Optics with Imperfect Mirrors
Part II Computational Project

29th April 2019

Abstract

A computational study of the effects of deformations in the surface of telescope mirrors is performed. The far-field diffraction pattern of mirrors is computed in a C++ program using discrete Fourier transforms. The depth of the defects causes a decrease in the central intensity of the diffraction pattern, thus raising the lower bound on the brightness of observable objects. The central amplitude follows: $\psi_0 \propto \exp\left(-(\sigma_\varepsilon/\lambda)^2/2\sigma_\psi^2\right)$, where σ_ε quantifies the typical defect depth. The spatial distribution of defects affects resolution, with smaller and denser deformations causing more spread-out fluctuations in the image.

---

1 Introduction

Modern telescopes use very finely polished curved mirrors to create sharp images. However, errors are an inherent property of any physical system and any manufacturing process. Thus, regardless of the way mirrors are manufactured (e.g. spin-casting or polishing), there are still imperfections in their shape.

The aim of this paper is a computational investigation of the effects of imperfections in the shape of a telescope mirror on the image. Several relationships are studied, as suggested in the projects manual [1]:

- the effects of tapered illumination on the size and central intensity of the image.
- the effect of a central hole in the mirror, due to the secondary mirror.
- the effects of random and correlated phase errors, due to physical dents in the surface of the mirror. The dents cause phase errors due to the change in path length with respect to the smooth shape.

A more detailed analysis of the theory and techniques used is given in Section 2. Implementation details are discussed in Section 3, and results are presented and discussed in Section 4.

2 Analysis

In the analysis of this problem, we treat the far-field diffraction pattern of the mirror under uniform or Gaussian illumination. In the far field regime, the diffraction pattern of an aperture is given by the Fourier Transform of the aperture function $A(x, y)$ [4, Chapter 10.2]. This is a reasonable approximation as long as:

$$\frac{R^2}{\lambda D} \ll 1, \tag{1}$$

known as the Fraunhofer limit, where R is the maximum extent of the aperture, λ is the wavelength of radiation and D is the distance from the aperture to the screen onto which the image is projected. In this problem we use:

$$\lambda = 1\,\text{mm}$$
$$R = 6\,\text{m},$$

which would require a distance on the order of 1 km for the far-field limit to hold.

However, considering parabolic mirrors brings the image plane closer. Besides, we can work in angular coordinates, with the image space spanned by:

$$p = k \sin\theta \approx \frac{kx'}{D}$$
$$q = k \sin\chi \approx \frac{ky'}{D},$$

where x' and y' are distances that would span a physical image plane, and k is the wavenumber. We thus eliminate the dependence on D. In these coordinates, the diffraction pattern is:

$$\psi(p,q) \propto \iint A(x,y) \exp\left(ipx + iqy\right) dxdy, \tag{2}$$

which is just a Fourier transform of A. The dimensions of the diffraction pattern will always be given in terms of p and q, and thus have units of m^{-1}.

NB: Seemingly we've also eliminated the dependence on λ, but that will be needed again in the analysis of dented mirrors (Section 2.5).

What is the meaning of this diffraction pattern? Since the incoming light is multiplied by the telescope's aperture function, the resulting image is a convolution of the astronomical objects being observed and the diffraction pattern of $A(x,y)$. Intuitively, for point-like stars, they will be seen through the telescope as "copies" of the diffraction pattern. Two factors are thus crucial:

- The width of the central disk, which limits the resolution of the telescope. If two objects are closer than this central width, they cannot be resolved [4, Section 10.2.6].
- The central intensity, which sets a lower bound on the brightness of objects that can be observed.

2.1 Discrete Fourier Transforms

The fact that the image is a Fourier transform of the aperture function is very useful. Fast Fourier Transform algorithms can compute discrete Fourier transforms of multi-dimensional data efficiently, and there exist many library implementations thereof. Here, the C++ `FFTW 3` library [2] was chosen, as suggested in the projects manual [1]. It is a well-established and well-tested library with very good computational efficiency.

Minor adaptations are required to use the DFT algorithms in the library. The 1D discrete Fourier transform is defined as [5, Chapter 12.1]:

$$H_k \propto \sum_{n=0}^{N-1} h_n \ \exp\left(2\pi i \frac{kn}{N}\right), \tag{3}$$

with the corresponding frequency values:

$$f_k = \frac{k}{N\Delta}, \quad k = 0..N-1, \tag{4}$$

where Δ is the sampling interval of the original signal. Comparing to a 1D discrete form of the diffraction integral (Equation 2):

$$\psi_k \propto \sum_{n=0}^{N-1} A_n \exp(ip_k x_n), \quad \text{where} \quad x_n = n\Delta \tag{5}$$

we see that we need to rescale:

$$p_k = 2\pi f_k. \tag{6}$$

2.2 Testing

To determine whether the program is outputting something sensible, we need to test it on a range of known results. Diffraction patterns for the following kinds of apertures are easy to compute analytically, and thus can be used for testing the program.

Rectangular aperture A rectangular aperture of size $-a < x < a$, $-b < y < b$ has diffraction pattern:

$$\psi(p,q) \propto \frac{\sin(pa)}{pa}\frac{\sin(qb)}{qb}, \tag{7}$$

thus it's expected to have the first zeros at:

$$p = \pm\frac{\pi}{a} \quad \text{and} \quad q = \pm\frac{\pi}{b}. \tag{8}$$

Circular aperture A circular aperture of radius R has a diffraction pattern called an Airy disc, with angular radius:

$$\sin\theta \approx 1.22\frac{\lambda}{2R} \quad \Rightarrow \quad p = 1.22\frac{\pi}{R}. \tag{9}$$

Both of these minima should appear in the diffraction patterns, and the scaling with the inverse of the aperture size should be observable.

2.3 Tapered aperture function

The amplitude of light illuminating a telescope mirror is often not uniform. A "taper" or "grading" is chosen, usually in the form of a Gaussian curve [7, Section 6.4]:

$$A(r) = \begin{cases} e^{-r^2/2\sigma^2} & , r \le R \\ 0 & , r > R \end{cases}, \quad \text{where} \quad r^2 = x^2 + y^2. \tag{10}$$

This is the multiplication of a Gaussian with a circular aperture of radius R. From the convolution theorem, its diffraction pattern is the convolution between another Gaussian and a Bessel function. Intuitively, this should lead to a "smearing" of the central maximum relative to a uniformly-lit mirror. Since the variance of a Gaussian is inversely proportional to the variance of its F.T., the smearing will be stronger at small σ.

Therefore, at small σ, the full width at half power (FWHP) should follow:

$$\Delta p_{\text{hp}} \propto \sigma^{-1}. \tag{11}$$

At large σ, as A approaches uniform illumination, the image tends to a Bessel function, and the FWHP will approach a constant value. Besides, since at smaller σ the integral of $|A|$ over the aperture is smaller, the central intensity should be smaller, tending to 0 as $\sigma \to 0$.

While the taper widens the central maximum, it also leads to suppression of the side lobes of the Airy pattern, thus possibly improving the resolution. See [4, Section 11.3] for an explanation.

2.4 Central hole

Classical telescopes, such as the Cassegrain design, have a central hole in the primary mirror due to the secondary mirror [7, Section 7.2.3]. The resulting aperture function can be viewed as the difference between that of the complete mirror (A_{complete}) and that of the hole (A_{hole}). Due to the linearity of the Fourier Transform, the diffraction pattern of this mirror should be the difference between the patterns of A_{complete} and A_{hole}.

We can quantify how similar these results are by subtracting the image produced by a holed aperture from the difference of the images produced by A_{complete} and A_{hole}.

Due to the Airy disc of the smaller A_{hole} being larger than that of the complete mirror, we expect the entire central disc to be dimmer due to the hole. The secondary rings should become irregular, because they have different frequencies in the patterns of A_{complete} and A_{hole}.

2.5 Bent mirrors

Mirror surfaces could have deformations in them due to manufacturing errors, altering the phase of light. As seen in Figure 1, the effect is that, for a dent of depth ε, there is a path difference of 2ε, leading to a phase difference of:

$$\Delta\varphi = 2k\varepsilon = \frac{4\pi}{\lambda}\varepsilon \tag{12}$$

We are going to investigate the effect of two types of phase errors:

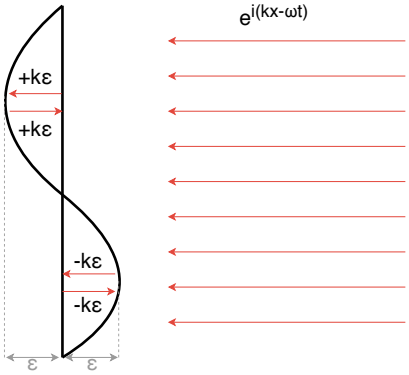


Figure 1: effect of deformations in the mirror surface on the phase of the wave. A deformation of depth ε produces an optical path difference of 2ε, one for the incident wave and one for the reflected wave.

- without spatial correlation, with a given RMS σ_ε
- spatially correlated on length l_c, to simulate deformations in the mirror.

The expected effect of these errors is to decrease the central intensity of the image, as they introduce decoherence to the light. We can also expect that errors on smaller correlation lengths will cause a wider spread of fluctuations in the beam pattern.

3 Implementation

More complete implementation details and instructions are found in the file `readme.md`. This section discusses details that are critical to the performance of the program.

The overall logic flow of the program is: The executable is invoked with one argument, naming a "configuration file". The instructions therein are read and executed in order, thus computing diffraction patterns for the described apertures. The results are printed to disk, and figures can then be produced by invoking the respective Python scripts for each problem.

C++ source files are in `src/cpp`, and plotting scripts in `src/scripts`. Sample configuration files can be found in `config`.

Note that, besides `FFTW`, I also used the GNU Scientific Library [3], mainly for random number generators and statistics.

3.1 Describing Apertures and Images

In the `FFTW` library two-dimensional N_x by N_y arrays are represented as one-dimensional arrays of complex numbers (the inbuilt `complex` type [6]) of length $N_x \times N_y$ [2, Section 3.2].

Because working directly with a 1D representation of 2D data can be clunky, I decided to wrap this functionality in a class called `Array2d`, declared in the header with the same name. The class stores the data internally in the 1D array representation, but has a more user-friendly interface. It defines the `[i][j]` and `(i, j)` operators for easy access to the element in the i^{th} row and j^{th} column.

Memory allocation One special feature of this class that breaks with convention is that the (compiler generated) copy constructor only performs a shallow copy, copying the pointer to the data array, but not the data itself. Only the explicitly defined constructor `Array2d::Array2d(int nx, int ny)` uses `fftw_alloc_complex` to allocate new memory for the array. This means that the user of the class has finer control over where memory is allocated, which is important in the case of large arrays. For example, using IEEE double-precision floating point, which occupies 64 bits of memory, a $2^{13} \times 2^{13}$ array of complex numbers occupies around 1 GB of memory.

For better understanding of when memory is allocated, compile the code with the variable `DEBUG_OUT` set to `true` in `array2d.cpp`. Then the calls to the constructor and destructor of `Array2d` will be printed to console.

Aperture generators These are a type of function that initialise an `Array2d` with a particular kind of aperture with given parameters. For example, there is a generator for circular apertures, one for circular Gaussian-illuminated, one for circular with random errors, etc.

3.2 Configuration Files

Configuration files are stored by convention in the `config` directory, and are a series of `key = value` lines. They contain:

- `nx` and `ny`. These are the size of input and output arrays.
- `tasks`, what actions to perform on each of the shapes
- `n_shapes`, the number of shapes
- for each shape, its properties: `type` (the name of the aperture generator), `lx` and `ly` (the domain of $A(x, y)$), and `params`, a list of parameters of the shape, e.g. the dimensions or the taper of Gaussian illumination.

3.3 Parallelization

The program can use multiple threads to process several shapes in parallel. The variable `N_WORKERS` in `main.cpp` controls the number of threads ("workers") used. Care must be taken to not run out of memory, as each thread allocates between 2 and 4 `Array2d` objects. The `FFTW` library only allows the creation of one Fourier transform plan at a time [2, Section 5]. This means that a larger number of threads has a larger initialization time, since each worker thread creates its own plans.

3.4 The `fftshift` operation

The result of this operation is shifting the zero-frequency component of the spectrum from the beginning to the middle of an array. This takes advantage of the fact that the Fourier frequencies (Equation 4) are cyclic with period $1/\Delta$ [5, Section 12.1.2]. Applying this is required to see a diffraction pattern as it would appear on a screen.

3.5 Correlating errors

As mentioned in Section 2.5, we want to investigate the effects of spatially-correlated phase errors. To produce such deformations, we convolve random gaussian-distributed numbers with a gaussian shape of the desired correlation width, as suggested in the manual [1]. Taking advantage of the convolution theorem, this is equivalent to multiplication of FTs of the two shapes, followed by a reverse FT. The numbers are then normalised to the desired RMS and set as the phase in the "mirror" array. The `corr_errors` aperture generator implements this.

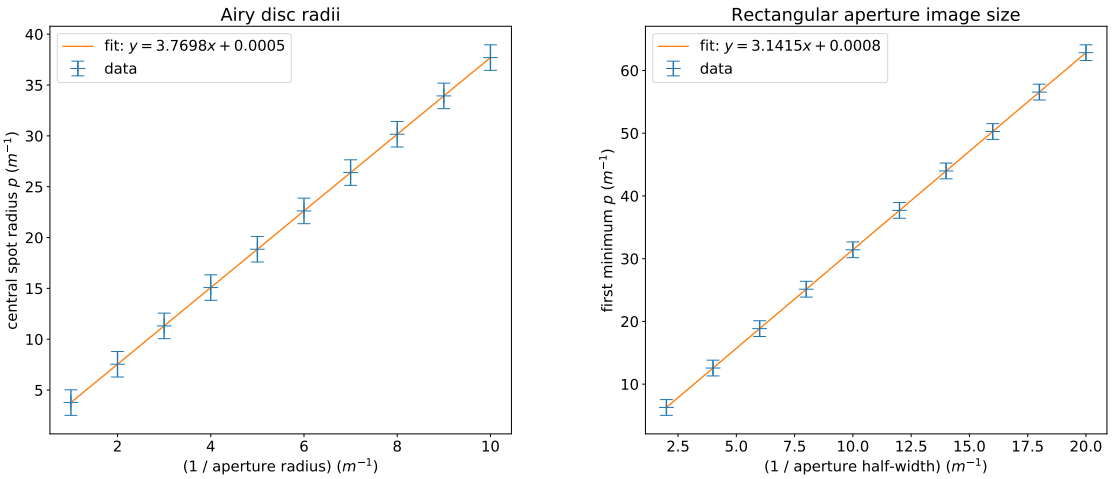
We want to ensure that the phase error produced by this method is the desired one, and that it doesn't change with the correlation length l_c. These properties are tested in Section 4.5.

