10}(I_0)) \quad (2)$$

$$= \log_{10}(I) - \log_{10}(I + dI) \quad (3)$$

$$= -\log_{10}\left(\frac{I + dI}{I}\right) \quad (4)$$

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  $dI \ll I$ .
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.

## 3 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.

### 3.1 Colormaps

Cite A colour scheme for the display of astronomical intensity images - D.A. Green [\[1\]](#)

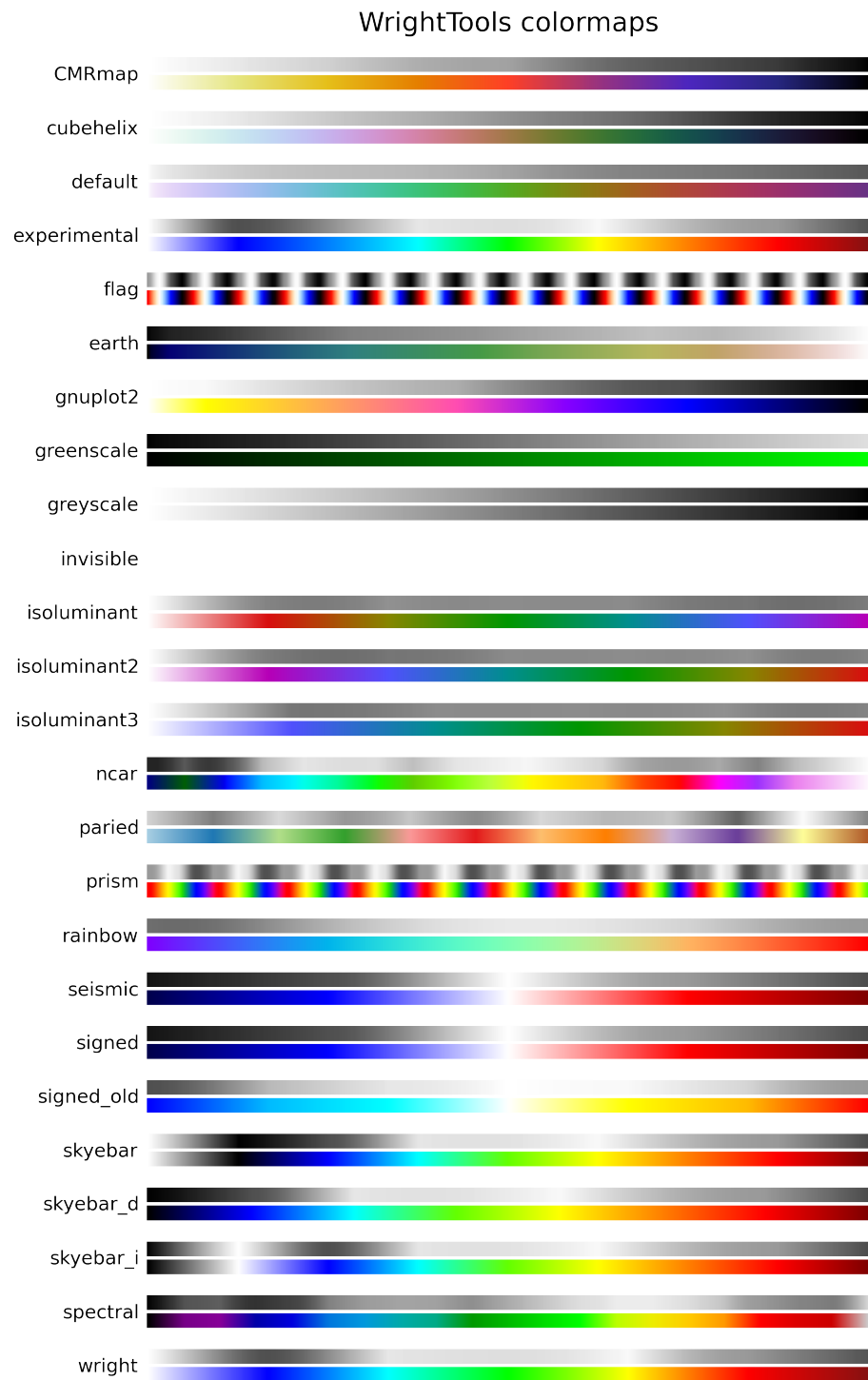


Figure 2: WrightTools colormaps as of version 1.5.0.

