Design of a high definition phase retrieval camera for quantum fluids of light

Tangui Aladjidi

Laboratoire Kastler Brossel - Paris

April-June 2020

Contents

Acknowledgements

I would first like to thank the whole Quantum Optics team who were kind enough to welcome me again. Even with the particular circumstances of the past few months, integration was seamless and allowed for a high productivity in my work even going as far as to ship the necessary equipment to my home ! I also thank warmly Yicheng Wu (main author of [1]) from the Rice University (Houston, USA) for sharing his MATLAB code with me, allowing for much faster progress when porting my code to a GPU impementation.

1 Introduction

Like cold atoms, paraxial fluids of light are a platform that allows to explore many-body quantum particles dynamics. However their beauty resides in the relative simplicity of the experimental setups needed [2]: paraxial fluids of light are generated in hot atomic vapors, using this non-linear and highly tunable medium to generate interactions between photons and thus collective effects such as the emergence of Bogoliubov excitations and superfluidity [3] [4] [5]. The typical setup of such an experiment is presented in fig 1 : a hot atomic vapor of Rubidium (^{85}Rb here) is pumped with a laser tuned near the D_2 line. This generates effective interactions between photons that allow for collective effects such as Bose-Einstein condensation (BEC) and superfluidity [2]. This quantum fluid of light can be characterized using a probe beam, with which one can measure the spectrum of the quantum fluid [3] or create an optical potential [4] on which the fluid will scatter.

In this framework, all of the information about interactions and the properties of the system are encoded in the light field at the output of the atomic vapor cell. It is why retrieval of the intensity and phase of the field would be extremely valuable. This then allows to compute directly many quantities such as correlation functions of the field, giving direct access to the details of the interactions inside of the cell. Measuring correlations as a way to investigate quantum behavior of the system has been proposed to detect exotic effects such as the dynamical Casimir effect [6]. As this will be one of the main topics of the PhD I will follow in the team, setting up this phase retrieval tool could be a critical asset in my future research.

For this purpose, commercially available options already exist for phase retrieval such as Shack-Hartmann sensors [7] or shearing interferometers [8]. However their resolution are limited, overall they offer very low definition (around 400×300 at best) and can be extremely expensive : up to 60000€. The first idea I had on the subject was to build a sort of Michelson interferometer following almost exactly the same setup presented in fig 12, where I would then modulate the phase using the spatial light modulator (SLM) until I extinguished the intensity on the camera (see fig 2). However this "brute force" approach would be quite hard to align, and then brute force search of the extinction would be very slow. Then I found the elegant solution recently presented in [1]. We subsequently decided to try and implement this model.

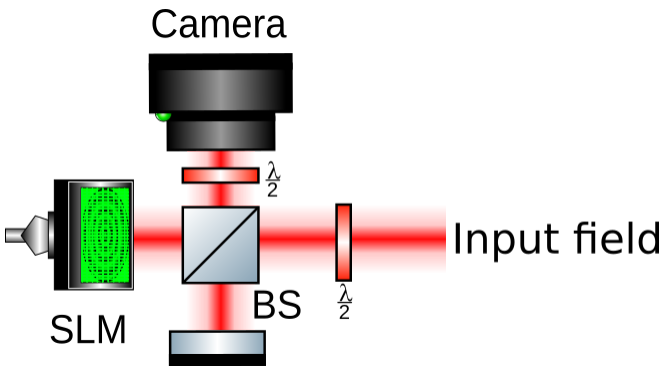


Figure 2: *Initial ideal of a modified Michelson interferometer. The input field to characterize is labeled on the right. It is then split by a non polarizing beam splitter (BS). One of the mirrors is replaced by an SLM. Polarization selection is ensured by a half wave plate (λ/2).*

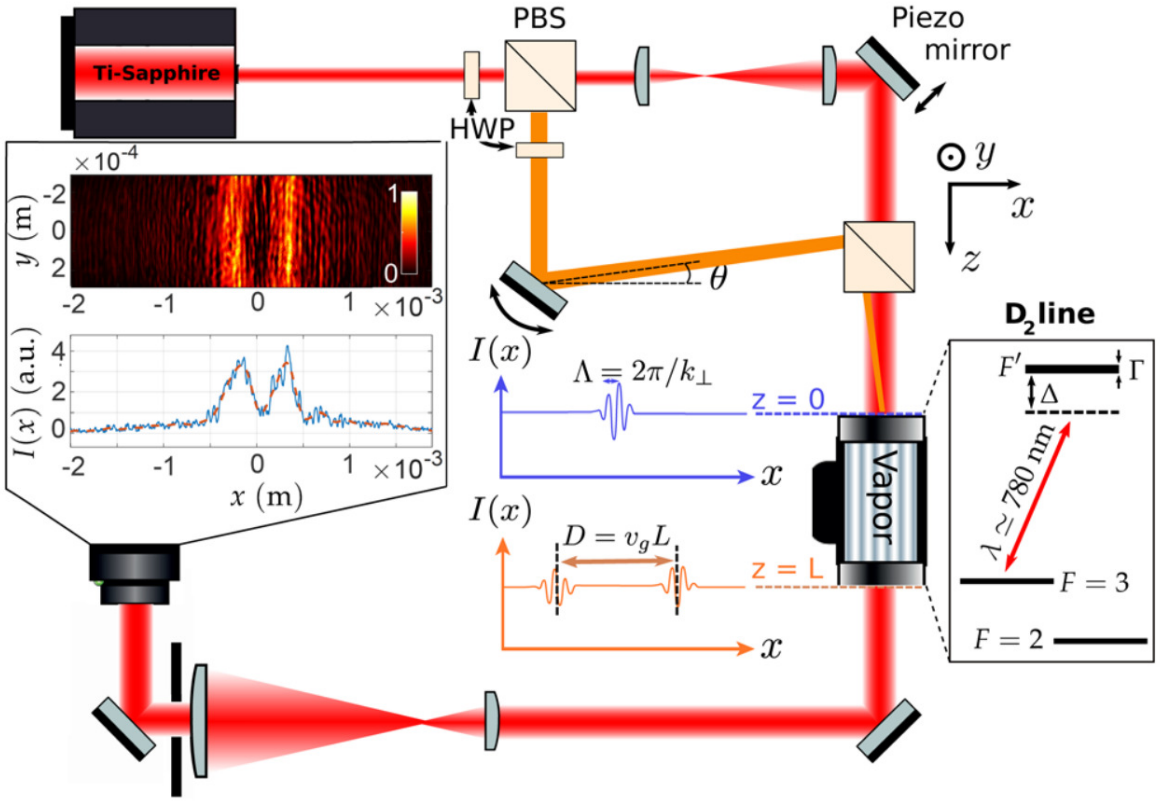


Figure 3: *Adapted from [3]. Experimental setup: polarized beam splitter (PBS) and half wave plate (HWP). θ is the angle between the probe (orange beam) and the optical axis defined by the pump (red beam). The probe interferes with the pump and slightly modulates its intensity. (Blue inset) Integrated intensity profile at the input of the medium (z=0).The wavelength λ of the density modulation is given by $2\pi/k_\perp$, where $k_\perp = k_0 \sin\theta$. (Orange inset) Integrated intensity profile at the output of the medium (z=L). Here the additional fine structure of the D_2 line of ^{85}Rb is represented. The distance D between the two wave packets gives access to the group velocity of the elementary excitations in the transverse plane. The output plane is imaged on a CMOS camera. (Inset, top left) Background-subtracted image obtained for $\theta \simeq 0$ rad and associated integrated envelope profile (blue: original; red dotted: high frequency filtered).*

2 Theory

The goal of the phase sensor is to allow for full field retrieval in a propagating superfluid of light experiment. The next section will detail how such a fluid of light can exist in a hot atomic vapor, and its properties. I will first detail the hydrodynamical equations for a fluid of light and show how superfluidity can occur, then I will expose the interest of correlation measurements and finally I will detail the phase retrieval procedure.

2.1 What is a quantum fluid of light ?

The starting point of the system's behavior is the propagation equation of an electric field in a non linear medium, and its quantum version. In our setup, the non-linear medium is a hot Rubidium (^{87}Rb or ^{85}Rb) vapor. This medium presents a third-order non-linearity (Kerr effect), meaning that the standard propagation equation needs to be modified in order to take the non-linearity into account. The following results and derivations are explained in great details in [4].

Paraxial propagation in a Kerr non-linear medium Let $\vec{E}(\vec{r},t)$ be the electric field of the laser of study. Within the paraxial approximation, it can be decomposed as follows :
$\vec{E}(\vec{r},t) = \mathcal{E}(\vec{r}_\perp,z)e^{i[k(\omega)z-\omega t]}\vec{e}$ where $\mathcal{E}(\vec{r}_\perp,z)$ is the enveloppe of the field and $\vec{e}$ some arbitrary polarization vector. Then, assuming that the enveloppe is varying slowly relative to the optical wavelength, the non-linear Schrödringer equation describes the propagation as follows :

$$i\partial_z\mathcal{E}(\vec{r}_\perp,z) = \Big[-\frac{1}{2k_0}\vec{\nabla}_\perp^2 - \frac{i\alpha}{2} - \frac{3}{8}\frac{k}{n_0^2}\chi^{(3)}(\omega)|\mathcal{E}(\vec{r}_\perp,z)|^2\Big]\mathcal{E}(\vec{r}_\perp,z). \tag{1}$$

Here k_0 is the wavevector of the field, α is the linear absorption of the medium, n_0 is the regular refractive index of the medium and $\chi^{(3)}(\omega)$ is the third order non-linear electric susceptibility of the medium. The validity of the paraxial treatment has been checked in reference [4]. One can also introduce an external potential (defect) in the cell by optically pumping some region of the cell with another beam closer to the D_2 line of Rb, creating a linear refractive index modulation δn (see fig 1). This adds a term as follows :

$$i\partial_z\mathcal{E}(\vec{r}_\perp,z) = \Big[-\frac{1}{2k_0}\vec{\nabla}_\perp^2 - \frac{i\alpha}{2} - k\frac{\delta n(\vec{r}_\perp,z)}{n_0} - \frac{3}{8}\frac{k}{n_0^2}\chi^{(3)}(\omega)|\mathcal{E}(\vec{r}_\perp,z)|^2\Big]\mathcal{E}(\vec{r}_\perp,z). \tag{2}$$

These equations allow to describe "classically" the behavior of the beam inside of the cell but do not allow to describe the quantum effects that take place. To describe these we need to look at the mapping between this non-linear Schrödringer equation and the Gross-Pitaevskii equation.

Gross-Pitaevskii mapping Formally, the third-order non-linearity introduces effective photon-photon interactions that lead to a whole range of effects, the most notable being the superfluid behavior of light inside the cell. Systems of interacting bosons are notoriously described with the Gross-Pitaevskii equation. For example, if $\Psi(\vec{r},t)$ is the wavefunction of a Bose-Einstein condensate (BEC) within the Hartree-Fock approximation, then $\Psi(\vec{r},t)$ follows the following equation :

$$i\hbar\partial_t\Psi(\vec{r},t) = \Big[-\frac{\hbar^2}{2m}\vec{\nabla}^2 + V(\vec{r}) + g|\Psi(\vec{r},t)|^2\Big]\Psi(\vec{r},t). \tag{3}$$

Here V is an external potential, g is the interaction constant and m the mass of the particles. Neglecting linear absorption ($\alpha \simeq 0$), one can introduce an effective mass, an effective coupling constant and map the propagation to an effective time $\tau = zn_0/c$. Introducing the normalized field enveloppe $\bar{\mathcal{E}} = \mathcal{E}/\int_S|\mathcal{E}|^2$ where S is the cell cross-section, (1) becomes :

$$i\hbar\partial_\tau\bar{\mathcal{E}}(\vec{r}_\perp,z) = \Big[-\frac{\hbar}{2(\hbar k/c)}\vec{\nabla}_\perp^2 - \hbar\omega\delta n(\vec{r}_\perp,z) - \hbar\omega n_2\mathcal{P}|\bar{\mathcal{E}}(\vec{r}_\perp,z)|^2\Big]\bar{\mathcal{E}}(\vec{r}_\perp,z). \tag{4}$$

One can identify $\bar{m} = \hbar k/c$ the effective mass and the interaction constant $\bar{g} = -\hbar\omega n_2 \mathcal{P}$ with the non-linear refractive index defined as $n_2 = \frac{3}{4c\epsilon_0 n_0^2}\chi^{(3)}$. This mathematically simple analogy is actually conceptually quite subtle as we switch from a 3D+1 geometry to a 2D+1 geometry. We will now display the hydrodynamics properties of the Gross-Pitaevskii equation using a Madelung transform.

Madelung transform and Euler equations One can represent the enveloppe function of the field $\mathcal{E}(\vec{r}_\perp, z)$ as a density $\rho(\vec{r}_\perp, z)$ and phase $\phi(\vec{r}_\perp, z)$: $\mathcal{E}(\vec{r}_\perp, z) = \sqrt{\rho(\vec{r}_\perp, z)}e^{i\phi(\vec{r}_\perp, z)}$. Injecting this form inside of equation (4) yields :

$$\begin{aligned} &\partial_\tau \rho + \vec{\nabla}_\perp \cdot (\rho \vec{v}) + \tilde{\alpha}\rho = 0 \\ &\frac{c}{n_0 k}\partial_\tau \phi + \frac{1}{2}\vec{v}^2 + \frac{c^2}{n_0^2}\Big(\frac{n_2}{n_0}\rho + \frac{1}{2k^2}\frac{\vec{\nabla}_\perp^2\sqrt{\rho}}{\sqrt{\rho}}\Big) = 0 \end{aligned} \tag{5}$$

Here $\vec{v} = \frac{c}{n_0 k}\vec{\nabla}_\perp \phi$ is the speed of the fluid and $\tilde{\alpha} = \alpha/cn_0$ are the losses. This expression is especially interesting because it shows phenomena that have no counterparts in classical hydrodynamics :

- The first equation describes the losses (as opposed to the classical conservation of mass equation) due to linear absorption and thus the Beer-Lambert law.
- The last term of the second equation (in between parenthesis) describes the quantum pressure. This has no classical counterpart and dominates on scales comparable to the healing length (described in the next section).

One finally sees from the expression of the velocity field $\vec{v}$ that to create a flow, one needs to have a gradient of phase, which can be easily achieved using an SLM (spatial light modulator) or the interference pattern between two beams.

Linearization and Bogoliubov transform The Euler equations give great insight into the fluid dynamics of light in a Kerr medium, but these equations have no analytical solutions in general. Furthermore, in order to understand more in details purely quantum effects, one would like a fully quantized equation.

