Master in High Performance Computing

Zernike mode analysis for advanced wavefront imaging

Supervisors:
Ivan Girotto,
Ricardo Oliva

Candidate:
Daniel Panea Lichtig

9th edition
2022–2023

Abstract

Wavefront sensing is a powerful technique to assess and quantify the aberrations present in optical systems. Turning this idea around and assuming an ideal optical system, it can also be used to accurately measure the three-dimensional shape of transparent or reflecting objects placed inside the optical setup. Subsequent processing of the obtained images often involves the separation of low and high frequency spatial modes, which for the case of circular objects or pupils are conventionally taken to be Zernike polynomials. In this work we develop fast and highly parallel algorithms to accurately compute the Zernike mode coefficients of a given wavefront image. We then extend this method to improve the performance in the more complicated and realistic cases when the pupil or the object are not perfectly circular.

Acknowledgements

I thank all my friends from MHPC for the memorable times in class, the boss for all the climbing, Ivan for the supervision and advice and Ricardo for the many helpful comments and ideas during the elaboration of this thesis.

Contents

Chapter 1

Introduction

Wavefront sensing

In physics, the wavefront of a time-varying wave field such as visible light denotes the locus of all points having the same phase.

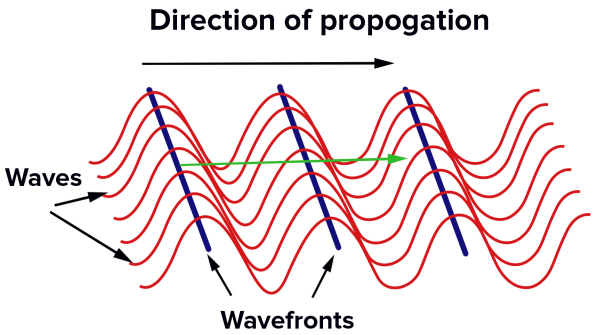

Figure 1.1: Wavefronts in wave physics. Image from Byju's [1].

In the case of geometric optics, where the propagation of light is approximately described by rays instead of waves, the wavefronts are the surfaces perpendicular to all light rays. The shape of the wavefront indicates how the rays are propagating: if all rays propagate in the same direction, parallel to each other, the wavefronts will be planar. Instead, if the rays are diverging or converging as an effect of a lens for example, the wavefronts will be spherical or of more complicated shapes.

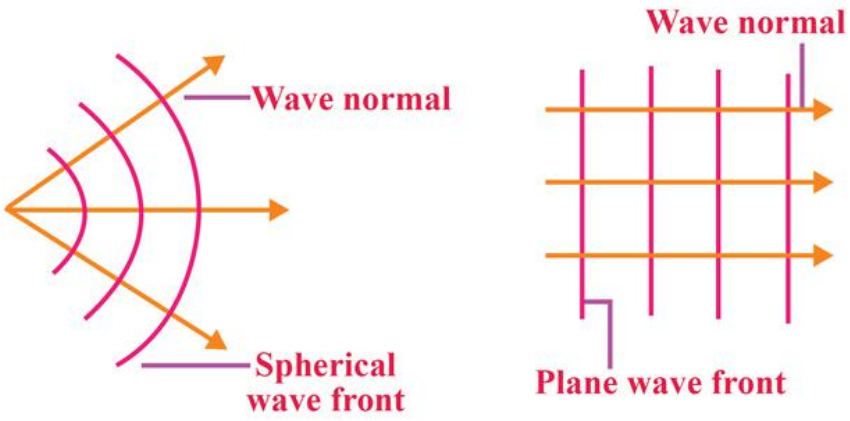

Figure 1.2: Wavefronts in geometric optics. Image from Toppr [2]

The shape of the wavefronts can therefore be used to describe the propagation of light rays. In particular, it can be used to describe the deviation of the light rays from the ideal, desired path in an optical system. Figure 1.3 shows the deviations of the wavefronts in real optical systems from the ideal, desired wavefront. In the first case, incoming parallel light rays are focused by an imperfect lens which does not produce a perfectly spherical wavefront, resulting in a defocused image since the light rays do not converge all to the same point. In the second case, light rays emanating from a point source form a spherical wavefront, which is then collimated by a lens. The resulting wavefront is not perfectly planar, meaning that the emerging rays do not propagate exactly parallel to each other. Measuring the resulting wavefronts therefore gives insights into the imperfections of an optical system. There are many ways of measuring wavefronts, one of the simplest and most classical ones is the Shack-Hartmann sensor [4, 5]. This setup uses a two-dimensional array of lenses (called lenslets) of the same focal length. If the incoming wavefronts are perfectly planar, each lenslet focuses the incoming light and produces a regular pattern of illuminated spots. If however the incoming wavefronts are not planar, the resulting spot pattern is irregular and contains information about the shape of the wavefronts. This is easier explained by an image, and is illustrated in figure 1.4.

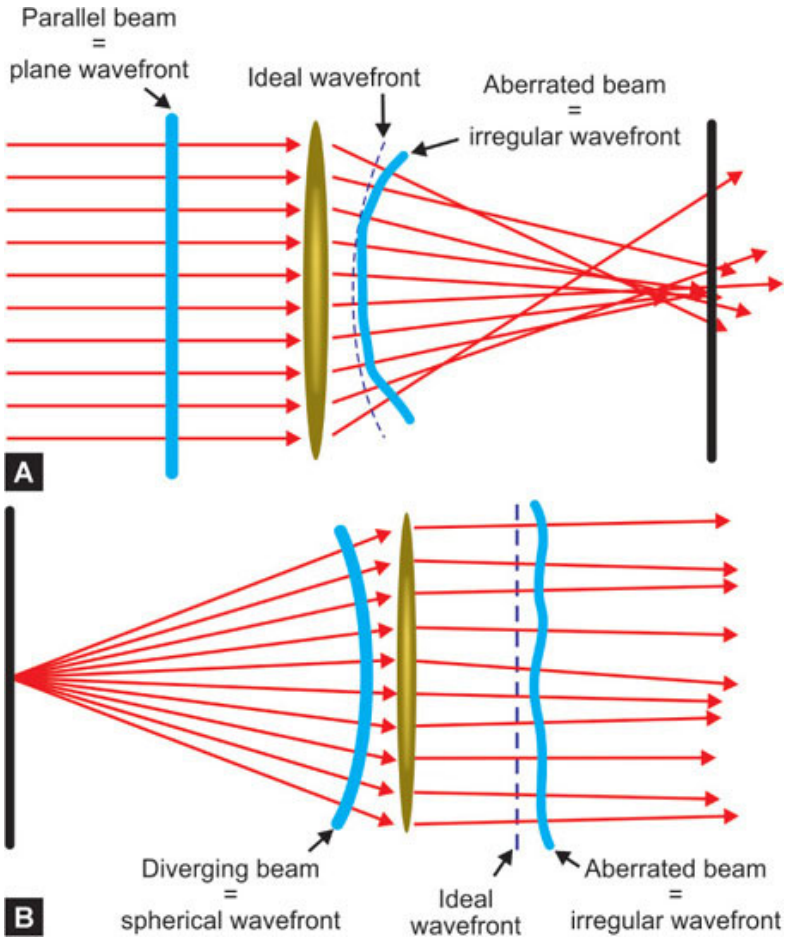

Figure 1.3: Wavefront aberrations. Image from Ortholibrary [3]

.

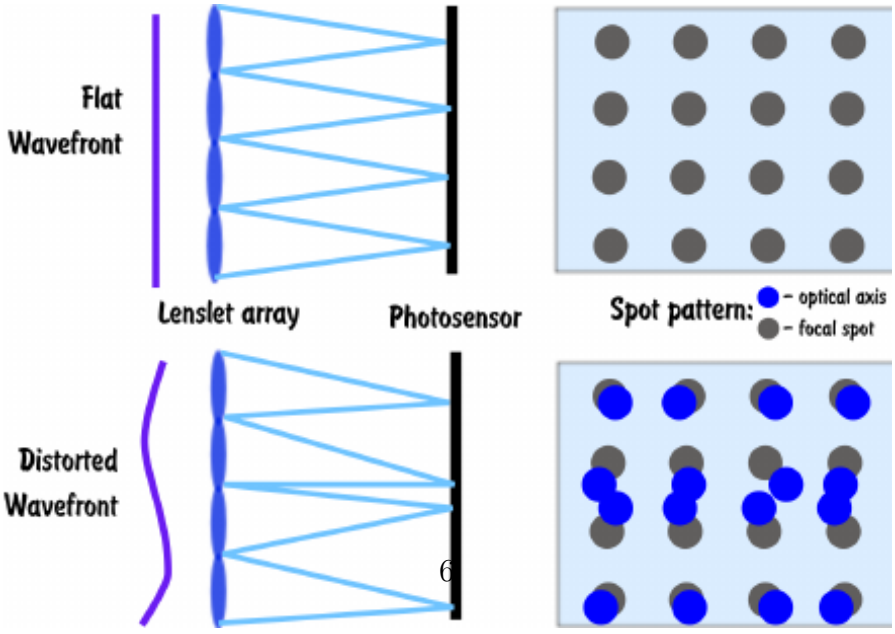

Figure 1.4: Shack-Hartmann sensor. Image from IEEE [6].

The main disadvantage of the Shack-Hartmann sensor is that the array of lenslets drastically reduces the resolution of the image, resulting in a resolution of as many pixels as there are lenslets. A more advanced setup for wavefront imaging which overcomes this problem is used and developed by Wooptix to measure the shape of transparent or reflecting objects. For the case of a reflecting object, this setup shines collimated light (planar wavefronts) onto a reflecting object. If this object were perfectly flat, the reflected light would have again planar wavefronts. But if its surface is curved, the light will be reflected in different directions depending on the point of the surface. These reflected rays are then measured in two different planes, as illustrated in figure 1.5.

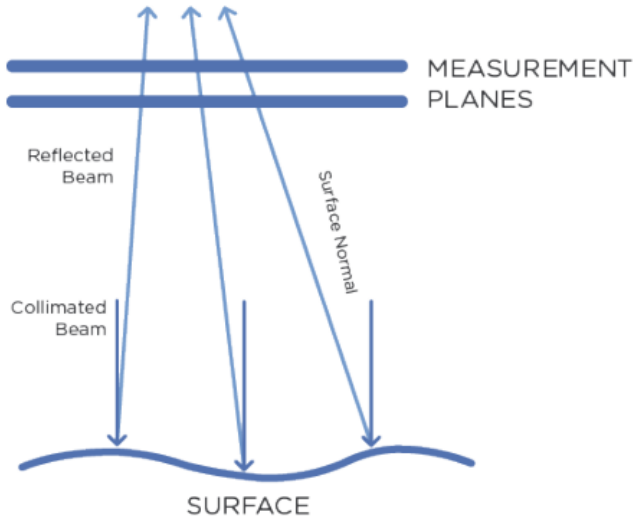

Figure 1.5: Wavefront sensor with two measurement planes. Image by Wooptix.

In the case depicted in the figure, since the rays are converging, the image produced in the second plane will be smaller but brighter than in the first plane, since the area covered by the rays is smaller, but the number of rays is the same. From the difference in size and intensity of the two images, the path of the light rays and therefore the incoming wavefront can be calculated, using for example the transport of intensity equation [7]. This in turn gives precise information on the shape of the reflecting surface. A similar setup can be applied to transparent instead of reflecting objects. This technology has many applications, ranging from the measurement of imperfections on silicon wafers used in the microchip industry to diagnosing aberrations in the human eye.

Zernike polynomials

In many applications it is of interest to encode the complicated wavefront information in a concise way, and to be able to separate large-scale deformations from small-scale structures. In the case of (nearly) circular objects or optical pupils, this is conventionally and conveniently done by expanding the resulting wavefront image in Zernike polynomials. These polynomials, named after optical physicist Frits Zernike [8], form a complete and orthogonal set of functions on the unit disk. Therefore any well-behaved function on the disk can be expanded in Zernike polynomials, analogously to how any periodic function can be expanded in sine and cosine waves using a Fourier series. This permits to encode the wavefront information in a comparatively small set of Zernike coefficients, truncating the series expansion at a given order of interest. Furthermore it separates the large-scale deformations which are encoded in the low Zernike polynomials from the small-scale details captured by the higher polynomials. Figure 1.6 shows the first 21 Zernike polynomials, ordered vertically by radial degree and horizontally by azimuthal degree.