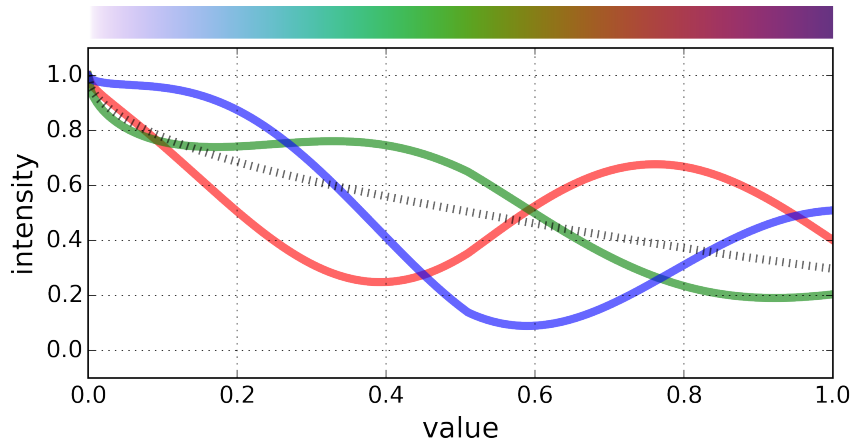


Figure 3: WrightTools default colormap RGB components.