The first natural way to treat these equations is to linearize them. Assuming small perturbations of the density and phase as follows :

$$\begin{aligned} \rho(\vec{r}_\perp, z) &= \rho_0(z) + \delta\rho(\vec{r}_\perp, z) \\ \phi(\vec{r}_\perp, z) &= \phi_0(z) + \delta\phi(\vec{r}_\perp, z) \end{aligned} \tag{6}$$

With $\delta\rho \ll \rho$ and $\delta\phi \ll \phi$. Keeping in mind the $\tau \leftrightarrow z$ mapping, we can then reinject this into (5) and get the linearized equations for $\delta\rho$ and $\delta\phi$ as a function of the propagation distance z. At zeroth order, this simply yields $\rho_0(z) = \rho_0(0)e^{-\alpha z}$ (Beer-Lambert law) and $\partial_z \phi_0 = k_0 n_2 \rho_0$. This last equation describes the Kerr dephasing that the wavefront undergoes. At first order we get :

$$\begin{aligned} &\partial_z \delta\rho + \frac{\rho_0}{k}\vec{\nabla}_\perp^2 \delta\phi + \alpha\delta\rho = 0 \\ &\partial_z \delta\phi - k\frac{n_2}{n_0}\delta\rho - \frac{1}{4k}\frac{\vec{\nabla}_\perp^2 \delta\rho}{\rho_0} = 0 \end{aligned} \tag{7}$$

We can then rewrite $\delta\rho$ and $\delta\phi$ in their second quantized form :

$$\begin{aligned} \delta\rho(\vec{r}_\perp, z) &= \sqrt{\rho_0}\int \frac{d\vec{k}_\perp}{(2\pi)^2}\Big[\hat{a}^\dagger(\vec{k}_\perp) f_+(\vec{k}_\perp, z)e^{-i\vec{k}_\perp\cdot\vec{r}_\perp} + h.c\Big] \\ \delta\phi(\vec{r}_\perp, z) &= \frac{1}{2i\sqrt{\rho_0}}\int \frac{d\vec{k}_\perp}{(2\pi)^2}\Big[\hat{a}^\dagger(\vec{k}_\perp) f_-(\vec{k}_\perp, z)e^{-i\vec{k}_\perp\cdot\vec{r}_\perp} + h.c\Big] \end{aligned} \tag{8}$$

Reinjecting these quantized forms into (7) allows us to derive the Bogoliubov-de Gennes matrix equation for the Fourier modes f_+ and f_- :

$$i\frac{\partial}{\partial z}\begin{pmatrix} f_+ \\ f_- \end{pmatrix} = [-i\frac{\alpha}{2} + \mathcal{H}_{\vec{k}_\perp}]\begin{pmatrix} f_+ \\ f_- \end{pmatrix} \text{ where } \mathcal{H}_{\vec{k}_\perp} = \begin{pmatrix} 0 & -\frac{\vec{k}_\perp^2}{2k} \\ -\frac{\vec{k}_\perp^2}{2k} + 2k_0\Delta n & 0 \end{pmatrix} \tag{9}$$

This matrix describes the mixing of Fourier modes into Bogoliubov modes which are actually **quantized excitations** of the fluid of light. Looking at the spectrum of this matrix allows us to retrieve the spectrum of these excitations (assuming the absorption is small enough for an adiabatic evolution) :

$$\Omega_B(\vec{k}_\perp, z) = i\frac{\alpha}{2} + \sqrt{-\frac{n_2}{n_0}\rho_0(z)\vec{k}_\perp^2 + \frac{\vec{k}_\perp^4}{4k^2}} \tag{10}$$

This dispersion relation has been experimentally measured in [3]. Its most notable feature is its linear behavior at low $k_\perp$, and quadratic behavior at high $k_\perp$. This effectively means that low $k_\perp$ excitations have a phonon-like massless behavior whereas the high $k_\perp$ excitations have a massive behavior. The transition occurs at $k_\xi = k\sqrt{\frac{\delta n}{n_0}}$ which is the inverse of the **healing length** ξ of the superfluid. As with atomic BEC's, the healing length is the minimum scale under which the order parameter of the fluid can heal [9], or in more trivial terms, the minimum wavelength of the Bogoliubov excitations. If an excitation has a wavelength under ξ, the excitation energy is too high and locally decondensates the quantum fluid. Once we have this spectrum, we are able to predict the emergence of a superfluid phase through the **Landau criterion for superfluidity** : the system can dissipate through Bogoliubov excitations only if this dissipation is energetically favorable i.e if the following condition is met : assuming we have a static fluid flowing at speed $\vec{v}$ in the transverse plane (following the notations of (5)), nucleating density excitations at $\vec{k}_\perp$ costs $\Omega_B|_{v\neq 0}(\vec{k}_\perp) = \Omega_B|_{v=0}(\vec{k}_\perp) + \vec{k}_\perp \cdot \vec{v}$. This cost must be negative for the nucleation to be energetically favorable. This is the Landau criterion. This criterion is met when $\vec{k}_\perp \cdot \vec{v} < 0$ and $\Omega_B|_{v=0}(\vec{k}_\perp) < |\vec{v}||\vec{k}_\perp|$ i.e when the excitations are emitted upstream and when the velocity exceeds a certain critical velocity v_c given by :

$$v_c = \min_{\vec{k}_\perp}\left(\frac{\Omega_B|_{v=0}(\vec{k}_\perp)}{|\vec{k}_\perp|}\right) \tag{11}$$

This velocity is referred as the **speed of sound** of the system because it delimits the normal fluid and the superfluid regime of the Bogoliubov dispersion. A remarkable display of the superfluid nature of the fluid of light is depicted in [4] and is reproduced in fig.4.

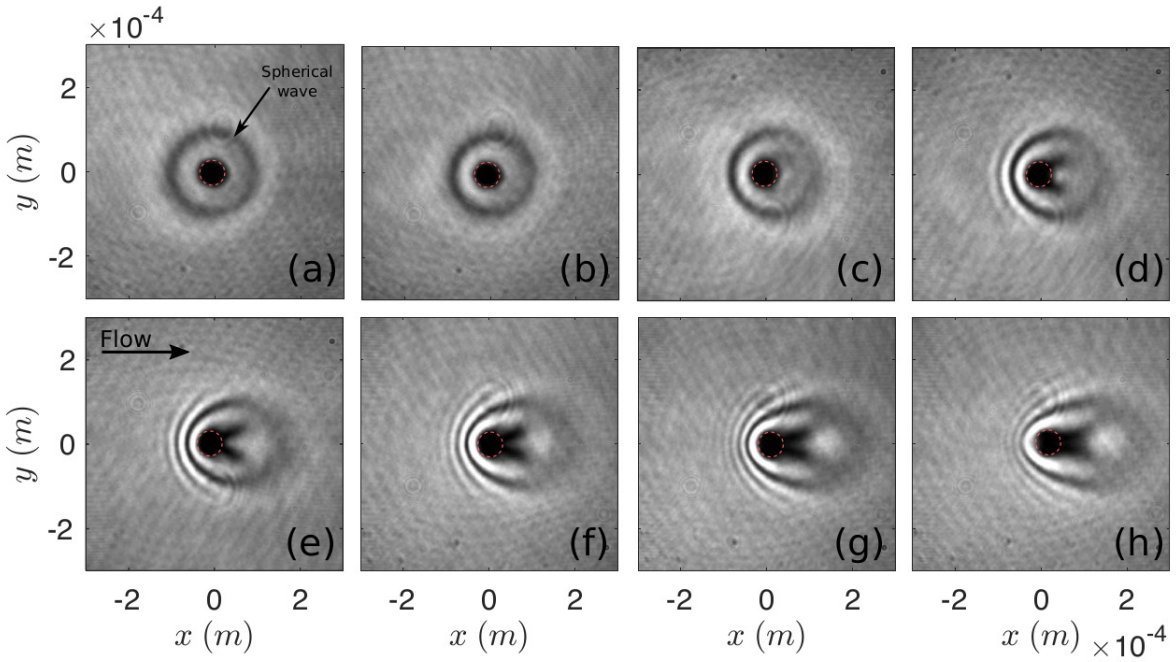


Figure 4: *Near-field scattering patterns at low background density ρ_0 . A defect beam generates a negative Gaussian index modulation ($\delta n < 0$), whose width $\omega_{0,d}$, is about 40 μm. The resulting potential is repulsive explaining why a dark spot is visible at the position where the defect lies (red dotted circle). (a): The photon fluid is at rest ($v = 0$). A spherical wave is created as soon as probe and defect enter the cell and propagate away from the defect. (b)-(h): The photon fluid is flowing rightward. The flow velocity steps up from one image to the next. On figure (d), interference fringes upstream from the obstacle start appearing. Light is scattered backward by the repulsive defect and interferes with the incoming fluid. On (g) and (h), the contrast of the fringes before the obstacle decreases. The kinetic energy of the photon fluid is large compared to the height of the potential and light just go through without being back-scattered. We can finally notice that the spherical wave visible on figure (a) is dragged with the flow and drifts rightward.*

As presented in fig.4, when a defect is introduced in the cell using an additional beam, one can see the flow of the fluid of light around the defect, very much like the soundwaves accumulating in front of a fighter jet.

2.2 Correlation measurements

One of the main goals of the paraxial fluid of light platform would be to observe beyond mean field effects, particularly higher-order non-classical correlations induced by the interactions of the Bogoliubov excitations inside of the cell. One way is to study the interferences of Bogoliubov excitations [10]. To that end, full field retrieval would be extremely valuable as it would not only give direct access to the non linear phase accumulated by the light through the cavity, but also to the correlations of the field. From the full field, one can then derive all the moments of the field, including the valuable $g^{(1)}$ and $g^{(2)}$ measurements that give a detailed look of the quantum properties of the system. Furthermore, doing this kind of measurement using full field retrieval would be considerably simpler that other cold atoms techniques that usually involve computing proxima of the actual moments (see [11] for instance).

Recently, quantum fluids of light have been proposed as an experimental platform to explore purely quantum effects from the field of analogue gravity such as quantum vacuum emission from analogue white holes [12] or Hawking radiation and the dynamical Casimir effect around the horizon of an analogue black hole [6]. All of these phenomenons display characteristic signatures in terms of density fluctuations i.e intensity fluctuations that are measured through a $g^{(2)}$ intensity correlation measurement. But a

direct $g^{(1)}$ field correlation measurement would give a more detailed view, especially concerning possible phase correlations and their significance, as well as giving access to all possible moments calculations.

2.3 Phase retrieval algorithm

Phase retrieval is a non trivial class of problems, and a very active branch of applied mathematics research. Several methods are available to solve it but in general it is a non convex optimization problem, and it is notoriously hard to solve. One way to solve is is the Gerchberg-Saxton algorithm [13]. However the convergence rate and noise sensitivity of this algorithm is not well controlled. If however one tries to recover the field from several modulated intensity samples i.e (N×N) measurements, it can be shown [14] that full phase retrieval is guarenteed **up to a global phase factor** with an arbitrarily high probability. Furthermore this becomes a convex relaxation of the original problem and can then be solved using existing convex optimization solvers [14]. As presented last year in [1] we use a mix of the two approaches. Following the Gerchberg-Saxton (GS) algorithm, we alternate projections between the source and image planes, but instead of using only one intensity measurement of the field, we propagate several intensity measurements of the **modulated** target field, averaging the result at each iteration.

I will now detail the phase retrieval algorithm. Let $E = Ae^{i\phi}$ be the $N \times N$ target field to recover. K random samples of the field's amplitude $\tilde{A}_i$ ($i \in [\![1, K]\!]$) and generated. The modulations are denoted by D_i for intensity masks and Φ_i for phases masks. The intensity modulations are a uniform random distribution of 1 and 0 (binary modulation), and the phase modulations are a uniform random distribution of angles in $[-\pi, \pi]$.The first modulation is kept at unity to initialize the amplitude. Let P_z be the propagation operator that propagates the field over a distance z. The phase retrieval problem is to find E such that $|P_z(ED_i)| = \tilde{A}_i$ (or $|P_z(E\Phi_i)| = \tilde{A}_i$). This is solved using a GS (Gerchberg-Saxton) loop as follows :

- **Initialization** : A non modulated sample $\tilde{A}_1(z)$ is back propagated to the source plane. This yields the initial $\tilde{E}^{(0)}(0)$ guess.
- **Propagation to the image plane** : the initial guess is propagated to the image plane with the modulations $\tilde{E}_i^{(0)}(z) = P_z[\tilde{E}^{(0)}(0)D_i]$. The mean of the root mean square errors of the field estimations $RMS = \frac{1}{N} <\| \tilde{E}^{(0)} - E \|_2>_i$ is computed
- **Amplitude constraint** : Impose the amplitude of the samples to be the measured samples $\tilde{E}_i^{(0)}(z) \leftarrow \tilde{A}_i e^{i[arg(\tilde{E}_i^{(0)}(z))]}$
- **Back-propagation to the source plane** : Back-propagate the fields to the source plane, taking in to account the modulation, and update the guesses $\tilde{E}_i^{(1)}(0) \leftarrow P_{-z}[\tilde{E}_i^{(0)}(0)]D_i$ (for the phase modulation $\tilde{E}_i^{(1)}(0) \leftarrow P_{-z}[\tilde{E}_i^{(0)}(0)]\Phi_i^{-1}$)
- **Averaging** : Average the guesses to obtain the first estimation of the field : $\tilde{E}^{(1)}(0) = \frac{1}{K}\sum_{i=1}^{K} \tilde{E}_i^{(1)}(0)$.

The loop stops when the algorithm has converged i.e when the relative variation of the RMS between two iterations falls below 5.10^{-5}. Note that I have found this threshold experimentally as the convergence of the algorithm usually plateaus in the middle, and thus one needs to define a threshold low enough that the algorithm goes past this plateau, but high enough that the algorithm does not continue to loop once retrieval has occured With a satisfactory precision. The flow chart of the loop can be found in fig.5.

A thorough mathematical analysis of the problem as presented in [14] allows to put precise bounds on the convergence of the algorithm as well as the accuracy of the phase retrieval. The dependence of the number of modulations required for recovery is **logarithmic** in the number of points. This is very interesting in terms of scaling. For a signal of N points, there exists a constant c such that the number of samples needed K follows $K \geq c\log^4(N)$, for a probability of a successful retrieval equals to $1 - \frac{1}{N}$.

Note that the value of the constant c is a priori unknown, but for a signal of 128 points, the authors of [14] find that only 6 samples are needed. For 4024x3036 image, this only goes up to 8 as shown in [1]. For intensity modulation schemes, more samples are needed because the intensity modulated samples carry less information about the field : the zeroes of the mask block half of the pixels. This particularity is discussed in the next section 3.