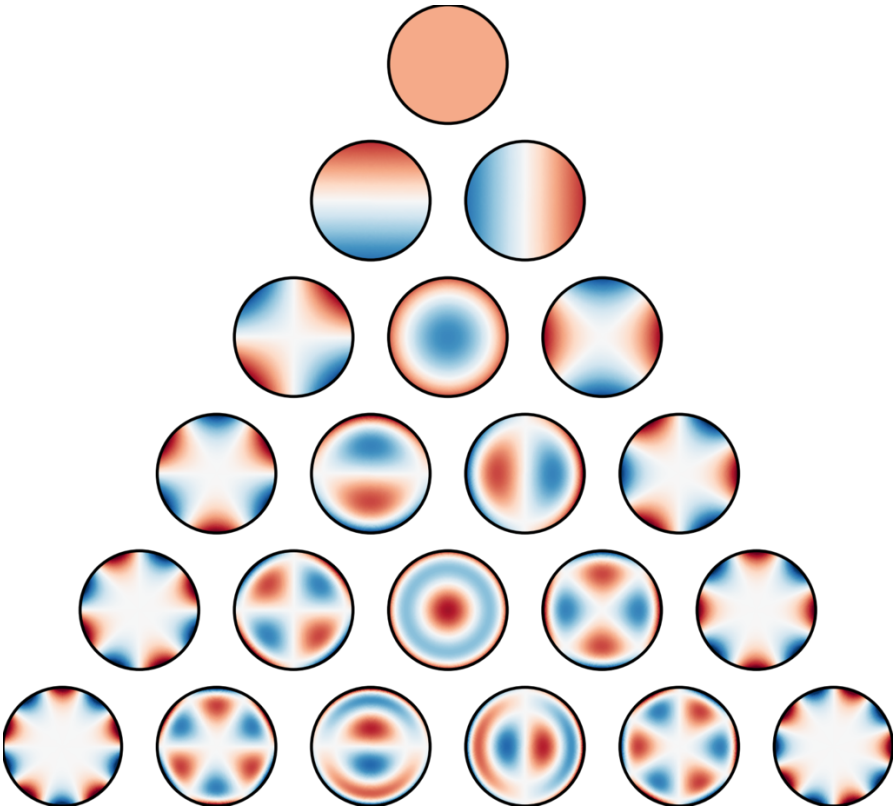

Figure 1.6: The first 21 Zernike polynomials. Image from Wikipedia [9].

Mathematically, the Zernike polynomials are defined as follows. There are even and odd polynomials, with the even polynomials defined as

$$Z_n^m(\rho,\phi) = R_n^m(\rho)\cos(m\phi)\,, \tag{1.1}$$

including the case $m = 0$, and the odd ones defined as

$$Z_n^{-m}(\rho,\phi) = R_n^m(\rho)\sin(m\phi)\,. \tag{1.2}$$

Here m and n are non-negative integers bounded by $n \geq m \geq 0$ with the condition $n - m$ an even number. For $n - m$ odd, the Zernike polynomials vanish. ρ is the radial coordinate on the unit disk, i.e. $0 \leq \rho \leq 1$ and ϕ is the azimuthal angle. The radial polynomials $R_n^m(\rho)$ are defined as

$$R_n^m(\rho) = \sum_{k=0}^{\frac{n-m}{2}} \frac{(-1)^k (n-k)!}{k!\left(\frac{n+m}{2}-k\right)!\left(\frac{n-m}{2}-k\right)!}\rho^{n-2k}\,. \tag{1.3}$$

As mentioned above, this is only defined for $n - m$ an even number, and vanishes otherwise. In applications it is often useful to use a single index j instead of the two indices n and m. In the following we will use the so-called OSA/ANSI standard indices. First, we define an azimuthal index l which covers both the even and odd cases by $m = |l|$, i.e. $n \geq l \geq -n$. Then the single-index j is given by

$$j = \frac{n(n+2)+l}{2}\,, \quad j \geq 0\,. \tag{1.4}$$

The importance of the Zernike polynomials comes from their orthogonality and completeness. Orthogonality is expressed by the integral of two polynomials over the unit disk:

$$\int_{disk} dA\, Z_n^l(\rho,\phi) Z_{n'}^{l'}(\rho,\phi) = \frac{\pi\epsilon_l}{2n+2}\delta_{nn'}\delta_{ll'}\,, \tag{1.5}$$

where ϵ_l is the so-called Neumann factor, taking the value 2 if $l = 0$ and 1 otherwise. Completeness means that any sufficiently smooth real-valued function f over the unit disk can be expressed in terms of its Zernike coefficients, in the same way as periodic functions can be expanded in Fourier series. Mathematically, in terms of the above single-index j we can write

$$f(\rho,\phi) = \sum_{j=0}^{\infty} c_j\, Z_j(\rho,\phi)\,. \tag{1.6}$$

Using the orthogonality relations (1.5), the coefficients can be calculated as inner products, i.e. integrals over the disk:

$$c_{j(n,l)} = \frac{2n+2}{\pi\epsilon_l}\int_{disk} dA\, f(\rho,\phi) Z_n^l(\rho,\phi)\,. \tag{1.7}$$

The normalization factor is expressed in terms of the radial and azimuthal indices n, l, so one has to express the single-index j in terms of those.

This work

The problems that are treated in this thesis can be divided into three parts. In the first part, we develop an algorithm to compute the Zernike coefficients c_j given a circular wavefront image. Previous algorithms developed by the group at Wooptix use a least-squares approach to fit the coefficients. This works well when the number of mode coefficients that are computed is close to the number of modes that are actually present in the image. However, if the image contains many high modes, and we are interested in extracting the lower coefficients, the least-squares computation of the low modes tries to fit also the high modes. This overfitting considerably worsens the performance. Instead, we opt for a direct approach and compute the coefficients using equation (1.7). We first develop a prototype in python, and then port it to C++. To further improve the performance, we parallelize the computations and perform them on a GPU. In chapter 2 we develop such an algorithm in CUDA and implement a python binding to be able to call the GPU functions in a simple way from a python program.

In the second part we start treating more realistic cases. In the application to measurements of silicon wafers, the images taken are not perfectly circular. Instead, the silicon wafer is held by three clips, which don't reflect light and produce dark indentations in the circle. This deviation from a perfect circle considerably lowers the accuracy of the developed algorithms which assume a circular image. Since the indentations are comparatively small, in chapter 3 we solve this problem by implementing a functionality that fills in the missing parts via barycentric interpolation. We evaluate its performance and find great improvement.

In the third and final part we consider a different application. Modern wavefront sensing technology allows to measure aberrations in the human eye with great precision, which is of interest in ophthalmological applications. The problem here is similar to the one before: human pupils are never perfectly circular. The difference lies in that we no longer have a circle with a few small indentations, but real deformations. In many cases these deformations are so pronounced that it no longer makes sense to consider the Zernike basis. The mathematically correct thing to do is to develop a new basis for each non-circular pupil. We do this in chapter 4 by developing a map between an arbitrary pupil (with some reasonable restrictions) and the disk, inspired by the algorithm described in [10]. We then simply take the usual Zernike basis functions on the disk and map it to the given pupil shape to obtain a new, adapted basis.

Chapter 2

Basic Zernike Fitter

In this chapter we describe the development of a simple Zernike fitter, which given a perfectly circular image extracts a given number of Zernike coefficients.

Python

We first develop a prototype in python. By putting the problem on a grid and therefore discretizing the integrals in (1.5) and (1.7), it reduces to a problem in linear algebra, which can easily be implemented using *numpy.sum* to compute the integrals. The main source of errors comes from the fact that when one leaves the continuum, the orthogonality relation (1.5) no longer holds exactly. However, this is not a problem for the applications that we have in mind since we will work with images with very high resolution. In this prototype code we also implement a least-squares fitter using the function *numpy.linalg.lstsq* apart from the direct fitter which uses (1.7) in order to compare both methods. We do not further describe the python code since it is straightforward and not particularly interesting.

C++

The C++ version is more interesting and provides a considerable speedup with respect to the python version. It is still fully serial and will be further improved upon by the CUDA version in the next section but it is useful in order to lay the foundation for the parallel version and can be used on machines that do not support CUDA. The basic object is a structure which we call *zernikeFitter*, with the following member variables and functions:

```
1 struct zernikeFitter{
2   int res; // resolution of the grid
3   int nmax; // maximum n index
```

```
  int lmax; // maximum l index
  int nmodes; // number of modes
  double* coefs; // coefficients
  double* wf; // wavefront (phase)
  double* rptable; // table of r^p
  double* phitable; // table of phi
  double* factorial; // factorial lookup table
  zernikeFitter(int res, int nmodes); // constructor
  ~zernikeFitter(); // destructor
  double* getCoeffs(double* wf); // get the coefficients
  double* getPhase(double* coefs); // get the wavefront
  void makePolartable(int res, int nmax, double* rptable, double*
   phitable); // make polar coordinates lookup table
};
```

Each instance of the zernikeFitter structure works on a square grid with a given resolution and an inscribed circle. In the constructor we also specify the maximum number of Zernike modes that we are interested in (corresponding to the single-index j defined in 1.4). The maximum radial and azimuthal indices n and l are then automatically calculated. We also allocate memory for the wavefront (the image) and the Zernike coefficients. More interestingly, we have three lookup tables: since we are interested in extracting many coefficients, of the order of $j \gtrsim 66$ we need to compute all of the corresponding polynomials, see (1.2, 1.3). Since their evaluation repeatedly involves computing factorials, powers of the radius and trigonometric functions of the angle, we compute them once and store their values in lookup tables. This considerably speeds up the computations when one deals with many modes. It is straightforward but worth mentioning that to make these lookup tables, we first need compute the maximum factorial and maximum power of the radius that we need, given *nmodes*.

The two main member functions are *getCoeffs*, which returns the Zernike coefficients (up to *nmodes*) of a given image *wf*, and *getPhase* which returns a wavefront (image) given some coefficients.

To compute the coefficients we simply loop over all modes and over each point of the grid. For a given mode and point we then compute the value of the Zernike polynomial using the corresponding lookup tables, multiply it by the value of the wavefront and the appropriate normalization factor:

```
double* zernikeFitter::getCoeffs(double* wf){
  // loop over all modes
  for (int n = 0; n <= nmax; n++){
    for (int l = -n; l <= n; l+=2){
      if (n == nmax && l > lmax) break; // break when all modes
   are done
      double N = 4.0*(2*n+2)/(M_PI*(res-1)*(res-1)); //
   normalization factor
      if (l == 0) N /= 2.0; // Neumann factor
```

```
      size_t idx = ZernIFromNL(n, l); // OSA index
      coefs[idx] = 0.0; // initialize coefficient

      // loop over mesh and perform integration
      for (int i = 0; i < res*res; i++){
        // get polar coordinates
        double r = rptable[i+res*res];
        double phi = phitable[i];
        double zvalue = 0.0;

        // set value to zero if outside unit circle
        if (r > 1.0) continue;

        // calculate value of Zernike polynomial at this point
        int m = abs(l);
        int sign = -1;

        for ( int s = 0; s <= (n-m)/2 ; s++ ){
          sign *= -1;
          zvalue += sign * factorial[n-s] / ( factorial[s] *
    factorial[(n+m)/2-s] * factorial[(n-m)/2-s] ) * rptable[i+(n-2*
    s)*res*res];
        }

        if (l < 0) zvalue *= sin(m*phi);
        else zvalue *= cos(m*phi);

        // add to coefficient
        coefs[idx] += N*zvalue*wf[i];
      }
     }
   }
   return coefs;
};
```

Similarly for *getPhase*. We also implement python bindings for these two member functions using *pybind11* [11] and its *array_t* template. In this way we can compile the C++ code as a library, import it as a python module and use these two functions in a simple way inside a python code.

