## Turbulence Simulator

A numerical simulator has been implemented in MATLAB in order to investigate the turbulence effects on the propagation of an optical beam. This simulator is capable to simulate:

- any shape of beam (Gaussian, Bessel, Orbital Angular Momentum...)
- any model for turbulence (Kolmogorov, Modified Von Karman...)
- any type of optical elements inserted in the propagation path (provided that the amplitude and phase transverse profile is known)
- all the parameters listed in **Error! Reference source not found.** section **Error! Reference source not found.**.

The implemented approach to solve the propagation wave equation is the Split-Step method. This technique involving the Fresnel-Kirchhoff diffraction integral is used to model the propagation through turbulent medium, where the propagated beam alternatively passes through a purely free space region (considering only the divergence effect) and a non-diffractive inhomogeneous medium (random phase screen with turbulence effects).

These phenomena are treated in the two conjugate domains: while the Fresnel integral is solved in the spatial frequency domain, turbulence is treated in the spatial one. Consequently, the link is partitioned in a sequence of sub-sections, whose number is reasonably chosen.

Each phase screen contains information about all the turbulence introduced by the subsection $\Delta z$. More precisely, assuming the Kolmogorov function (1) as the PSD of turbulence, each phase screen is randomly generated by a seed, a random complex array which is low-pass-filtered, and then transformed back to the space domain. As a result, different patterns, all following same statistics, are generated.

The number of phase screens $N_{ps}$ is decided in order to guarantee that each one introduces small turbulence, meaning that the $\sigma_R^2$ should be less than 0.1 [6]. For instance, if L = 320 m, the turbulence regime is low enough (see Figure 5) that a single phase screen can be used. Since $\sigma_R^2$ scales up with the square of L, the number of phase screens $N_{ps}$ scales quadratically with the length of the link and hence for longer links multiple phase screens are needed. Anyway, a convergence study has been done and the result are quite robust (see section 3.4).

To introduce time correlation, at each time step the phase screen is given by a linear combination of the current phase screen with a random uncorrelated phase screen, through the employment of a correlation coefficient

$$PS_{k+1}(t_{k+1}) = \rho \cdot PS_k + (1-\rho)^{1/2} \cdot PS_N \qquad (1)$$

where $PS_N$ is a random generated phase screen. Moreover, $\rho$ ranges from 0, which means no time-correlation, to 1, corresponding to maximum correlation between consecutive phase screens.

In general, all the phase screens, being randomly created, are uncorrelated one each other. It is possible to monitor the temporal evolution of the beam, generating phase screens with a controlled correlation.

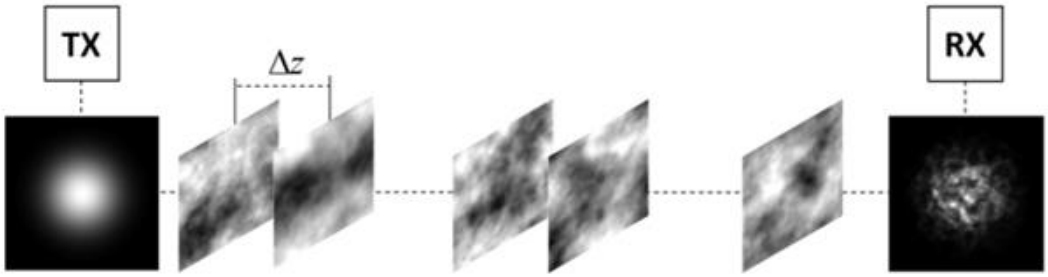


*Figure 1: Propagation of a beam through turbulence can be simulated using phase screens, placed at Δz one each other. It is clear that the received beam is distorted resulting in fading [7].*

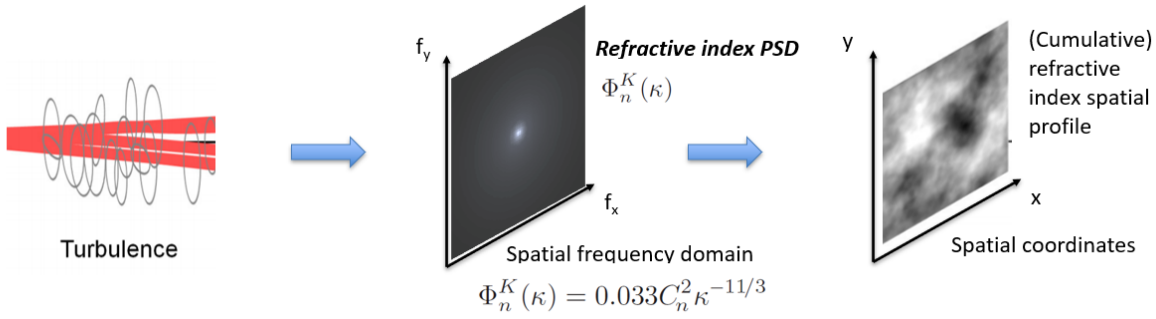


*Figure 2: General scheme depicting process of creation of a phase screen.*

---

## User Manual for Turbulence Simulator

As presented in section 0, the turbulence simulator is able to simulate the propagation of a laser beam through a turbulent medium, modelled as a sequence of random phase screens. The divergence of the beam is clearly taken into account while the backreflections are neglected.