4 Results and Discussion

4.1 Tests

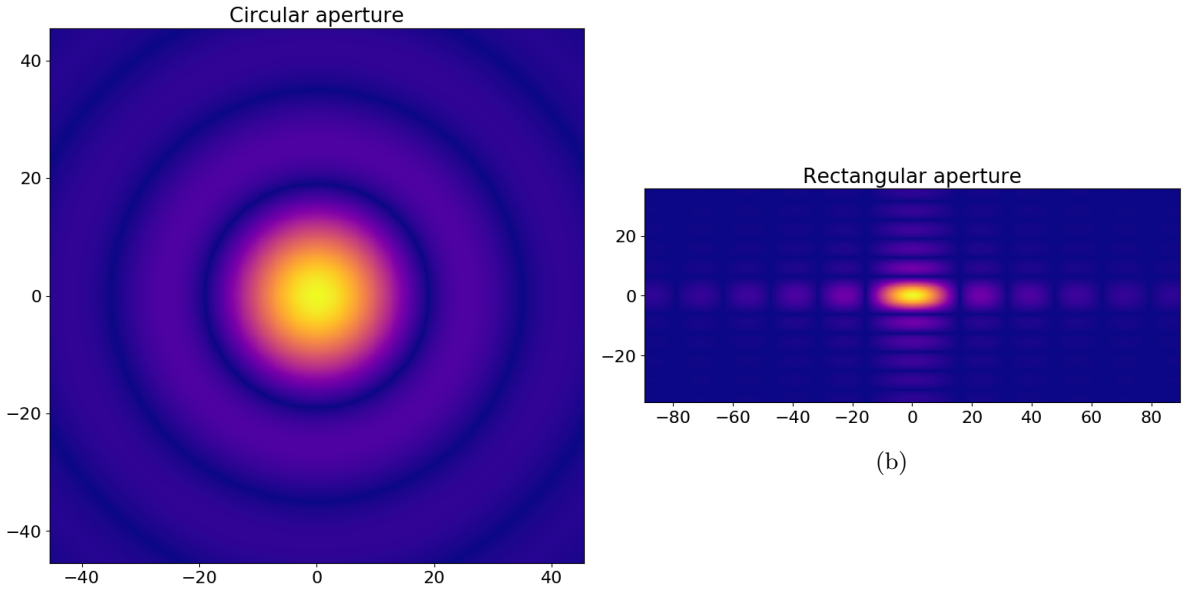
First, I ran the program on series of circular and rectangular apertures described in `config/circular_mins.txt` and `config/rectangle_mins.txt`, expecting to see a linear relationship between the extent of the central maximum and the inverse of the aperture size (see Section 2.2). As seen in Figure 2, these linear relationships hold. The line slopes are close to the expected values of 1.22π for the Airy disc and π for rectangular shapes.

Figure 3 shows that the diffraction patterns look as expected. Most notably, the image from a vertical rectangle is longer along the horizontal axis, as expected from Equation 8.



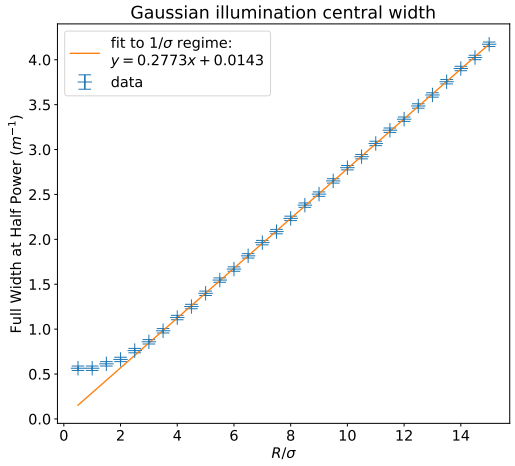
(a) (b)

Figure 2: Sizes of central spots as function of aperture size, for both round and rectangular apertures.

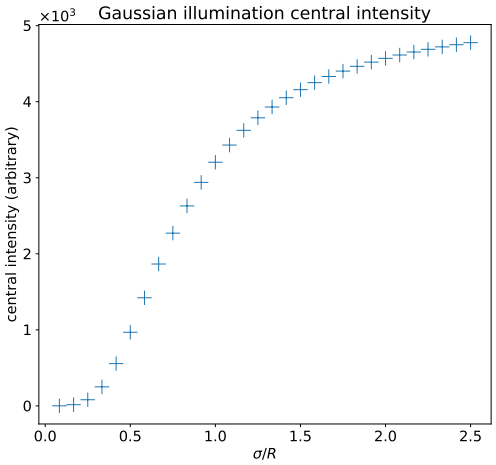


(a) (b)

Figure 3: Typical diffraction patterns of circular and vertical rectangular apertures.

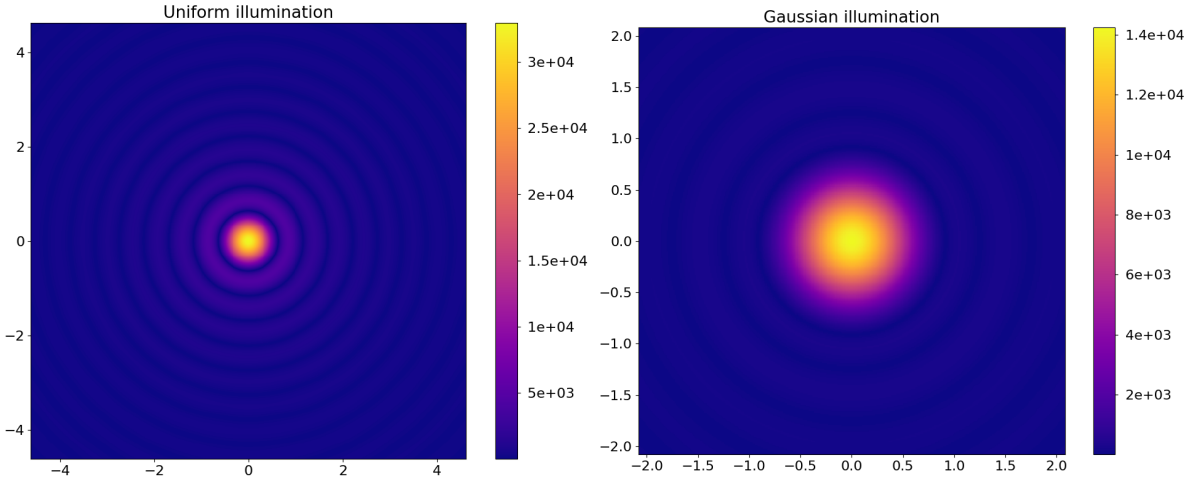


(a) Dependence of full width at half maximum on aperture taper. The fit is for $R/\sigma > 3$.



(b) Central intensity variation with aperture taper.

Figure 4: Properties of Gaussian illumination diffraction pattern.



(a) (b)

Figure 5: Amplitude diffraction patterns of two different illumination patterns of the same mirror of radius $R = 6\,\mathrm{m}$. Note the different scales of the images.

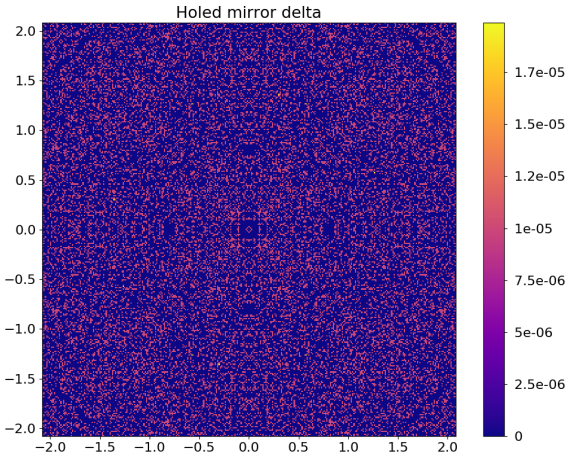


Figure 6: The diffraction pattern of a mirror with a central hole of radius $r = 0.5\,\mathrm{m}$ was subtracted from the difference of diffraction patterns of a complete mirror and just the central hole.

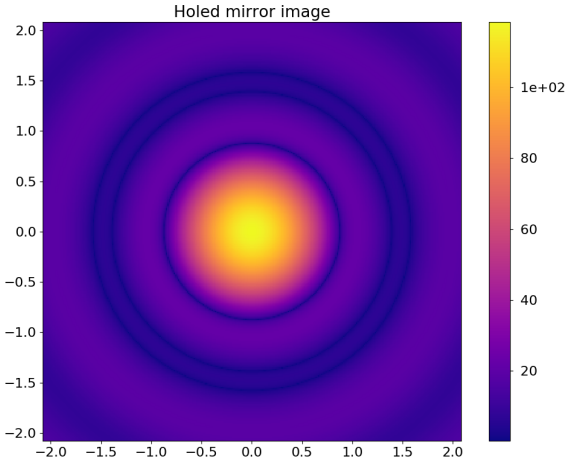


Figure 7: diffraction pattern of holed mirror. The amplitude was raised to the power 1/2 for better visibility.

4.2 Gaussian Illumination

Figure 4 shows the results anticipated in Section 2.3, especially the low and high σ limits. At low σ, the FWHM $\propto 1/\sigma$, and the central intensity tends to zero. At high σ, both properties approach their uniform illumination values:

$$\mathrm{FWHP} \to (0.641 \pm 0.014)\,\mathrm{m}^{-1}$$
$$I(0,0) \to\sim 30 \times 10^3.$$

These results support the expected behaviour of Gaussian-illuminated apertures. Note that the shape in Figure 4b is inherently hard to quantify since it results from the integral of a Gaussian, which has no analytical form.

Figure 5 shows the images produced by illuminating the same mirror uniformly or with a taper. The uniformly lit mirror produces much sharper secondary rings. This suggests that the reason for using a taper is to concentrate more of the power in the central spot, and avoid the effects of the secondary peaks on image quality.

4.3 Central hole

The diffraction pattern of a mirror with a central hole is seen in Figure 7. The expected irregular rings are visible.

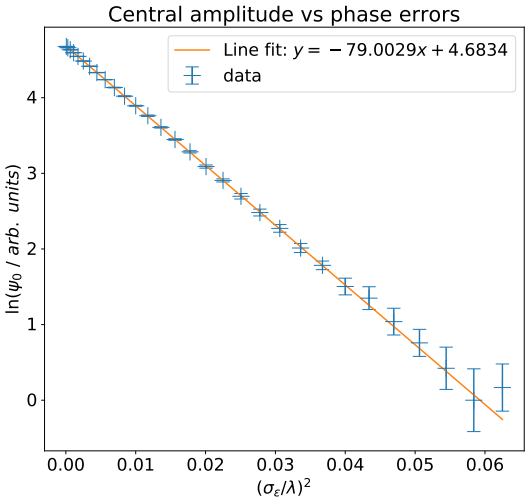
Figure 6 shows the results of the calculation suggested in Section 2.4. We can see the amplitude of the difference reaching at most around 2×10^{-5}. In the original diffraction patterns, the central amplitude is on the order of 10^4, so indeed the two patterns are equal to a very high accuracy. The differences are on the order of the precision to which the numbers were printed before plotting, around 10^{-6}.

This confirms that the F.T. operations we use are linear to a good accuracy.

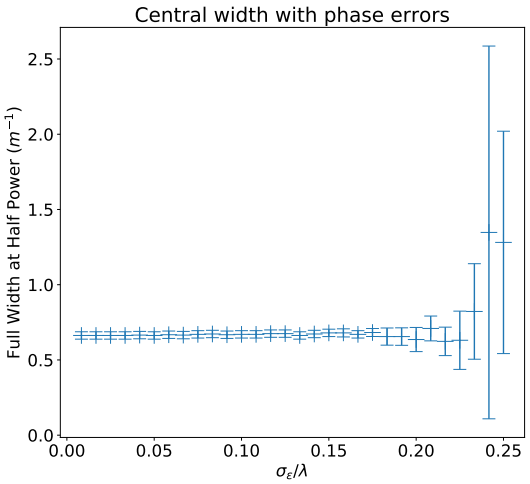
4.4 Random errors

Typical mirror and image shapes for random phase errors are seen in Figure 13.

Figure 8 shows how properties of the image are affected by random errors. The width of the central disc is unaffected, but the central amplitude (ψ_0) decreases with the RMS error σ_ε. When $4\pi\sigma_\varepsilon/\lambda = \sigma_\varphi \to \pi$, the amplitude tends to zero. The linear relationship in Fig. 8a indicates that



(a) dependence of the central amplitude on phase (wavefront) error. Note logarithmic scale of the vertical axis.



(b) (lack of) relationship of full-width-at-half-power with random phase errors.

Figure 8

the amplitude is a Gaussian function of the wavefront error:

$$\psi_0 \propto \exp\left(-\frac{(\sigma_\varepsilon/\lambda)^2}{2\sigma_\psi^2}\right), \tag{13}$$

with

$$\sigma_\psi = (79.56 \pm 0.03) \times 10^{-3} \tag{14}$$

given by the line slope.

The implication for telescopes is that larger wavefront errors will decrease the apparent brightness of the observed objects.

4.5 Error correlation tests

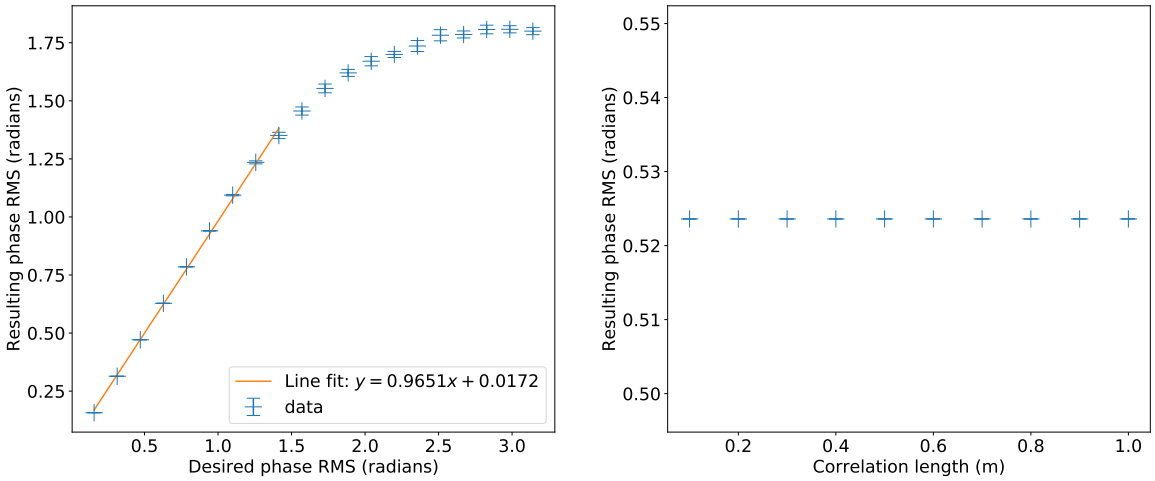
In Figure 9a we see that the produced RMS phase error is close to the desired one for phases of up to $\frac{\pi}{2}$. It saturates to about 1.8 rad at larger desired σ_ε, probably due to some numbers falling outside the $(-\pi,\ \pi)$ range, and overflowing when they are converted to complex phase. This is expected behaviour.

Figure 9b shows that the RMS wavefront error does not depend on correlation length, which confirms normalisation is correct.

4.6 Spatially correlated errors

Defect length l_c	Decay width σ_ψ
0.25 m	$(79.6 \pm 0.2) \times 10^{-3}$
0.50 m	$(79.8 \pm 0.3) \times 10^{-3}$
0.75 m	$(80.0 \pm 0.8) \times 10^{-3}$
1.00 m	$(81.6 \pm 0.9) \times 10^{-3}$
1.25 m	$(83 \pm 1) \times 10^{-3}$

Table 1: decay widths of on-axis amplitude with wavefront error.



(a) (b)

Figure 9: dependence of RMS (root mean square) phase on desired RMS and correlation length.