CUDA

Next, we turn to the CUDA version of the Zernike fitter. Since we work with images and most computations are pixel-wise, this problem lends itself to parallelization on a GPU. The general idea is the same as before, just that now many functions are implemented as CUDA kernels. To compute the coefficients for example, we

still loop serially over all modes. For each mode, the pixel-wise operations of computing the Zernike polynomial at a given point and multiplying it by the wavefront are done by CUDA kernels. Also the final reduction, i.e. the sum over all grid points corresponding to the discretization of the integral in (1.7) is done by a kernel function. The fact that we loop serially over the modes, instead of performing the computation of all modes in parallel is due to memory bounds. As a rough estimate, the computation of a single mode for an image with a resolution of 6000×6000 pixels involves at least the memory needed to store the values of the polynomials at all grid points, which at double precision amounts to 288 MB. Computing all modes in parallel would exceed the 5928 MB memory capacity of the NVIDIA GeForce GTX 1660 SUPER GPU that I am using for as little as 21 modes, even without taking into account the memory needed for the lookup tables. Furthermore, since the number of CUDA cores of this GPU is 1408, the threads would be scheduled to run sequentially anyways.
The first step is to allocate the necessary memory for the coefficients, the wavefront, the Zernike polynomials and the lookup tables on the GPU. Then, in the case of computing the coefficients, we need to copy the wavefront image from host to device. To optimize the code, this memory copy is overlapped with the independent computation of the lookup tables for the angle and the powers of the radius by using two different CUDA streams. Finally, the computation of the coefficients looks as follows:

```
// loop over all Zernike modes
for (int n = 0; n <= nmax; n++){
  for (int l = -n; l <= n; l += 2){
    if (n == nmax && l > lmax) break; // break when all modes are done
    double N = 4.0*(2*n+2)/(M_PI*(res-1)*(res-1)); // normalization factor
    if (l == 0) N /= 2.0; // Neumann factor
    size_t idx = ZernIFromNL(n, l); // OSA index

    // make zernike polynomial
    makeZernike<<<nb,nth>>>(n, l, res, fact_gpu, rptable, phitable, zernike);

    // multiply zernike with wavefront on GPU (result is stored in zernike again)
    zxwf<<<nb,nth>>>(res, N, zernike, wf_gpu);

    // reduce result on GPU (sum over all pixels) and copy to host
    int nblocks_red = nb/2;
    int idx_red = 0;
    int size = res*res;
```

```
19      reduce<<<nblocks_red,nth,nth*sizeof(double)>>>(zernike,
    tmp_gpu[idx_red], size);
20      while (nblocks_red > 2){ // reduce until only two blocks are
     left (several kernel calls are necessary for block
    synchronization)
21          size = nblocks_red;
22          nblocks_red = (nblocks_red+2*nth-1)/(2*nth);
23          reduce<<<nblocks_red,nth,nth*sizeof(double)>>>(tmp_gpu[
    idx_red], tmp_gpu[(idx_red+1)%2], size);
24          idx_red = (idx_red+1)%2;
25      }
26      size = nblocks_red;
27      nblocks_red = (nblocks_red+2*nth-1)/(2*nth);
28      reduce<<<nblocks_red,nth,nth*sizeof(double)>>>(tmp_gpu[
    idx_red], &coefs_gpu[idx], size);
29     }
30  }
```

The kernels are called with the maximum number of threads available per block, in our case $nth = 1024$ and the minimum number of blocks such that $nb * nth \geq res * res$ in order to fit all pixels. The kernel *makeZernike* computes the Zernike polynomial at each point and stores it in the device array *zernike*. The kernel *zxwf* then multiplies the Zernike with the wavefront. The non-trivial part is hidden in the *reduce* kernel: we need to sum the result of the multiplication for all pixels and therefore need to communicate between threads. Since thread synchronization works most efficiently on the block-level, we first reduce all pixels inside each block and then repeat the procedure on the result until we are left with just a single block which completes the reduction. The specifics of the parallel reduction kernel are taken from an NVIDIA webinar on parallel reductions [12] and look like the following:

```
1  __global__ void reduce(double *g_idata, double *g_odata, int size,
       int stride=1) {
2    extern __shared__ double sdata[];
3    // each thread loads one element from global to shared mem
4    unsigned int tid = threadIdx.x;
5    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
6    // check if index is in bounds
7    if (i >= size){
8      sdata[tid] = 0.0;
9      return;
10   }
11   sdata[tid] = g_idata[i*stride];
12   if (i + blockDim.x < size) sdata[tid] += g_idata[(i+blockDim.x)*
      stride];
13   __syncthreads();
14   // do reduction in shared mem
15   for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
```

```
16      if (tid < s) sdata[tid] += sdata[tid + s];
17      __syncthreads();
18    }
19    if (tid < 32) warpReduce(sdata, tid); // unroll last warp
20    // write result for this block to global mem
21    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
22  };
```

Let us elaborate on this code snippet. Since the threads in the reduction repeatedly access the memory, we use a shared memory array *sdata*, whose size is specified when calling the kernel, and is equal to the number of threads in each block (times $sizeof(double)$). In order to avoid divergent branches and shared memory bank conflicts as discussed on pages 9-12 in [12], we use a sequential addressing scheme for the threads instead of interleaved addressing (see figure 2.1).

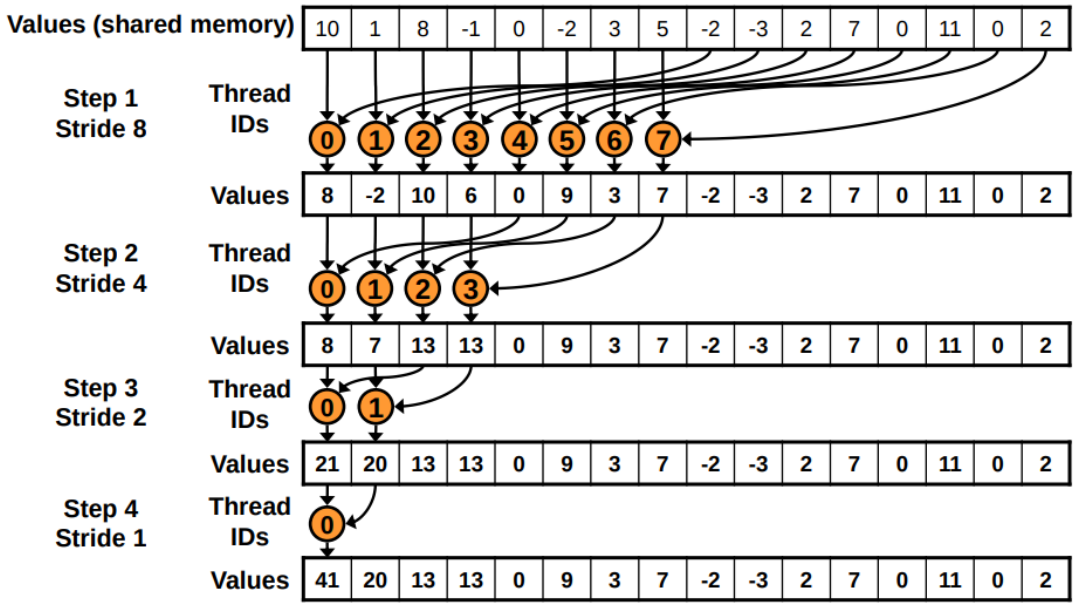

Figure 2.1: Sequential addressing scheme. Image from NVIDIA [12].

The kernel can perform reductions with strided memory accesses which will be needed later. Considering for now the default case of *stride=1* for simplicity, each thread then loads the corresponding input data value into its shared memory slot and (if in bounds) directly adds the next value in our indexing scheme. This prevents half of the threads being idle in the first iteration as in figure 2.1: in step 1 the second half of the threads performs no operations after loading the values into the shared memory. We then synchronize the threads and repeat the procedure, reducing the offset for the addition by powers of two until we reach

32. At this point, we are left with a single warp and can further optimize the reduction: since instructions are SIMD synchronous within a warp, we don't need to *__syncthreads()* and we don't need the *if (tid<s)* branch, since it would not save any work. We can simply unroll the last six steps of the loop and write

```
__device__ void warpReduce(volatile double* sdata, int tid) {
  sdata[tid] += sdata[tid + 32];
  sdata[tid] += sdata[tid + 16];
  sdata[tid] += sdata[tid + 8];
  sdata[tid] += sdata[tid + 4];
  sdata[tid] += sdata[tid + 2];
  sdata[tid] += sdata[tid + 1];
}
```

The *volatile* keyword prevents the compiler from keeping *sdata[tid]* in register, which would break the implicit SIMD memory synchronization. Finally, after the warp reduction, the last step is to have thread 0 write its result into the corresponding output memory location. Since this reduction kernel just reduces all values inside each block, we need to keep calling the kernel on the result until we are left with a single block and have completed the reduction of the full grid. The *getPhase* function works in a similar way, but in this case we do not need to perform any reduction. As before, we also implement a python binding for these functions using *pybind11* [11].

Results and benchmarks

Let us show an example of the results of the Zernike fitter developed above. We generate a synthetic wavefront from 91 known, randomly sampled Zernike coefficients and use the Zernike fitter to compute the first 66 coefficients from the resulting image. These two numbers are triangular numbers, meaning that for the given radial order of the Zernike polynomials we take all the possible angular modes. In figure 2.2 we plot the original wavefront generated from 91 coefficients, the wavefront generated from the lowest 61 modes whose coefficients were extracted by the Zernike fitter, the difference of the two images, which is non-zero because we did not fit the high modes, and the difference between the fitted wavefront and the wavefront generated from only the first 66 original coefficients.

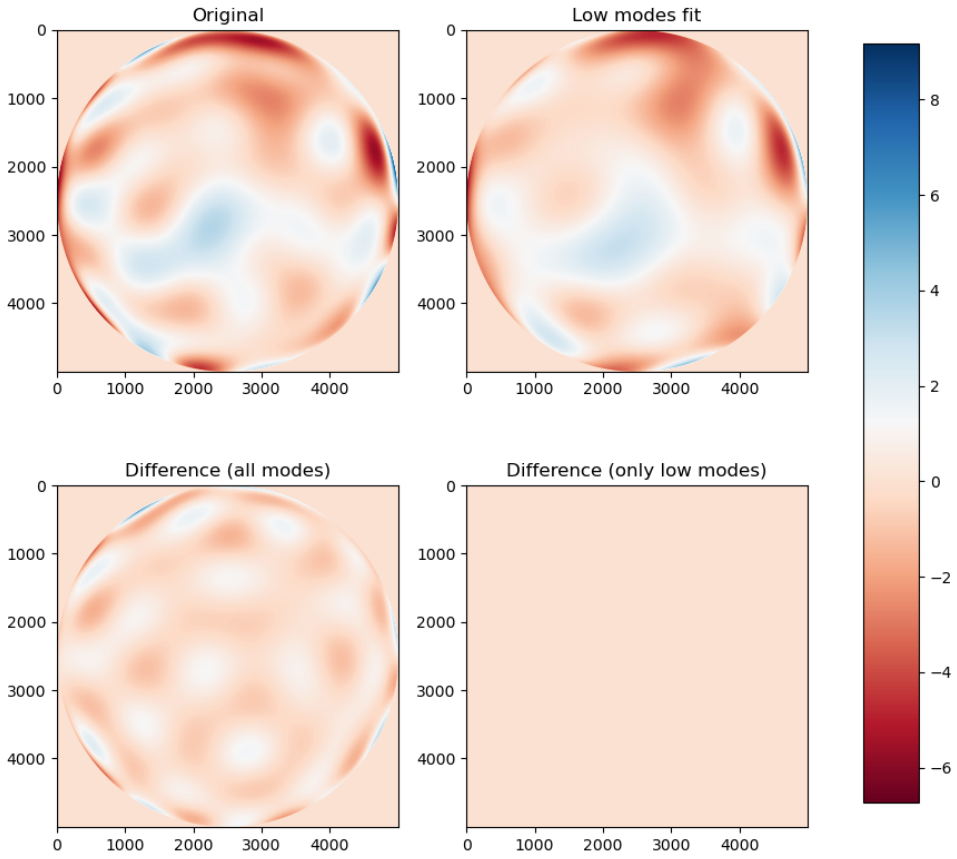

Figure 2.2: Top left: synthetic wavefront generated from 91 coefficients. Top right: wavefront generated by 66 coefficients, computed from the first. Bottom left: difference between the two images. Bottom right: difference between the top right image and the equivalent of the top left, using only the first 66 coefficients.

In figure 2.3 we plot the values of the original coefficients, the computed coefficients and their difference, for all modes and for only the modes that have been computed.

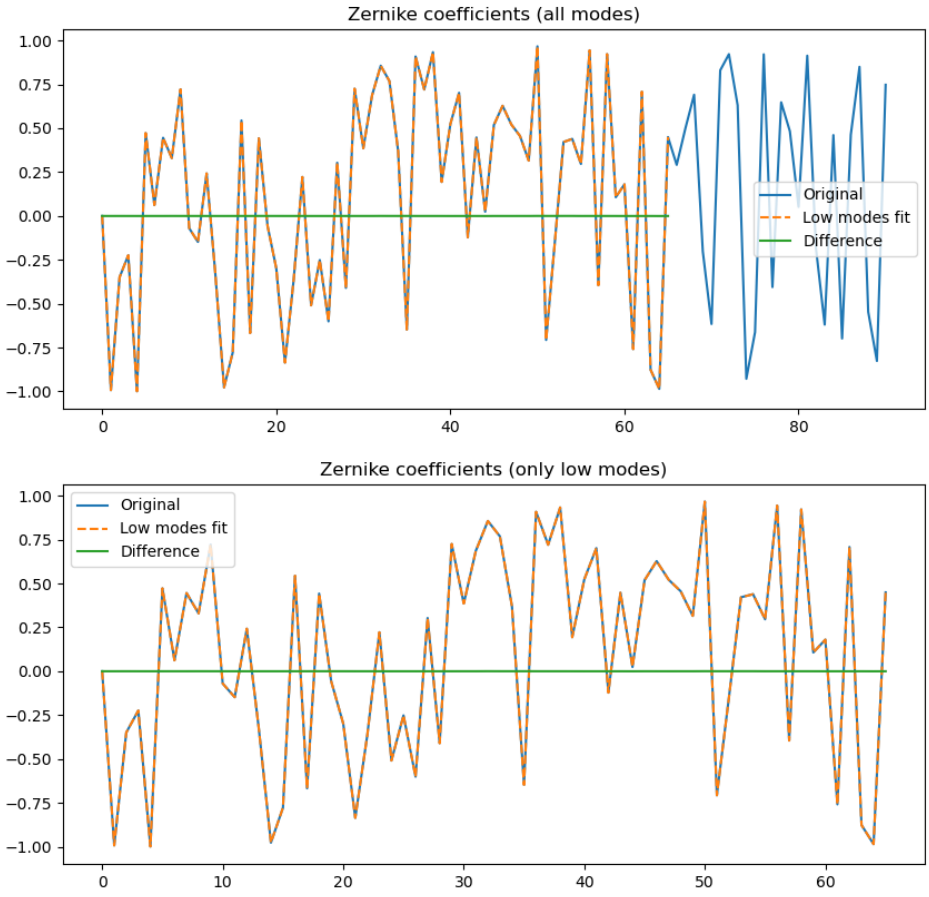

Figure 2.3: Original Zernike coefficients used to generate the wavefront in figure 2.2, Zernike coefficients computed by the Zernike fitter, and their differences.