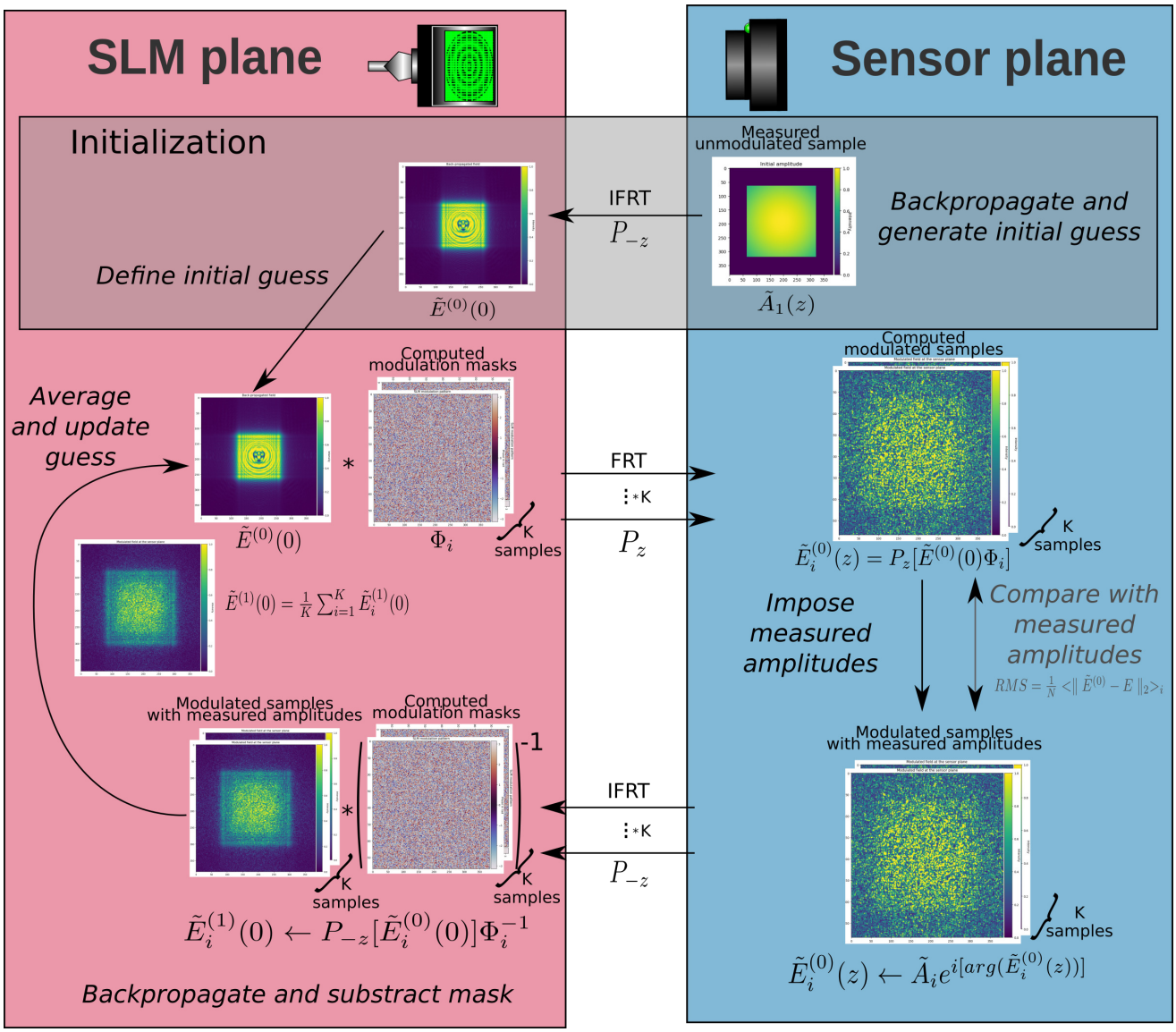


Figure 5: *Flow chart of the phase retrieval algorithm : what is the phase associated with the measured unmodulated amplitude $\tilde{A}_1(z)$? FRT and IFRT mean the forward and inverse Fresnel transform i.e the propagation operator over a distance z.*

3 Numerical simulations

Given the lack of access to the lab, I had to devise a simulation code in order to develop the software needed for the sensor as well as benchmark its performance. The performance of the code is a critical aspect and is discussed more in detail in subsection 3.2. To generate the modulations of the field realistically, I will simulate the effect of a spatial light modulator (SLM) and a digital micro-mirror device (DMD). An SLM is a liquid crystal screen that allows to apply an arbitrary phase mask to a field, each pixel allowing 256 levels of modulation between 0 and 2π. A DMD is an array of microscopic

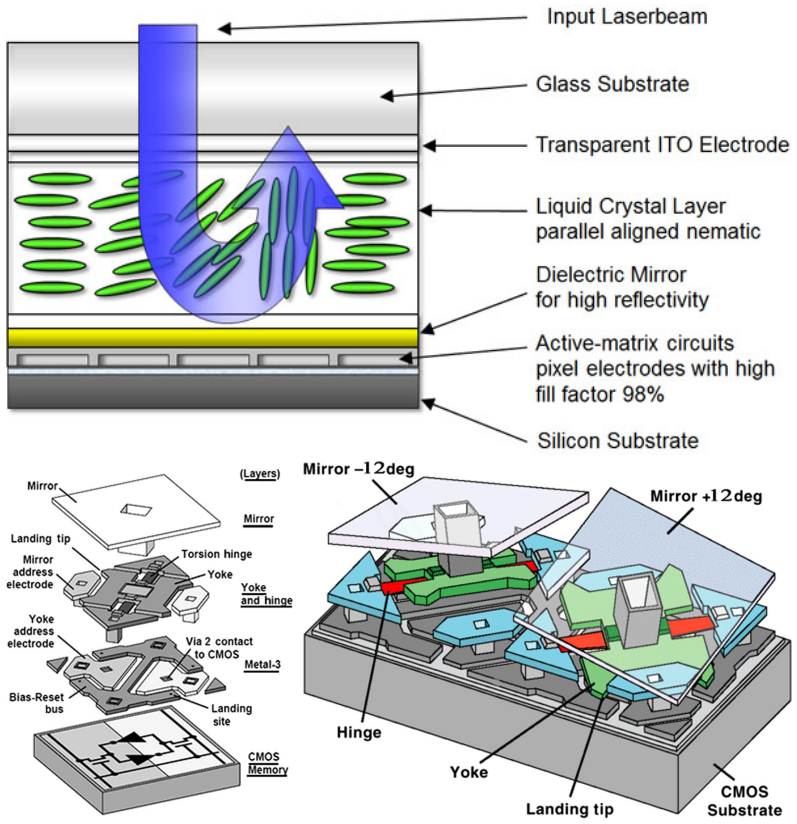


Figure 6: *Working principles of an SLM (top) and DMD (bottom). For the SLM, electrodes allow to control the angle of nematic liquid crystals, thus locally modulating the refractive index of the medium. For the DMD, each pixel is simply a mirror placed on a hinge that can be switched from one angle to the other using two electrodes.*

mirrors that can be set to two incidence angles : 12 and -12 degrees. This allows to apply an intensity mask to a field by turning "on" or "off" each micro-mirror of the array. The design of these is depicted in fig 6.

3.1 Program structure

The code I've been using for the last month of my internship is divided in two main parts :

- Image production : simulate some signal from which to retrieve the phase map, including the rescaling steps taking into account the various definitions of the camera, SLM or DMD
- Phase retrieval

All of the code is available in the Addendum section, and its latest version can be found on the project's GitHub. Physical parameters and other relevant hard coded parameters are read from a configuration file (see `wish.conf`). The main guideline throughout development was to write a code that would be directly deployable in for the future production code. This meant starting from the sensor plane as this is the main limiting factor in the experimental setup, and anticipating the image processing routines needed for real data acquisition. For this, the help from Yicheng Wu (author of [1]) allowed for massive time savings.

3.1.1 Image production

I start by loading an arbitrary image that will be the signal that we observe on the camera and from which we want to retrieve the phase. I also load some phase pattern of the same size, this phase pattern will at the end be compared to the retrieved wavefront. Then, these images are padded in order to leave some room for the modulated wavefronts. Also as we deal with a lot of Fourier transforms, it is better to introduce protection bands in order to avoid unwanted reflection aliasing (lines 41 to 58 in `WISH_measurement.py`). Then these images are backpropagated to the source plane using the function `u4Tou3` (l.221 of `WISH_lkb.py`). Propagating duties are handled by the `frt` function (l.150/176 of `WISH_lkb.py`). From the field in the source plane, SLM or DMD modulations are generated and applied to the field using `modulate` or `modulate_binary` (l.106/119 of `WISH_lkb.py`). Examples of such modulations can be found in fig 9.

As the different planes of the experiment have different spacings in practice, it is of capital importance that all fields are rescaled relative to one another according to these different spacings to ensure the correct numerical simulation of propagation. More explicitely, the SLM pixel pitch for instance is 12.5 μm, whereas the camera pixel pitch is 5.5 μm. This means that if we want to numerically simulate the effect of the SLM on the wavefront, one needs to resize it so that one pixel of the field has the right physical size when applying the phase mask. As the pixel pitch of the SLM is roughly twice the camera's, this means that the SLM pattern needs to be scaled by a factor of two. As the SLM has an apparent spacing imposed by the Fresnel transform (see equation 12), this means that numerically, the SLM pattern will need to be rescaled by a factor d_{SLM}/d' if d_{SLM} is the SLM pixel pitch, and d' the apparent pitch of the SLM as seen by the camera. If this is not done, the calculation cannot carry on as the SLM has a 1280×1024 definition, whereas the camera has a 2048×2048. This resizing routine is done by the function `process_SLM` (l.231 of `WISH_lkb.py`).

Finally, modulated fields are propagated to the sensor plane. All of this is done in the `gen_ims` function (l.299 of `WISH_lkb.py`). When propagating, an optional `noise` parameter allows to add a gaussian noise to the field to simulate sensor noise. It can then generate multiple noised copies to then be able to simulate noise averaging. These three steps are presented in fig 7.

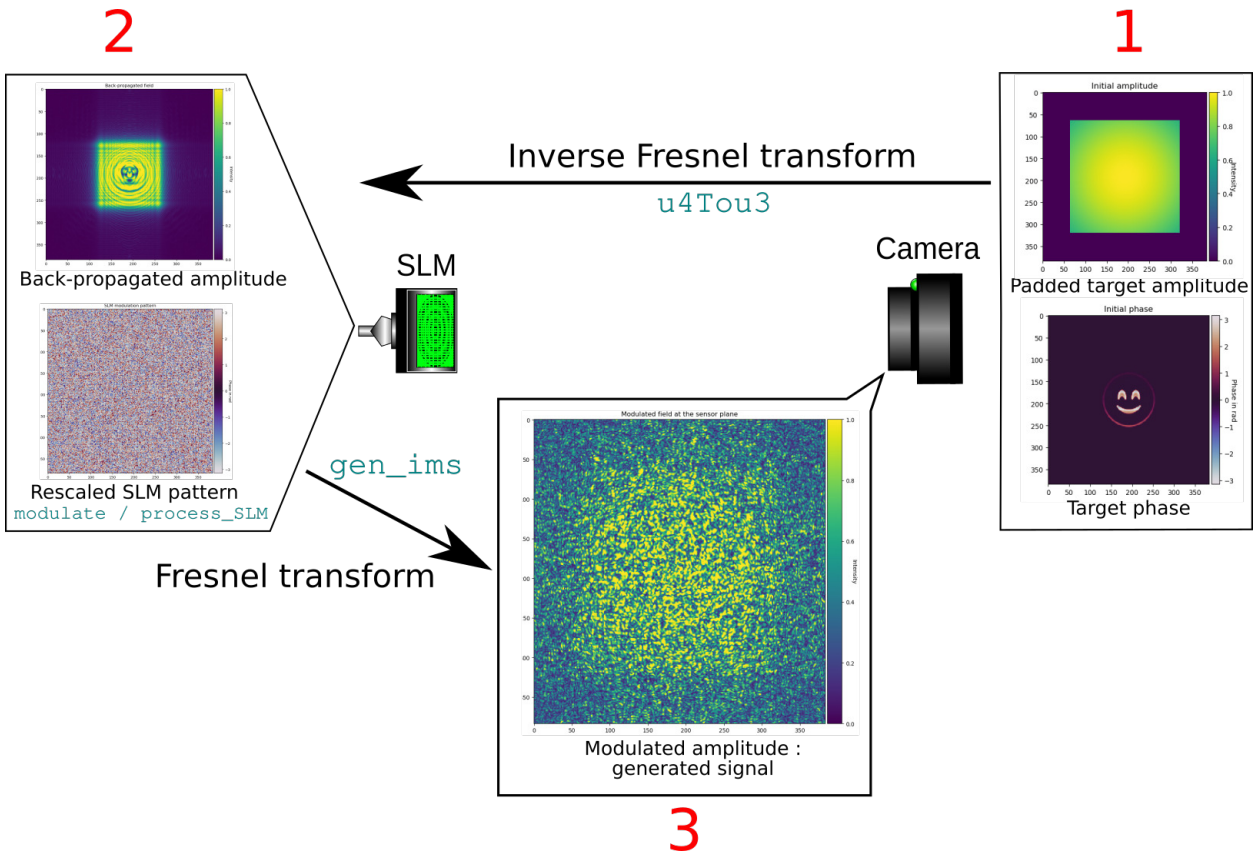


Figure 7: *Flowchart of the signal generation. The target field amplitude and phase are first imposed (1) then they are backpropagated to the source plane. At the source plane, the modulation patterns are generated (2) and rescaled according to the relevant physical scales. Finally, the modulation is applied to the field and it is propagated to the sensor plane (3).*

3.1.2 Phase retrieval

When the field samples are generated, they are then fed to the main workhorse of the code : `WISHrun` (l.351 of `WISH_lkb.py`). This function is a strict implementation of the algorithm workflow presented in fig 5. It is mainly composed of a double `for` loop : one for the Gerchberg-Saxton (GS) iterations, and inside it, another loop for the samples. Inside the GS loop, propagation is kept to its bare minimum using and optimized propagation function `frt_gpu_s` (l.204 in `WISH_lkb.py`). As we go back and forth between the source and sensor planes, the quadratic and constant phase factors applied in the full Fresnel transform in eq.12 cancel each other out. We are left with almost a bare FFT, only supplemented by two calls to `fftshift` to keep the physical orders of the images. This function is explored more in detail in the next section 3.2 as it has been the subject of a lot of thoughts for the code to run in a reasonable amount of time.

Convergence of the loop is controlled by computing the mean of the normalized root mean square (RMS) deviation between the target amplitudes and current guess, using the built-in `norm` function of Numpy. This RMS is computed only in the signal region, i.e outside of the padding where the intensity is zero. The convergence criterion is when the relative difference of the RMS between two iterations falls below 5.10^{-5} : $\frac{|RMS_{i+1}-RMS_i|}{RMS_i} < 5.10^{-5}$. If the convergence criterion is not met in 10000 iterations, the loop stops. Usually, the number of GS iterations is between 40 and 3000 for realistic data sets.

3.2 Need for speed : optimization strategies

A critical aspect of this project is the optimization of the code. For the simulation to run on realistic sets of data, it is crucial that the code is both very fast, but also memory efficient. Effectively, we need to compute at two FFT per sample observation, per GS iteration, as all numerical propagation methods use at least one FFT. In the end, if N_{os} is the number of observations per sample (several images of one modulated samples are taken to try and average noise see section 3.3), N_{mod} the number of modulations and N_{GS} the number of GS iterations, this makes for $N_{os} \times N_{mod} \times N_{GS}$ FFT's. For typical numbers, this makes for $2 \times 16 \times 300 = 9600$ FFT's ! All of this for 2048×2048 pictures or even more depending on the camera ! While this is not computer intensive *per se*, if we want the computation to take less than one minute on a laptop for a "lab compatible" speed, this needs quite a lot of thinking to reach these speeds in Python.