## 4 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:

---

```

1      function = wt.fit.Gaussian() # create an instance of Gaussian class
2      center, width, amplitude, baseline = function.fit(yi, xi) # do fit on your 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.



## 5 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.

### 5.1 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 allows curves to share common code operations. Like the `wt.data.Data` class, `Curve` is a container for other objects: the `wt.tuning.curve.Motor` class. `Curve` objects are not meant to be instantiated directly. Instead they are returned by the ‘from’ methods of the `wt.tuning.curve` module.

Curve objects are useful for visualizing and working with tuning curve data.

Curve objects also serve as a convenient backend for OPA hardware in PyCMDS. `curve.get_motor_positions` and `curve.get_color` are made for this purpose.

### 5.2 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.<sup>z</sup>

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.

#### 5.2.1 Signal Preamp

In the TOPAS-C preamp white light is mixed non-collinearly 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.

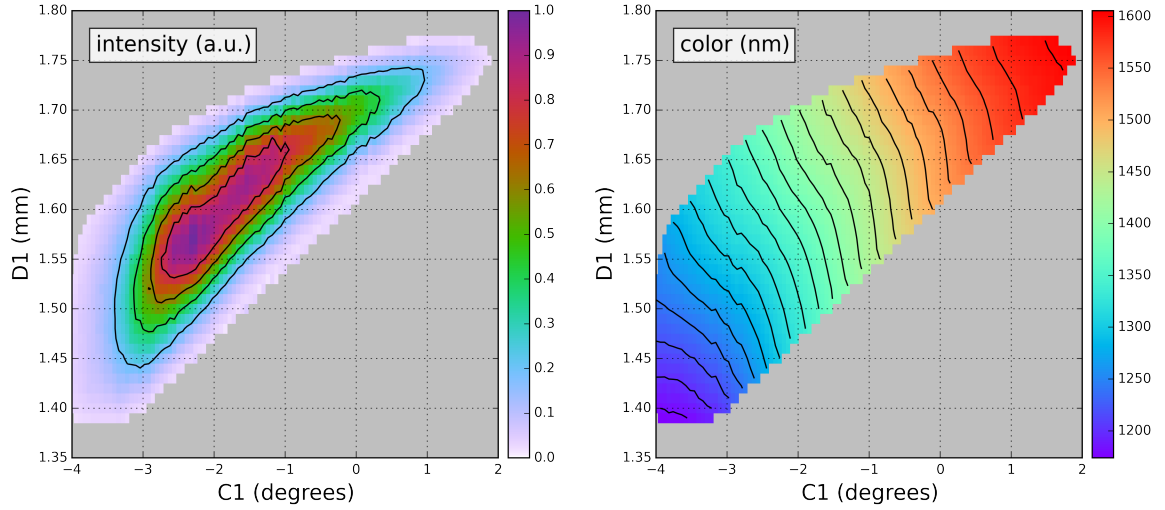