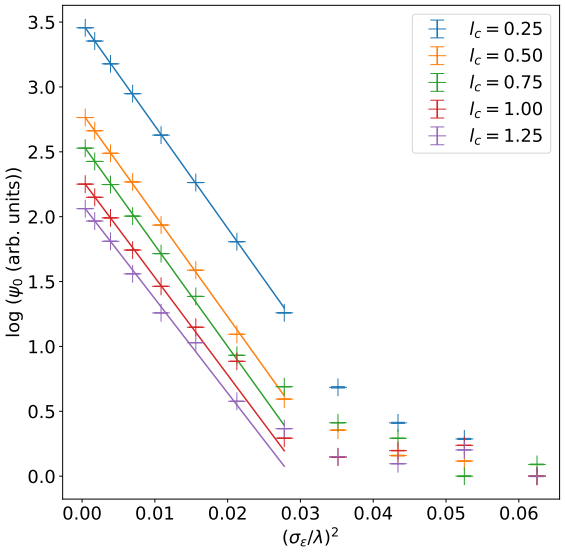


Figure 10: dependence of central amplitude on the RMS of correlated phase errors, for different defect lengths (in meters). Values are normalised to the smallest in the series.

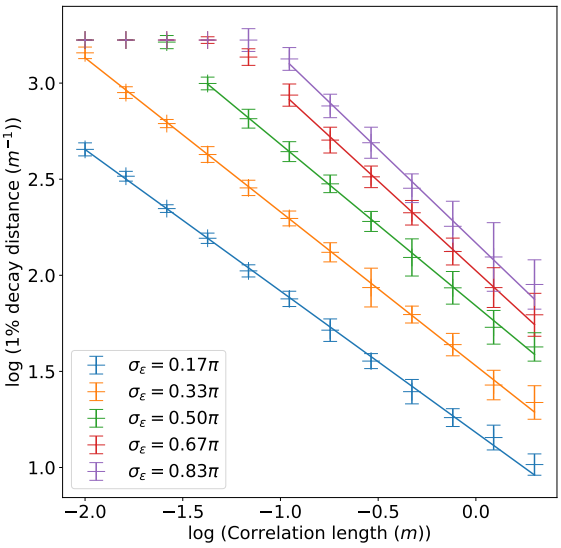


Figure 11: farthest distance from the centre where an amplitude at least 1 % of the central one is found, as a function of error correlation length (or defect 'width'). Error bars are the standard deviation of 20 runs. Three of the series are seen to saturate at the same value, reaching the maximum extent of the discrete Fourier transform.

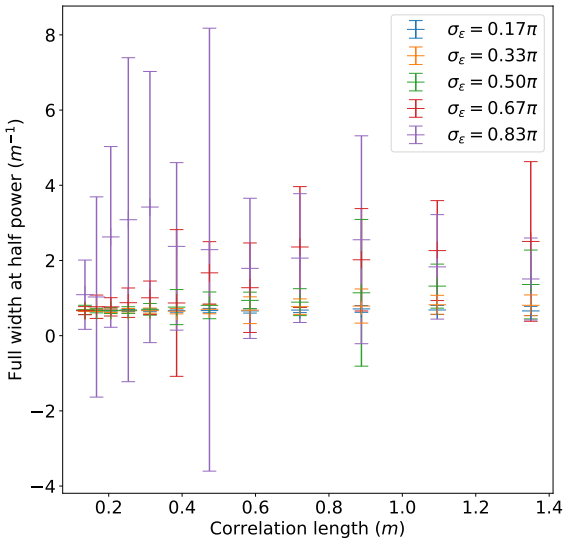


Figure 12: the full width at half power as a function of correlation length, for different values of the RMS wavefront error. Error bars are the standard deviation of 20 runs.

RMS phase ($4\pi\sigma_\varepsilon/\lambda$)	exponent (n)
$\pi/6$ rad	0.736 ± 0.008
$\pi/3$ rad	0.801 ± 0.008
$\pi/2$ rad	0.84 ± 0.01
$2\pi/3$ rad	0.93 ± 0.02
$5\pi/6$ rad	0.98 ± 0.04

Table 2: exponents in the relation of 1 % decay distance with defect correlation length.

Finally, we explore the effects of finite-width defects in the mirror surface. Figure 14 shows some typical diffraction patterns at different defect lengths.

Figure 10 shows that the exponential relationship in Equation 13 does not hold for the entire range $4\pi\sigma_\varepsilon/\lambda \in (0, \pi)$, but rather only to around $3\pi/4$. The decay rates σ_ψ of amplitude with wavefront error are in Table 1. At small correlation lengths, these are similar to the one found in Section 4.4, which is good limiting behaviour.

Another interesting relation emerges: I measured the maximum distance from the centre it takes the amplitude to decay to 1 % of the maximum value. The results in Figure 11 show that this decay distance ($p_{1/100}$) decreases with the defect width, and increases with the wavefront error, i.e. with the depth. The fitted lines show that:

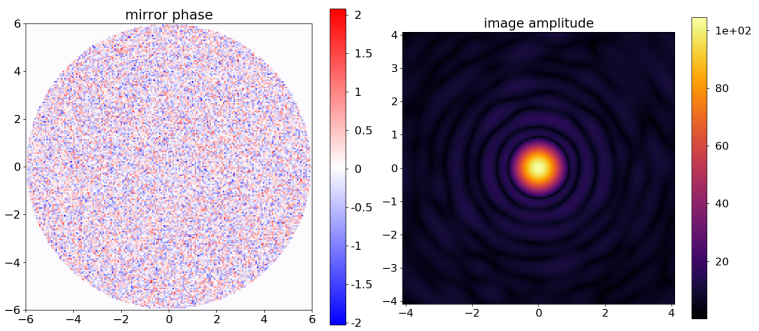
$$p_{1/100} \propto l_c^{-n}, \quad \text{with} \quad n \in (2/3, 1). \tag{15}$$

The calculated values of the exponent n are in Table 2.

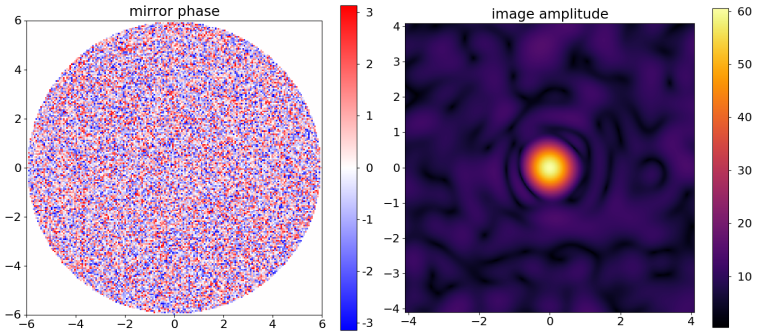
For telescope manufacturing, the implication is that, at the same wavefront precision, many narrow defects will spread energy farther from the centre of the diffraction pattern compared to fewer wider defects. In the light of the inverse relationship between the sizes of mirror shapes and their Fourier Transforms, this makes intuitive sense.

Interestingly, the full width at half-power of the central diffraction peak was found to be unaffected by the correlation length, at least in the range of lengths studied (see Figure 12). A possible explanation is that, since the typical size of the defects is smaller than the mirror size ($l_c \in 0.1\,\text{m}$ to $1.5\,\text{m}$, while $R = 6\,\text{m}$), their effects on the image are on a larger length scale than the central maximum.

The on-axis amplitude was also found to not vary with the defect size. If the normalisation in Figure 10 is removed, all the data series approximately coincide. Again, the intuitive explanation would be that, since ψ_0 is the integral of the aperture function (i.e. in Equation 2 the exponent is 0), the spatial distribution of complex phase is not important.

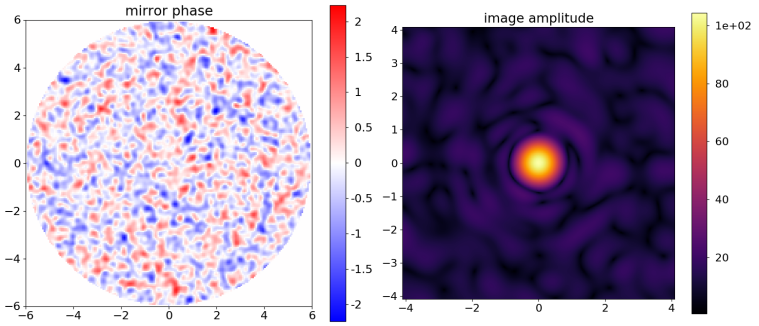


(a) $\sigma_\varepsilon = \pi/6$

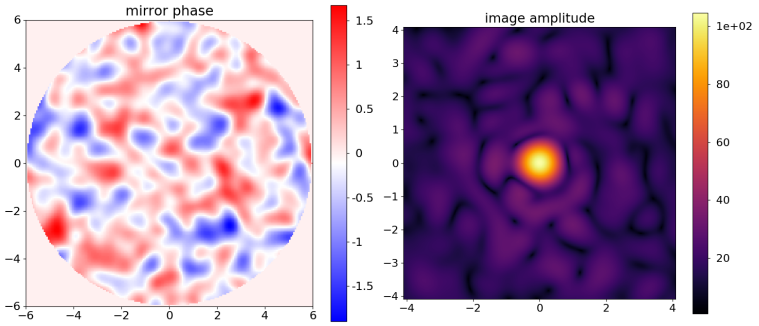


(b) $\sigma_\varepsilon = \pi/2$

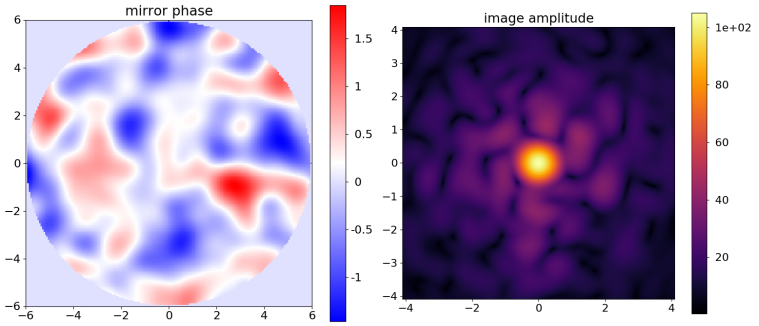
Figure 13: Typical random error aperture functions and resulting diffraction patterns. Amplitudes are raised to the power 1/2 to make features easier to see.



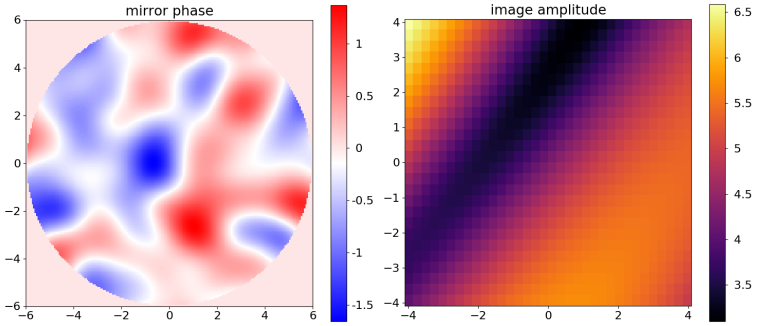
(a) $\sigma_\varepsilon = \pi/6, \quad l_c = 0.1\,\mathrm{m}$



(b) $\sigma_\varepsilon = \pi/6, \quad l_c = 0.3\,\mathrm{m}$



(c) $\sigma_\varepsilon = \pi/6, \quad l_c = 0.5\,\mathrm{m}$



(d) $\sigma_\varepsilon = \pi/6, \quad l_c = 0.8\,\mathrm{m}$

Figure 14: Typical correlated error aperture functions and resulting diffraction patterns. Amplitudes are raised to the power 1/2 to make features easier to see. Note that at larger correlation length, the central peak is absent.

5 Conclusion

I performed a computational study of the effects of wavefront error, such as those produced by deviations from the parabolic shape, on the images produced by telescope mirrors. A program was written in C++ to calculate the far-field diffraction pattern of a general mirror shape using a Fast Fourier Transform algorithm. It was tested with known mirror shapes and found to perform as expected.

The effect of gaussian-tapered illumination on the diffraction pattern was investigated. It was found to slightly widen the central maximum and decrease the peak intensity, but greatly reduce the sharpness of secondary rings found in a typical Airy pattern. A central hole, such as the one due to a secondary mirror, was found to have minimal effects on the beam.

The effects of two types of wavefront error were studied: random and spatially correlated. In both cases, an increase in the typical depth of the defects, i.e. the RMS phase error, was found to decrease the central amplitude of the diffraction pattern, thus decreasing the apparent brightness of objects seen through the telescope.

The spatial distribution of errors was seen to affect the spatial distribution of fluctuations in the image, but not its brightness or the width of the central maximum. For wider deformations, the fluctuations were concentrated closer to the central peak, whereas for denser and smaller deformations the fluctuations were spread out farther in the image plane. Thus, small and dense deformations have worse effects on the telescope's resolution.

Word count: 2949

References

[1] David Buscher. *Part II Computational Physics Projects*. Cavendish Laboratory, University of Cambridge. Lent 2019. URL: `https://www-teach.phy.cam.ac.uk/dms/dms_getFile.php?node=20927` (visited on 03/04/2019).

[2] Matteo Frigo and Steven G. Johnson. *FFTW. (version 3.3.8)*. [library documentation]. Massachusetts Institute of Technology. May 2018. URL: `http://www.fftw.org/fftw3.pdf` (visited on 04/04/2019).

[3] Mark Galassi et al. *GNU Scientific Library Reference Manual*. June 2018. ISBN: 0954612078. URL: `https://www.gnu.org/software/gsl/` (visited on 29/04/2019).

[4] Eugene Hecht. *Optics*. 5th edition. Pearson Education, 2017. ISBN: 978-1-292-09693-3.

[5] William H. Press et al. *Numerical Recipes. The Art of Scientific Computing*. 3rd edition. Cambridge University Press, 2007. ISBN: 9780521880688.

[6] *std::complex*. cppreference. URL: `https://en.cppreference.com/w/cpp/numeric/complex` (visited on 26/04/2019).

[7] Thomas L. Wilson, Kristen Rohlfs and Susanne Hüttemeister. *Tools of Radio Astronomy*. 5th edition. Berlin Heidelberg: Springer, 2009. ISBN: 978-3-540-85121-9.