In this section a quick tutorial illustrating its main features is provided.

1. The first step is running the `startup.m` script, which loads the necessary dependencies in the MATLAB path and define a set of global constants and settings.
2. After completing point n. 1. It is necessary to declare the simulation window dimensions. This is done by instantiating the `SimulationRegion` class. Once a `SimulationRegion` object is declared, this will provide all the information about the computational grid used to simulate the propagation of the laser beam.
3. Once the simulation window is defined, the initial optical field $U_{\text{in}}$ entering inside the channel has to be declared. This is done by specifying a $M \times N$ complex matrix, whose

size must be compatible with the size specified in the SimulationRegion object. A good practice to avoid bugs is to use the meshgrids provided by SimulationRegion in the definition of $U_{\text{in}}$, as in the example below

```
% Define a simulation window 5cmx6cm, with 512 horizontal and
% vertical samples.
Lx = 5e-2;
Ly = 6e-2;
Nx = 512;
Ny = 512;
sim_reg = SimulationRegion(Lx, Ly, Nx, Ny);

% Generate a gaussian beam with beam waist of 3mm centered
% inside the simulation region
w_in = 3e-3; % beam spot size
U_in= exp(-(sim_reg.X.^2 + sim_reg.Y.^2)/(w_in^2));
% Normalize the gaussian beam
U_in = U_in/sqrt(int2(abs(U_in).^2, sim_reg));
```

4. The propagation of the laser beam is done by two functions: free_prop for the propagation inside a linear isotropic and homogeneous medium and turbulent_prop for the propagation inside a turbulent medium. Both functions accept as a parameter an electric field distribution U_in specified as a $M \times N$ complex matrix as in the last step, a SimulationRegion object, and a positive number L representing the length for which the beam propagate. They always return a $M \times N$ complex matrix U_out which contains the electric field distribution after propagation. The following snippet, provides an example for the propagation in a turbulent medium.

```
% Turbulence parameters (Modified Von Karman)
Cn2=1e-10; % [m^(-2/3)]
D0=1000; % Outer scale   [m]
d0=1e-6; %  Inner scale  [m]
corr_coeff = 0.9; % temporal correlation coefficient
L = 10; % Propagation distance [m]

% Simulation parameters
n_screen=1; % Number of phase screen
n_iter = 200; % Iteration number

% Preallocate memory
U_out = zeros(sim_reg.N_x, sim_reg.N_y, n_iter);
phase_no_turb = angle(free_prop(U_in, sim_reg, L));
U_rec = receiver.U.*exp(1i*phase_no_turb);
phase_screen = zeros(sim_reg.N_x, sim_reg.N_y, n_screen,
n_iter);
```

```
c = zeros(1,n_iter);
phase_screen_old=zeros(sim_reg.N_x, sim_reg.N_y);

tic;

for i = 1:n_iter
    % propagation through a concentrated turbulent medium
    [U_out(:, :, i), phase_screen_old] = turbulent_prop(U_in,
sim_reg, ...
        L, Cn2, D0, d0, n_screen, phase_screen_old,
corr_coeff);
    phase_screen(:, :, :, i) = phase_screen_old;
end

computation_time = toc;

fprintf("computation time [s]: %f \n", computation_time)
```

5. As it can be observed in the last snippet, the function `turbulent_prop` adopts a MVK model, so it is necessary to specify the $C_n^2$, but the inner scale and outer scale dimensions. Moreover, it is possible to specify the number of phase screen employed. Under the hook, everytime the function `turbulent_prop` is called, a set of random phase screens obeying the MVK PSD is generated and the split-step method the beam propagates through them. For this reason it is important to repeat the execution of the algorithm a for a reasonable number of iterations, such that a proper statistic is obtained.

6. In case time-correlated phase screens are employed, it is necessary to reuse the phase screens of the last iteration for generating the new ones. To do so, the `turbulent_prop` also returns, a $M \times N \times L$ tensor, where $L$ is the number of phase screens employed, containing the phase screen used and they can be passed again to the function through the parameter `phase_screen_old`

The simulator also contains some utilities function which can help in the post-processing of the turbulence simulation. They can help in computing the main turbulence parameters such as the fried parameter, or the scintillation index. For a complete list, look at src folder and to the documentation of each method.

It is a good practice to pre-allocate the memory for the array used to store the optical field and the phase screens. It is also suggested to store them into a $M \times N \times L$ tensor (as it is done in the last snippet), since most of the post-processing functions assumes that the data are stored in 3-rank tensors this way and in particular that the last index corresponds to different realization of the turbulent process. An example of the use of these functions is provided in `simulate_artificial_turbulence.m` in the `example` folder.