The first step was to depart from external libraries in order to simulate light propagation. At first, I used the Lightpipes library for light propagation with Fresnel transforms. While being very practical and accurate, this library was way too slow and exhaustive for our goals. The order of magnitude of runtime for realistic sets was from 30 minutes to several hours depending on the size of the set. I subsequently coded my own propagation routine using the Fresnel transform presented in [15] :

$$E(x,y,z) = \frac{e^{ikz}}{i\lambda z} e^{i\frac{k}{2z}(x^2+y^2)} \mathcal{F}\{E(x',y',0)e^{i\frac{k}{2z}(x'^2+y'^2)}|_{p=\frac{x}{\lambda z}, q=\frac{y}{\lambda z}}. \tag{12}$$

Here E is the scalar electric field, x, y, z are the spatial coordinates and x' and y' are the coordinates in the origin plane. Note that this expression is very convenient for numerical calculations because this means that input and output planes (at 0 and at z) can have different spacings, furthermore this expression can be efficiently computed using fast Fourier transforms (FFT) and a (N, N) discretization of the field. To avoid loss of information however, we will always compute this transform such that the two spacings are Fourier conjugated i.e if the spacing in the sensor plane is d, the reciprocal spacing in the source plane will be $d' = \frac{\lambda z}{Nd}$. After implementing eq.12, I could then convert the code to run on a graphics card (GPU), which are notoriously efficient for array computations such as FFT's.

GPU computing Having no experience in GPU programming, I tried to find the simplest interface to allow me to run calculations on my laptop's GPU (Nvidia MX150, 384 CUDA cores, 2 GB video RAM). Having a Nvidia GPU, I had to find an API (application programming interface) that was using a CUDA C backend. As I had a lot of experience using Numpy for scientifical computing, CuPy [16] was the most straightforward option. This library is a drop-in replacement for Numpy and also allows direct access to low level CUDA utilities. This was a game changer and brang a 10× speed up for FFT calculations on 2048×2048 images. Moreover, the speedup gets dramatically larger as size increases (up to ×210 for a 800MB array). The main limitation of GPU computing on my machine was its very small memory capacity (compared to the CPU's RAM) : I have 16GB of RAM on my laptop vs. only 2GB on my GPU, 700MB of which are used by the graphical interface of my operating system. This forced me to work a lot on memory management.

Memory handling After switching to GPU computing, memory management became very important because of the very limited memory "budget" : at first, I could not run any calculation with realistic sizes because my code was so inefficient. The first step was to hunt down useless copies, and other unnecessary uses of memory. Then I reduced the precision wherever possible. As the camera output is coded with a maximum precision of 10 bits, it is meaningless to carry calculations with a precision over 32 bits (for complex numbers). Due to the limitations of the Numpy/CuPy datatypes, and hardware limitations of my GPU (`float16` computations are not really supported on consumer-grade GPU's), I could not go below the `float32` / `complex64` precision. Still, this makes for a factor of two in memory.

Finally, in order to get the best speed, there is a tradeoff between memory economy on the GPU, and memory tranfer times. With CuPy, the data needs to be transfered in the GPU's memory prior

to calculation. This means that it is possible to precisely load only what is needed in the GPU in order to spare memory, but then transfer times penalize execution speed. If on the opposite side, all calculations are ran in a vectorized fashion, meaning by directly feeding a big array with all the samples to the GPU, speed is a lot better, but image size is more limited. Practically, in my code, fields are stored as three dimensional arrays of shape (N, N, N_{im}) where N is the image size, and N_{im} is the total image number ($N_{mod} \times N_{os}$). I can either load this array inside the GPU prior to the GS loop and do all the calculations on the GPU directly, or load individually each batch of N_{os} images inside the GS loop. The first strategy is 35% faster than the second for same size arrays, but the second allows for substantially larger arrays (images up to 2400×2400) or more samples, versus 2048×2048 for the first implementation.

Numba and Cython Python is a flexible and expressive language, however it is not well known for its performance (even though it can be very fast when used properly). This is because Python is a high-level interpreted language and its flexibility actually slows it down drastically. Python spends a lot of time figuring out what the user wants to do. When running a Python code, the code is first compiled in to byte code, then fed to the interpreter that will in term convert it to machine code and actually run it. Upon compilation, one of the most important tasks of the compiler is **type inference** i.e figuring out what types (`float`, `int` etc ...) the variables and various objects should be, allocating memory etc ... After having optimized all functions in terms of algorithmic, this is where we want to try and save some time. There are several ways to speed up this compilation process, but I will only present the two main ones : Numba [17] and Cython [18].

The first way to gain some time on the compilation is to pre-compile some functions that will be heavily used. This can be done with Numba. Numba is a just-in-time compiler, meaning that when running the code, it will compile the properly decorated functions, and store the resulting LLVM (low level virtual machine) code in a cache for further use. This LLVM code is actually the last step before machine code so when the functions run again, it will directly access the cached LLVM code and run drastically faster.

Using Numba is particularly convenient because one only needs to decorate the functions we want to compile just in time ("jit") using the `@jit` command. It can also be used to vectorize functions, and parallelize them. However it is limited in the sense that calls to external libraries are somewhat impossible outside of a few compatible libraries (Numpy and Scipy most notably). Furthermore, one needs to be quite cautious of the way we write functions. Speed up is not automatic and if things are not done properly, one can quite easily end up with a much slower function that what we started with.

Another major way to improve execution speed of Python is to use Cython. Cython allows to use C typing and functions, and compile whole Python scripts or libraries (with their dependences). Actually, almost all Python libraries are compiled using Cython upon installation. Starting from a python script, it is then possible to statically type all of the script by writing a `.pyx` extension that specifies all types and function signatures, very much like a C `.h` header file. Cython also allows to import directly C or C++ modules inside a Python code. This is especially efficient on "light" code where the code in itself does not do heavy computation, where the interpreter overhead is big. Knowing this, I wrote the `.pyx` extension for my `WISH_sensor` class, and then compiled it. Unfortunately, speed up was marginal at best. There are several reasons for this. The bulk of the work of my code is done by either Numpy or CuPy which are two already heavily optimized libraries. The other reason is that CuPy has no official Cython interface, this means that I could not properly type all of the CuPy arrays, nor write custom optimized functions directly using low-level C/Cython code from Cupy, in term greatly hindering the efficiency of the Cythonization.

As a conclusion, as of now, my code is running the fastest it can. It could only run faster if I optimize the `WISHrun` function by rewriting it in CUDA C. I chose not to pursue as it would be a significant step in terms of development time and technicality. As an order of magnitude, the runtime is now around several seconds to 10 minutes depending on the dataset size. For a 300×400 array, phase

retrieval simulation takes 0.2 second with intensity modulations. Compared to the several hours of the first code, this is quite satisfactory. On the hardware side, I did some testing of the code on some more powerful GPU (2018 Nvidia Quadro P2000, 1024 CUDA cores, 5 GB GDDR5) from École Polytechnique computers and I did get a $\times 3$ to $\times 4$ speedup depending on the dataset size. So this means that on more recent GPU's, we could potentially get much more speedup as newer Nvidia GPU's have much larger CUDA core counts (more than $3\times$ larger), significantly higher clock speeds(40% higher), and much larger memory buses ($3\times$ larger). Speedups ranging from $\times 8$ to $\times 16$ seem to be a reasonable estimate, as the added memory would allow to fully vectorize the computations. This would bring the runtime to several seconds for full size 2048×2048 calculations, possibly much less for smaller resolutions.

3.3 Performance benchmarks

In this part I will detail the performance of the phase retrieval procedure for various experimental parameters : number of modulation patterns, type of modulation and noise levels. For benchmarking, I will use a gaussian target amplitude, and the phase pattern presented in fig 8. Besides its comical appearance, I chose an easily recognizable phase pattern with sharp edges and high frequency details in order to test both the accuracy and resolution of the phase retrieval.

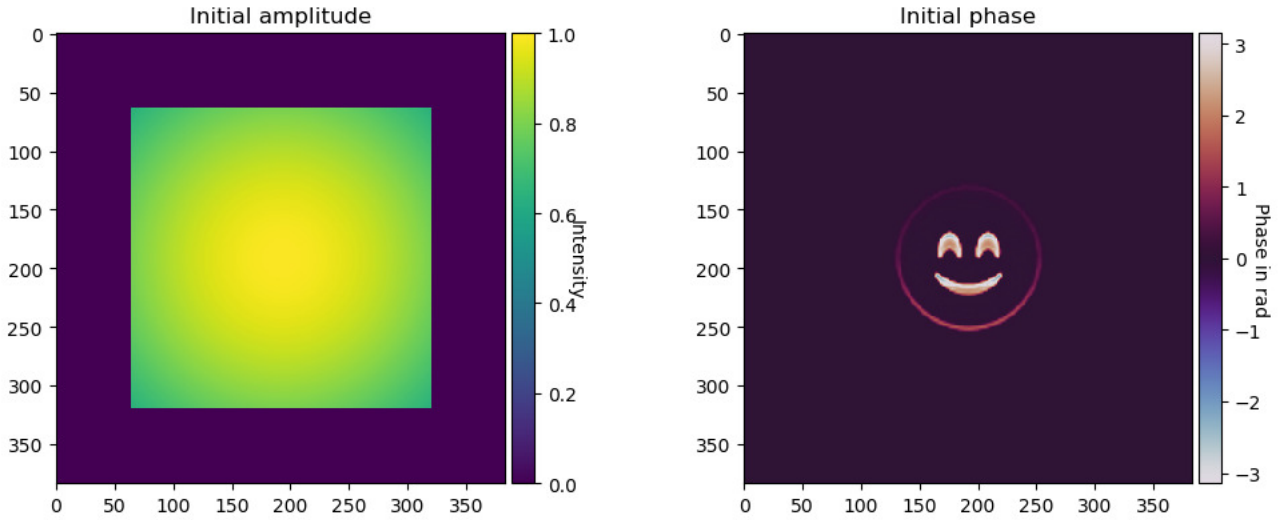


Figure 8: *Initial amplitude (left) and initial phase (right). Both images are 256×256 with an additional padding of 64 pixels.*

The other experimental parameters follow the actual experimental setup (see section 4) : wavelength λ is 634.9 nm, the SLM pixel pitch is 12.5 μm, the DMD pixel pitch is 5.5 μm, the camera pixel pitch is 5.5 μm and the propagation distance is set to 30 mm. As phase is recovered up to a constant phase factor, accuracy of the phase recovery is estimated by simply computing the minimal normalized RMS error when adding a constant phase factor to the recovered phase :

$$RMS_{phase} = \min_{\theta\in[0,2\pi]} \Big[\frac{1}{2\pi N} \parallel arg(E_{target}) - arg(e^{i\theta} E_{recovered}) \parallel_2 \Big]. \tag{13}$$

Here E_{target} is the target field, $E_{recovered}$ is the recovered field, and N is the size of the region where the RMS error is calculated (here 256).

Phase modulation (SLM) Let us start by assessing the effect of the modulation size. Theoretically [14], each pixel should be modulated randomly, however in a physical version of the experiment, the modulation should not diffract the beam so much that it is much larger than the sensor. If this is the

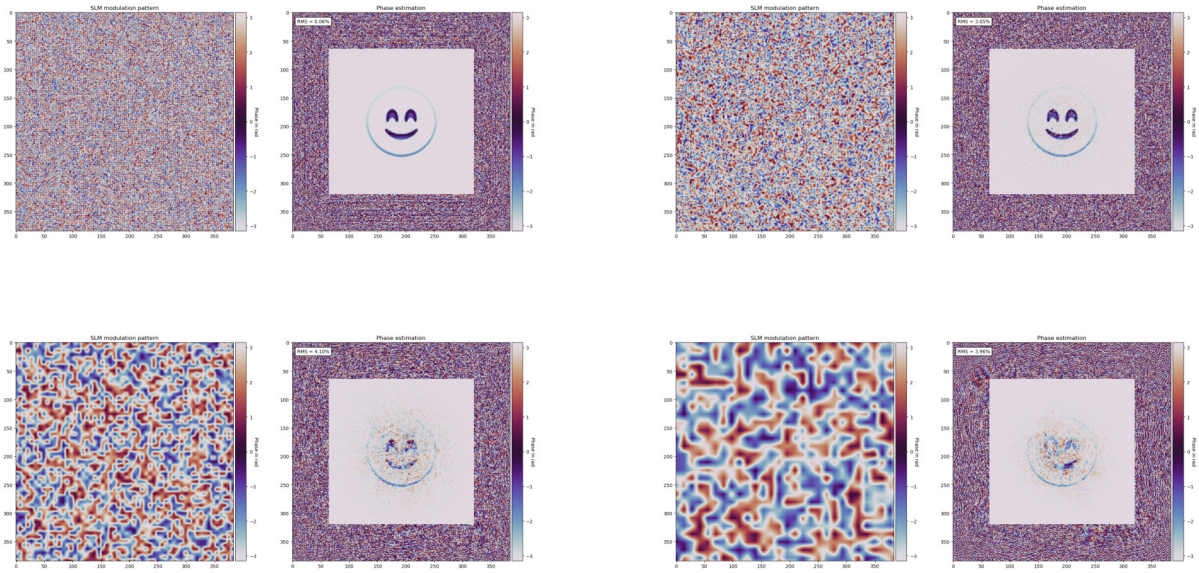


Figure 9: *Impact of the modulation pixel size on phase retrieval. Sample number is 8 and no noise is added. The RMS of the recovered phases from top left to bottom right : 0.06%, 3.05%, 4.10% and 3.96%.*

case, the sensor then acts as a kind of Fourier low-pass filter and prevents the retrieval of fine details. In the same way, if the modulation size is too big, the field will not interfere enough and the retrieval will not be perfect (or not occur at all). For repeatability, the seed of the random number generator is fixed. We compare the results of various pixel sizes in fig 9, setting the modulation number to 8 as recommended in [1].

We see in fig 9 that the quality of the recovered wavefront quickly deteriorates with the increasing pixel size. The RMS in the last 2 picture remains relatively "low" as there is a lot of the target phase pattern that is at $-\pi$, however the loss in quality is very clear. Note that the authors of [1] recommend a pixel size of 10 pixels to avoid excessive diffraction, but their SLM has a pixel size of 5.5 μm, which is significantly smaller than ours.

We can now look at the influence of the number of modulations on reconstruction accuracy. Taking the SLM modulation size at 1 pixel, we get the following RMS for various sample numbers :

Number of samples K	RMS of the recovered phase
2	0.078 %
8	0.059 %
16	0.049 %

Table 1: *Influence of sample number on recovery for phase modulations*

Augmenting the number of samples improves the convergence, but it also slows it down as the algorithm needs more iterations to converge and the iterations become slower because of the increased number of samples to process. This effect is more pronounced on larger images.