A Code listing

A.1 Numerical routines in C++

A.1.1 src/cpp/main.cpp

```cpp
#include <cstdio>
#include <cmath>
#include <complex>
#include <queue>
#include <thread>

#include <gsl/gsl_sf_trig.h>
#include <gsl/gsl_math.h>
#include <fftw3.h>

#include "array2d.h"
#include "util.h"

using namespace std;

#define N_THREADS 1
#define N_WORKERS 2

#define INFO_OUT true

/** This block defines a struct that has an index and a string;
 * It's meant to hold a line of data to be printed to the data file.
 * The comparator is used to sort data lines by the index in the priority queue, so that
 * if we use multiple workers, the data lines are still printed in order when
 * main() does the dequeueing.
 */
struct DataLine{
    unsigned int idx;
    string line;
};

auto dlcmp = [](DataLine a, DataLine b) { return a.idx > b.idx; };
priority_queue<DataLine, vector<DataLine>, decltype(dlcmp)> dataq(dlcmp);


/** Process the shapes between start and end (inclusive, exclusive) in the config.
 * n_proc is the processor number, used in the logger name for debugging
 * Push the data results to the data queue; Array printing is handled here;
 */
void shapes_worker(const Config& conf, unsigned int n_proc, unsigned int start, unsigned int end) {
    // init a logger for each processor
    string logname = "work_" + to_string(n_proc);
    Logger proc_log(stdout, logname.c_str(), INFO_OUT);

    proc_log("Started on shapes " + to_string(start) + " to " + to_string(end));

    // declarations
    Array2d in(conf.nx, conf.ny);
    Array2d out(conf.nx, conf.ny);
    fftw_plan plan;

    // plan
    planner_mtx.lock();
    proc_log("Locked. Planning...");
    plan = fftw_plan_dft_2d(conf.nx, conf.ny, in.ptr(), out.ptr(), FFTW_FORWARD, FFTW_MEASURE);
    planner_mtx.unlock();
    proc_log("Unlocked. Planning done.");

    for(unsigned int shape_idx = start; shape_idx < end; shape_idx ++ ) {
        proc_log("===== Shape " + to_string(shape_idx) + " =====");
        ShapeProperties sp = conf.shapes[shape_idx];

        // construct new data line
        DataLine dl{shape_idx, to_string(shape_idx)};

        // calculate x and y values for both in and out
        // the division by 2pi is because p and q are angular frequencies,
        // whereas the FFT produces number frequencies
        vector<double> xs = coords(sp.lx, conf.nx);
```

```
70          vector<double> ys = coords(sp.ly, conf.ny);
71          vector<double> ps = fftfreq(conf.nx, sp.lx/(double)conf.nx/(2*M_PI));
72          vector<double> qs = fftfreq(conf.ny, sp.ly/(double)conf.ny/(2*M_PI));
73          // the printing boundaries of the arrays
74          Limits in_lims, out_lims;
75
76          // fill in the input
77          proc_log("Initializing input...");
78          generators[sp.generator_key](in, xs, ys, sp.shape_params);
79
80          proc_log("Executing...");
81          fftw_execute(plan);
82
83          proc_log("Resolving tasks:");
84          if(contains(conf.tasks, "params")) {
85              // print shape parameters
86              for(unsigned int ip = 0; ip < sp.shape_params.size(); ip ++ )
87                  dl.line += "\t" + to_string(sp.shape_params[ip]);
88          }
89          if(contains(conf.tasks, "find_min")) {
90              // print size of central spot and error
91              ValueError<double> min_pos = find_first_min(myabs, out, ps);
92              dl.line += "\t" + to_string(min_pos.val) + "\t" + to_string(min_pos.err);
93          }
94          if(contains(conf.tasks, "fwhp")) {
95              // print coordinate of full-width at half-power along horizontal.
96              // times by 2 for FULL width (function gives half width)
97              ValueError<double> res = hwhp(out, ps);
98              dl.line += "\t" + to_string(res.val * 2) + "\t" + to_string(res.err * 2);
99          }
100         if(contains(conf.tasks, "fwhp_y")) {
101             // print coordinate of FWHP along vertical
102             ValueError<double> res = hwhp(out, qs, true);
103             dl.line += "\t" + to_string(res.val * 2) + "\t" + to_string(res.err * 2);
104         }
105         if(contains(conf.tasks, "central_amplitude")) {
106             // print absolute value of central spot
107             dl.line += "\t" + to_string(myabs(out(0, 0)));
108         }
109         if(contains(conf.tasks, "in_phase_stat")) {
110             // print the mean and RMS of phase errors in input array
111             ValueError<double> stat = mean_stddev(myarg, in, xs, ys, sp.shape_params[0]);
112             dl.line += "\t" + to_string(stat.val) + "\t" + to_string(stat.err);
113         }
114
115         // find interesting limits if printing is needed. This next bit is ugly, I know.
116         if(any_begins_with(conf.tasks, "print_in")) {
117             // look for in limits
118             proc_log("\tin limits");
119             in_lims = in.find_interesting(myabs, conf.abs_sens, conf.rel_sens);
120         }
121
122         if(any_begins_with(conf.tasks, "print_out") || contains(conf.tasks, "out_lims")) {
123             // this screws up out
124             proc_log("\tfftshift(out)");
125             fftshift(out);
126             ps = fftshift(ps); qs = fftshift(qs);
127
128             // look for out limits
129             proc_log("\tout limits");
130             out_lims = out.find_interesting(myabs, conf.abs_sens, conf.rel_sens);
131         }
132
133         if(contains(conf.tasks, "out_lims")) {
134             // record the boundaries of the image that are above the given sensitivity
135             // reminder: lims = {imin, imax, jmin, jmax}
136             int imin = out_lims[0], imax = out_lims[1];
137             int jmin = out_lims[2], jmax = out_lims[3];
138             double p1 = ps[jmin], p2 = ps[jmax - 1];
139             double q1 = qs[imin], q2 = qs[imax - 1];
140
141             dl.line += "\t" + to_string(p1) + "\t" + to_string(p2);
142             dl.line += "\t" + to_string(q1) + "\t" + to_string(q2);
143         }
144
```

```
145         // DO the array printing.
146         if(contains(conf.tasks, "print_in_abs")) {
147             proc_log("\tprint_in_abs");
148             // print aperture amplitude
149             string in_fname = conf.out_prefix + to_string(shape_idx) + "in_abs.txt";
150             FILE * in_filep = fopen(in_fname.c_str(), "w");
151             print_lim_array(in_filep, myabs, in, xs, ys, in_lims);
152             fclose(in_filep);
153         }
154         if(contains(conf.tasks, "print_in_phase")) {
155             proc_log("\tprint_in_phase");
156             // print aperture phase
157             string in_fname = conf.out_prefix + to_string(shape_idx) + "in_phase.txt";
158             FILE * in_filep = fopen(in_fname.c_str(), "w");
159             print_lim_array(in_filep, myarg, in, xs, ys, in_lims);
160             fclose(in_filep);
161         }
162         if(contains(conf.tasks, "print_out_abs")) {
163             proc_log("\tprint_out_abs");
164             // print image amplitude
165             string out_fname = conf.out_prefix + to_string(shape_idx) + "out_abs.txt";
166             FILE * out_filep = fopen(out_fname.c_str(), "w");
167             print_lim_array(out_filep, myabs, out, ps, qs, out_lims);
168             fclose(out_filep);
169         }
170         if(contains(conf.tasks, "print_out_phase")) {
171             proc_log("\tprint_out_phase");
172             // print image phase
173             string out_fname = conf.out_prefix + to_string(shape_idx) + "out_phase.txt";
174             FILE * out_filep = fopen(out_fname.c_str(), "w");
175             print_lim_array(out_filep, myarg, out, ps, qs, out_lims);
176             fclose(out_filep);
177         }
178         dataq.push(dl);
179     }
180 
181     proc_log("Done. Cleaning up...");
182     fftw_destroy_plan(plan);
183 }
184 
185 
186 int main(int argc, char * argv[]) {
187     // open logger
188     Logger main_log(stdout, "main.cpp", INFO_OUT);
189 
190     if(argc != 2) {
191         main_log("Incorrect number of arguments. Provide one config file.");
192         return 1;
193     }
194 
195     // init fftw threads
196     int threads_status = fftw_init_threads();
197     if(threads_status == 0) {
198         main_log("Thread initialisation failed!");
199         return 1;
200     }
201     fftw_plan_with_nthreads(N_THREADS);
202     main_log("Thread initialisation successful.");
203 
204     // parse command line config
205     Config conf(argv[1]);
206     main_log("Configured");
207 
208     // Multithread the shape processing
209     unsigned int shapes_per_thread = conf.shapes.size() / N_WORKERS;
210     vector<thread> worker_threads;
211 
212     main_log("Spawning worker threads");
213     for(unsigned int i_th = 0; i_th < N_WORKERS; i_th ++ ) {
214         unsigned int start = i_th * shapes_per_thread;
215         unsigned int end = (i_th + 1) * shapes_per_thread;
216         // the last worker has to finish the shapes
217         if(i_th == N_WORKERS - 1) end = conf.shapes.size();
218 
219         if(start < end)
```

```
            // only start workers if they have something to do
            worker_threads.push_back(thread(shapes_worker, conf, i_th, start, end));
    }

    // join everything when it's done
    for(vector<thread>::iterator th = worker_threads.begin(); th != worker_threads.end(); th++ )
        th->join();

    main_log("Writing data results");
    // open data file;
    string data_filename = conf.out_prefix + "dat.txt";
    FILE * data_filep = fopen(data_filename.c_str(), "w");

    // print the data
    DataLine dl;
    while(!dataq.empty()) {
        dl = dataq.top();
        fprintf(data_filep, "%s\n", dl.line.c_str());
        dataq.pop();
    }
    fclose(data_filep);

    main_log("Done. Exiting.");
    return 0;
}
```

A.1.2 src/cpp/util.h

```
#ifndef MYUTIL
#define MYUTIL

#include<cstdio>
#include<cstring>
#include<string>
#include<vector>
#include<map>
#include<algorithm>
#include<complex>
#include<mutex>

using namespace std;

#define EPS 1e-6
#define DBL_EQ(a, b) (abs(a-b) < EPS)

#define CONV_KEY "corr_errors"

// a mutex locks thread execution to only allow one thread at a time to access a resource
// here it's needed because FFTW only allows one thread to plan FFTs at a time
extern mutex planner_mtx;

/** Type that takes complex argument and returns real number.
 * These are functions such as abs, real, imag, arg, etc...
 */
using complex_to_real = double (*)(complex<double>);
extern complex_to_real myabs, myarg, myre, complexness;

/** Find if s exists in a vector of strings */
bool contains(const vector<string> &v, const char * s);
/** Find if any string in a vector begins with s */
bool any_begins_with(const vector<string> &v, const char * s);

/** Holds the properties of an aperture shape.
 * lx, ly are the lengths of the sides of the board
 * shape holds all the numbers necessary to make the shape, and it's passed
 * to the generator function.
 * generator_key is a key in the name-function map of aperture generators
 */
struct ShapeProperties {
    string generator_key;
    double lx, ly;
    vector<double> shape_params;
};


```

```
48  /**
49   * Struct containing the configuration of the program.
50   * nx and ny are the dimensions of the arrays used
51   * tasks is the list of things to do with each shape
52   * out_prefix is a prefix for the files where to print data
53   * shapes is a vector of shapes to process
54   * abs_sens and rel_sens are the sensitivities at printing. Use 0 to print everything.
55   * convolution is a flag describing whether a convolution in the input array is needed. If yes, we'll need a second FFT
     ↪  plan for transforming backwards, because convolution is done by multiplying the FFT results.
56   */
57  struct Config {
58      Config(const char * filename);
59
60      string out_prefix;
61      vector<string> tasks;
62      vector<ShapeProperties> shapes;
63
64      int nx, ny;
65      double abs_sens, rel_sens;
66  };
67
68
69  /**
70   * My own utility logger, than can be turned on or off
71   * whenever. It writes to the given file pointer, which can
72   * be stdout as well, and prepends its name.
73   */
74  struct Logger {
75      FILE * filep;
76      string name;
77      bool enabled;
78      Logger(FILE * filep, const char * name, bool enabled);
79      void write(const char * message) const;
80      void operator()(const char * message) const;
81      void operator()(const string message) const;
82  };
83
84
85  template <typename T>
86  struct ValueError {
87      T val;
88      T err;
89  };
90
91
92  /**
93   * Generate the FT frequencies corresponding to n time-samples spaced by dt.
94   * For even n, the positive frequencies are [1 .. n/2 - 1] / (n*dt)
95   * For odd n, the positive frequencies are [1 .. (n-1)/2] / (n*dt)
96   */
97  vector<double> fftfreq(int n, double dt=1.0);
98
99  /**
100  * Create new vector where the zero-component freuqency is shifted to the middle
101  */
102 template <typename T> vector<T> fftshift(const vector<T> &v) {
103     int n = v.size();
104     vector<T> shifted(v);
105     int new_first = (n+1) / 2;
106
107     rotate(shifted.begin(), shifted.begin() + new_first, shifted.end());
108
109     return shifted;
110 }
111
112
113 /**
114  * Calculate the coordinates of n evenly distributed points between -l/2 and l/2
115  */
116 vector<double> coords(double l, int n);
117
118 #endif
```

A.1.3 src/cpp/util.cpp

```cpp
#include "util.h"

// define the mutex
mutex planner_mtx;

complex_to_real myabs = [](complex<double> z) -> double {return abs(z);};
complex_to_real myarg = [](complex<double> z) -> double {return arg(z);};
complex_to_real myre = [](complex<double> z) -> double {return real(z);};
complex_to_real complexness = [](complex<double> z) -> double {return abs(imag(z)/real(z));};


bool contains(const vector<string> &v, const char * s) {
    string str = string(s);
    return any_of(v.begin(), v.end(), [&](string element){ return element == s; });
}

bool any_begins_with(const vector<string> &v, const char * s) {
    string str = string(s);
    return any_of(v.begin(), v.end(), [&](string element){ return element.find(s) == 0; });
}


#define OPTION_ERROR "Incorrect option. Expected %s, got %s"

inline void option_error(const char * optname, const char * readname) {
    char msg[100];
    sprintf(msg, OPTION_ERROR, optname, readname);
    throw runtime_error(msg);
}

/** Overloaded utility function used to parse one line of the the config file
 * It expects a format like
 * `name = value`
 * where `name` has to match `optname` exactly. Only if that happens, `value` is
 * written to `option`.
 */
void read_option(FILE * filep, const char * optname, int &option) {
    char readname[64];
    int readval;
    fscanf(filep, " %s = %d ", readname, &readval);

    if(strcmp(readname, optname) != 0) option_error(optname, readname);
    option = readval;
}

void read_option(FILE * filep, const char * optname, double &option) {
    char readname[64];
    double readval;
    fscanf(filep, " %s = %lf ", readname, &readval);

    if(strcmp(readname, optname) != 0) option_error(optname, readname);
    option = readval;
}

void read_option(FILE * filep, const char * optname, string &option) {
    char readname[64];
    char readval[64];
    fscanf(filep, " %s = %s ", readname, readval);

    if(strcmp(readname, optname) != 0) option_error(optname, readname);
    option = string(readval);
}

void read_option(FILE * filep, const char * optname, vector<double> &option) {
    char readname[64], delim[2];
    double readval;
    fscanf(filep, " %s = ", readname);

    if(strcmp(readname, optname) != 0) option_error(optname, readname);

    while(fscanf(filep, " %lf", &readval) == 1) {
        option.push_back(readval);
        // consume terminating newline
```

```
74          if(fscanf(filep, "%1[\n]", delim) == 1) break;
75      }
76  }
77
78  void read_option(FILE * filep, const char * optname, vector<string> &option) {
79      char readname[64], delim[2];
80      char readval[64];
81      fscanf(filep, " %s = ", readname);
82
83      if(strcmp(readname, optname) != 0) option_error(optname, readname);
84
85      while(fscanf(filep, " %s", readval) == 1) {
86          option.push_back(string(readval));
87          // consume the terminating newline
88          if(fscanf(filep, "%1[\n]", delim) == 1) break;
89      }
90  }
91
92  /**
93   * Parse configuration file `filename` and construct the Config object.
94   */
95  Config::Config(const char * filename) {
96      string cnf_filename = filename;
97
98      int n_shapes = 0;
99
100     FILE * cnf_filep = fopen(cnf_filename.c_str(), "r");
101     if(cnf_filep == NULL) perror("Could not open config file.");
102
103     read_option(cnf_filep, "nx", nx);
104     read_option(cnf_filep, "ny", ny);
105     read_option(cnf_filep, "prefix", out_prefix);
106     read_option(cnf_filep, "tasks", tasks);
107     read_option(cnf_filep, "rel_sens", rel_sens);
108     read_option(cnf_filep, "abs_sens", abs_sens);
109     read_option(cnf_filep, "n_shapes", n_shapes);
110
111     for(int i = 0; i < n_shapes; i ++ ) {
112         ShapeProperties sp;
113         read_option(cnf_filep, "type", sp.generator_key);
114         read_option(cnf_filep, "lx", sp.lx);
115         read_option(cnf_filep, "ly", sp.ly);
116         read_option(cnf_filep, "params", sp.shape_params);
117
118         shapes.push_back(sp);
119     }
120 }
121
122 /** Construct logger that writes to file pointer filep,
123  *  prepending name given, only if enabled == true
124  */
125 Logger::Logger(FILE * filep, const char * name, bool enabled) :
126     filep(filep),
127     name(name),
128     enabled(enabled) {}
129
130 /* Write logger name + given message to file pointer, only if enabled */
131 void Logger::write(const char * message) const {
132     if(enabled) {
133         fprintf(filep, "%s: %s\n", name.c_str(), message);
134     }
135 }
136
137 /* Same as calling write */
138 void Logger::operator()(const char * message) const {
139     (*this).write(message);
140 }
141
142 void Logger::operator()(const string message) const {
143     (*this).write(message.c_str());
144 }
145
146
147 vector<double> fftfreq(int n, double dt) {
148     vector<double> v(n, 0.0);
```

```
    int halfpoint = (n+1)/2;

    for(int i = 0; i < halfpoint; i++)
        v[i] = i / (dt*n);

    for(int i = halfpoint; i < n; i ++)
        v[i] = (i - n) / (dt*n);

    return v;
}


/**
 * Calculate the coordinates of n evenly distributed points between -l/2 and l/2
 */
vector<double> coords(double l, int n) {
    vector<double> x(n);
    for(int i = 0; i < n; i ++ )
        x[i] = (i - n/2) * l/n;

    return x;
}
```