We see that at least qualitatively, the Zernike fitter developed in the previous section does a good job at extracting the correct coefficients. We now assess its performance more quantitatively, considering four different versions:

1. Python (least squares)
2. Python (direct)
3. C++ (direct)

4. CUDA (direct)

The first computes the Zernike coefficients using the least squares approach, while the others use the direct approach, assuming the discretized Zernike polynomials form an orthogonal basis. We analyze the time to fit and the normalized RMS error. The original Zernike coefficients (c_j) used to generate the wavefront are sampled randomly and uniformly from the interval $[-1, 1]$. If we denote the fitted coefficients by c'_j, we define the normalized RMS error as

$$NRMS = \frac{\sum_j (c'_j - c_j)^2}{\sum_j c_j^2} \,. \tag{2.1}$$

Fixed resolution, varying number of modes

First we consider a fixed resolution of 1024×1024 pixels which is about the highest resolution for which the python codes run in a reasonable time. We generate a synthetic wavefront with 66 modes with randomly sampled coefficients and then use our Zernike fitter to extract a varying number of coefficients, from 3 to 36, and compare their values with the original known coefficients. For each number of extracted modes, we generate 50 different wavefronts. In figures 2.4 and 2.5 we plot the NRMS error and the logarithm of the time that it takes to compute the coefficients versus the number of modes that are fitted.

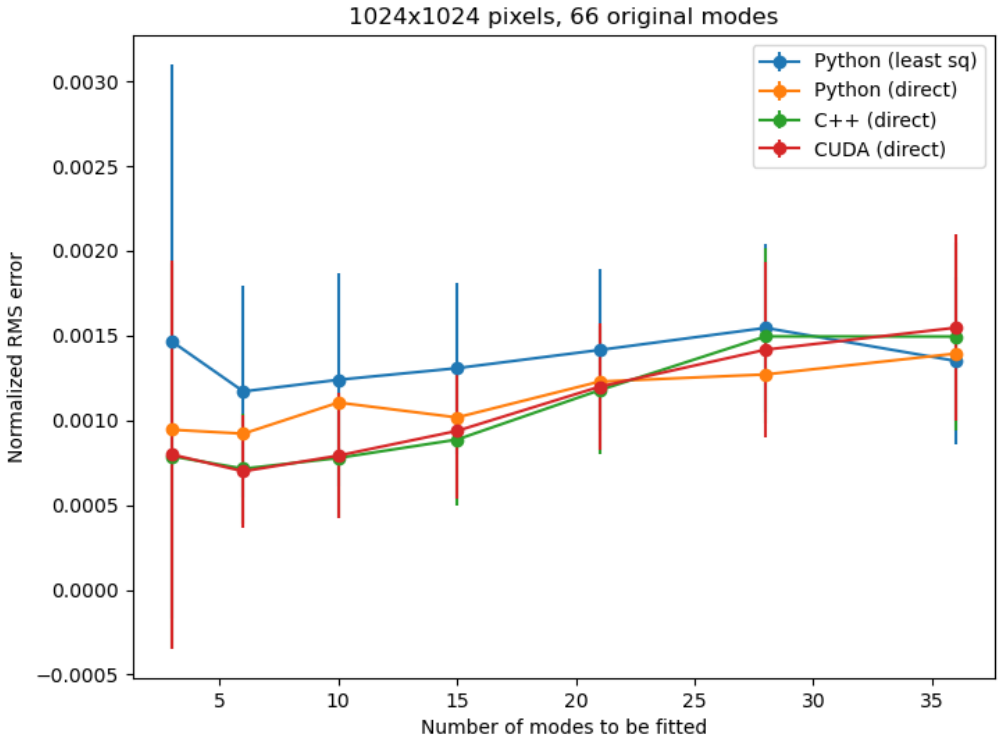

Figure 2.4: NRMS error vs number of fitted modes. Points represent the mean value over 50 runs and the error bars the standard deviation.

We see that all four versions have similar NRMS errors. For a low number of modes, the least-squares approach is the worst, but it gets better as we approach the number of original modes contained in the wavefront. The standard deviation indicated by the error bars is quite large since for each data point we consider 50 different wavefronts.

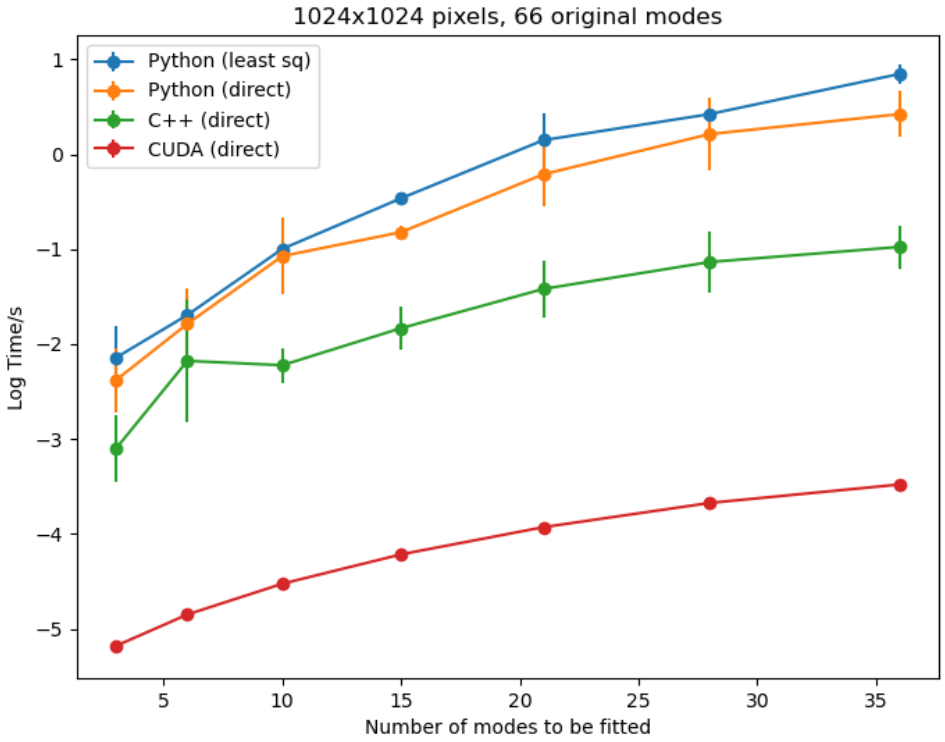

Figure 2.5: Logarithm of time needed to extract coefficients vs number of fitted modes. Points represent the mean value over 50 runs and the error bars the standard deviation.

This plot nicely showcases the speedups between the different versions: the least-squares approach is computationally more expensive since it involves computing the pseudo-inverse of a matrix. The direct approach in python is therefore slightly faster, while also being more accurate for a low number of modes as represented in 2.4. The C++ version is about an order of magnitude faster, while the CUDA versions speeds up the process by about another two orders of magnitude.

Varying resolution, fixed number of modes

Next we consider a fixed number of modes and vary the resolution from 128 to 4096 for the C++ and CUDA versions, while for the python version we stop at 2048 since for higher resolutions the computation time becomes unfeasible. For each value of the resolution, we again generate 50 different wavefronts from 66 known, randomly sampled coefficients. We then extract the 36 lowest coefficients

and compare them with the original ones, plotting the logarithm of the NRMS error and the logarithm of the computation time.

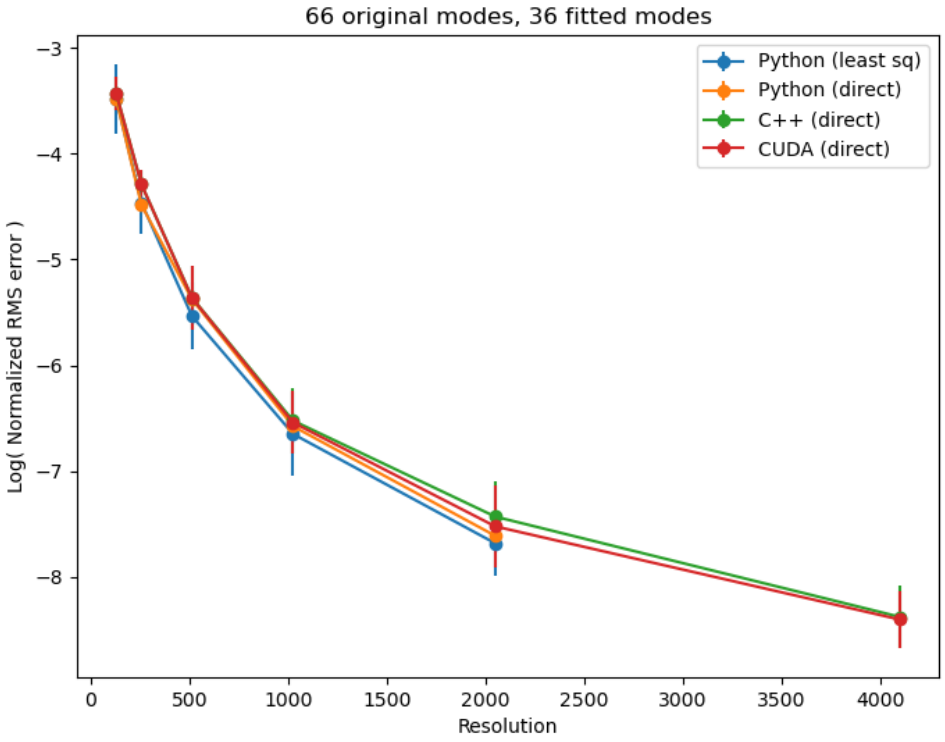

Figure 2.6: Log NRMS error vs resolution of image. Points represent the mean value over 50 runs and the error bars the standard deviation.

Note that this plot is logarithmic in the y-axis. We appreciate that the NRMS error drops sharply with increasing resolution, as expected: the main source of error is the coupling between Zernike modes (i.e. the loss of orthogonality) due to the discretization of the grid. This error quickly decreases as we increase the resolution and approach the continuum.

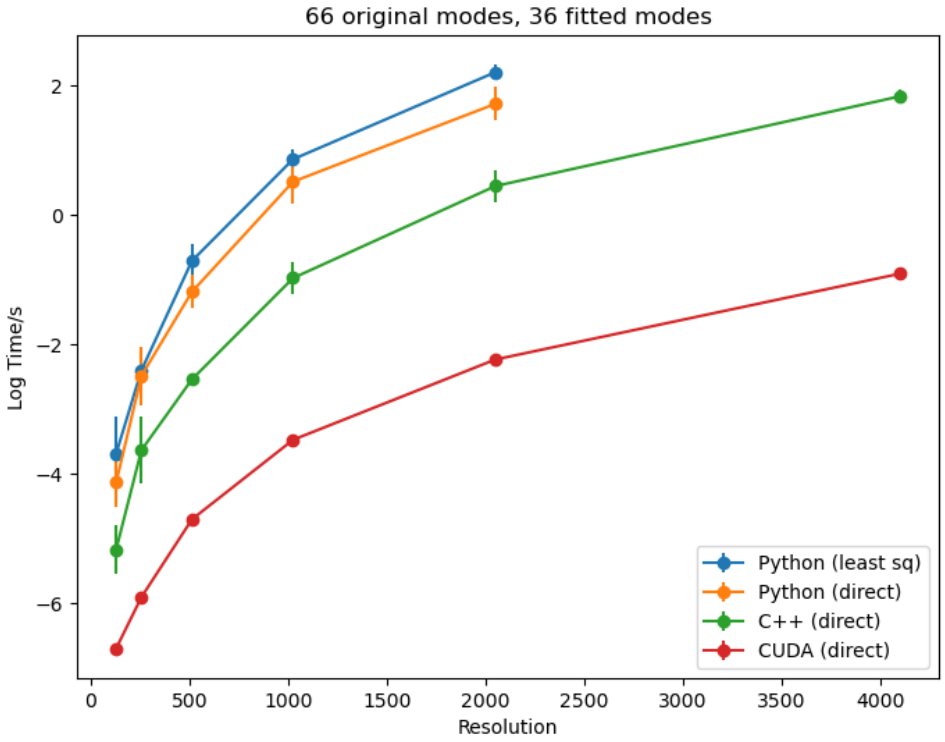

Figure 2.7: Logarithm of time needed to extract coefficients vs resolution of image. Points represent the mean value over 50 runs and the error bars the standard deviation.