Finally, let us see now the effects of noise on the retrieval. Looking at the specifications of our camera (10 bit Point Grey Grasshopper NIR, 2048×2048 CMOS 1" sensor), we know that the dark noise is 0.02%. We can then realistically look at noise levels from 0.02% to 0.2%. The SLM modulation is kept at its finest size i.e one pixel. Firstly, we test the influence of noise without any noise averaging scheme (taking multiple pictures of the same SLM pattern to average noise). The results can be found

in table 2. We see that the retrieval is quite robust to noise : even at ten times the dark noise of the camera, we still recover a faithfull estimate of the target phase map. However the performance of the retrieval is of course reduced. We then redo the same experiment averaging the noise with 10 images for each sample. The results can be found in table 3. This has close to no effect (it even is a bit detrimental), even if we do the average of target amplitudes before the GS loop using 100 images. Also, we note that in the presence of noise, the algorithm stagnates more quickly. Note that in these "realistic" conditions, the performance of the retrieval stays on par with commercial devices such as the Phasics SID4-HR phase camera (following the design presented in [8]), whose rated accuracy on phase recovery is 3.75% .

Noise level	RMS of the recovered phase
0.2%	2.11 %
0.5%	2.81 %
1%	3.67 %
2%	4.12 %

Table 2: *Influence of noise on phase recovery for phase modulations*

Noise level	RMS of the recovered phase
0.2%	2.48 %
0.5%	3.60 %
1%	3.97 %
2%	4.13 %

Table 3: *Influence of noise on phase recovery with a 10 image averaging for phase modulations*

Intensity modulation (DMD) Because of the finer pitch of the DMD, the field gets much more diffracted after reflection. For this reason padding needs to be increased substantially in order to capture a satisfying amount of the field. The following benchmarks will use the same target field as previously, only with a padding of 256 pixels. Let us first study the influence of the number of samples. From an information theory standpoint, as binary intensity modulation give less information about the field than phase modulations, we expect to need more samples to retrieve the phase of the field. We know that on average only half of the pixels are "on", thus to get the information from all the pixels, we need one pattern and its opposite, so we need to multiply by two the number of pattern compared to phase modulations. But then, only half of the pixels have interfered together : "complementary pixels" (from one mask and its opposite) did not interfere together as displayed in fig 10.This is why we need to again multiply by two the number of samples. We subsequently start testing the modulation pixel size with 32 samples. The results of this test are in fig 11.

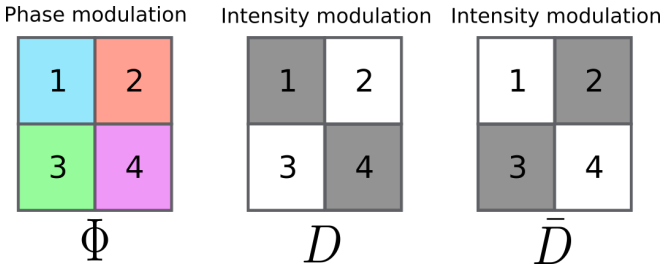


Figure 10: *Comparison of the retrieved information between phase and intensity modulations. With the phase modulation, all pixels (1,2,3,4) can interfere together. With intensity modulation, with pattern D only 2 and 3 can interfere, and in $\bar{D}$ only 1 and 4. This means that taking into accound D and $\bar{D}$ we have all pixels but interferences only for the couples (2,3) and (1,4) : we miss the couples (1,2) and (3,4).*

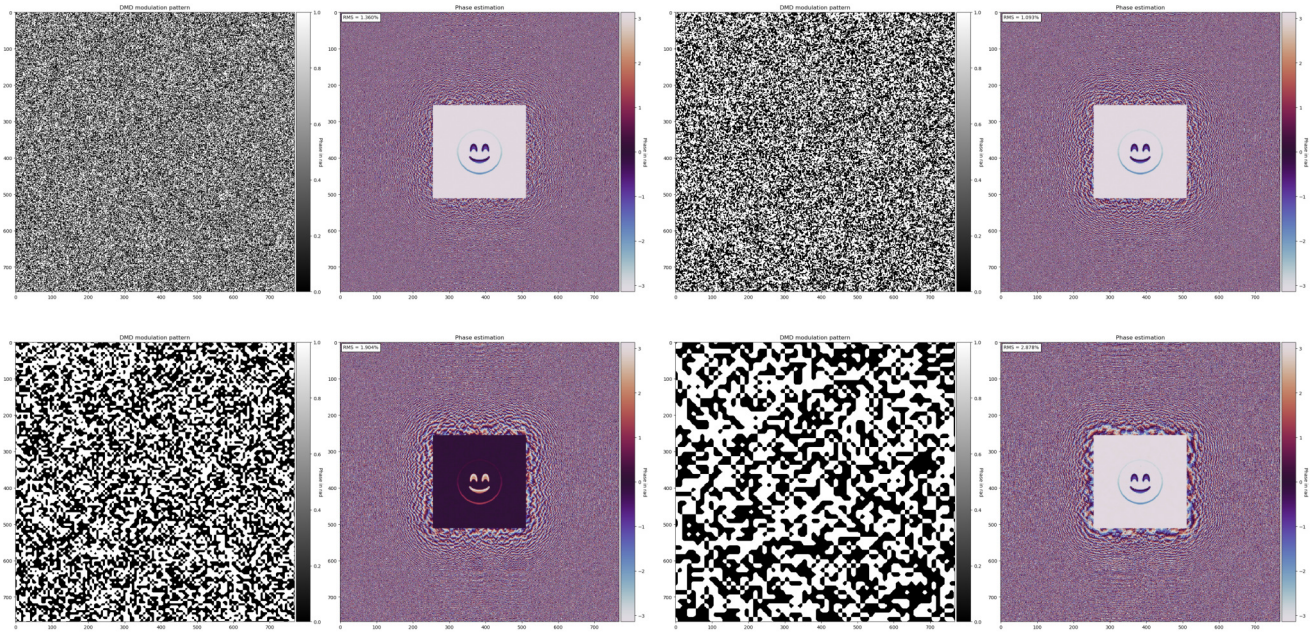


Figure 11: *Impact of the modulation pixel size on phase retrieval. Number of samples is 32 and no noise is added. The RMS of the recovered phases from top left to bottom right : 1.36%, 1.09%, 1.90% and 2.88%.*

We can see that with the intensity modulation, the optimal size is now at 2 pixels with a padding of 256 pixels. This was expected, as mentioned before, as the field is diffracted a lot more, 2 pixels is the best balance between collection efficiency and sampling of the wavefront. We notice also that the recovery is less sensitive to the change in size of the modulation, certainly due to the increased size of the padding increasing the total size of the images. As before in table 1 we now test the reconstruction accuracy versus the number of samples for a modulation size of 2 pixels : We see that below 8 samples, the recovery breaks down, which was to be expected as it is four times the minimal number theoretically possible, as seen previously with the SLM modulations. Finally, let us assess the robustness of the recovery against noise for 32 samples in tables 5 6.

Then again, the effect of noise averaging is marginal at best, which is quite surprising. Compared to the phase modulations, intensity modulations do seem to be very slightly less sensitive to noise. This can be explained by the higher modulation count.

The conclusion of these benchmarks is that the phase recovery from modulated sample is a very robust and efficient method, having no real bottleneck outside of experimental noise or computational

Number of samples K	RMS of the recovered phase
4	5.78 %
8	3.00 %
16	1.83 %
32	1.10 %
64	1.08 %
128	0.60 %

Table 4: *Influence of the number of samples on recovery for intensity modulations*

Noise level	RMS of the recovered phase
0.2%	1.61 %
0.5%	2.04 %
1%	2.57 %
2%	3.33 %

Table 5: *Influence of noise on the recovery for intensity modulations*

power, achieving sub 1% RMS accuracy for phase recovery.

4 Experimental results

Given the lockdown situation, I could not access the lab at the time I am writing these ligns. This means that actually getting an experimental setup to work was challenging to say the least. Hopefully, thanks to the amazing commitment of the lab, the necessary optical components, camera and SLM were sent at my home and I could patch up a working prototype. However due to shipping times and little issues with Thorlabs, I can only present very preliminary results.

Noise level	RMS of the recovered phase
0.2%	1.62 %
0.5%	2.03 %
1%	2.51 %
2%	3.30 %

Table 6: *Influence of noise on the recovery after 10 image averaging for intensity modulations*

4.1 Experimental setup

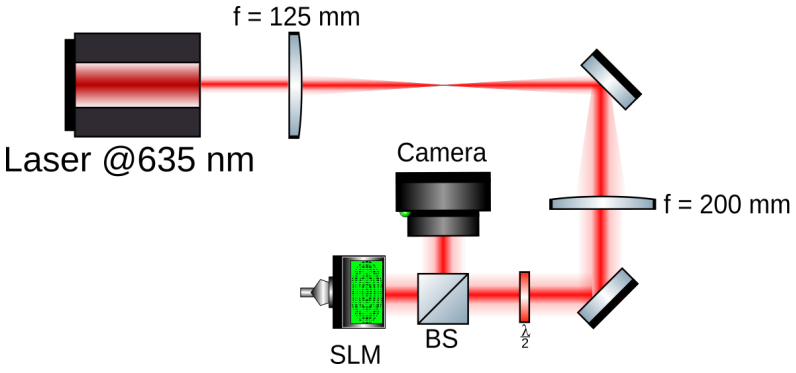


Figure 12: *Left : Experimental setup. The source laser is a Thorlabs CP635R laser diode. It is then expanded by a ×1.6 telescope. Optimal polarization for maximal SLM efficiency is selected using a half-wave plate (λ/2). The beam is then redirected after reflection to the camera using a non polarizing beam splitter (BS). Right : Picture of the actual setup.*

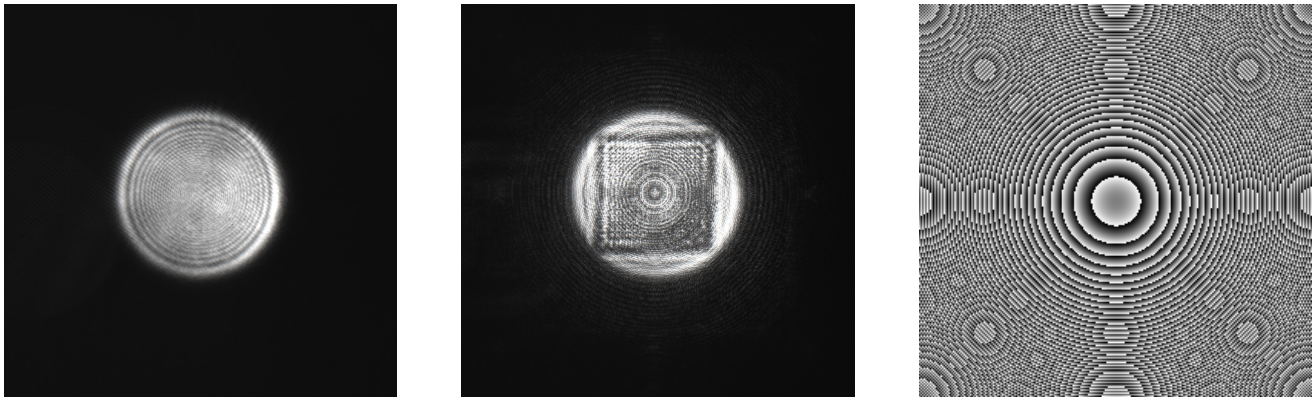
One of the main appeals of this method for phase retrieval is the simplicity of its experimental setup. As can be seen in 12, the actual measurement setup is composed only of a beam splitter (BS), an SLM and a camera. The test beam is generated using a Thorlabs CP635 red laser diode, then it is expanded using an afocal telescope from two 2 inches plano convex lenses ($f_1 = 125$ mm, $f_2 = 200$ mm). It is then filtered using a half waveplate, and is reflected by the SLM (Hammamatsu LCOS SLM X13138-01, 1280×1024, 12.5μm pitch). Finally the modulated beam is captured by a CMOS camera (10 bit Point Grey Grasshopper NIR, 2048×2048 CMOS 25.4 mm sensor).

4.2 SLM flatness measurement attempts

The first (and only) measurement I attempted to do was to retrieve the flatness of the SLM. The simplest known dephasing that the beam undergoes is the dephasing due to the SLM surface flatness upon reflection. This information is given by the manufaturer and we can thus check our results against it. Calibration of the SLM head is showed in fig 13. The goal of this measurement is to try and find the phase at the sensor plane, then try to back-propagate it to the SLM plane and compare it with the manufacturer calibration.

Figure 13: *Calibration of the SLM. The flatness of the SLM is here presented in a gray level picture, white is a phase of 0 rad, and black* 2π *rad.*

The setup is aligned so that the center of the beam coincides with the center of the SLM, and the center of the camera using a Fresnel lensed phase pattern. The images are presented in fig 14. The beam is not perfect and has a high-order Laguerre-Gauss profile, but this is actually beneficial as it helps for the alignment procedure by giving good reference points.



(a) *(b)* *(c)*

Figure 14: *(a) : Camera view when nothing is displayed on the SLM. (b) : Camera view when the Fresnel pattern is displayed. (c) : Fresnel pattern.*

After this optical alignment procedure that is done by hand, the residual misalignment is taken care of using the Python computer vision library OpenCV. I used the built-in tools of centroid detection to build a simple live centroid detector in order to sight the center of the beam (`alignment.py`), and then translate the image such as the centroid of the beam coincides with the center of the camera. For this I only capture an image and then artificially augment the contrast to facilitate centroid detection. Of course, this introduces artificial margins that are filled with 0, but in any case these areas should not have signal. This numerical alignment procedure is presented in fig 15. This alignment is quite critical as it ensures that when the modulation is applied to the numerical fields, it corresponds to the actual modulation recorded by the camera.