A.1.4 src/cpp/array2d.h

```
#ifndef ARRAY2D
#define ARRAY2D

#include <cstdio>
#include <complex>
#include <vector>
#include <array>
#include <string>

#include <fftw3.h>

#include "util.h"
using namespace std;

#define PRINT_FORMAT "% 6.5f\t"

// type containing i and j limits of interest in an Array2d
using Limits = array<int, 4>;
string lims_to_str(const Limits &l);

/** THE class that stores a 2D nx by ny array of complex<double> numbers
 * internally represented as a 1D array of length (nx*ny). It offers access
 * to elements in mutable and immutable ways, (approximate) equality comparison.
 *
 * NB it doesn't follow the rule of 3 for classes having pointer members.
 * This means that the (compiler-generated) copy constructor will not deep-copy
 * the data stored within, but rather just copy the pointer arr. This is done
 * on purpose to save time and memory when deep-copying isn't necessary.
 */
class Array2d {
private:
    int nx, ny;
    complex<double> * arr;

public:
    Array2d(int size_x, int size_y);
    ~Array2d();

    complex<double> * operator[](int ix);
    complex<double> operator()(int ix, int iy) const;
    int mult(const Array2d& a);
    int mult_each(complex<double> c);
    int divide_each(complex<double> c);
    friend bool operator==(const Array2d &a, const Array2d &b);

    fftw_complex * ptr();
    Limits find_interesting(complex_to_real fun, double abs_sens, double rel_sens) const;
    void print_prop(complex_to_real fun, FILE * out_file) const;
    void print_prop(complex_to_real fun, const Limits &lim, FILE * out_file) const;

```

```
    int copy_into(Array2d &a) const;

    friend void fftshift(Array2d &a);
};

/**
 * Find the first minimum of fun(z) along the horizontal axis of a
 * and the associated error
 */
ValueError<double> find_first_min(complex_to_real fun, const Array2d &a, const vector<double> &xs);

/**
 * Find the x-coordinate of the first half-power point and the error
 */
ValueError<double> hwhp(const Array2d &a, const vector<double> &coord, bool vertical = false);

/**
 * Calculate the mean and standard deviation of fun within a given radius
 */
ValueError<double> mean_stddev(complex_to_real fun, const Array2d &a, const vector<double>& xs, const vector<double>& ys,
↪  double radius);

/**
 * Print the limits in two dihections of the 2d array, then the array itself,
 * in standard formatted way.
 * Only print stuff within x and y limits given by lims.
 */
void print_lim_array(FILE * filep, complex_to_real fun, const Array2d &a, const vector<double> &xs, const vector<double>
↪  &ys, const Limits &lims);

/**
 * Type of function that writes and aperture an aperture given
 * the list of x and y coordinates and a vector of parameters.
 */
using aperture_generator = int (*)(Array2d& arr, const vector<double>& xs, const vector<double>& ys, const
↪  vector<double>& params);

/** map the name found in config files to the actual function pointer
 * for dynamically choosing which functions to run
 */
extern map<string, aperture_generator> generators;

#endif
```

A.1.5 src/cpp/array2d.cpp

```
#include "array2d.h"

#include<gsl/gsl_rstat.h>

#define DEBUG_OUT false
Logger arr2dlog(stdout, "arr2d", DEBUG_OUT);

string lims_to_str(const Limits &l) {
    char buff[100];
    sprintf(buff, "%d\t%d\t%d\t%d", l[0], l[1], l[2], l[3]);
    return string(buff);
}

Array2d::Array2d(int size_x, int size_y) : nx(size_x), ny(size_y) {
    // log
    char msg[64];
    sprintf(msg, "constructed array %d x %d", nx, ny);
    arr2dlog(msg);

    // allocate memory
    arr = (complex<double>*) fftw_alloc_complex(nx * ny);
}

Array2d::~Array2d() {
    // log
    char msg[64];
    sprintf(msg, "destructed array %d x %d", nx, ny);
    arr2dlog(msg);

```

```
30      // free memory
31      fftw_free(ptr());
32  }
33
34  /** Return the value of element [ix][iy] through round bracket operator
35   * Use like a(ix, iy). Return is immutable, good for use with const Array2d &.
36   */
37  complex<double> Array2d::operator()(int ix, int iy) const {
38      int idx = ny*ix + iy;
39      return arr[idx];
40  }
41
42  /** Return a pointer to row number ix
43   * Use like a[ix][iy] for value of element, like a normal 2d array.
44   * Return is mutable.
45   */
46  complex<double> * Array2d::operator[](int ix) {
47      return (arr + (ny*ix));
48  }
49
50  /** Multiply the first array with the second element-wise,
51   * storing results in the first.
52   * Returns 0 if succesful or something else if failed.
53   */
54  int Array2d::mult(const Array2d& a) {
55      if(nx != a.nx) return -1;
56      if(ny != a.ny) return -1;
57
58      for(int i = 0; i < nx; i ++ )
59          for(int j = 0; j < ny; j ++ )
60              (*this)[i][j] *= a(i, j);
61      return 0;
62  }
63
64  /** Multiply every element in the array by a scalar c */
65  int Array2d::mult_each(complex<double> c) {
66      for(int i = 0; i < nx; i ++ )
67          for(int j = 0; j < ny; j ++ )
68              (*this)[i][j] *= c;
69      return 0;
70  }
71
72  /** Divide every element in the array by a scalar c */
73  int Array2d::divide_each(complex<double> c) {
74      return this->mult_each(1.0 / c);
75  }
76
77  /** Test for near equality to within EPS */
78  bool operator==(const Array2d &a, const Array2d &b) {
79      if(a.nx != b.nx) return false;
80      if(a.ny != b.ny) return false;
81
82      int nx = a.nx, ny = a.ny;
83
84      for(int i = 0; i < nx; i ++ )
85          for(int j = 0; j < ny; j ++ ) {
86              if( ! DBL_EQ(a(i, j), b(i, j)))
87                  return false;
88          }
89      return true;
90  }
91
92  /** Cast the pointer to fftw_complex*,
93   * such that it can be used with fftw library
94   */
95  fftw_complex * Array2d::ptr() {
96      return (fftw_complex*) arr;
97  }
98
99
100 /** Find the x and y bounds within which the absolute value of property fun
101  *  is greater than some fraction (rel_sens) of the maximum value
102  *  OR just greater than abs_sens.
103  *
104  *  To ignore one of absolute or relative sensitivities, set them to 0.
```

```
 */
Limits Array2d::find_interesting(complex_to_real fun, double abs_sens, double rel_sens) const {
    // check if caller wants to ignore one criterion
    if(abs_sens == 0.0) abs_sens = INFINITY; // nothing is greater than inf
    if(rel_sens == 0.0) rel_sens = 2.0;      // nothing is greater than 2*max

    int imin = nx, imax = 0, jmin = ny, jmax = 0;
    double fun_max = 0.0, abs_here;

    // walk the array once and find max abs value of fun
    for(int i = 0; i < nx; i ++ ) {
        for(int j = 0; j < ny; j ++ ) {
            abs_here = abs(fun((*this)(i, j)));
            if(abs_here > fun_max)
                fun_max = abs_here;
        }
    }

    // walk again and record where abs value of fun
    // is greater than fraction of maximum found earlier
    for(int i = 0; i < nx; i ++ ) {
        for(int j = 0; j < ny; j ++ ) {
            abs_here = abs(fun((*this)(i, j)));
            if(abs_here > rel_sens * fun_max || abs_here > abs_sens) {
                if(i > imax) imax = i;
                if(i < imin) imin = i;
                if(j > jmax) jmax = j;
                if(j < jmin) jmin = j;
            }
        }
    }
    // increase maxima by one to follow inclusive-exclusive convention
    imax++; jmax++;

    // log
    char msg[100];
    sprintf(msg, "\t\tLimits: rows %d -- %d ; cols %d -- %d", imin, imax, jmin, jmax);
    arr2dlog(msg);

    return Limits{imin, imax, jmin, jmax};
}


/** Print function fun applied to all the elements, formatted as 2d array
 */
void Array2d::print_prop(complex_to_real fun, FILE * out_file) const {
    for(int ix = 0; ix < nx; ix++) {
        for(int iy = 0; iy < ny; iy++)
            fprintf(out_file, PRINT_FORMAT, fun((*this)(ix, iy)));
        fprintf(out_file, "\n");
    }
}


/* Print function applied to all elements within limits specified in lim */
void Array2d::print_prop(complex_to_real fun, const Limits &lim, FILE * out_file) const {
    int imin = lim[0], imax = lim[1], jmin = lim[2], jmax = lim[3];

    for(int i = imin; i < imax; i++) {
        for(int j = jmin; j < jmax; j++)
            fprintf(out_file, PRINT_FORMAT, fun((*this)(i, j)));
        fprintf(out_file, "\n");
    }
}

/** Make a deep copy of this into another array, which must have the same size.
 * returns non-zero if copy failed.
 */
int Array2d::copy_into(Array2d &a) const {
    arr2dlog("copy_into called");
    if(a.nx != (*this).nx || a.ny != (*this).ny) return 1;

    for(int i = 0; i < nx; i ++ )
        for(int j = 0; j < ny; j ++ )
            a[i][j] = (*this)(i, j);
```

```
180
181     return 0;
182 }
183
184 /**
185  * The zero-frequency component is shifted to the middle, IN PLACE!!!
186  * The function allocates another array of the same size as a, so be careful
187  * to not run out of memory.
188  **/
189 void fftshift(Array2d &a) {
190     arr2dlog("fftshift(Array2d) call");
191     Array2d b(a.nx, a.ny);
192
193     int shift_x = (a.ny+1) / 2;
194     int shift_y = (a.nx+1) / 2;
195     for(int i = 0; i < a.nx; i ++ )
196         for(int j = 0; j < a.ny; j ++ ) {
197             int source_i = (i + shift_y) % a.nx;
198             int source_j = (j + shift_x) % a.ny;
199             b[i][j] = a(source_i, source_j);
200         }
201
202     b.copy_into(a);
203 }
204
205 /** Find the first minimum of abs(fun) along the horizontal axis in the first row */
206 ValueError<double> find_first_min(complex_to_real fun, const Array2d &a, const vector<double> &xs) {
207     int i = 0, j = 1;
208     int nx = xs.size();
209
210     // keep walking along the row until you find a local min
211     while(j < nx-1)
212         if(fun(a(i, j-1)) > fun(a(i, j)) && fun(a(i, j)) < fun(a(i, j+1)))
213             break;
214         else
215             j++;
216
217     ValueError<double> res;
218     res.val = xs[j];
219     res.err = abs(xs[j] - xs[j-1]);
220     return res;
221 }
222
223 /**
224  * Find the x-coordinate of the first half-power point along the first horizontal and the error
225  * If vertical is true, the y-coordinate along the first vertical is found instead.
226  * coord are the x-positions (or y-positions) of the points, depending on vertical
227  */
228 ValueError<double> hwhp(const Array2d &a, const vector<double> &coord, bool vertical) {
229     int j = 0;
230     int n = coord.size();
231
232     double max_abs = abs(a(0, 0));
233     double half_power = max_abs / sqrt(2.0);
234
235     // walk along the first row or column until abs(a) drops below half_power
236     while(j < n-1)
237         if(!vertical && abs(a(0, j)) < half_power)
238             break;
239         else if(vertical && abs(a(j, 0)) < half_power)
240             break;
241         else
242             j++;
243
244     ValueError<double> res;
245     res.val = coord[j];
246     res.err = abs(coord[j] - coord[j-1]);
247     return res;
248 }
249
250 /**
251  * Calculate the mean and standard deviation of fun within a given radius
252  */
253 ValueError<double> mean_stddev(complex_to_real fun, const Array2d &a, const vector<double>& xs, const vector<double>& ys,
    ↪  double radius) {
```

```
    int n_cols = xs.size(), n_rows = ys.size();
    double rsq;

    // running statistics initialization
    gsl_rstat_workspace * rstat = gsl_rstat_alloc();

    // walk the array, and add arg only if number is larger than eps
    for(int i = 0; i < n_rows; i ++ )
        for(int j = 0; j < n_cols; j ++ ) {
            rsq = xs[j] * xs[j] + ys[i] * ys[i];
            if(rsq < radius * radius)
                gsl_rstat_add(fun(a(i, j)), rstat);
        }

    // cleanup and return
    ValueError<double> res;
    res.err = gsl_rstat_sd(rstat);
    res.val = gsl_rstat_mean(rstat);
    gsl_rstat_free(rstat);
    return res;
}


/**
 * Print the limits in two directions of the 2d array, then the array itself,
 * in standard formatted way.
 * Only print stuff within x and y limits given by lims.
 */
void print_lim_array(FILE * filep, complex_to_real fun, const Array2d &a, const vector<double> &xs, const vector<double>
↪  &ys, const Limits &lims) {
    int imin = lims[0], imax = lims[1], jmin = lims[2], jmax = lims[3];

    fprintf(filep, "% 6.5f\t% 6.5f \n", xs[jmin], xs[jmax-1]);
    fprintf(filep, "% 6.5f\t% 6.5f \n", ys[imin], ys[imax-1]);
    a.print_prop(fun, lims, filep);
}
```