This plot of the computation time again highlights the speedups gained by passing from the least-squares to the direct approach in python and then to C++ and CUDA.

Memory usage

We now consider the memory usage of our codes as a function of the resolution and of the number of modes to be fitted. For the "fast" direct version, i.e. where we keep a lookup table for the powers of the radius in memory as described in the previous section, the memory usage is plotted below in figure 2.8. This approach is quite economical in terms of memory usage since we just need to store a few lookup tables and one Zernike polynomial at a time.

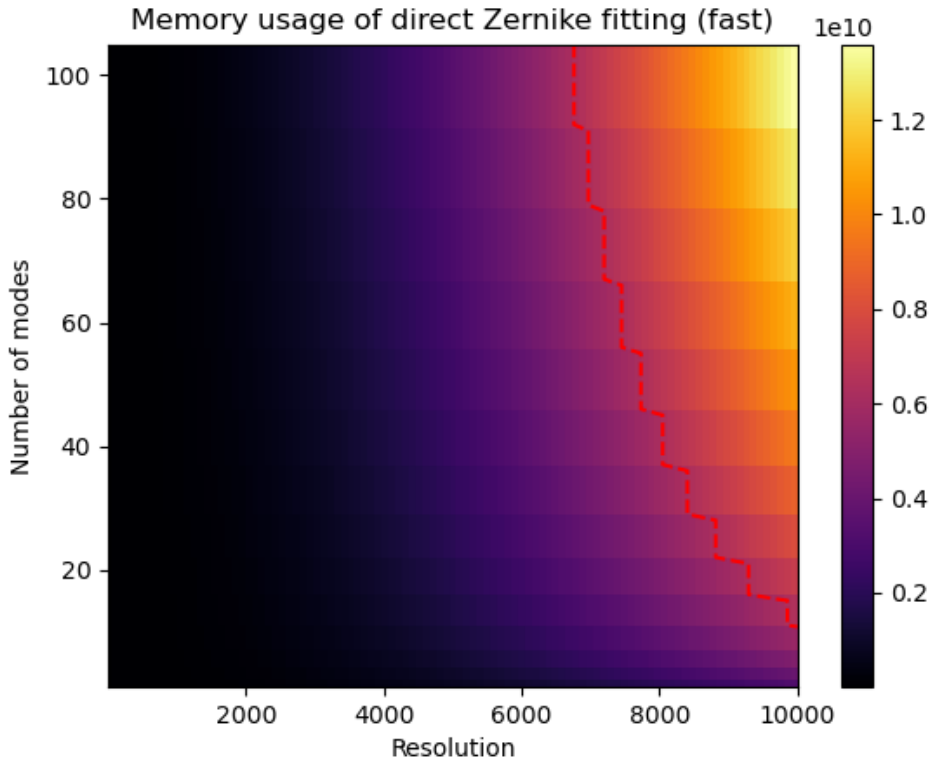

Figure 2.8: Memory usage of the "fast" direct Zernike fitter as a function of the resolution and the number of modes to be fitted. The color bar indicates memory usage in bytes.

The red line indicates the 5928 MB memory limit of the NVIDIA GeForce GTX 1660 SUPER GPU that I am using and the stepped structure comes from the fact that the number of powers of the radius that we need to store in memory scales with the radial index n and is independent of l and therefore jumps at each triangular number. We see that at the typical resolution of the images that we are interested in for applications, which is about 6000, we can extract more than 100 modes.
For completeness we also consider the memory usage of the least-squares fitter. Since in this case we need to store *all* relevant Zernike polynomials in memory in order to compute the pseudo-inverse matrix needed for the least-squares fit, the memory usage quickly increases with resolution and number of modes as plotted below in figure 2.9.

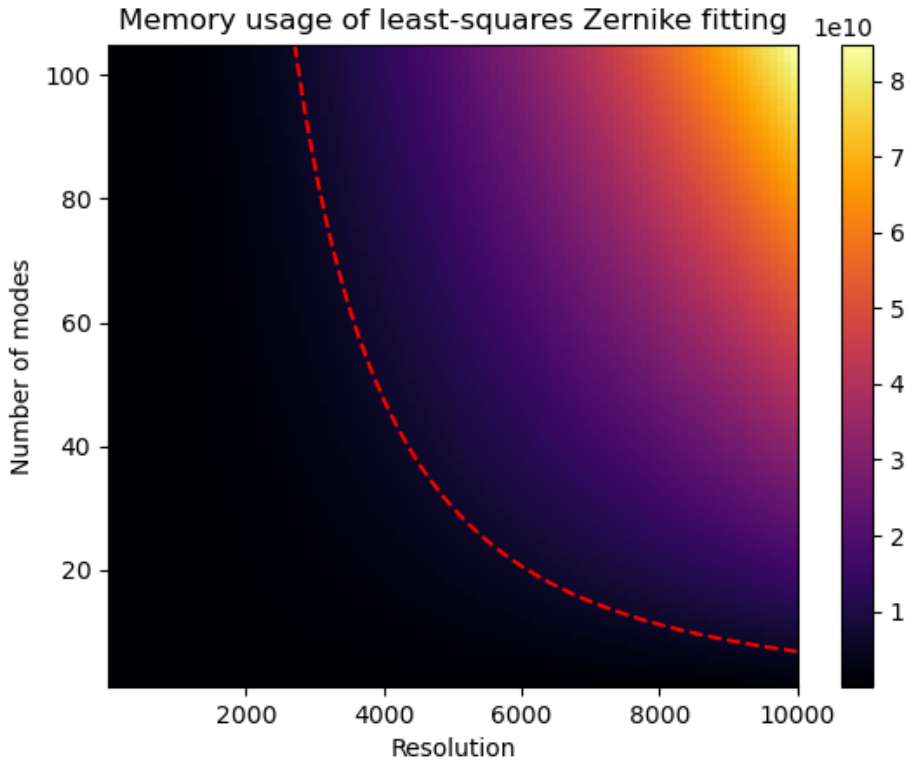

Figure 2.9: Memory usage of the least-squares Zernike fitter as a function of the resolution and the number of modes to be fitted. The color bar indicates memory usage in bytes.

Again, the red line indicates the 5928 MB memory limit of the NVIDIA GeForce GTX 1660 SUPER GPU that I am using. We see that for a moderate resolution and number of modes, the least-squares approach quickly becomes unfeasible due to the elevated memory usage, apart from the fact that it is slower than the direct approach.

Conclusion

All in all, we conclude that the direct approach is more suited for the applications we have in mind. First of all, because the typical images encountered in applications have many small-scale details and therefore a least-squares approach would produce a considerable overfitting when extracting only a limited number of low modes. Even if we would extract many modes, in which case the least-squares approach outperforms the direct approach, it is considerably slower due to an

elevated computational cost and uses much more memory, making it quickly practically unfeasible. On top of that, the computational error of the direct approach quickly drops as we increase the resolution. Between the three direct approaches, the CUDA version is the fastest, running in about 10^{-1} seconds for a resolution of 4096 and fitting 36 modes. This is more than sufficient for applications.

Chapter 3

Wafer indentations

Generalities

In this chapter we deal with the more realistic case of a concrete application. As mentioned in the introduction, one of the applications of the wavefront imaging technology is to silicon wafers. This is a thin, circular slice of crystalline silicon which is widely used in the semiconductor industry as a substrate for microelectronic devices and circuits which are etched or printed on top of the wafer. These processes require the wafers to be as flat and polished as possible. The advanced wavefront imaging technology of Wooptix permits capturing millions of data points in a few milliseconds with sub-nanometer height accuracy and 100 micrometer lateral resolution [13]. The resulting images contain information about both the global bow and deformation of the wafer and nanoscale polishing imperfections. An example is shown in figure 3.1, where the top right image shows the global bow of the wafer and the top center image highlights the polishing imperfections.

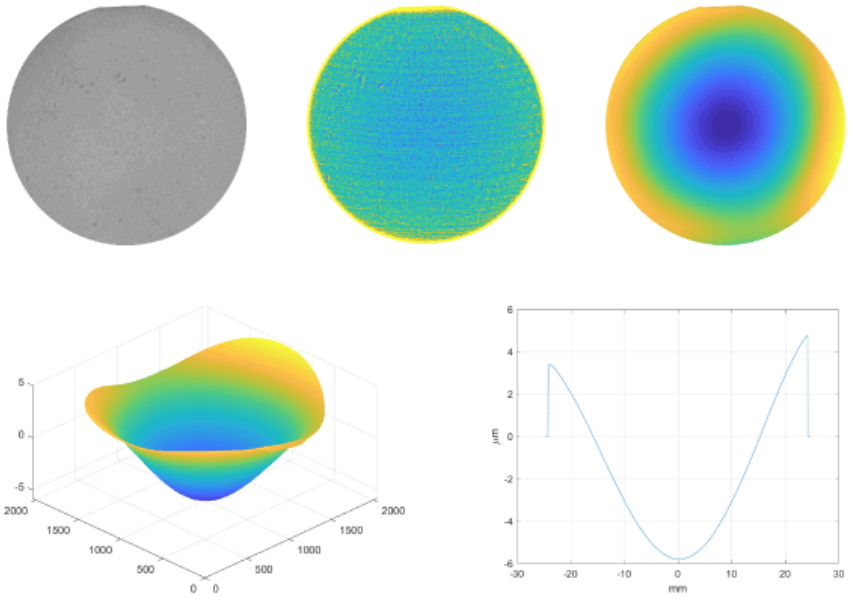

Figure 3.1: Wavefront image showing deformations of a silicon wafer, highlighting the global bow and polishing imperfections. Image by Wooptix.

To accurately separate the global, large-scale bow of the wafer from the nanoscale polishing imperfections, a mode decomposition is used. The Zernike fitter developed in the previous chapter is a first step towards this goal. However, its performance is significantly impacted by the fact that to capture the wavefront image of the wafer, it needs to be held in place by three small grips. These are non-reflecting and therefore produce dark indentations along the border of the wafer. An example is shown below.

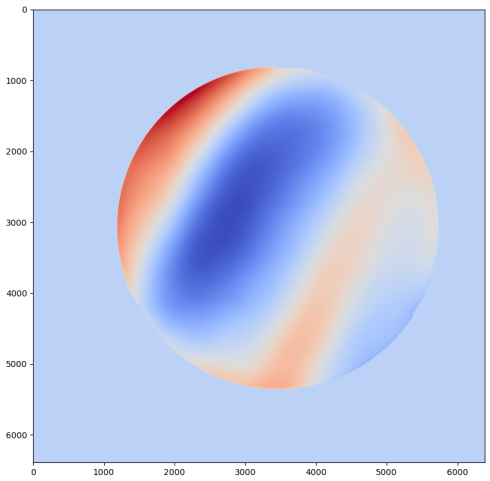

Figure 3.2: Wafer with indentations. Image by Wooptix.

The indentations can be appreciated at about 12, 4 and 8 o' clock. Albeit small, these indentations cover part of the wafer and produce spurious data points of zero elevation which misguide the Zernike fitter algorithm developed in the previous section, significantly altering the extracted mode coefficients. To overcome this problem, the straightforward solution is to try to recover the missing, covered parts of the image. We do so by barycentric interpolation, described in the next section.

Barycentric interpolation

Barycentric interpolation is a way to perform fast interpolation on arbitrary grids, see for example [14]. It extends linear interpolation to arbitrary dimensions, in particular to our two-dimensional case. To understand and perform the interpolation we first need to introduce the *barycentric coordinates* (see for example [15]). These coordinates can be used to specify the location of a point in the plane with respect to the three vertices of a triangle. Given a triangle with vertices located at $\vec{r}_i$, $i = 1, 2, 3$, the position $\vec{r}$ of any point in the plane of the triangle can be given as

$$\vec{r} = \sum_{i=1,2,3} \lambda_i \vec{r}_i \,, \qquad \text{with} \quad \sum_{i=1,2,3} \lambda_i = 1 \,. \tag{3.1}$$

The three numbers λ_i are referred to as the barycentric coordinates with respect to the triangle $\vec{r}_i$. All three coordinates are positive if and only if the point lies

inside the triangle, while they can be negative if it lies outside of it. Two examples are shown in figure 3.3 below.