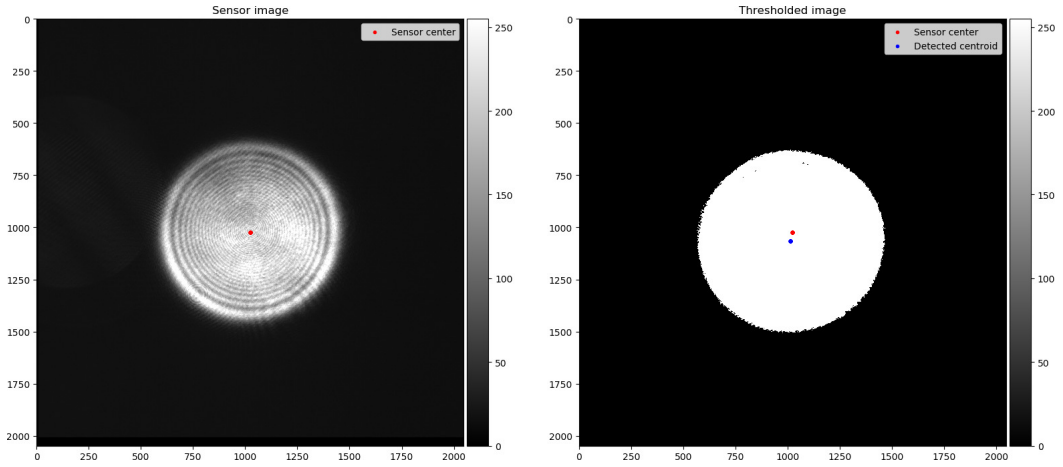
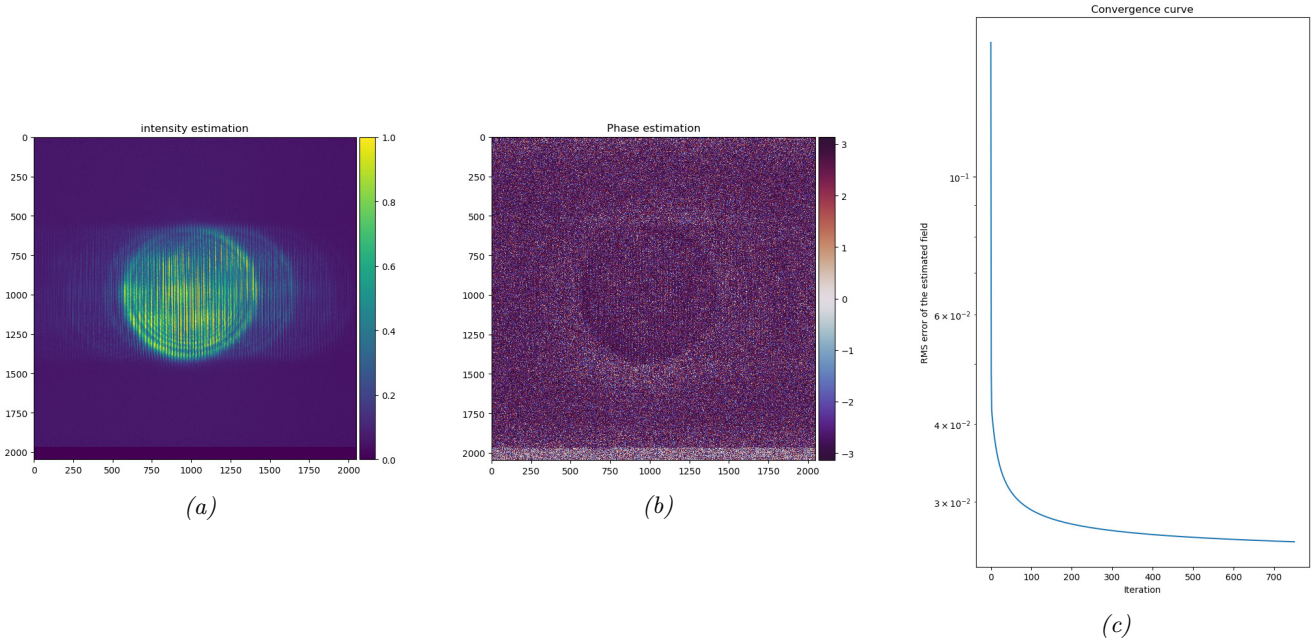


Figure 15: *Centroid detector. The code computes the offset between the red and blue points and can then shift the sensor image so that it is centered.*

The problem with this method is that the centroid detection fails to detect the actual center of the beam. There is a residual error of approximately 5 pixels. This can be seen in fig 15 where the offset between the blue and red points (on the right) is greater than the actual offset that can be seen (on the left). Attempting a measure with 8 samples (the maximum my laptop could handle), one observation per sample gives the following results in fig 16. We see that the amplitude reconstruction is blurred by interference fringes. I suspect they come from the offset between the image and phase field as explained previously. As this image is the mean of the recovery for each sample, the slight misalignment make the different samples interfere. Phase recovery is thus greatly hampered. Looking at the convergence curve, we see however that accuracy on the reconstruction of each sample is quite good as the RMS goes down under 3%. The misalignment might also be due to vibrations during the acquisition as my actual setup is quite shaky due to the fact that I unfortunately do not own an air stabilized optical table at home.

As a conclusion, the sensor does not work for the moment. However, even though the results presented in fig 16 do not show any successful phase retrieval, they confirm that at least amplitude recovery is functionnal, thus providing no infirmation of the theoretical simulations. Beam alignment seems to be the critical parameter that remains to be solved. For now, I have put very little effort in to this, and I am confident that I will find a more elegant solution to this alignment problem. Finally, I did not take the time to calibrate the SLM as the algorithm recovers the phase up to a global offset, and as the SLM calibration curve is linear, I thought that this would not be an issue at first. It is quite clear that it will be.



(a) *(b)* *(c)*

Figure 16: *Phase retrieval measurement attempt with 8 modulation samples. (a) : Reconstructed amplitude. (b) : Reconstructed phase.(c) : Convergence curve (log scale).*

5 Conclusion and outlook

In spite of the very particular circumstances of these first two months of my internship, the workload has been quite intensive and I think a lot a progress was made for this project. I had already a strong background in writing Python for physics and experiment control applications, but this project allowed me to bridge a new step in my coding by discovering GPU programming. If I get the time to do so, I would also like to try and implement this algorithm using a convex solver instead of a Gerchberg-Saxton loop to see if there is any improvement in performance. The material constraints have actually been a very strong catalyser as they forced me to write more efficiently, implementing practical and simple solutions. Furthermore on the experimental side, these conditions are also a good test as the intent of this wavefront sensor on the long run (if its efficiency is validated) would be to be deployed as a cheap and modular solution in the team.

On the more fundamental side, unlocking full phase retrieval would be a big achievement for the main Rubidium experiment, as it would unlock almost direct access to a lot of parameters that were previously hard to derive. Furthermore, it could also be used to set up a retroaction loop, feeding the output field at the input of the cell, thus turning the setup into an actual quantum simulator. This project is also the perfect way for me to set up a lot of the tools and knowledge I will need during my PhD in the team. Besides the purely technical aspect, it allowed me to adapt and integrate in the team in which I will spend the next three years working hard.

References

[1] Y. Wu, M. K. Sharma, and A. Veeraraghavan, "WISH: wavefront imaging sensor with high resolution," *Light Sci Appl*, vol. 8, no. 1, p. 44, 2019.

[2] I. Carusotto and C. Ciuti, "Quantum fluids of light," *Rev. Mod. Phys.*, vol. 85, no. 1, pp. 299–366, 2013.

[3] Q. Fontaine, T. Bienaimé, S. Pigeon, E. Giacobino, A. Bramati, and Q. Glorieux, "Observation of the bogoliubov dispersion relation in a fluid of light," *Physical Review Letters*, vol. 121, no. 18, 2018.

[4] Q. Fontaine, *Paraxial fluid of light in hot atomic vapor*. PhD thesis, Sorbonne Université, 2019.

[5] D. Gerace and I. Carusotto, "Analog hawking radiation from an acoustic black hole in a flowing polariton superfluid," *Phys. Rev. B*, vol. 86, no. 14, p. 144505, 2012.

[6] I. Carusotto, R. Balbinot, A. Fabbri, and A. Recati, "Density correlations and analog dynamical casimir emission of bogoliubov phonons in modulated atomic bose-einstein condensates," *Eur. Phys. J. D*, vol. 56, no. 3, pp. 391–404, 2010.

[7] B. C. Platt and R. Shack, "History and principles of shack-hartmann wavefront sensing," *J Refract Surg*, vol. 17, no. 5, pp. S573–S577, 2001.

[8] J.-C. Chanteloup, "Multiple-wave lateral shearing interferometry for wave-front sensing," *Appl. Opt.*, vol. 44, pp. 1559–1571, Mar 2005.

[9] L. P. Pitaevskij and S. Stringari, *Bose-Einstein condensation and superfluidity*. No. 164 in International series of monographs on physics, Oxford University Press, reprinted (with corrections) ed., 2016.

[10] Q. Fontaine, P.-E. Larré, G. Lerario, T. Bienaimé, S. Pigeon, D. Faccio, I. Carusotto, E. Giacobino, A. Bramati, and Q. Glorieux, "Interferences between bogoliubov excitations and their impact on the evidence of superfluidity in a paraxial fluid of light," *arXiv:2005.14328 [cond-mat.quant-gas]*, 2020.

[11] J. Armijo, T. Jacqmin, K. V. Kheruntsyan, and I. Bouchoule, "Probing three-body correlations in a quantum gas using the measurement of the third moment of density fluctuations," *Physical Review Letters*, vol. 105, Nov 2010.

[12] S. Finazzi and I. Carusotto, "Spontaneous quantum emission from analog white holes in a nonlinear optical medium," *Physical Review A*, vol. 89, no. 5, p. 053807, 2014.

[13] R. Gerchberg and W. Saxton, "A practical algorithm for the determination of phase from image and diffraction plane pictures," *Optik*, vol. 35, pp. 227–246, 1972.

[14] E. J. Candès, T. Strohmer, and V. Voroninski, "PhaseLift: Exact and stable signal recovery from magnitude measurements via convex programming," *Communications on Pure and Applied Mathematics*, vol. 66, no. 8, pp. 1241–1274, 2013.

[15] S. Mehrabkhani and M. Kuester, "Optimization of phase retrieval in the fresnel domain by the modified gerchberg-saxton algorithm," *arXiv:1711.01176 [physics]*, 2017.

[16] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-compatible library for NVIDIA GPU calculations," in *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017)*, p. 7, 2017.

[17] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[18] L. Dalcin, R. Bradshaw, K. Smith, C. Citro, S. Behnel, and D. Seljebotn, "Cython: The best of both worlds," *Computing in Science Engineering*, vol. 13, no. 02, pp. 31–39, 2011.