A.1.6 src/cpp/generators.cpp

```
#include "array2d.h"

#include<gsl/gsl_rng.h>
#include<gsl/gsl_randist.h>

#define INFO_OUT true
#define DEBUG_OUT false

// logger
Logger dbglog(stdout, "generator_dbg", DEBUG_OUT);
Logger genlog(stdout, "generator", INFO_OUT);

/** Just a circle at the origin. Params[0] is the radius. */
int circular(Array2d& in, const vector<double>& xs, const vector<double>& ys, const vector<double>& params) {
    int nx = xs.size(), ny = ys.size();
    double radius = params[0], r;

    for(int i = 0; i < nx; i ++ ) {
        for(int j = 0; j < ny; j ++ ) {
            r = xs[j] * xs[j] + ys[i] * ys[i];
            if(r <= radius*radius)
                in[i][j] = 1.0;
            else
                in[i][j] = 0.0;
        }
    }
    return 0;
}

/** A rectange of dimensions params[0] x params[1] */
int rectangle(Array2d& in, const vector<double>& xs, const vector<double>& ys, const vector<double>& params) {
    int nx = xs.size(), ny = ys.size();
    double ax = params[0], ay = params[1];

    for(int i = 0; i < nx; i ++ ) {
        for(int j = 0; j < ny; j ++ ) {
```

```
            if(abs(xs[j]) <= ax/2.0 && abs(ys[i]) <= ay/2.0)
                in[i][j] = 1.0;
            else
                in[i][j] = 0.0;
        }
    }
    return 0;
}

/** Gaussian illuminated circular aperture. params[0] is radius and params[1] is sigma */
int gaussian(Array2d& in, const vector<double>& xs, const vector<double>& ys, const vector<double>& params) {
    int nx = xs.size(), ny = ys.size();
    double R = params[0], sig = params[1];

    double R_sq = R * R;
    double sigsq2 = 2.0 * sig * sig;
    double rsq;

    for(int i = 0; i < nx; i ++ ) {
        for(int j = 0; j < ny; j ++ ) {
            rsq = xs[j] * xs[j] + ys[i] * ys[i];
            if(rsq <= R_sq)
                in[i][j] = exp(-rsq / sigsq2);
            else
                in[i][j] = 0.0;
        }
    }
    return 0;
}

/** A circular Gaussian aperture with a hole in the middle
 * params[0] is the radius. params[1] is sigma. params[2] is the hole radius.
 */
int gaussian_hole(Array2d& in, const vector<double>& xs, const vector<double>& ys, const vector<double>& params) {
    int nx = xs.size(), ny = ys.size();
    double R_ext = params[0], sig = params[1], R_int = params[2];

    double R_ext_sq = R_ext * R_ext;
    double R_int_sq = R_int * R_int;
    double sigsq2 = 2.0 * sig * sig;
    double rsq;

    for(int i = 0; i < nx; i ++ ) {
        for(int j = 0; j < ny; j ++ ) {
            rsq = xs[j] * xs[j] + ys[i] * ys[i];
            if(rsq <= R_ext_sq && rsq >= R_int_sq)
                in[i][j] = exp(-rsq / sigsq2);
            else
                in[i][j] = 0.0;
        }
    }
    return 0;
}

/** A circular and holed aperture, with random errors on it.
 * params[0] is the outer radius. params[1] is sigma. params[2] is the inner(hole) radius.
 * params[3] is the sigma of the phase errors. params[4] is (optionally) the RNG seed.
 */
int rand_errors(Array2d& in, const vector<double>& xs, const vector<double>& ys, const vector<double>& params) {
    // this is just unpacking the arguments
    int nx = xs.size(), ny = ys.size();
    double R_ext_sq = params[0] * params[0];
    double sig_sq2 = 2 * params[1] * params[1];
    double R_int_sq = params[2] * params[2];
    double err_sigma = params[3];
    double rsq, phi;

    // initiate a random number generator for the errors
    gsl_rng * rng = gsl_rng_alloc(gsl_rng_default);
    if(params.size() > 4) {
        gsl_rng_set(rng, (unsigned long int)params[4]);
    }

    for(int i = 0; i < nx; i ++ ) {
        for(int j = 0; j < ny; j ++ ) {
```

```
            rsq = xs[j] * xs[j] + ys[i] * ys[i];
            if(rsq <= R_ext_sq && rsq >= R_int_sq) {
                phi = gsl_ran_gaussian(rng, err_sigma);
                in[i][j] = polar(exp(-rsq / sig_sq2), phi);
            }
            else
                in[i][j] = 0.0;
        }
    }

    gsl_rng_free(rng);
    return 0;
}


/** Helper: A gaussian mask to convolve with.
 * params[0] is the correlation length.
 * Void return so it can't be exposed via the names map at the bottom.
 */
void gauss_mask(Array2d &in, const vector<double>& xs, const vector<double>& ys, const vector<double>& params) {
    // unpack arguments
    int n_cols = xs.size(), n_rows = ys.size();
    double lc = params[0];          // the correlation length
    double sig_sq2 = 2 * lc * lc;   // the 2 sigma squared in the gaussian
    double rsq;

    // set the array values
    for(int i = 0; i < n_rows; i ++ )
        for(int j = 0; j < n_cols; j ++ ) {
            rsq = xs[j] * xs[j] + ys[i] * ys[i];
            in[i][j] = exp( - rsq / sig_sq2);
        }
}


/** Helper: round shape of radius params[0], filled with random real numbers.
 * The randomness is gaussian-distributed, with mean 0 and sigma given by params[1].
 * params[2] is the rng seed.
 *
 * This is useful for convolving with the gaussian mask to produce correlated errors.
 * Also void return so it can't be included in the map.
 */
void real_errors(Array2d &in, const vector<double> &xs, const vector<double> &ys, const vector<double> &params) {
    // unpack the arguments
    int n_cols = xs.size(), n_rows = ys.size();

    double R_ext_sq = params[0] * params[0];
    double err_sigma = params[1];
    unsigned long seed = (unsigned long)params[2];

    double rsq;

    // initialise the rng with the given seed
    gsl_rng * rng = gsl_rng_alloc(gsl_rng_default);
    gsl_rng_set(rng, seed);

    // put a random number at each point
    for(int i = 0; i < n_rows; i ++ )
        for(int j = 0; j < n_cols; j ++ ) {
            rsq = xs[j] * xs[j] + ys[i] * ys[i];
            if(rsq <= R_ext_sq)
                in[i][j] = gsl_ran_gaussian(rng, err_sigma);
        }
    gsl_rng_free(rng);
}


/** Correlated errors - bumps in the surface
 * This is achieved by taking random errors and convolving (multiplying in Fourier space)
 * with a gaussian. It's quite resource intensive.
 * WARNING: There's an assumption that nx, ny will always be the same, which should be true
 * as long as main isn't modified!
 *
 * params are the same as above for 0 -- 4, and params[5] is the correlation length.
 */
```

```
187 int corr_errors(Array2d &in, const vector<double>& xs, const vector<double>& ys, const vector<double>& params) {
188     // unpack the arguments
189     int n_cols = xs.size(), n_rows = ys.size();
190 
191     double R_ext_sq = params[0] * params[0];
192     double sig_sq2 = 2 * params[1] * params[1];
193     double R_int_sq = params[2] * params[2];
194     double err_sigma = params[3];
195     double seed = params[4];
196     double lc = params[5];
197 
198     double rsq, rho, phi;
199 
200     // declare the secondary array and fftw plans
201     static thread_local Array2d sec_array(n_rows, n_cols);
202     static thread_local fftw_plan fwd_plan, rev_plan, sec_plan;
203     static thread_local bool init = false;
204 
205     // Planning: this only runs on the first call
206     if(!init) {
207         init = true;
208 
209         planner_mtx.lock();
210         genlog("\t\tLocked. Planning convolution FFTs.");
211 
212         // to check behaviour in multithreading, print array addresses
213         char buff[100];
214         sprintf(buff, "In array ptr: %p", (void*)in.ptr());
215         dbglog(buff);
216         sprintf(buff, "Sec array ptr: %p", (void*)sec_array.ptr());
217         dbglog(buff);
218 
219         fwd_plan = fftw_plan_dft_2d(n_rows, n_cols, in.ptr(), in.ptr(), FFTW_FORWARD, FFTW_MEASURE);
220         rev_plan = fftw_plan_dft_2d(n_rows, n_cols, in.ptr(), in.ptr(), FFTW_BACKWARD, FFTW_MEASURE);
221         sec_plan = fftw_plan_dft_2d(n_rows, n_cols, sec_array.ptr(), sec_array.ptr(), FFTW_FORWARD, FFTW_MEASURE);
222 
223         planner_mtx.unlock();
224         genlog("\t\tUnlocked. Planning done.");
225     }
226 
227     // init the arrays
228     // note that real_errors writes real numbers to the array - the depth
229     gauss_mask(sec_array, xs, ys, {lc});
230     real_errors(in, xs, ys, {params[0] + 3*lc, err_sigma, seed});
231 
232     // execute forward ffts
233     fftw_execute(fwd_plan);
234     fftw_execute(sec_plan);
235 
236     // multiply and reverse FT, then shift to proper place
237     in.mult(sec_array);
238     fftw_execute(rev_plan);
239     fftshift(in);
240 
241     // calculate the current RMS and normalize to get the desired RMS error
242     double depth_sigma = mean_stddev(myre, in, xs, ys, params[0]).err;
243     in.mult_each(err_sigma / depth_sigma);
244 
245     // walk the array and set the depth as the phase,
246     // and the amplitude as a gaussian taper
247     for(int i = 0; i < n_rows; i ++ )
248         for(int j = 0; j < n_cols; j ++ ) {
249             rsq = xs[j] * xs[j] + ys[i] * ys[i];
250             if(rsq <= R_ext_sq && rsq >= R_int_sq) {
251                 phi = real(in(i, j));
252                 rho = exp(- rsq / sig_sq2);
253                 in[i][j] = polar(rho, phi);
254             }
255             else
256                 in[i][j] = 0;
257         }
258 
259     return 0;
260 }
261 
```

```
map<string, aperture_generator> generators = {
    {"circular", circular},
    {"rectangle", rectangle},
    {"gaussian", gaussian},
    {"gaussian_hole", gaussian_hole},
    {"rand_errors", rand_errors},
    {"corr_errors", corr_errors}
};
```

A.2 Plotting in Python

A.2.1 src/scripts/util.py

```
# utilities used for reading config files and such

import re
import numpy as np

SEP = "="*30

def print_warning(configs):
    print(SEP)
    print("Warning: for the script to run correctly, data must be produced by running the executable on: ")
    for conf in configs:
        print(conf)
    print("If you've done that already, ignore this message")
    print(SEP)

def extract_value(line):
    """ Helper that returns the value from a `key = value` string """
    return re.split(r'\s+=\s+', line)[-1].strip()

def parse_config(filename):
    """
    Parses a configuration file and returns the values of some of the options
    It returns the number of shapes, the prefix for the files produced,
    and the figures produced
    e.g. if there is a task called "print_in_phase", figs will contain "in_phase"
    """
    n_shapes = None
    prefix = None
    tasks = None

    with open(filename) as fin:
        lines = fin.readlines()
        for line in lines:
            if line.startswith("n_shapes"):
                n_shapes = int(extract_value(line))
            elif line.startswith("prefix"):
                prefix = extract_value(line)
            elif line.startswith("tasks"):
                tasks = extract_value(line)

    # extract what figures are produced for each shape
    figs = []
    for task in tasks.split():
        if task.startswith("print_"):
            figs.append(task[6:])

    return (n_shapes, prefix, figs)


def read_image(filename):
    """ Read image from a data file """

    print("Reading data from " + filename)
    with open(filename) as fin:
        lines = fin.readlines()
        data = [[float(x) for x in line.split()] for line in lines]
        xlim = data[0]
        ylim = data[1]
        data = np.array(data[2:])
    return xlim, ylim, data


```

```
63  def read_data(config_filename):
64      """ Read the data file created by running the executable on given config """
65      _, prefix, _ = parse_config(config_filename)
66
67      data_fname = prefix + "dat.txt"
68      with open(data_fname) as fin:
69          lines = fin.readlines()
70          data = np.array([[float(x) for x in line.split()] for line in lines])
71      return np.ndarray.transpose(data)
72
73  def accumulate_data(data, xidx, yidx, dyidx=None):
74      """
75      This function takes data where multiple points may have the same x value, and optionally with error bars.
76      It accumulates the points with the same x into one data point,
77      with error given by the larger of standard deviation or intrinsic error.
78      xidx, yidx, dyidx are the indices of the columns to use as x, y and error respectively
79      """
80      xs = data[xidx]
81      ys = data[yidx]
82      if dyidx:
83          dys = data[dyidx]
84      else:
85          dys = np.zeros(len(ys))
86      res = ([], [], [])
87
88      tmp = []
89      x_current = xs[0]
90
91      for i, x in enumerate(xs):
92          if x == x_current:
93              tmp.append(ys[i])
94          else:
95              # if the current x has changed, push the values to res and start a new run
96              res[0].append(x_current)
97              res[1].append(np.average(tmp))
98              res[2].append(np.maximum(np.std(tmp), dys[i]))
99
100             x_current = x
101             tmp = [ys[i]]
102
103     # flush what's left
104     res[0].append(x_current)
105     res[1].append(np.average(tmp))
106     res[2].append(np.maximum(np.std(tmp), dys[i]))
107
108     return res
109
110
111 def group_data(data, idx):
112     """
113     Group the rows in a 2d data array based on column number idx,
114     and return a dict whose values are sub-arrays of data that have
115     the same value in the idx-th column, and whose keys are that common value.
116     """
117     res = {}
118     # data is originally read in column-first representation, so needs transposing
119     data = np.ndarray.transpose(data)
120
121     # walk the data row-wise and add each row to the corresponding entry in res
122     for row in data:
123         key = row[idx]
124         if key in res:
125             res[key].append(row)
126         else:
127             res[key] = [row]
128
129     # transpose back to column-first for easy plotting
130     for key in res.keys():
131         res[key] = np.array(res[key])
132         res[key] = np.ndarray.transpose(res[key])
133
134     return res
135
136
137 def relim(data_shape, old_lim, new_lim):
```

```
    """
    Take a data array with known x and y limits
    and return the data between different x and y limits
    of course, the new limits should be included in the old ones
    Returns two tuples:
        (min_row, max_row, min_col, max_col)
    And then the coordinates:
        (xmin, xmax, ymin, ymax)
    """
    x1, x2, y1, y2 = old_lim
    xn1, xn2, yn1, yn2 = new_lim
    n_rows, n_cols = data_shape

    # the delta is the distance divided by the number of intervals
    dx = abs(x2 - x1) / (n_cols-1)
    dy = abs(y2 - y1) / (n_rows-1)

    # find the indices of the new coordinates
    min_col = int(np.ceil((xn1 - x1) / dx))
    max_col = int(np.ceil((xn2 - x1) / dx))

    min_row = int(np.ceil((yn1 - y1) / dy))
    max_row = int(np.ceil((yn2 - y1) / dy))

    xlim = (x1 + dx * min_col), (x1 + dx * (max_col-1))
    ylim = (y1 + dy * min_row), (y1 + dy * (max_row-1))

    return (min_row, max_row, min_col, max_col), xlim+ylim
```