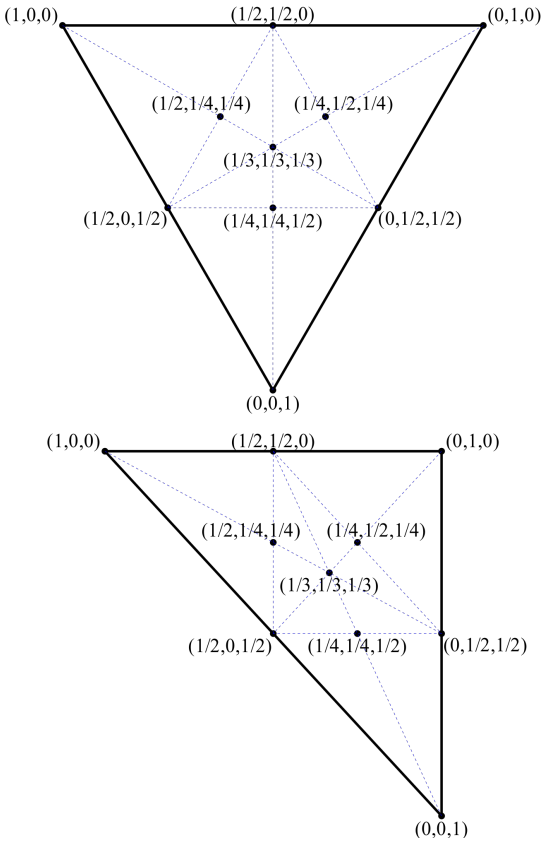

Figure 3.3: Barycentric coordinates on an equilateral triangle and on a right triangle. Image by Rubybrian [16].

Given a point $\vec{r}$ in the triangle's plane with Cartesian coordinates (x, y) one can obtain the barycentric coordinates λ_i from it and vice versa. The first part of (3.1) gives two constraints and the second part a third constraint on the three unknowns. Writing (3.1) in terms of the cartesian coordinates (x_i, y_i) of the triangles vertices $\vec{r}_i$, we have

$$\begin{aligned} x &= \sum_{i=1,2,3} \lambda_i x_i \\ y &= \sum_{i=1,2,3} \lambda_i y_i \,. \end{aligned} \tag{3.2}$$

The solution to this linear system of equations is written succinctly in terms of a

matrix $\mathbf{T}$ defined as

$$\mathbf{T} = \begin{pmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{pmatrix} . \tag{3.3}$$

This matrix is invertible as long as $\vec{r}_1$, $\vec{r}_2$ and $\vec{r}_3$ are not collinear, in which case they would not form a triangle. The barycentric coordinates can then be computed as

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} = \mathbf{T}^{-1} \cdot (\vec{r} - \vec{r}_3) , \qquad \lambda_3 = 1 - \lambda_1 - \lambda_2 . \tag{3.4}$$

Now that we have defined the barycentric coordinates, we can interpolate (or extrapolate) the value of a function at any point in terms of its values at the three nearest points forming a triangle by using the barycentric coordinates as weights in the interpolation:

$$f(\vec{r}) \approx \sum_{i=1,2,3} \lambda_i f(\vec{r}_i) . \tag{3.5}$$

For a linear function, this interpolation is exact, so it makes sense to use it to approximate functions which are roughly piecewise linear in the region of interest. We assume this to be the case in our example of wafer indentations, since the indentations are small compared to the whole image and so we don't expect large fluctuations in those parts.

The algorithm

The algorithm is a straightforward implementation of the above definition of barycentric interpolation. The first step is to implement a kernel which crops the image such that the wafer is perfectly inscribed in a square and makes a mask which delimits the ideal, circular wafer. We then implement a kernel which spawns a thread for each pixel in the image. Each thread checks if its assigned pixel lies inside the mask delimiting the boundary of the ideal circular wafer and if so, whether or not it is equal to zero in which case it is classified as a dark indentation. If a pixel belongs to an indentation, the corresponding thread starts searching for the three nearest neighbors in order to perform the barycentric interpolation.
It starts looking for points with a non-zero value in annuli of a single-pixel width and increasing radius, centered around the starting pixel. The search continues until three non-zero points are found and their locations are stored. We then check if the three points are collinear by computing the determinant of the matrix $\mathbf{T}$ defined in 3.3. If so, we search for another point until we find three of them that form a triangle and we complete the barycentric interpolation by computing the interpolated value using 3.4 and assigning it to the starting pixel.

```
1 int r = 1;
```

```
int dis[3];
int djs[3];
int count = 0;

// find closest three neighbors
while (count < 3){
for (int di=-r; di<=r; di++){
  for (int dj=-r; dj<=r; dj++){
    if (count == 3) break;
    // skip exterior circle
    if ( (di*di + dj*dj) > r*r ) continue;
    // skip interior circle
    if ( (di*di + dj*dj) <= (r-1)*(r-1) ) continue;

    // check if neighbor is in bounds
    if ( (i+di < 0) || (i+di >= res) || (j+dj < 0) || (j+dj >= res
   ) ) continue;
    // check if neighbor is not an indentation
    if ( wf_gpu[(i+di)*res + (j+dj)] != 0.0 ) {
      dis[count] = di;
      djs[count] = dj;
      count++;
    }
    // check if points are collinear
    if (count == 3) {
      double detT = (double)((djs[1]-djs[2])*(dis[0]-dis[2]) - (
   djs[0]-djs[2])*(dis[1]-dis[2]));
      if (detT == 0) count = 2;
    }
  }
}
r++;
}
```

Results and benchmarks

We test the performance of the barycentric interpolation algorithm described above by taking eight real wafer measurements with dark indentations and creating masks representing their non-circular shapes. We then generate synthetic wavefront images from known, randomly sampled Zernike coefficients with the shape of these masks and try to extract these coefficients from the image by

1. using the Zernike fitter on this non-circular image and
2. filling in the indentations by barycentric interpolation and then using the Zernike fitter on the resulting interpolated, circular image.

Below we show an example of such a synthetic wavefront. We plot the original circular image, the masked image with the shape of a real wafer measurement, the interpolated result, and the difference between the interpolated image and the original one, magnified 100× to highlight the difference. For the same synthetic wavefront we also plot the errors in the coefficients extracted from the original synthetic circular image, in the coefficients extracted from the indented wavefront and in the ones computed from the interpolated image.

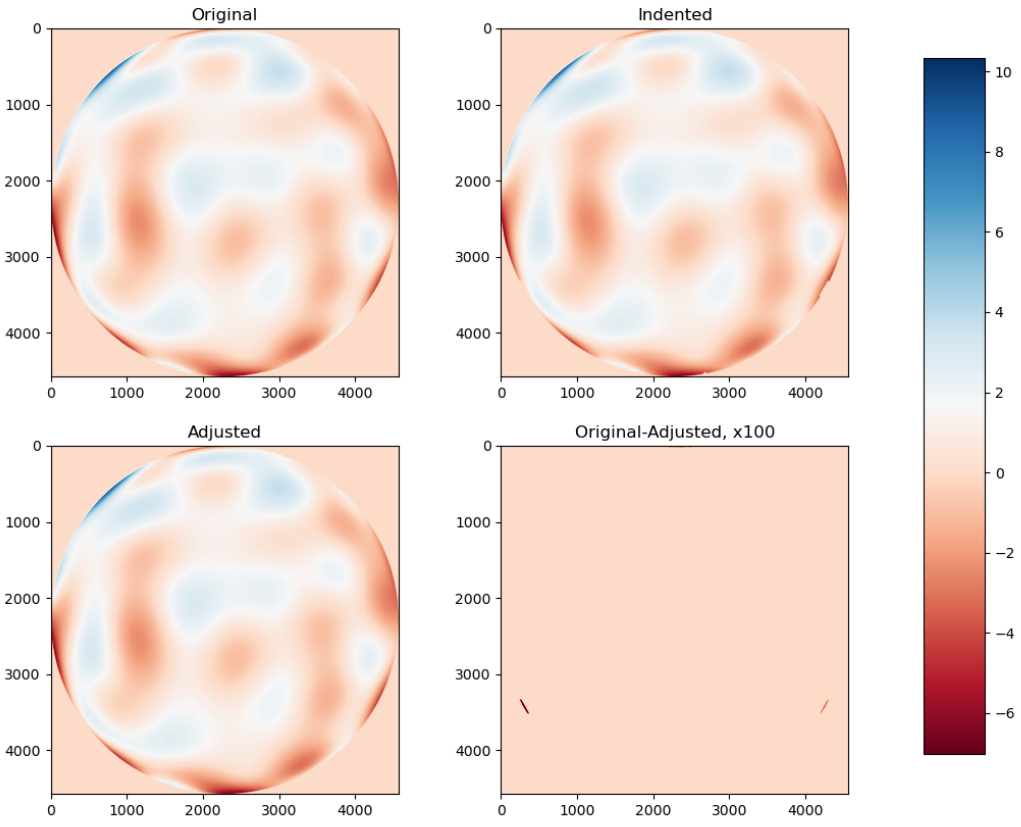

Figure 3.4: Synthetic wavefront with interpolated indentations.

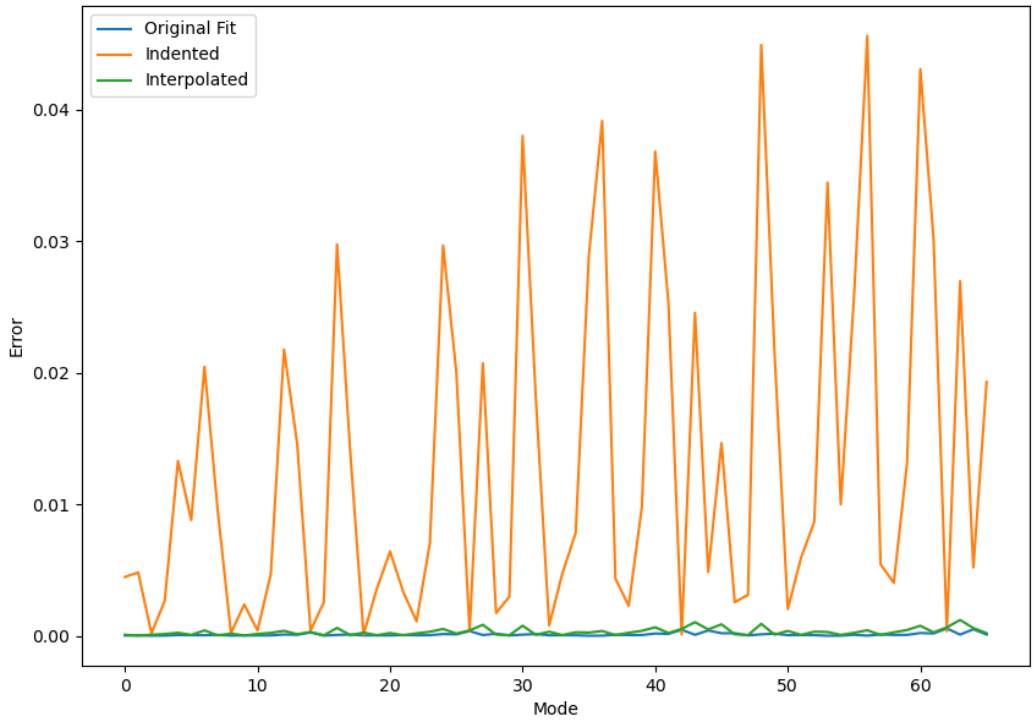

Figure 3.5: Zernike coefficients extracted from the original, indented and interpolated images. The error represents the absolute value of the difference between the computed coefficients and the original coefficients used to generate the synthetic wavefront.

From this typical example we can already appreciate that the interpolation greatly improves the computation of the coefficients. Below we plot the statistics of the error in recovering the coefficients for 10 different random, synthetic wavefronts for each of the 8 masks obtained from real wafer measurements.

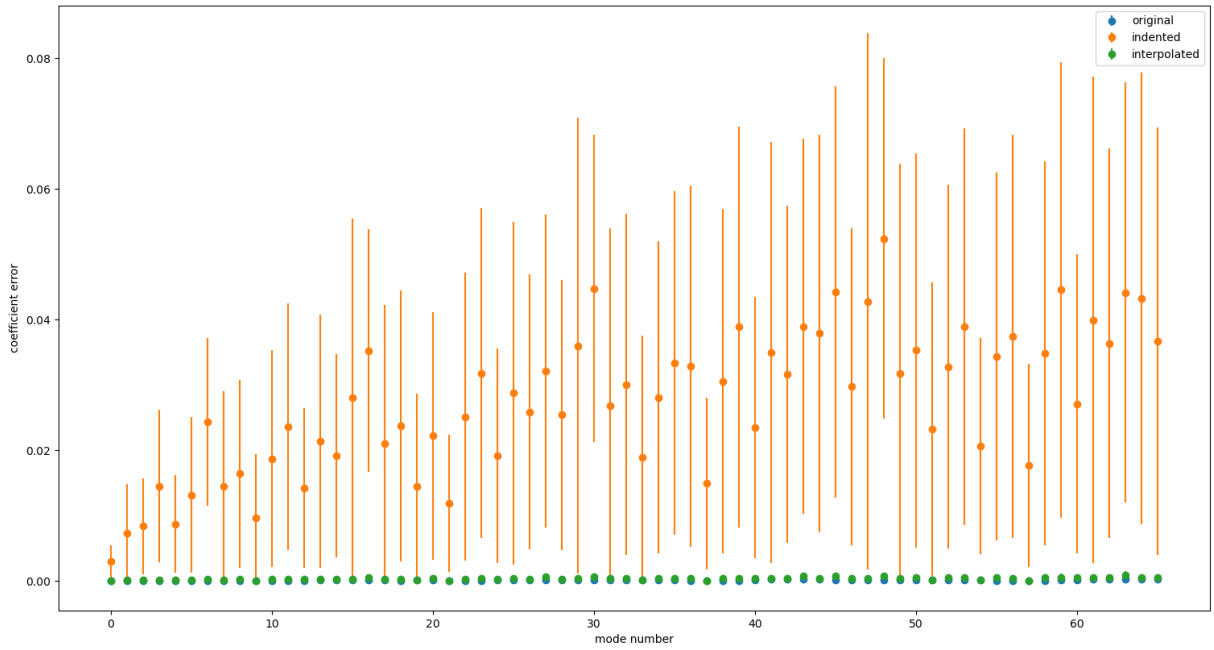

Figure 3.6: Statistics of errors in coefficients extracted from the original, indented and interpolated images. The dots represent the mean error error as the absolute value of the difference between the computed coefficients and the original coefficients used to generate the synthetic wavefront, while the error bars indicate the standard deviation.