Addendum : Python codes

Listing 1: *WISH_lkb.py*

```python
# -*- coding: utf-8 -*-
"""
@author : Tangui ALADJIDI
After the Matlab code from Yicheng WU
"""


import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from time import time
from mpl_toolkits.axes_grid1 import make_axes_locatable
import time
import sys
import configparser
import cupy as cp
from cupyx.scipy.ndimage import zoom



"""
IMPORTANT NOTE : If the cupy module won't work, check that you have the right version of CuPy installed for you version
of CUDA Toolkit : https://docs-cupy.chainer.org/en/stable/install.html
If you are sure of you CuPy install, then it is possible that your nvidia kernel module froze or that some program
bars the access to CuPy. In this case reload your Nvidia module using these commands (in Unix) :
    sudo rmmod nvidia_uvm
    sudo modprobe nvidia_uvm
This usually happens after waking up you computer. A CPU version of the code is also available WISH_lkb_cpu.py
"""


class WISH_Sensor:
    def __init__(self, cfg_path):
        conf = configparser.ConfigParser()
        conf.read(cfg_path)
        self.d_SLM = float(conf["params"]["d_SLM"])
        self.d_CAM = float(conf["params"]["d_CAM"])
        self.wavelength = float(conf["params"]["wavelength"])
        self.z = float(conf["params"]["z"])  # propagation distance
        self.N_gs = int(conf["params"]["N_gs"])  # number of GS iterations
        self.N_mod = int(conf["params"]["N_mod"])  # number of modulation steps
```

```
        self.N_os = int(conf["params"]["N_os"])   #number of observations per image
        ↪  (to avg noise)
        self.Nim = self.N_mod * self.N_os
        self.threshold = float(conf['params']['mask_threshold'])  # intensity
        ↪  threshold for the signal region
        self.noise = float(conf['params']['noise'])
    def define_mask(self, I: np.ndarray, plot: bool = False):
        """
        A function to define the signal region automatically from the provided
    ↪  intensity and threshold
        :param I: intensity from which to define a signal region
        :param threshold: intensities below threshold are discarded
        :param plot: Plot or not the defined mask
        :return: mask_sr the defined mask
        """
        threshold = self.threshold
        h, w = I.shape
        mask_sr = np.zeros((h, w))
        # detect outermost non zero target intensity point
        non_zero = np.array(np.where(I > self.threshold))
        non_zero_offset = np.zeros(non_zero.shape)
        # offset relative to center
        non_zero_offset[0] = non_zero[0] - (h / 2) * np.ones(len(non_zero[0]))
        non_zero_offset[1] = non_zero[1] - (w / 2) * np.ones(len(non_zero[1]))
        # Determine radii of each non-zero point
        R_non_zero = np.sqrt(non_zero_offset[0] ** 2 + non_zero_offset[1] ** 2)
        R_max = np.where(R_non_zero == np.max(abs(R_non_zero)))[0][
            0]
        # if there are several equally far points, it takes the
        # first one
        i_max, j_max = int(h / 2 + int(abs(non_zero_offset[0][R_max]))), int(
            w / 2 + int(abs(non_zero_offset[1][R_max])))
        i_min, j_min = int(h / 2 - int(abs(non_zero_offset[0][R_max]))), int(
            w / 2 - int(abs(non_zero_offset[1][R_max])))
        delta_i = int(i_max - i_min)
        delta_j = int(j_max - j_min)
        if delta_i > delta_j:
            mask_sr[i_min:i_max, i_min:i_max] = 1
            k,l = i_min, i_max
        else:
            mask_sr[j_min:j_max, j_min:j_max] = 1
            k,l = j_min, j_max
        if plot:
            fig = plt.figure(0)
            ax1 = fig.add_subplot(121)
            ax2 = fig.add_subplot(122)
            divider1 = make_axes_locatable(ax1)
            divider2 = make_axes_locatable(ax2)
            cax1 = divider1.append_axes('right', size='5%', pad=0.05)
```

```
            cax2 = divider2.append_axes('right', size='5%', pad=0.05)
            im1=ax1.imshow(I, cmap="viridis")
            ax1.set_title("Intensity")
            im2=ax2.imshow(mask_sr, cmap="viridis", vmin=0, vmax=1)
            ax2.set_title(f"Signal region (Threshold = {threshold})")
            scat = ax2.scatter(non_zero[0][R_max], non_zero[1][R_max], color='r')
            scat.set_label('Threshold point')
            ax2.legend()
            fig.colorbar(im1, cax=cax1)
            fig.colorbar(im2, cax=cax2)
            plt.show()
        return mask_sr, k,l
    def crop_center(self, img, cropx, cropy):
        y, x = img.shape
        startx = x // 2 - (cropx // 2)
        starty = y // 2 - (cropy // 2)
        return img[starty:starty + cropy, startx:startx + cropx]
    @staticmethod
    def modulate(shape: tuple, pxsize: int = 10):
        """
        A function to randomly modulating a phase map without introducing too much
↪   high frequency noise
        :param phi: Phase map to be modulated
        :return: phi_m a modulated phase map to multiply to phi
        """
        # generate (N/10)x(N/10) random matrices that will then be upscaled through
        ↪  interpolation
        h, w = int(shape[0] / pxsize), int(shape[1] / pxsize)
        cp.random.seed(1)
        M = cp.random.rand(h, w)  # random matrix between [-1 , 1]
        phi_m = cp.asnumpy(zoom(M, (shape[0]/M.shape[0], shape[1]/M.shape[1])))
        return phi_m
    @staticmethod
    def modulate_binary(shape: tuple, pxsize: int = 10):
        """
        A function to randomly modulating a phase map without introducing too much
↪   high frequency noise
        :param phi: Phase map to be modulated
        :return: phi_m a modulated phase map to multiply to phi
        """
        # generate (N/10)x(N/10) random matrices that will then be upscaled through
        ↪  interpolation
        h, w = int(shape[0] / pxsize), int(shape[1] / pxsize)
        cp.random.seed(1)
        M = cp.random.choice(cp.asarray([0,1]), (h,w))   # random intensity mask
        #phi_m = np.kron(M, np.ones((10, 10)))
        phi_m = cp.asnumpy(zoom(M, shape[0]/M.shape[0]))
        return phi_m
    def gaussian_profile(self, I: np.ndarray, sigma: float):
```

```
        """

        :param I: Intensity to which a gaussian profile is going to be applied
        :param sigma: Standard deviation of the gaussian profile, in fraction of the provided intensity size
        :return: I_gauss : the "gaussianized" intensity
        """
        h, w = I.shape
        # define a radial position matrix
        R = np.zeros((h, w))
        for i in range(h):
            for j in range(w):
                R[i, j] = np.sqrt((h / 2 - i) ** 2 + (w / 2 - j) ** 2)
        sig = sigma * max(h, w)
        G = np.exp(-R ** 2 / (2 * sig ** 2))
        I_gauss = I * G
        return I_gauss
    @staticmethod
    def frt( A0: np.ndarray, d1: float, wv: float, z: float):
        """
        Implements propagation using Fresnel diffraction
        :param A0: Field to propagate
        :param d1: Sampling size of the field A0
        :param z : Propagation distance in metres
        :return: A : Propagated field
        """
        k = 2*np.pi / wv
        N = A0.shape[0]
        x = np.linspace(0, N - 1, N) - (N / 2) * np.ones(N)
        y = np.linspace(0, N - 1, N) - (N / 2) * np.ones(N)
        d2 = wv * z / (N*d1)
        X1, Y1 = d1 * np.meshgrid(x, y)[0], d1 * np.meshgrid(x, y)[1]
        X2, Y2 = d2 * np.meshgrid(x, y)[0], d2 * np.meshgrid(x, y)[1]
        R1 = np.sqrt(X1 ** 2 + Y1 ** 2)
        R2 = np.sqrt(X2 ** 2 + Y2 ** 2)
        D = 1 /(1j*wv*abs(z))
        Q1 = np.exp(1j*(k/(2*z))*R1**2)
        Q2 = np.exp(1j*(k/(2*z))*R2**2)
        if z >=0:
            A = D * Q2 * (d1**2) * np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(A0 * Q1, axes=(0,1)), axes=(0,1)), axes=(0,1))
        elif z<0:
            A = D * Q2 * ((N*d1) ** 2) * np.fft.fftshift(np.fft.ifft2(np.fft.ifftshift(A0 * Q1, axes=(0,1)), axes=(0,1)), axes=(0,1))
        return A
    @staticmethod
    def frt_gpu(A0: np.ndarray, d1: float, wv: float, z: float):
        """
```

```
        Implements propagation using Fresnel diffraction. Runs on a GPU using CuPy with a CUDA backend.
        :param A0: Field to propagate
        :param d1: Sampling size of the field A0
        :param wv: Wavelength in m
        :param z : Propagation distance in metres
        :return: A : Propagated field
        """
        k = 2*np.pi / wv
        N = A0.shape[0]
        x = cp.linspace(0, N - 1, N) - (N / 2) * cp.ones(N)
        y = cp.linspace(0, N - 1, N) - (N / 2) * cp.ones(N)
        d2 = wv * z / (N*d1)
        X1, Y1 = d1 * cp.meshgrid(x, y)[0], d1 * cp.meshgrid(x, y)[1]
        X2, Y2 = d2 * cp.meshgrid(x, y)[0], d2 * cp.meshgrid(x, y)[1]
        R1 = cp.sqrt(X1 ** 2 + Y1 ** 2)
        R2 = cp.sqrt(X2 ** 2 + Y2 ** 2)
        D = 1 /(1j*wv*abs(z))
        Q1 = cp.exp(1j*(k/(2*z))*R1**2)
        Q2 = cp.exp(1j*(k/(2*z))*R2**2)
        if z >=0:
            A =D * Q2 * (d1**2) * cp.fft.fftshift(cp.fft.fft2(cp.fft.ifftshift(A0 * Q1, axes=(0,1)), axes=(0,1)), axes=(0,1))
        elif z<0:
            A =D * Q2 * ((N*d1) ** 2) * cp.fft.fftshift(cp.fft.ifft2(cp.fft.ifftshift(A0 * Q1, axes=(0,1)), axes=(0,1)), axes=(0,1))

        return A
    @staticmethod
    def frt_gpu_s(A0: np.ndarray, d1: float, wv: float, z: float, plan = None):
        """
        Simplified Fresnel propagation optimized for GPU computing. Runs on a GPU using CuPy with a CUDA backend.
        :param A0: Field to propagate
        :param d1: Sampling size of the field A0
        :param wv: Wavelength in m
        :param z : Propagation distance in metres
        :return: A : Propagated field
        """
        N = A0.shape[0]
        D = 1 /(1j*wv*abs(z))
        if z >=0:
            A =cp.multiply(D*(d1**2), cp.fft.fftshift(cp.fft.fft2(cp.fft.ifftshift(A0, axes=(0,1)), axes=(0,1)), axes=(0,1)))
        elif z<0:
            A =cp.multiply(D * ((N*d1) ** 2), cp.fft.fftshift(cp.fft.ifft2(cp.fft.ifftshift(A0, axes=(0,1)), axes=(0,1)), axes=(0,1)))
```

```
219
220             return A
221         def u4Tou3(self, u4: np.ndarray, delta4: float, z3: float):
222             """
223             Propagates back a field from the sensor plane to the SLM plane
224             :param u4: Field to propagate back
225             :param delta4: Sampling size of the field u4
226             :param z3: Propagation distance in metres
227             :return: u3 the back propagated field
228             """
229             u3 = self.frt(u4, delta4, self.wavelength, -z3);
230             return u3
231         def process_SLM(self, slm: np.ndarray, N: int, delta3: float, type: str):
232             """
233             Scales the pre submitted SLM plane field (either amplitude of phase) to the
    ↪  right size taking into account the
234             apparent size of the SLM in the sensor field of view.
235             :param slm: Input SLM patterns
236             :param N: Size of the calculation (typically the sensor number of pixels)
237             :param N_batch: Number of images to generate
238             :param delta3: Sampling size of the SLM plane (typically the "apparent"
    ↪  sampling size wvl*z/N*d_Sensor )
239             :param type : "amp" / "phi" amplitude or phase pattern.
240             :return SLM: Rescaled and properly shaped SLM patterns of size (N,N,N_batch)
241             """
242             delta_SLM = self.d_SLM
243             N_batch = self.N_mod
244             if slm.dtype == 'uint8':
245                 slm = slm.astype(float)/256.
246             if slm.ndim == 3:
247                 #slm2 = slm[:, 421 : 1501, 0:N_batch] #takes a 1080x1080 square of the
                    ↪  SLM
248                 slm2 = slm[:, :, 0:N_batch]
249                 slm3 = np.empty((N,N,N_batch))
250                 #scale SLM slices to the right size
251                 for i in range(N_batch):
252                     slm1 = cp.asnumpy(zoom(cp.asarray(slm2[:,:,i]), delta_SLM / delta3))
253                     if slm1.shape[0]>N or slm1.shape[1]>N:
254                         #print("\rWARNING : The propagation distance must be too small
                            ↪  and the field on the sensor is cropped !")
255                         slm3[:,:,i]=self.crop_center(slm1, N, N)
256                     else :
257                         slm1 = np.pad(slm1, (int(np.ceil((N - slm1.shape[0]) / 2)), \
258                                              int(np.ceil((N - slm1.shape[1]) / 2))))
259                     if slm1.shape[0] > N and slm1.shape[1] > N:
260                         slm3[:, :, i] = slm1[0:N, 0:N]
261                     elif slm1.shape[0] > N:
262                         slm3[:,:,i] = slm1[0:N, :]
263                     elif slm1.shape[1] > N:
```

```
                    slm3[:,:,i] = slm1[:, 0:N]
                else :
                    slm3[:,:,i] = slm1
            if type == "phi":
                SLM = np.exp(1j * 2 * np.pi * slm3).astype(np.complex64)
            elif type == "amp":
                SLM = slm3.astype(np.complex64)
            else :
                print("Wrong type specified : type can be 'amp' or 'phi' ! ")
                raise
        elif slm.ndim == 2:
            #slm2 = slm[:, 421:1501]
            slm2 = slm
            slm3 = np.empty((N, N))
            # could replace with my modulate function
            # scale SLM slices to the right size
            slm1 = zoom(slm2, delta_SLM / delta3)
            slm1 = np.pad(slm1, (int(np.ceil((N - slm1.shape[0]) / 2)), \
                                 int(np.ceil((N - slm1.shape[1]) / 2))))
            if slm1.shape[0] > N and slm1.shape[1] > N:
                slm3 = slm1[0:N, 0:N]
            elif slm1.shape[0] > N:
                slm3 = slm1[0:N, :]
            elif slm1.shape[1] > N:
                slm3 = slm1[:, 0:N]
            else:
                slm3 = slm1
            if type == "phi":
                SLM = np.exp(1j * 2 * np.pi * slm3).astype(np.complex64)
            elif type == "amp":
                SLM = slm3.astype(np.complex64)
            else:
                print("Wrong type specified : type can be 'amp' or 'phi' ! ")
                raise
        return SLM
    def gen_ims(self, u3: np.ndarray, slm: np.ndarray, z3: float, delta3: float,
    ↪  noise: float):
        """
        Generates dummy signal in the sensor plane from the pre generated SLM
    ↪  patterns
        :param u3: Initial field in the SLM plane
        :param phi0 : Initial phase typically the calibration of the SLM
        :param slm : Pre generated slm patterns
        :param z3: Propagation distance in metres
        :param delta3: "apparent" sampling size of the SLM plane (as seen by the
    ↪  image plane from z3 m away)
        :param Nim: Number of images to generate
        :param noise: Intensity of the gaussian noise added to the images
        :return ims: Generated signal in the sensor plane of size (N,N,Nim)
```

```
        """
        N = u3.shape[0]
        Nim = self.Nim
        N_batch = self.N_mod
        N_os = self.N_os
        delta_SLM = self.d_SLM
        L_SLM = delta_SLM * 1080
        x = np.linspace(0, N - 1, N) - (N / 2) * np.ones(N)
        y = np.linspace(0, N - 1, N) - (N / 2) * np.ones(N)
        XX, YY = np.meshgrid(x,y)
        A_SLM = (np.abs(XX) * delta3 < L_SLM / 2) * (np.abs(YY) * delta3 < L_SLM /
        ↪  2)

        if slm.dtype=='uint8':
            slm = slm.astype(float)/256
        ims = np.zeros((N, N, Nim), dtype=float)
        for i in range(Nim):
            sys.stdout.write(f"\rGenerating image {i+1} out of {Nim} ...")
            sys.stdout.flush()
            a31 = u3 * A_SLM * slm[:,:,i//N_os]
            a31 = cp.asarray(a31)  #put the field in the GPU
            a4 = self.frt_gpu(a31, delta3, self.wavelength, z3)
            w = noise * cp.random.rand(N, N)
            ya = cp.abs(a4)**2 + w
            ya[ya<0]=0
            ims[:,:, i] = cp.asnumpy(ya)
            del a31, a4, ya
        return ims
    def process_ims(self, ims: np.ndarray, N: int):
        """
        Converts images to amplitudes and eventually resizes them.
        :param ims: images to convert
        :param N: Size of the sensor
        :return y0 : Processed field of size (N,N, Nim)
        """
        if ims.dtype!=float:
            ims=(ims/256).astype(float)
        y0 = np.real(np.sqrt(ims)); # change from intensity to magnitude
        y0 = np.pad(y0, (round((N - y0.shape[0]) / 2), round((N - y0.shape[1]) /
        ↪  2)))
        if y0.shape[0] > N:
            y0=y0[0:N,0:N,:]
        return y0
    def WISHrun(self, y0: np.ndarray, SLM: np.ndarray, delta3: float, delta4: float,
    ↪  plot: bool=True):
        """
        Runs the WISH algorithm using a Gerchberg Saxton loop for phase retrieval.
        :param y0: Target modulated amplitudes in the sensor plane
        :param SLM: SLM modulation patterns
```

```
        :param delta3: Apparent sampling size of the SLM as seen from the sensor plane
        :param delta4: Sampling size of the sensor plane
        :param N_os: Number of observations per image
        :param N_iter: Maximal number of Gerchberg Saxton iterations
        :param N_batch: Number of batches (modulations)
        :param plot: If True, plots the advance of the retrieval every 10 iterations
        :return u4_est, idx_converge: Estimated field of size (N,N) and the convergence indices to check convergence
                                      speed
        """
        wvl = self.wavelength
        z3 = self.z
        ## parameters
        N = y0.shape[0]
        N_batch = self.N_mod
        N_os = self.N_os
        N_iter = self.N_gs
        u3_batch = cp.zeros((N, N, N_os), dtype=cp.complex64) # store all U3 gpu
        u4 = cp.zeros((N, N, N_os), dtype=cp.complex64) # gpu
        y = cp.zeros((N, N, N_os), dtype=cp.complex64) # store all U3 gpu
        ## initilize a3
        k = 2 * np.pi / wvl
        xx = cp.linspace(0, N - 1, N, dtype=cp.float) - (N / 2) * cp.ones(N, dtype=cp.float)
        yy = cp.linspace(0, N - 1, N, dtype=cp.float) - (N / 2) * cp.ones(N, dtype=cp.float)
        X, Y = float(delta4) * cp.meshgrid(xx, yy)[0], float(delta4) * cp.meshgrid(xx, yy)[1]
        R = cp.sqrt(X ** 2 + Y ** 2)
        Q = cp.exp(1j*(k/(2*z3))*R**2)
        del xx, yy, X, Y, R
        SLM = cp.asarray(SLM)
        y0 = cp.asarray(y0)
        SLM_batch = SLM[:, :, 0]
        for ii in range(N_os):
            y0_batch = y0[:,:, ii]
            u3_batch[:,:, ii] = self.frt_gpu_s(y0_batch/Q, delta4, self.wavelength, -z3) * cp.conj(SLM_batch) #y0_batch gpu
        u3 = cp.mean(u3_batch, 2)
        #i_mask, j_mask = self.define_mask(np.abs(y0[:, :, 0]) ** 2, plot=True)[1:3]
        ## Recon run : GS loop
        idx_converge = np.empty(N_iter)
        for jj in range(N_iter):
            sys.stdout.flush()
            u3_collect = cp.zeros(u3.shape, dtype=cp.complex64)
            idx_converge0 = np.empty(N_batch)
            for idx_batch in range(N_batch):
                # put the correct batch into the GPU
```

```
            SLM_batch = SLM[:,:, idx_batch]
            y0_batch = y0[:,:, int(N_os * idx_batch): int(N_os * (idx_batch+1))]
            for _ in range(N_os):
                u4[:,:,_] = self.frt_gpu_s(u3 * SLM_batch, delta3,
                ↪  self.wavelength, z3) # U4 is the field on the sensor
                y[:,:,_] = y0_batch[:,:,_] * cp.exp(1j * cp.angle(u4[:,:,_]))
                ↪  #impose the amplitude
                #[:,:,_] = u4[:,:,_]
                #y[i_mask:j_mask,i_mask:j_mask,_] =
                ↪  y0_batch[i_mask:j_mask,i_mask:j_mask,_] \
                #                                   * cp.exp(1j *
                ↪  cp.angle(u4[i_mask:j_mask,i_mask:j_mask,_]))
                u3_batch[:,:,_] = self.frt_gpu_s(y[:,:,_], delta4,
                ↪  self.wavelength, -z3) * cp.conj(SLM_batch)
            u3_collect = u3_collect + cp.mean(u3_batch, 2) # collect(add) U3
            ↪  from each batch
            # convergence index matrix for each batch
            idx_converge0[idx_batch] = (1/N)*\
            cp.linalg.norm((cp.abs(u4)-(1/N**2)*cp.sum(cp.abs(SLM_batch))*
            y0_batch)*(y0_batch>0)) #eventual mask absorption

        u3 = (u3_collect / N_batch) # average over batches
        idx_converge[jj] = np.mean(idx_converge0) # sum over batches
        sys.stdout.write(f"\rGS iteration {jj + 1}")
        sys.stdout.write(f"  (convergence index : {idx_converge[jj]})")



        if jj % 10 == 0 and plot:
            u4_est = cp.asnumpy(self.frt_gpu_s(u3, delta3, self.wavelength, z3)
            ↪  * Q)
            plt.close('all')
            fig=plt.figure(0)
            fig.suptitle(f'Iteration {jj}')
            ax1=fig.add_subplot(121)
            ax2=fig.add_subplot(122)
            im=ax1.imshow(np.abs(u4_est), cmap='viridis')
            ax1.set_title('Amplitude')
            ax2.imshow(np.angle(u4_est), cmap='viridis')
            ax2.set_title('Phase')

            fig1=plt.figure(1)
            ax = fig1.gca()
            ax.plot(np.arange(0,jj,1), idx_converge[0:jj], marker='o')
            ax.set_xlabel('Iterations')
            ax.set_ylabel('Convergence estimator')
            ax.set_title('Convergence curve')
            plt.show()
            time.sleep(2)
```

```
            # exit if the matrix doesn't change much
            if jj > 1:
                if cp.abs(idx_converge[jj] - idx_converge[jj - 1]) /
                ↪  idx_converge[jj] < 1e-5:
                #if cp.abs(idx_converge[jj]) < 5e-3:
                #if idx_converge[jj]>idx_converge[jj-1]:
                    print('\nConverged. Exit the GS loop ...')
                    #idx_converge = idx_converge[0:jj]
                    idx_converge = cp.asnumpy(idx_converge[0:jj])
                    break
        u4_est = self.frt_gpu_s(u3, delta3, self.wavelength, z3) * Q #propagate
        ↪  solution to sensor plane
        return u3, u4_est, idx_converge