A.2.2 src/scripts/write_config.py

```
#!/usr/bin/python3
# Utility script to make config files programatically
# Change to your leisure

import os
import numpy as np

################ leave these alone ################
PARAM_STR = "{:s} = {}\n"

CONF_DIR = "config"
DATA_DIR = "data"
os.makedirs(CONF_DIR, exist_ok=True)
os.makedirs(DATA_DIR, exist_ok=True)

# write one key = value line
def write_option(fout, name, value):
    fout.write(PARAM_STR.format(name, value))

################ change these ################
FILENAME = "corr_sig.txt"

# write the RMS errors
sigmas = [(i+1)/12 * np.pi for i in range(12)]
print(sigmas)
# for each, write the correlation lengths
ls = [(i+1) * 0.25 for i in range(5)]
print(ls)

# repeat each this many times - to deal with the randomness
N_DIFF_SHAPES = len(ls) * len(sigmas)
N_REPEAT = 20

NX, NY = 2**12, 2**12
DATA_PREFIX = os.path.join(DATA_DIR, "corr_sig")
TASKS = "params fwhp fwhp_y central_amplitude out_lims"
REL_SENS, ABS_SENS = 0.01, 0

TYPE = "corr_errors"
LX, LY = 512, 512

################ leave this ################
with open(os.path.join(CONF_DIR, FILENAME), "w") as fout:
    write_option(fout, "nx", NX)
```

```
    write_option(fout, "ny", NY)
    write_option(fout, "prefix", DATA_PREFIX)
    write_option(fout, "tasks", TASKS)
    write_option(fout, "rel_sens", REL_SENS)
    write_option(fout, "abs_sens", ABS_SENS)
    write_option(fout, "n_shapes", N_DIFF_SHAPES * N_REPEAT)

    for s_eps in sigmas:
        for l_corr in ls:
            for _ in range(N_REPEAT):
                fout.write("\n")
                write_option(fout, "type", TYPE)
                write_option(fout, "lx", LX)
                write_option(fout, "ly", LY)
                ################ and change this: ################
                r = 6           # telescope radius
                sig = 3         # taper
                r_int = 0       # interior hole
                seed = np.random.randint(1, 32e3)        # rng seed
                write_option(fout, "params", "{: 5.5f} {: 5.5f} {: 5.5f} {: 5.5f} {:05d} {: 5.5f}".format(r, sig, r_int,
                ↪  s_eps, seed, l_corr))
```

A.2.3 src/scripts/make_pictures.py

```
#!/usr/bin/python3

import re
import os

import numpy as np
import matplotlib.pyplot as plt
import matplotlib

from util import *

COLORMAP = matplotlib.cm.inferno
PHASE_COLORMAP = matplotlib.cm.bwr
FIGSIZE = (9, 8)
FONTSIZE = 16

def colour_plot(data, limits, title, colorbar=False, colormap=COLORMAP):
    matplotlib.rcParams.update({'font.size': FONTSIZE})

    fig, ax = plt.subplots(figsize=FIGSIZE)
    im = ax.imshow(data, cmap=colormap, interpolation='nearest', extent=limits)

    # draw colorbar
    if colorbar:
        fig.colorbar(im, ax=ax, format="% .2g")

    ax.set_title(title)


SAVE_DIR = os.path.join("fig", "error_pics")

if __name__ == "__main__":
    os.makedirs(SAVE_DIR, exist_ok=True)

    # colour_plot(os.path.join("data", "gauss0out_abs.txt"), "Uniform illumination", colorbar=True)
    # plt.savefig(os.path.join(SAVE_DIR, "uniform.png"), bbox_inches='tight')

    # colour_plot(os.path.join("data", "gauss1out_abs.txt"), "Gaussian illumination", colorbar=True)
    # plt.savefig(os.path.join(SAVE_DIR, "gaussian.png"), bbox_inches='tight')

    # uncomment this to automagically plot everything created by a certain config
    # otherwise do it manually with your own tweaks
    n_shapes, prefix, figs = parse_config("config/errors.txt")
    for i in range(n_shapes):
        for fig in figs:
            filename = prefix + str(i) + fig + ".txt"
            title = fig.replace("_", " ").replace("in", "mirror").replace("out", "image").replace("abs", "amplitude")

            # read the data
            xlim, ylim, data = read_image(filename)
            coord_lim = xlim+ylim
```

```
            # select the colormap
            if fig.endswith("phase"):
                colormap = PHASE_COLORMAP
            else:
                colormap = COLORMAP

            # relimit to make all the same size
            if fig.startswith("out"):
                idx_lim, coord_lim = relim(data.shape, xlim+ylim, (-4.1, 4.1, -4.1, 4.1))
                min_row, max_row, min_col, max_col = idx_lim
                data = np.sqrt(data[min_row:max_row, min_col:max_col])

            colour_plot(data, coord_lim, title, colorbar=True, colormap=colormap)

            # save the figure
            figname = filename.replace(".txt", ".png").replace("data/", "")
            plt.savefig(os.path.join(SAVE_DIR, figname), bbox_inches="tight")
        plt.show()
```

A.2.4 src/scripts/tests.py

```
#!/usr/bin/python3

import re
import os

import numpy as np
import matplotlib.pyplot as plt
import matplotlib

from util import *

SIZES_CONFIG = "config/circular_mins.txt"
AMP_CONFIG = "config/circular_amps.txt"
RECT_CONFIG = "config/rectangle_mins.txt"
SAVE_DIR = "fig/tests"

FONTSIZE = 16
CAPSIZE = 7
FIT_LABEL = "fit: $y={:4.4f}x{:+4.4f}$"
FIGSIZE = (9, 8)

SEP = "="*30

def plot_sizes(data):
    xs = 1/data[1]      # inverse aperture diameter
    ys = data[2]        # image size
    dys = data[3]       # image size error

    # formatting niceness
    matplotlib.rcParams.update({'font.size': FONTSIZE})
    matplotlib.rcParams.update({'errorbar.capsize': CAPSIZE})

    # plot the data
    fig, ax = plt.subplots(figsize=FIGSIZE)
    ax.errorbar(xs, ys, yerr=dys, fmt='+', label="data", markersize=FONTSIZE)

    # fit a line
    coefs = np.polyfit(xs, ys, 1, w=1/dys)
    print("Airy disc fit coefficients, largest power first: ", coefs)
    fit_label = FIT_LABEL.format(coefs[0], coefs[1])
    ax.plot(xs, np.polyval(coefs, xs), '-', label=fit_label)

    # label the plot
    ax.set_xlabel("(1 / aperture radius) $(m^{{-1}})$")
    ax.set_ylabel("central spot radius $p$ $(m^{{-1}})$")
    ax.set_title("Airy disc radii")
    ax.legend()

    plt.savefig(os.path.join(SAVE_DIR, "airy.pdf"))


def plot_rect_sizes(data):
    xs = 2/data[1]      # inverse aperture size/2
```

```
    ys = data[3]        # image size
    dys = data[4]       # image size error

    # formatting niceness
    matplotlib.rcParams.update({'font.size': FONTSIZE})
    matplotlib.rcParams.update({'errorbar.capsize': CAPSIZE})

    # plot the data
    fig, ax = plt.subplots(figsize=FIGSIZE)
    ax.errorbar(xs, ys, yerr=dys, fmt='+', label="data", markersize=FONTSIZE)

    # fit a line
    coefs = np.polyfit(xs, ys, 1, w=1/dys)
    fit_label = FIT_LABEL.format(coefs[0], coefs[1])
    print("Rectangular mirror fit coefs, largest power first: ", coefs)
    ax.plot(xs, np.polyval(coefs, xs), '-', label=fit_label)

    # label the plot
    ax.set_xlabel("(1 / aperture half-width) $(m^{{-1}})$")
    ax.set_ylabel("first minimum $p$ $(m^{{-1}})$")
    ax.set_title("Rectangular aperture image size")
    ax.legend()

    plt.savefig(os.path.join(SAVE_DIR, "rect.pdf"))


def plot_amplitudes(data):
    xs = np.power(data[1], 2)       # square aperture size
    ys = data[2]/np.min(data[2])    # central amplitude

    # formatting
    matplotlib.rcParams.update({'font.size': FONTSIZE})

    # plot
    fig, ax = plt.subplots(figsize=FIGSIZE)
    ax.plot(xs, ys, '+', label="data", markersize=FONTSIZE)

    # label the plot
    ax.set_xlabel("(aperture size)$^2$ $(m^2)$")
    ax.set_ylabel("central amplitude (arbitrary)")
    ax.set_title("Central amplitude of circular aperture")
    # ax.legend()

    plt.savefig(os.path.join(SAVE_DIR, "amp.pdf"))


if __name__ == "__main__":
    print_warning([SIZES_CONFIG, AMP_CONFIG, RECT_CONFIG])

    os.makedirs(SAVE_DIR, exist_ok=True)

    plot_sizes(read_data(SIZES_CONFIG))
    plot_amplitudes(read_data(AMP_CONFIG))
    plot_rect_sizes(read_data(RECT_CONFIG))
    plt.show()
```

A.2.5 src/scripts/gauss.py

```
#!/usr/bin/python3

import os

import numpy as np
import matplotlib.pyplot as plt
import matplotlib

from util import *

FONTSIZE = 16
CAPSIZE = 7
FIT_LABEL = "$y={:4.4f}x{:+4.4f}$"
FIGSIZE = (9, 8)

def plot_sizes(data):
    rs = data[1]            # radii
```

```
    sig = data[2]           # taper
    mins = data[3]          # minima location
    sig_mins = data[4]      # error

    xs = np.divide(rs, sig)
    ys = np.array(mins)
    dys = np.array(sig_mins)

    # formatting niceness
    matplotlib.rcParams.update({'font.size': FONTSIZE})
    matplotlib.rcParams.update({'errorbar.capsize': CAPSIZE})

    # plot the data
    fig, ax = plt.subplots(figsize=FIGSIZE)
    ax.errorbar(xs, ys, yerr=dys, fmt='+', label="data", markersize=FONTSIZE)

    # find limit to first regime
    i_min = [i for (i, x) in enumerate(xs) if x > 3][0]

    # fit a line to first regime
    coefs = np.polyfit(xs[i_min:], ys[i_min:], 1, w=1/dys[i_min:])
    print("FWHP largest power first: ", coefs)
    fit_label = "fit to 1/$\\sigma$ regime:\n" + FIT_LABEL.format(coefs[0], coefs[1])
    ax.plot(xs, np.polyval(coefs, xs), '-', label=fit_label)

    # label the plot
    ax.set_xlabel("$R / \\sigma$")
    ax.set_ylabel("Full Width at Half Power $(m^{{-1}})$")
    ax.set_title("Gaussian illumination central width")
    ax.legend()

    plt.savefig(os.path.join(SAVE_DIR, "size.pdf"))


def plot_intensities(data):
    rs = data[1]                        # radii
    sig = data[2]                       # taper
    rel_amp = data[5]/np.min(data[5])   # relative amplitude
    ys = np.power(rel_amp, 2)           # relative intensity

    xs = np.divide(sig, rs)

    # formatting
    matplotlib.rcParams.update({'font.size': FONTSIZE})

    # plot
    fig, ax = plt.subplots(figsize=FIGSIZE)
    ax.plot(xs, ys, '+', label="data", markersize=FONTSIZE)

    # label the plot
    ax.set_xlabel("$\\sigma / R$")
    ax.set_ylabel("central intensity (arbitrary)")
    ax.set_title("Gaussian illumination central intensity")
    ax.ticklabel_format(axis='y', style='sci', scilimits=(-2,3), useMathText=True)
    # ax.legend()

    plt.savefig(os.path.join(SAVE_DIR, "int.pdf"))

CONFIG1 = "config/gauss_hm.txt"
CONFIG2 = "config/gauss_amp.txt"
SAVE_DIR = os.path.join("fig", "gauss")

if __name__ == "__main__":
    print_warning([CONFIG1, CONFIG2])

    os.makedirs(SAVE_DIR, exist_ok=True)

    plot_sizes(read_data(CONFIG1))
    plot_intensities(read_data(CONFIG2))
    plt.show()
```

A.2.6 src/scripts/hole.py

```
#!/usr/bin/python3

```

```
import re
import os

import numpy as np
import matplotlib.pyplot as plt
import matplotlib

from util import *

COLORMAP = matplotlib.cm.plasma
FIGSIZE = (12, 9)
FONTSIZE = 16

def colour_plot(data, limits, title, colorbar=False, logdata=False, colormap=COLORMAP):
    matplotlib.rcParams.update({'font.size': FONTSIZE})

    if logdata:
        data = np.log(data)

    fig, ax = plt.subplots(figsize=FIGSIZE)
    im = ax.imshow(data, cmap=COLORMAP, interpolation='nearest', extent=limits)

    # draw colorbar
    if colorbar:
        fig.colorbar(im, ax=ax, format="% .2g")

    ax.set_title(title)


SAVE_DIR = os.path.join("fig", "hole")

if __name__ == "__main__":
    os.makedirs(SAVE_DIR, exist_ok=True)

    """
    What I'm doing here: take read the amplitude and phase of the small and large
    mirror patterns, then compute the difference between them. Compare that
    to the mirror-with-hole pattern to see how different it is.
    """

    # read in the large mirror pattern
    xlim, ylim, l_abs = read_image("data/hole0out_abs.txt")
    xlim, ylim, l_arg = read_image("data/hole0out_phase.txt")
    # make it complex
    l_true = np.multiply(l_abs, np.exp(1j * l_arg))

    # read in the small mirror pattern
    xs, ys, s_abs = read_image("data/hole1out_abs.txt")
    (row_min, row_max, col_min, col_max), newlims = relim(s_abs.shape, xs+ys, xlim+ylim)
    s_abs = s_abs[row_min:row_max, col_min:col_max]

    xs, ys, s_arg = read_image("data/hole1out_phase.txt")
    s_arg = s_arg[row_min:row_max, col_min:col_max]
    # make complex
    s_true = np.multiply(s_abs, np.exp(1j * s_arg))

    # take the difference
    diff = np.abs(l_true - s_true)

    # read the holed thing
    xs, ys, h_abs = read_image("data/hole2out_abs.txt");
    (row_min, row_max, col_min, col_max), newlims = relim(h_abs.shape, xs+ys, xlim+ylim)
    h_abs = h_abs[row_min:row_max, col_min:col_max]
    colour_plot(np.sqrt(h_abs), newlims, "Holed mirror image", colorbar=True)
    plt.savefig(os.path.join(SAVE_DIR, "image.png"), bbox_inches='tight')

    colour_plot(np.abs(diff - h_abs), newlims, "Holed mirror delta", colorbar=True)
    plt.savefig(os.path.join(SAVE_DIR, "delta2.png"), bbox_inches='tight')

    plt.show()
```