In terms of the NRMS errors in extracting the coefficients, we have in fitting the

- original, circular wavefront: $NRMS = 0.0002924 \pm 0.0001013$
- indented wavefront: $NRMS = 0.0604414 \pm 0.0269938$
- interpolated wavefront: $NRMS = 0.0008062 \pm 0.0004229$.

These statistics again highlight the improvement of the Zernike fitter that the interpolation permits. Finally, a comment on the time of computation is in order. The masks used above have a resolution of about 4500×4500 pixels and from the resulting synthetic wavefront we extract 66 mode coefficients. This takes the CUDA Zernike fitter approximately 0.94 seconds while the barycentric interpolation takes about 0.12 seconds so it does not considerably slow down the fitting process, while it greatly improves its results.

Chapter 4

Adaptable Zernike basis

Generalities

We have seen in the previous chapter that for non-circular objects the computation of the Zernike coefficients becomes significantly worse. In the case of the wafer indentations, we have solved this problem by filling in the missing pieces via barycentric interpolation. When considering other applications however, this problem becomes more pronounced and interpolation is no longer the best option. In this chapter we deal with the application of wavefront sensing to Ophthalmology. This technology permits to accurately and objectively measure optical aberrations in the human eye. Here we are presented with a similar problem as before, but of intrinsically different nature: the pupils of the human eye are not perfectly circular, rather they are of some complicated, vaguely circular shape. The difference with respect to the case of the wafers treated in the previous chapter is that there is no fundamental, circular object which is obscured by something else like the grips. Rather the object itself is fundamentally non-circular, which makes interpolation meaningless since we are interpolating non-existing data. The better choice in this case is to use a new basis, adapted to the given non-circular pupil. This is what we will do in the present chapter. The basic idea is simple: find a mapping between the disk and the pupil and use it to map the Zernike basis from the disk to the given pupil shape. Conversely, we can also map the wavefront image from the pupil to the disk.

The algorithm

The core of the problem lies in finding a mapping between the pupil shape and the disk. For simple, non-circular shapes such as ellipses this can be done analytically, but for more complicated shapes this quickly becomes intractable. Furthermore,

we have to deal with the pixelated nature of the problem: ideally, we would want a 1:1 mapping between pixels, in order not to lose any information. The solution presented in the present chapter is inspired by [10] and is essentially a simplified version of the algorithm described in that paper, which considers more complicated cases than we need to deal with. The fundamental idea is the following: peel the pupil and the disk into one-pixel wide layers, and map layer by layer. This peeling is easily done by what is called an erosion, with a 3×3 square kernel: the erosion function scans the kernel over the image, computes the minimal pixel value overlapped by the kernel and replaces the central pixel with that minimal value. In our case this has the effect of setting the pixels on the border of the disk or pupil to zero. Subtracting the image before and after the erosion gives us the outermost layer. By repeating this procedure we peel the images until we reach the center and store the coordinates of the pixels in each layer in a corresponding array. The effect of this layering procedure can be seen in figure 4.1.

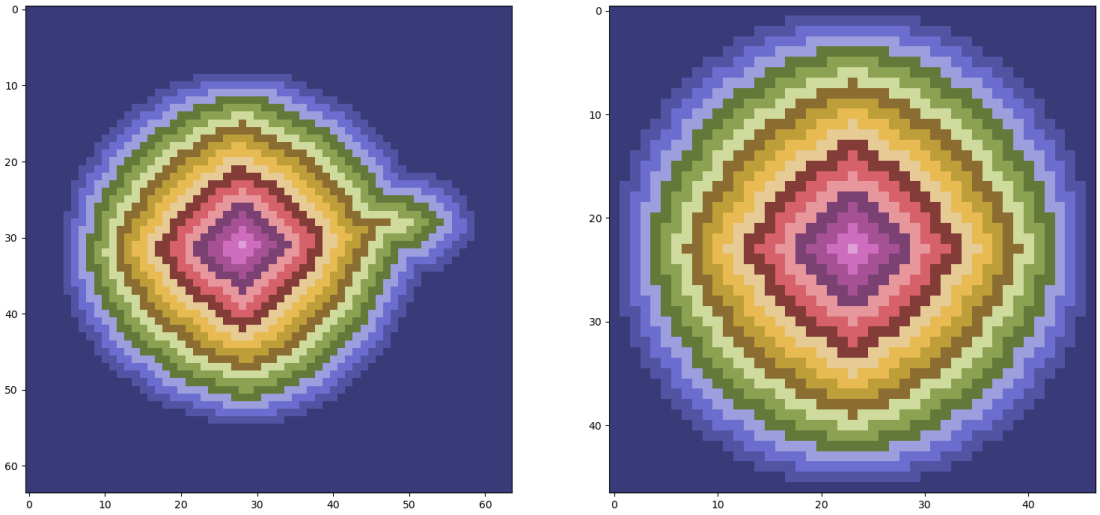

Figure 4.1: "Onions" corresponding to a non-circular pupil and a disk. Here we show them with very low resolution for visualization purposes.

In order to have the best possible mapping, given a pupil shape we generate a disk with a resolution such that it has the same number of layers as the pupil. Now the only part that is left to do is to compute a mapping between the pixels in the corresponding layers. For each layer, we take all the pixels in that layer, order them by their angle with respect to the central pixel (the pixel in the innermost layer) and index them in that order. Note that this procedure produces the desired result of a continuous line of pixels only if the shape is approximately circular and does not have any "S-like" shapes. More general cases are treated in [10] but are

not needed for our application. Since in general the number of pixels in a given layer of the pupil and the disk will be different, we need to interpolate the values. Consider mapping the disk to the pupil: to do so, we first "stretch" the indices of the disk layer to fit the number of pixels in the corresponding pupil layer as shown in figure 4.2 below.

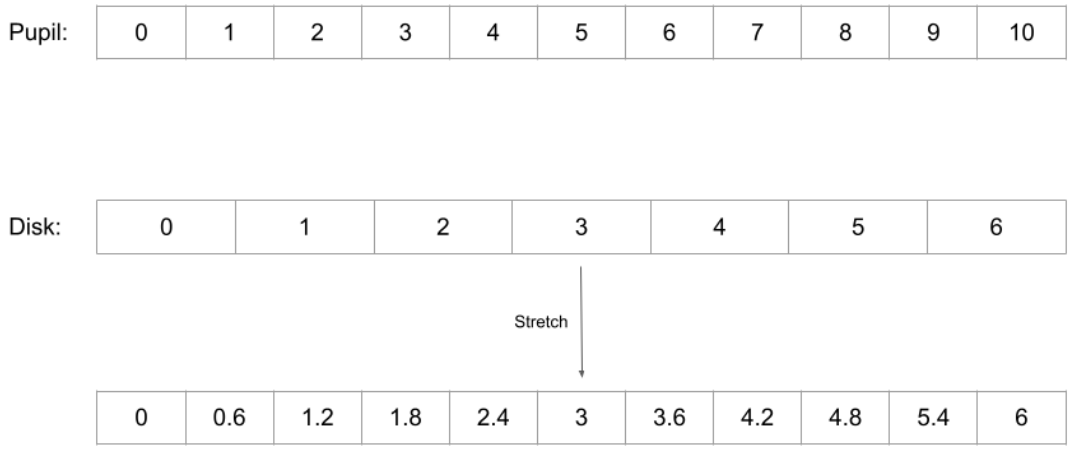

Figure 4.2: Indices corresponding to an 11 pixel long layer in the pupil, a 7 pixel long layer in the disk, and its corresponding "stretch".

This stretching gives us the weights for the interpolation by computing the distance of the fractional index from the previous and next integers. Denoting the array of stretched disk indices by SI_d and the values of the corresponding pupil and disk pixels by V_p and V_d, we have for a given pupil pixel with index i:

$$\begin{aligned} w_1[i] &= \lceil SI_d[i] \rceil - SI_d[i]\,, \quad w_2[i] = SI_d[i] - \lfloor SI_d[i] \rfloor \\ V_p[i] &= w_1[i] \cdot V_d\left[\lfloor SI_d[i] \rfloor\right] + w_2[i] \cdot V_d\left[\lceil SI_d[i] \rceil\right]\,. \end{aligned} \tag{4.1}$$

Here $\lceil x \rceil$ and $\lfloor x \rfloor$ denote the ceiling and floor functions, respectively and w_1, w_2 are the weights in the interpolation. The above formula becomes much clearer by considering an example. Take for example the pupil pixel with index $i = 4$ in figure 4.2: its corresponding stretched disk index is $SI_d = 2.4$. Formula (4.1) then

becomes:

$$\begin{aligned} w_1[4] &= \lceil 2.4 \rceil - 2.4 = 0.6\,, \quad w_2[4] = 2.4 - \lfloor 2.4 \rfloor = 0.4 \\ V_p[4] &= 0.6 \cdot V_d\,[2] + 0.4 \cdot V_d\,[3]\ . \end{aligned} \tag{4.2}$$

The mapping repeats this procedure for each layer, appropriately stretching the indices and returning an array of weights and coordinates of the next and previous pixels used for the interpolation. The same reasoning applies to the inverse mapping from the pupil to the disk, or if the indices need to be squeezed. Applying these maps we can map the Zernike polynomials defined on the disk to an arbitrary pupil shape, or map the pupil image to the disk.

Results and benchmarks

Below we show the result of mapping the first 15 Zernike polynomials from the disk to an example pupil shape.

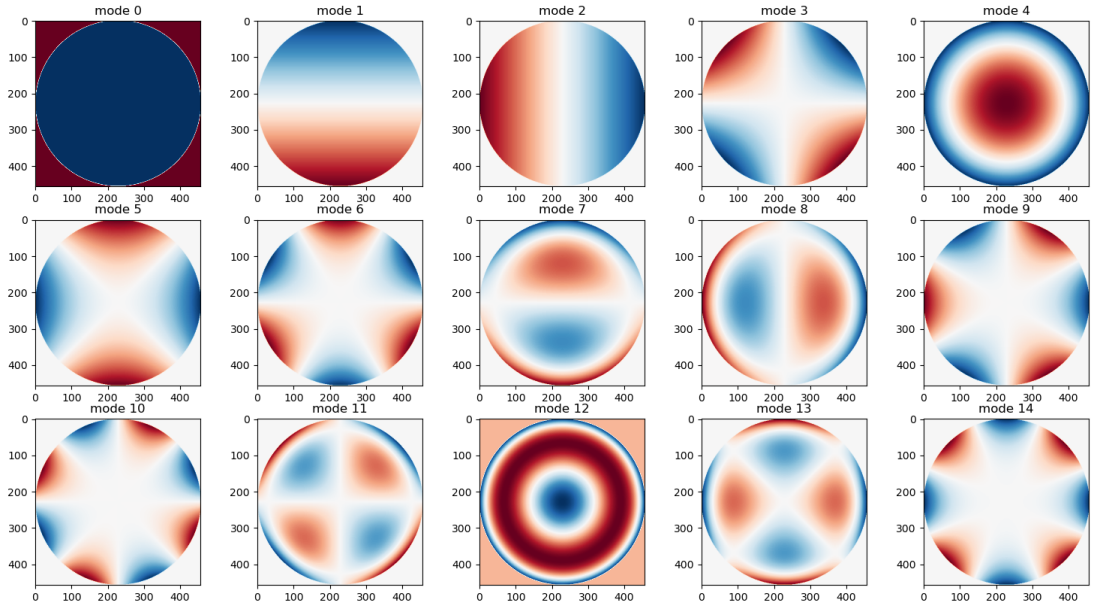

Figure 4.3: First 15 Zernike polynomials on the disk.