```

Listing 2: *WISH_measurement.py*

```
# -*- coding: utf-8 -*-
"""
Created by Tangui Aladjidi at 28/05/2020
"""

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from time import time
from mpl_toolkits.axes_grid1 import make_axes_locatable
import time
import sys
import configparser
import cupy as cp
from scipy.ndimage import zoom
from WISH_lkb import WISH_Sensor
import slmpy
import cv2
import EasyPySpin

#WISH routine
def alignment(frame):
    frame_blurred = cv2.blur(frame, (12, 12))
    ret1, thresh = cv2.threshold(frame, 70, 255, 0)
    thresh_blurred = cv2.blur(thresh, (12, 12))
    M = cv2.moments(thresh_blurred)
    cX = int(M["m10"] / M["m00"])
    cY = int(M["m01"] / M["m00"])
    T = np.float32([[1, 0, int(frame.shape[1]/2) - cX], [0, 1, int(frame.shape[0]/2)
    ↪  - cY]])
```

```
    frame_s = cv2.warpAffine(frame, T, frame.shape)
    return frame_s
def main():
    #start timer
    T0 = time.time()
    #instantiate WISH
    Sensor = WISH_Sensor("wish_3.conf")
    wvl = Sensor.wavelength
    z3 = Sensor.z
    delta4 = Sensor.d_CAM
    '''
    To generate images from scratch
    #I0 = np.array(Image.open('intensities/harambe_512_full.bmp'))[:,:,0]
    #I0 = I0.astype(float)/256
    #I0 = np.pad(I0.astype(np.float) / 256, (256, 256))  # protection band
    im = np.array(Image.open('intensities/I0_256_full.bmp'))[:,:,0]
    phi0 = np.array(Image.open('phases/smiley_256.bmp'))[:,:,0]
    im = cp.asnumpy(zoom(cp.asarray(im), 2))
    phi0 = cp.asnumpy(zoom(cp.asarray(phi0), 2))
    u40 = np.pad(im.astype(np.float)/256, (768, 768)) #protection band
    u40 = Sensor.gaussian_profile(u40, 0.5)
    phi0 = np.pad(phi0.astype(np.float)/256, (768,768)) #protection band
    u40 = u40 * (np.exp(1j * phi0 * 2 * np.pi))
    u40=u40.astype(np.complex64)
    N = u40.shape[0]
    delta3 = wvl * z3 / (N * delta4)
    u30 = Sensor.u4Tou3(u40, delta4, z3)
    ## forward prop to the sensor plane with SLM modulation
    print('Generating simulation data images ...')
    noise = Sensor.noise
    '''
    slm = np.zeros((1080, 1920,Sensor.N_mod))
    slm_type = 'SLM'
    #Setting up the camera for acquisition
    Cam = EasyPySpin.VideoCapture(0) #by default camera 0 is the laptop webcam

    N = int(Cam.get(cv2.CAP_PROP_FRAME_WIDTH)*1)
    delta3 = wvl * z3 / (N * delta4)
    ims = np.zeros((N,N,Sensor.Nim))
    if slm_type=='DMD':
        slm = np.ones((1080, 1920, Sensor.N_mod))
        slm_display = slmpy.SLMdisplay()
        slm_display.updateArray(slm[:, :, 0])
        print(f"Displaying 1 st SLM pattern")
        for obs in range(Sensor.N_os):
            ret, frame = Cam.read()
            frame = alignment(frame)
            ims[:, :, obs] = zoom(cv2.flip(frame, 0), 0.5)
        for i in range(1,int(Sensor.N_mod/2)):
```

```
            slm[:, :, 2 * i] = Sensor.modulate_binary((1080, 1920), pxsize=1)
            for obs in range(Sensor.N_os):
                ret, frame = Cam.read()
                frame = alignment(frame)
                ims[:,:,2 * i+obs]= zoom(cv2.flip(frame, 0), 0.5)
            slm[:, :, 2 * i + 1] = np.ones((1080, 1920)) - slm[:, :, 2 * i]
            for obs in range(Sensor.N_os):
                ret, frame = Cam.read()
                frame = alignment(frame)
                ims[:,:,2 * i + 1 + obs]= zoom(cv2.flip(frame, 0), 0.5)
    elif slm_type=='SLM':
        slm = np.ones((1024, 1280, Sensor.N_mod))
        slm_display = slmpy.SLMdisplay()
        slm_display.updateArray(slm[:, :, 0])
        print(f"Displaying 1 st SLM pattern")
        for obs in range(Sensor.N_os):
            ret, frame = Cam.read()
            frame = alignment(frame)
            ims[:, :, obs] = zoom(cv2.flip(frame, 0), 1)
        for i in range(1,Sensor.N_mod):
            slm[:,:,i]=Sensor.modulate((1024,1280), pxsize=1)
            slm_display.updateArray(slm[:,:,i])
            for obs in range(Sensor.N_os):
                ret, frame = Cam.read()
                frame = alignment(frame)
                ims[:,:,i+obs]= zoom(cv2.flip(frame, 0), 1)
            print(f"Displaying {i+1} th SLM pattern")
        slm_display.close()
        Cam.release()
    if slm_type =='DMD':
        SLM = Sensor.process_SLM(slm, N, delta3, type="amp")
        SLM[np.abs(SLM) > 0.5] = 1 + 1j*0
        SLM[SLM <= 0.5] = 0 + 1j*0
        #fig = plt.figure(1)
        #ax1 = fig.add_subplot(121)
        #ax2 = fig.add_subplot(122)
        #ax1.imshow(np.abs(SLM[:, :, Sensor.N_os]), vmin=0, vmax=1)
        #ax2.imshow(np.abs(u30), vmin=0, vmax=1)
        #plt.show()
    elif slm_type == 'SLM':
        SLM = Sensor.process_SLM(slm, N, delta3, type="phi")
        #fig = plt.figure(1)
        #ax1 = fig.add_subplot(121)
        #ax2 = fig.add_subplot(122)
        #ax1.imshow(np.angle(SLM[:,:,Sensor.N_os]), vmin=-np.pi, vmax = np.pi)
        #ax2.imshow(np.abs(u30), vmin=0, vmax=1)
        #plt.show()
    #ims = Sensor.gen_ims(u30, SLM, z3, delta3, noise)

```

```
    print('\nCaptured images are simulated')
    #reconstruction
    #process the captured image : converting to amplitude and padding if needed
    ims=(ims/255.0).astype(np.float)
    y0 = Sensor.process_ims(ims, N)
    plt.imshow(y0[:,:,Sensor.N_os], vmin=0, vmax=1)
    plt.scatter(N/2,N/2, color='r', marker='.')
    plt.show()
    ##Recon initialization
    T_run_0=time.time()
    u3_est, u4_est, idx_converge = Sensor.WISHrun(y0, SLM, delta3, delta4,
    ↪  plot=False)
    T_run=time.time()-T_run_0
    #phase_rms = cp.corrcoef(cp.ravel(cp.angle(cp.asarray(u40))),
    ↪  cp.ravel(cp.angle(u4_est)))[0,1]
    u3_est = cp.asnumpy(u3_est)
    u4_est = cp.asnumpy(u4_est)
    #phase_RMS =(1/N) * np.array(
    #    [np.linalg.norm((np.angle(u40)-np.angle(np.exp(1j*th)*u4_est))*(np.abs(u40)
    ↪  > 0)) for th in
    #     np.linspace(-np.pi, np.pi, 256)])
    #phase_rms = np.min(phase_RMS)
    #phase_rms
    ↪  =(1/N)*np.linalg.norm((np.angle(u30)-np.angle(u3_est))*(np.abs(u30)>0))
    #print(f"\n Phase RMS is : {phase_rms}")
    #total time
    T= time.time()-T0
    print(f"\n Time spent in the GS loop : {T_run} s")
    print(f"\n Total time elapsed : {T} s")
    fig=plt.figure()
    #ax1 = fig.add_subplot(231)
    #ax2 = fig.add_subplot(232)
    #ax3 = fig.add_subplot(233)
    #ax4 = fig.add_subplot(234)
    #ax5 = fig.add_subplot(236)
    ax3 = fig.add_subplot(131)
    ax4 = fig.add_subplot(132)
    ax5 = fig.add_subplot(133)
    #divider1 = make_axes_locatable(ax1)
    #cax1 = divider1.append_axes('right', size='5%', pad=0.05)
    #divider2 = make_axes_locatable(ax2)
    #cax2 = divider2.append_axes('right', size='5%', pad=0.05)
    divider3 = make_axes_locatable(ax3)
    cax3 = divider3.append_axes('right', size='5%', pad=0.05)
    divider4 = make_axes_locatable(ax4)
    cax4 = divider4.append_axes('right', size='5%', pad=0.05)
    #im1=ax1.imshow(np.abs(u40), cmap='viridis', vmin=0, vmax=1)
    #ax1.set_title('Amplitude GT')
    #im2=ax2.imshow(np.angle(u40), cmap='twilight_shifted',vmin=-np.pi, vmax=np.pi)
```

```
    #ax2.set_title('Phase GT')
    im3=ax3.imshow(abs(u4_est)**2, cmap='viridis', vmin=0, vmax=1)
    ax3.set_title('intensity estimation')
    im4=ax4.imshow(np.angle(u4_est), cmap='twilight_shifted', vmin=-np.pi,
    ↪  vmax=np.pi)
    ax4.set_title('Phase estimation')
    ax5.plot(np.arange(0, len(idx_converge),1), idx_converge)
    ax5.set_title("Convergence curve")
    ax5.set_xlabel("Iteration")
    ax5.set_ylabel("RMS error of the estimated field")
    #fig.colorbar(im1, cax=cax1)
    #fig.colorbar(im2, cax=cax2)
    fig.colorbar(im3, cax=cax3)
    fig.colorbar(im4, cax=cax4)
    plt.show()
if __name__=="__main__":
    main()
```

Listing 3: *wish.conf*

```
[params]
#sensor pixel size
d_CAM = 5.5e-6
#d_CAM = 6.4e-6
#SLM pixel size
#d_SLM = 6.4e-6
d_SLM = 12.5e-6
#in m
wavelength = 635.9e-9
#propagation distance in m
z = 30e-3
 #max number of GS iterations
N_gs = 1000
#modulation number 1 is disabled.
N_mod = 32
#number of observations per image (to avg noise)
N_os = 1
mask_threshold = 5e-2
#artificial noise added to the generated images
noise = 0
```

Listing 4: *alignment.py*

```
# -*- coding: utf-8 -*-
"""
Created by Tangui Aladjidi at 28/05/2020
"""
```

```

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import cv2
import EasyPySpin

Cam = EasyPySpin.VideoCapture(0)
fig=plt.figure(0)
ax = fig.add_subplot(121)
ax1 = fig.add_subplot(122)
divider = make_axes_locatable(ax)
cax = divider.append_axes('right', size='5%', pad=0.05)
divider1 = make_axes_locatable(ax1)
cax1 = divider1.append_axes('right', size='5%', pad=0.05)

frame_nbr=0
while True:
    frame_nbr+=1
    ret, frame = Cam.read()
    frame=cv2.flip(frame, 0)
    frame_blurred = cv2.blur(frame, (12,12))
    ret1, thresh = cv2.threshold(frame, 70, 255, 0)
    thresh_blurred = cv2.blur(thresh, (12, 12))
    im1 = ax1.imshow(thresh, cmap='gray')
    ax1.set_title("Thresholded image")
    M = cv2.moments(thresh_blurred)
    cX = int(M["m10"] / M["m00"])
    cY = int(M["m01"] / M["m00"])
    T = np.float32([[1, 0, 1024-cX], [0, 1, 1024-cY]])
    frame_s = cv2.warpAffine(frame, T, frame.shape)
    im = ax.imshow(frame_s, cmap='gray')
    ax.set_title("Sensor image")
    scat=ax.scatter(1024,1024, color='r', marker='.')
    scat10=ax1.scatter(1024,1024, color='r', marker='.')
    scat11=ax1.scatter(cX,cY, color='b', marker='.')
    scat.set_label('Sensor center')
    scat10.set_label('Sensor center')
    scat11.set_label('Detected centroid')
    if frame_nbr==1:
        ax.legend()
        ax1.legend()
        plt.colorbar(im, cax)
        plt.colorbar(im1, cax1)
    plt.pause(0.01)
plt.show()
Cam.release()
cv2.destroyAllWindows()
```