Figure 4: OPA output intensity and color as a function of C1 and D1. Measured on OPA1 (serial number 10743) 2016/01/13. Plotted pixels correspond directly to actual measured OPA outputs. The OPA output was measured using an array detector and fit to a single Gaussian lineshape. Grey pixels gave bad fits 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.

The measured dependence of output intensity and color on C1 and D1 positions is shown in Figure 5.2.1.

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 5.2.1.

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).

### 5.2.2 Signal Poweramp

## 5.3 OPA800 Motortune Processing

OPA800

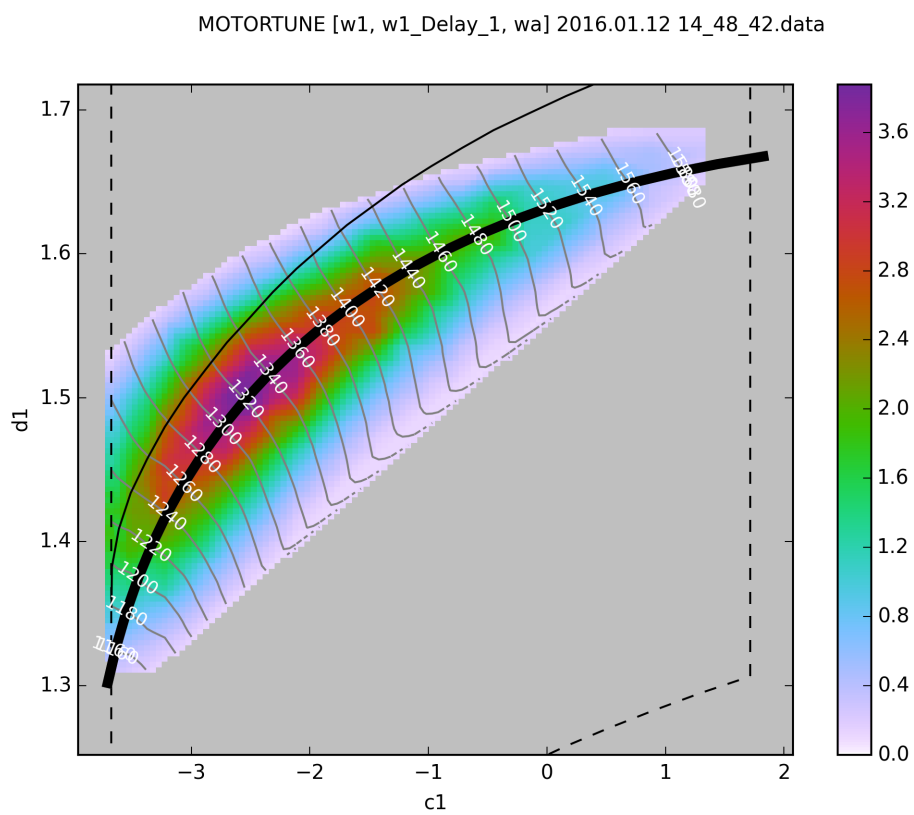


Figure 5:

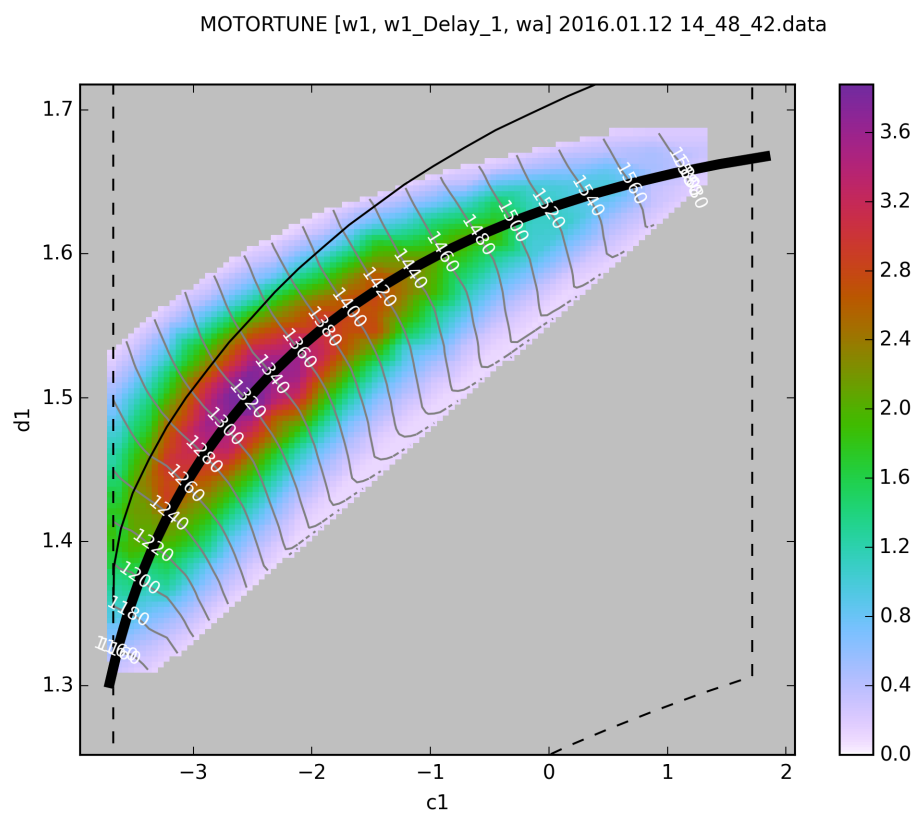


Figure 6:

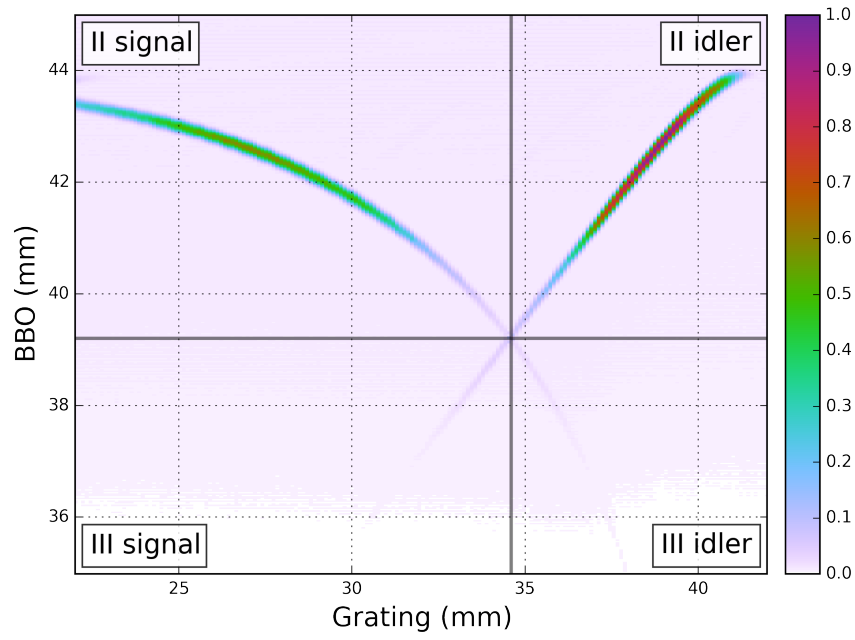


Figure 7: OPA800 (OPA2) signal and idler as measured by pyro. Data recorded 2015/10/15. Four quadrants are labeled with type and seed identification. Data taken with PyCMDS MOTORTUNE module.

### 5.3.1 Signal & Idler

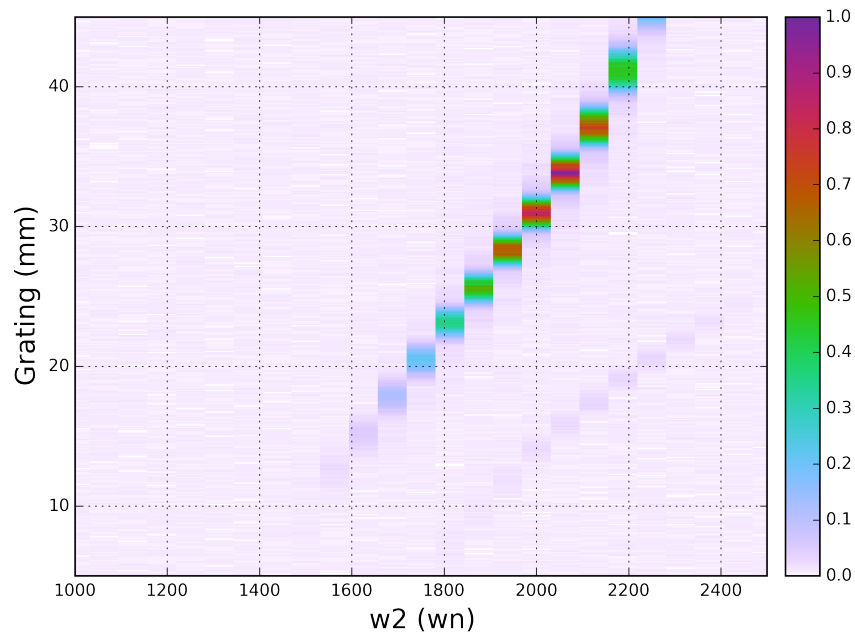


Figure 8: OPA800 (OPA2) DFG as measured by pyro. Data recorded 2015/10/16. Data taken with PyCMDS MOTORTUNE module.

## 6 Diagrams

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

## References

- [1] Nobody Jr. *My Article*. 2006.