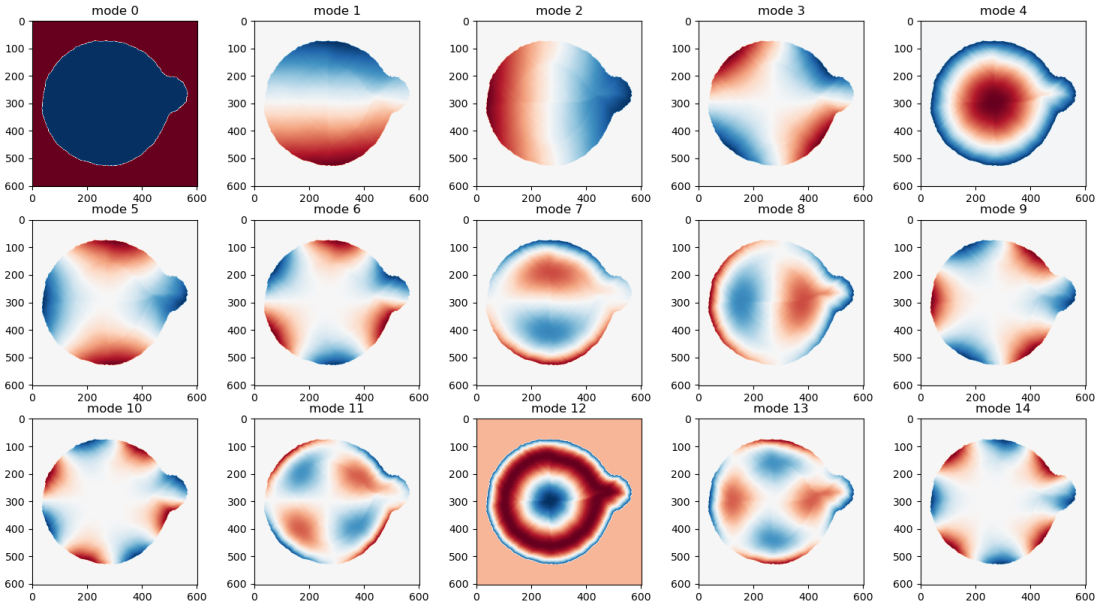

Figure 4.4: First 15 Zernike polynomials, mapped to a non-circular pupil

These adapted Zernike polynomials form a good basis for functions on the given pupil shape. For the example above, we can compute the Gram matrix of the basis functions, defined as

$$G_{ij} = \sqrt{\frac{2n_i + 2}{\pi \epsilon_{l_i}}} \sqrt{\frac{2n_j + 2}{\pi \epsilon_{l_j}}} \int_{disk} dA \, Z_i(\rho, \phi) Z_j(\rho, \phi) \,, \tag{4.3}$$

where as always i, j are the single-indices, from which the radial and azimuthal indices n_i, l_i, n_j, l_j can be computed. Ideally, for an orthonormal basis this Gram matrix would be a diagonal matrix with all ones along the diagonal. For our discrete case, the integral becomes a sum over pixels and introduces some error. The deviation of this real Gram matrix from the ideal one characterises the quality of our basis. Below we plot the Gram matrix of the first 66 Zernike polynomials on the disk, and of the same polynomials mapped to the pupil shown above.

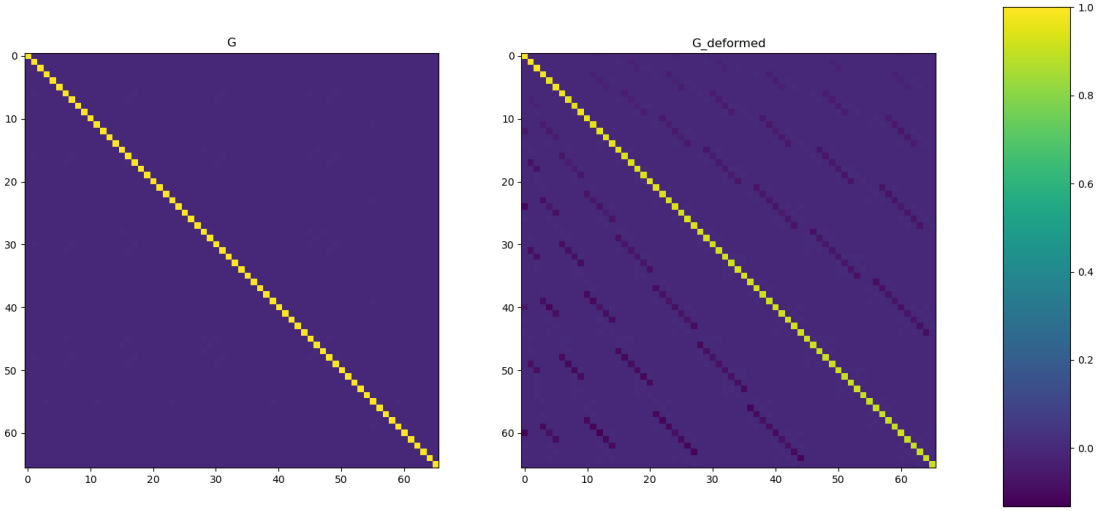

Figure 4.5: Gram matrices for the first 66 Zernike polynomials on the disk (left) and on the pupil shown in 4.4 (right).

We see that the mapping introduces a small source of error but still produces a high-quality basis. In terms of numbers, we are working with a disk of resolution 457×457 which has the same number of layers as the pupil. At this resolution, the deviations of the 66×66 Gram matrix from the ideal one are the following:

- On the disk:

  Average value on the diagonal: 0.997435

  Maximum deviation from 1 on the diagonal: 0.007654

  Average value on the off-diagonal: 0.000279

  Maximum deviation from 0 on the off-diagonal: 0.005915

- On the pupil:

  Average value on the diagonal: 0.931825

  Maximum deviation from 1 on the diagonal: 0.101562

  Average value on the off-diagonal: 0.002996

  Maximum deviation from 0 on the off-diagonal: 0.101651 .

Up to now we have considered mapping the Zernike polynomials from the disk to the pupil shape, in order to get a new basis adapted to the pupil. This provides a nice visualization of the aberration modes on a given pupil. On the other hand, given an application where we want to extract a number of mode coefficients from a given non-circular image, it makes more sense to map the image itself to the disk

and use the usual Zernike basis. In this way we transform only one image, instead of many Zernike polynomials. An example of this inverse mapping is shown in figure 4.6.

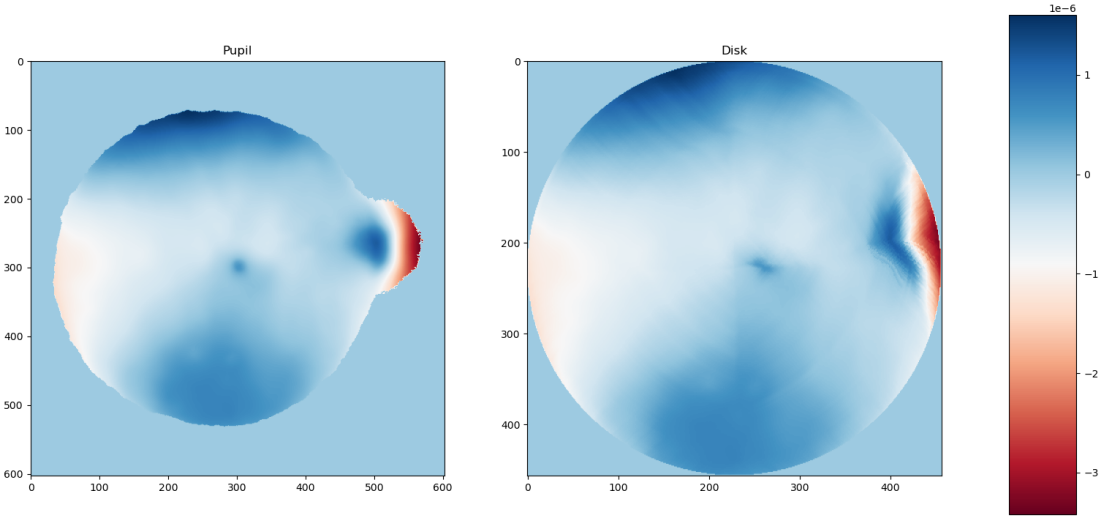

Figure 4.6: Mapping of a wavefront image with a non-circular pupil to the disk.

Conclusions

In this thesis we introduced the basics of modern wavefront sensing and the mathematics of Zernike polynomials, commonly used to analyze the measured wavefronts. In chapter 2 we developed a Zernike fitter, which given a wavefront extracts the coefficients of a given number of Zernike polynomials or inversely, given some coefficients reconstructs the wavefront. It was implemented in python, C++ and CUDA, where the latter uses the full power of a GPU to efficiently perform the image processing. We assessed the performance of the python, C++ and CUDA versions by producing synthetic images with known Zernike coefficients and comparing them with the coefficients extracted by the different programs. We looked at the RMS error and the time of computation, finding similar RMS errors throughout all versions, and a considerable speedup when passing from python to C++ and to CUDA, as expected. We also looked at memory usage to inspect the viability of a least-squares approach in the case of images with not too many modes. In chapter 3 we extended this basic Zernike fitter to be applicable to the analysis of silicon wafers. The presence of indentations in these wafers make them non-circular which results in a dramatic loss of accuracy of the previously developed Zernike fitter. To solve this problem, we implemented a functionality which fills the indentations using barycentric interpolation. Testing this solution we found a significant improvement and were able to recover the Zernike coefficients with an accuracy close to the ideal, circular case.
Finally, in chapter 4 we considered an even more complicated wavefronts, where the non-circularity is inherent and interpolation does not make sense. We developed a method to extend the Zernike basis to non-circular shapes by explicitly constructing a mapping from the wavefront shape to the disk. We then analyzed the orthonormality of the resulting new basis, finding satisfactory results.

Bibliography

[1] Byju's. *Wavefront of light waves*. [Online; last accessed July 27, 2023]. URL: https://byjus.com/question-answer/what-is-the-wave-front-of-the-light-waves/.

[2] Toppr. *Wavefronts in geometric optics*. [Online; last accessed July 27, 2023]. URL: https://www.toppr.com/ask/content/posts/wave-optics/notes-28049/.

[3] Ortholibrary. *Wavefront aberrations*. [Online; last accessed July 27, 2023]. URL: https://ortholibrary.in/book/9789351522478/chapter/ch6.

[4] J. Hartmann. "Objektivuntersuchungen". In: *Zeitschrift für Instrumentenkunde* 24 (1904), 1–25, 33–47, 97–117. URL: https://archive.org/details/zeitschriftfrin09gergoog.

[5] R.V. Shack. "Production and use of a lenticular Hartmann screen". In: *Journal of the Optical Society of America* 61 (1971). URL: http://www.opticsinfobase.org/josa/abstract.cfm?uri=josa-61-5-648.

[6] Mikhail Konnik and Jose Dona. "Waffle mode mitigation in adaptive optics systems: A constrained Receding Horizon Control approach". In: June 2013. DOI: 10.1109/ACC.2013.6580355.

[7] Chao Zuo et al. "Transport of intensity equation: a tutorial". In: *Optics and Lasers in Engineering* 135 (2020), p. 106187. ISSN: 0143-8166. DOI: https://doi.org/10.1016/j.optlaseng.2020.106187. URL: https://www.sciencedirect.com/science/article/pii/S0143816619320858.

[8] von F. Zernike. "Beugungstheorie des Schneidenverfahrens und seiner verbesserten Form, der Phasenkontrastmethode". In: *Physica* 1.7 (1934), pp. 689–704. ISSN: 0031-8914. DOI: https://doi.org/10.1016/S0031-8914(34)80259-5. URL: https://www.sciencedirect.com/science/article/pii/S0031891434802595.

[9] Wikipedia. *Zernike polynomials*. [Online; last accessed July 27, 2023]. URL: https://en.wikipedia.org/wiki/Zernike_polynomials#/media/File:Zernike_polynomials_with_read-blue_cmap.png.

[10] G. Wolberg. “Image Warping Among Arbitrary Planar Shapes”. In: *New Trends in Computer Graphics*. Ed. by Nadia Magnenat-Thalmann and Daniel Thalmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 209–218.

[11] Jakob Wenzel. *pybind11 — Seamless operability between C++11 and Python*. URL: https://github.com/pybind/pybind11.

[12] Mark Harris. *Optimizing Parallel Reduction in CUDA*. [Online; last accessed July 31, 2023]. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[13] Wooptix. *Semiconductor Metrology*. [Online; last accessed August 1, 2023]. URL: https://wooptix.com/solutions/semiconductor-metrology/.

[14] Simon Barthelmé. *Barycentric interpolation*. [Online; last accessed August 1, 2023]. URL: https://dahtah.wordpress.com/2013/03/06/barycentric-interpolation-fast-interpolation-on-arbitrary-grids/.

[15] Wikipedia. *Barycentric coordinate system*. [Online; last accessed August 1, 2023]. URL: https://en.wikipedia.org/wiki/Barycentric_coordinate_system.

[16] CC BY-SA 3.0 Rubybrian Own work. *Barycentric coordinates*. [Online; last accessed August 1, 2023]. URL: https://commons.wikimedia.org/w/index.php?curid=4842309.