A.2.7 src/scripts/rand.py

```
#!/usr/bin/python3

```

```
import os

import numpy as np
import matplotlib.pyplot as plt
import matplotlib

from util import *

FONTSIZE = 18
CAPSIZE = 7
FIT_LABEL = "$y={:4.4f}x{:+4.4f}$"
FIT_INFO = "Slope: {:4.6f} +/- {:4.6f}\nIntercept: {:4.6f} +/- {:4.6f}"
X_LABEL = "$\\sigma_{{\\epsilon}}/\\lambda$"
FIGSIZE = (9, 8)

def plot_sizes(data):
    xs = np.divide(data[0], 4 * np.pi)    # phase error std dev
    ys = data[1]          # fwhp
    dys = data[2]         # error

    # formatting niceness
    matplotlib.rcParams.update({'font.size': FONTSIZE})
    matplotlib.rcParams.update({'errorbar.capsize': CAPSIZE})

    # plot the data
    fig, ax = plt.subplots(figsize=FIGSIZE)
    ax.errorbar(xs, ys, yerr=dys, fmt='+', label="data", markersize=FONTSIZE)

    # label the plot
    ax.set_xlabel(X_LABEL)
    ax.set_ylabel("Full Width at Half Power $(m^{{-1}})$")
    ax.set_title("Central width with phase errors")
    # ax.legend()

    plt.savefig(os.path.join(SAVE_DIR, "size.pdf"), bbox_inches="tight")


def plot_intensities(data):
    sigerr = np.divide(data[0], 4 * np.pi)  # phase error std dev
    amp = data[1] / np.min(data[1])         # amplitude normalised
    sig_amp = data[2] / np.min(data[1])     # error in amplitudes normalised

    # plot log of amplitude vs square of phase deviation
    xs = np.power(np.array(sigerr), 2)
    ys = np.log(amp)
    dys = np.divide(sig_amp, amp)

    # formatting
    matplotlib.rcParams.update({'font.size': FONTSIZE})

    # plot
    fig, ax = plt.subplots(figsize=FIGSIZE)
    ax.errorbar(xs, ys, yerr=dys, fmt='+', label="data", markersize=FONTSIZE)

    # fit a line
    coefs, covar = np.polyfit(xs, ys, 1, w=1/dys, cov=True)
    fit_label = "Line fit: " + FIT_LABEL.format(coefs[0], coefs[1])
    print(FIT_INFO.format(coefs[0], np.sqrt(covar[0, 0]), coefs[1], np.sqrt(covar[1, 1])))
    ax.plot(xs, np.polyval(coefs, xs), '-', label=fit_label)

    # label the plot
    ax.set_xlabel("({:s})$^2$".format(X_LABEL))
    ax.set_ylabel("$\\ln(\\psi_0\\ /\\ arb.\\ units)$")
    ax.set_title("Central amplitude vs phase errors")
    ax.ticklabel_format(axis='y', style='sci', scilimits=(-2,3), useMathText=True)
    ax.legend()

    plt.savefig(os.path.join(SAVE_DIR, "int.pdf"), bbox_inches="tight")


CONFIG1 = "config/rand.txt"
# CONFIG2 = "config/gauss_amp.txt"
SAVE_DIR = os.path.join("fig", "rand")

if __name__ == "__main__":
```

```
    print_warning([CONFIG1])

    os.makedirs(SAVE_DIR, exist_ok=True)

    data = read_data(CONFIG1)
    plot_sizes(accumulate_data(data, 4, 6, 7))
    plot_intensities(accumulate_data(data, 4, 8, None))
    plt.show()
```

A.2.8 src/scripts/rms_test.py

```
#!/usr/bin/python3

import os

import numpy as np
import matplotlib.pyplot as plt
import matplotlib

from util import *

FONTSIZE = 18
CAPSIZE = 7
FIT_LABEL = "$y={:4.4f}x{:+4.4f}$"
FIT_INFO = "Slope: {:4.6f} +/- {:4.6f}\nIntercept: {:4.6f} +/- {:4.6f}"
FIGSIZE = (9, 8)


def plot_depth(data):
    xs = np.array(data[0])    # desired phase error
    ys = np.array(data[1])    # obtained phase error
    dys = np.array(data[2])   # error

    # formatting niceness
    matplotlib.rcParams.update({'font.size': FONTSIZE})
    matplotlib.rcParams.update({'errorbar.capsize': CAPSIZE})

    # plot the data
    fig, ax = plt.subplots(figsize=FIGSIZE)
    ax.errorbar(xs, ys, yerr=dys, fmt='+', label="data", markersize=FONTSIZE)

    # fit a line
    # first, find the xs between 0 and pi/2
    idx = [i for i, x in enumerate(xs) if x >= 0 and x <= np.pi/2]

    coefs, covar = np.polyfit(xs[idx], ys[idx], 1, cov=True)
    print(FIT_INFO.format(coefs[0], np.sqrt(covar[0, 0]), coefs[1], np.sqrt(covar[1, 1])))

    fit_label = "Line fit: " + FIT_LABEL.format(coefs[0], coefs[1])
    ax.plot(xs[idx], np.polyval(coefs, xs[idx]), '-', label=fit_label)

    # label the plot
    ax.set_xlabel("Desired phase RMS (radians)")
    ax.set_ylabel("Resulting phase RMS (radians)")
    ax.legend()

    plt.savefig(os.path.join(SAVE_DIR, "depth.pdf"), bbox_inches="tight")


def plot_lc(data):
    xs = data[0]    # correlation length
    ys = data[1]    # obtained phase error
    dys = data[2]   # error

    # formatting niceness
    matplotlib.rcParams.update({'font.size': FONTSIZE})
    matplotlib.rcParams.update({'errorbar.capsize': CAPSIZE})

    # plot the data
    fig, ax = plt.subplots(figsize=FIGSIZE)
    ax.errorbar(xs, ys, yerr=dys, fmt='+', label="data", markersize=FONTSIZE)

    # label the plot
    ax.set_xlabel("Correlation length (m)")
    ax.set_ylabel("Resulting phase RMS (radians)")
```

```
    # ax.legend()

    plt.savefig(os.path.join(SAVE_DIR, "l_corr.pdf"), bbox_inches="tight")


CONFIG1 = "config/rms_test_depth.txt"
CONFIG2 = "config/rms_test_lc.txt"
SAVE_DIR = os.path.join("fig", "rmstest")

if __name__ == "__main__":
    print_warning([CONFIG1, CONFIG2])

    os.makedirs(SAVE_DIR, exist_ok=True)

    data = read_data(CONFIG1)
    plot_depth(accumulate_data(data, 4, 8, None))

    data = read_data(CONFIG2)
    plot_lc(accumulate_data(data, 6, 8, None))
    plt.show()
```

A.2.9 src/scripts/corr.py

```
#!/usr/bin/python3

import os
import itertools

import numpy as np
import matplotlib.pyplot as plt
import matplotlib

from util import *

FONTSIZE = 18
CAPSIZE = 7
FIT_LABEL = "$y={:4.4f}x{:+4.4f}$"
FIT_INFO = "Slope: {:4.6f} +/- {:4.6f};\tIntercept: {:4.6f} +/- {:4.6f}"
FIGSIZE = (9, 9)

X_LABEL = "Correlation length $(m)$"

def add_means(data):
    """
    here is some pre-processing
    i want to calculate the average and error of x and y FWHP
    and the average and error of 1% fall-off distace in +/-x and +/-y
    and put them into their own rows.
    """
    # here the fhwp in x and y are aggregated into an average and error
    fwhp_row = np.mean([ data[7], data[9] ], axis=0)
    fwhp_sigma_row = np.std([ data[7], data[9] ], axis=0)
    fwhp_sigma_row = np.max([fwhp_sigma_row, data[8], data[10]], axis=0)
    # they will be rows 16 and 17
    data = np.append(data, [fwhp_row], axis=0)
    data = np.append(data, [fwhp_sigma_row], axis=0)

    # here the 1% falloff distances are aggregated into an average and error
    falloff_distances = [data[12], data[13], data[14], data[15]]
    dist_row = np.mean(np.abs(falloff_distances), axis=0)
    dist_sigma_row = np.std(np.abs(falloff_distances), axis=0)
    # this will be rows 18 and 19
    data = np.append(data, [dist_row], axis=0)
    data = np.append(data, [dist_sigma_row], axis=0)

    return data


def get_keys_by_index(dict_keys, idx):
    """
    Sort the list of dict keys and return the ones of index idx
    """

    if idx is None:
        idx = range(len(dict_keys))
```

```
53
54      sorted_keys = sorted(dict_keys)
55      return [key for i, key in enumerate(sorted_keys) if i in idx]
56
57
58  def plot_runs_v_lc(data_dict, x_idx, y_idx, dy_idx=None, runs=None, x_label=X_LABEL, y_label=None, x_log=False,
    ↪  y_log=False):
59      """
60      The data is grouped by group_data into runs with different RMS errors
61      We want to plot a dataset per run, and need to accumulate each run first
62      x_idx, y_idx, dy_idx are the indices of x, y and errorbar columns respectively
63      runs is a list of indices of which runs to plot (in the order of increasing key)
64      """
65
66      # check for None's
67      if y_label is None:
68          y_label = ""
69
70      # formatting niceness
71      matplotlib.rcParams.update({'font.size': FONTSIZE})
72      matplotlib.rcParams.update({'errorbar.capsize': CAPSIZE})
73
74      # open the plot
75      fig, ax = plt.subplots(figsize=FIGSIZE)
76
77      # iterate over the separate runs -- the key is RMS of phase
78      run_keys = get_keys_by_index(data_dict.keys(), runs)
79
80      for key in run_keys:
81          label =  "$\\sigma_\\epsilon = {:.2f} \\pi$".format(key / np.pi)
82
83          acc = accumulate_data(data_dict[key], x_idx, y_idx, dyidx=dy_idx)
84
85          xs = np.array(acc[0])
86          ys = np.array(acc[1])
87          dys = np.array(acc[2])
88
89          ax.errorbar(xs, ys, yerr=dys, fmt='+', label=label, markersize=FONTSIZE)
90
91          # label the plot
92          ax.set_xlabel(x_label)
93          ax.set_ylabel(y_label)
94          ax.legend()
95
96
97  def plot_falloff_v_lc(data_dict, runs=None):
98      """
99      Plots grouped data runs like the generic function, but with some extra tweaking
100     and line fitting specific for this plot.
101     """
102
103     # set local vars
104     x_idx = 6
105     y_idx = 18
106     dy_idx = 19
107
108     # formatting niceness
109     matplotlib.rcParams.update({'font.size': FONTSIZE})
110     matplotlib.rcParams.update({'errorbar.capsize': CAPSIZE})
111
112     # open the plot
113     fig, ax = plt.subplots(figsize=FIGSIZE)
114
115     # iterate over the separate runs -- the key is RMS of phase
116     run_keys = get_keys_by_index(data_dict, runs)
117
118     for run_no, key in enumerate(run_keys):
119         label =  "$\\sigma_\\epsilon = {:.2f} \\pi$".format(key / np.pi)
120
121         acc = accumulate_data(data_dict[key], x_idx, y_idx, dyidx=dy_idx)
122
123         xs = np.log(acc[0])       # correlation length
124         ys = np.log(acc[1])       # falloff distance
125         dys = np.divide(acc[2], acc[1])      # relative error
126
```

```
        scatter = ax.errorbar(xs, ys, yerr=dys, fmt='+', label=label, markersize=FONTSIZE)[0]

        # get the x values for which to calculate the fit
        # this is empirical, and should be changed for new data
        if run_no in [0, 1]:
            # first 2 runs: take all the xs
            idxs = range(len(xs))
        elif run_no == 2:
            idxs = [i for i, x in enumerate(xs) if x > -1.5]
        else:
            idxs = [i for i, x in enumerate(xs) if x > -1.1]

        # fit a line and plot it
        coefs, covar = np.polyfit(xs[idxs], ys[idxs], 1, w=1/dys[idxs], cov=True)

        print(SEP)
        print("Run: sigma_eps = {: .2f} pi".format(key / np.pi))
        print(FIT_INFO.format(coefs[0], np.sqrt(covar[0, 0]),
              coefs[1], np.sqrt(covar[1, 1])))

        ax.plot(xs[idxs], np.polyval(coefs, xs[idxs]), '-', color=scatter.get_color())

        # label the plot
        ax.set_xlabel("log ({:s})".format(X_LABEL))
        ax.set_ylabel("log (1% decay distance $(m^{{-1}})$)")
        ax.legend()


def plot_amp_v_sig(data_dict, runs=None):
    """
    Now data is grouped by correlation length, and we want to plot central
    amplitude v RMS phase.
    """

    # local vars
    x_idx = 4
    y_idx = 11

    # formatting niceness
    matplotlib.rcParams.update({'font.size': FONTSIZE})
    matplotlib.rcParams.update({'errorbar.capsize': CAPSIZE})

    # open the plot
    fig, ax = plt.subplots(figsize=FIGSIZE)

    # iterate over the separate runs -- the key is correlation length
    run_keys = get_keys_by_index(data_dict.keys(), runs)

    for key in run_keys:
        label =  "$l_c = {:.2f}$".format(key)

        acc = accumulate_data(data_dict[key], x_idx, y_idx)

        norm = np.min(acc[1])   # normalization factor for the amplitude

        xs = np.power(np.divide(acc[0], 4 * np.pi), 2)  # phase error squared
        ys = np.log(acc[1] / norm)      # normalised amplitude
        dys = np.divide(acc[2] / norm, acc[1])   # error in amplitude

        scatter = ax.errorbar(xs, ys, yerr=dys, fmt='+', label=label, markersize=FONTSIZE)[0]

        # find xs to use for line fit
        idxs = [i for i, x in enumerate(xs) if x < 0.03]

        # fit
        coefs, covar = np.polyfit(xs[idxs], ys[idxs], 1, w=1/dys[idxs], cov=True)

        print(SEP)
        print("Run: l_c = {:.4f}".format(key))
        print(FIT_INFO.format(coefs[0], np.sqrt(covar[0, 0]),
              coefs[1], np.sqrt(covar[1, 1])))

        # print the decay width sigma_psi
        slope, err_slope = coefs[0], np.sqrt(covar[0, 0])
        sig_psi = np.sqrt(- 1 / 2 / slope)
```

```
        err_sig_psi = 1/2 * sig_psi * err_slope / (- slope)
        print("sig_psi = {:4g} +/- {:4g}".format(sig_psi, err_sig_psi))

        ax.plot(xs[idxs], np.polyval(coefs, xs[idxs]), '-', color=scatter.get_color())

        # label the plot
        ax.set_xlabel("$(\\sigma_\\epsilon / \\lambda)^2$")
        ax.set_ylabel("log ($\\psi_0$ (arb. units))")
        ax.legend()


CONFIG1 = "config/corr_lc.txt"
CONFIG2 = "config/corr_sig.txt"
SAVE_DIR = os.path.join("fig", "corr")

if __name__ == "__main__":
    print_warning([CONFIG1, CONFIG2])
    os.makedirs(SAVE_DIR, exist_ok=True)

    # FIRST: consider runs of same sigma_err, different lc
    data = add_means(read_data(CONFIG1))
    # group by the RMS phase error
    sig_eps_groups = group_data(data, 4)

    # plot the FWHP versus correlation length
    fwhp_label = "Full width at half power $(m^{{-1}})$"
    plot_runs_v_lc(sig_eps_groups, 6, 16, dy_idx=17, y_label=fwhp_label)
    plt.savefig(os.path.join(SAVE_DIR, "fwhp.pdf"), bbox_inches="tight")

    # plot the 1% falloff distance vs correlation length
    plot_falloff_v_lc(sig_eps_groups)
    plt.savefig(os.path.join(SAVE_DIR, "dist.pdf"), bbox_inches="tight")

    # SECOND: consider runs of same lc, different sigma_err
    data = add_means(read_data(CONFIG2))
    # group data by the correlation length
    lc_groups = group_data(data, 6)
    # plot the central amplitude vs RMS phase
    plot_amp_v_sig(lc_groups)
    plt.savefig(os.path.join(SAVE_DIR, "amp.pdf"), bbox_inches="tight")

    plt.show()
```