Wavefront Sensor-less Adaptive Optics. Report 2019-2020

Abstract

This is an overview of my attempt at wavefront sensor-less adaptive optics and the work carried out to accomplish this. Although tests failed for a real adaptive optics setup, it appears to be possible and relatively quick in simulations. Following from previous attempts, a few optimisation algorithms are considered including the Nelder Mead simplex method, gradient descent, and my very own coordinate search. In this report I will be explaining in detail how the optimisation algorithms work, how to apply them in adaptive optics, how to simulate basic adaptive optics and how to apply machine learning.

Contents

- Introduction 1
- Numerical optimisation 2 - 6
- Sensor-less phase optimisation 7 - 14
- Deformable mirror and far-field simulation 15 - 26
- Neural Networks 27 - 33
- Application of Deep learning to phase optimisation 34 - 40

Introduction

The goal is to perform Adaptive Optics without the use of a wavefront sensor. Removing the need for a sensor is not just a way to lower the cost of the AO setup, but also frees up space for more optics on the optical table.

Normally, the wavefront sensor measures the influence of every actuator on the deformable mirror and the influence matrix is constructed. By getting a modal description of the influences, a command matrix is computed that returns the required voltages to flatten a given wavefront.

To perform sensorless correction means that you have to get some information about the quality of the wavefront indirectly. The farfield is probably the best way to do that. It is also a diagnostic that all laser labs use, so no new equipment is added to the setup. When the wavefront is highly aberrated, the resulting focal spot will be of low quality. This means that a high-quality spot, corresponds to a high quality (or flat) wavefront.

By using an optimisation algorithm with the objective function being the farfield quality, sensorless phase optimisation is possible.

Author

Petros Oratiou, CALTA
06/07/2020

Optimisation Algorithms

Optimisation is a very basic concept, however the methods that achieve it can be very complicated. From a simple hill climbing algorithm to a deep neural network, the idea to optimise a function. Optimisation refers to finding the minimum or maximum of that function.

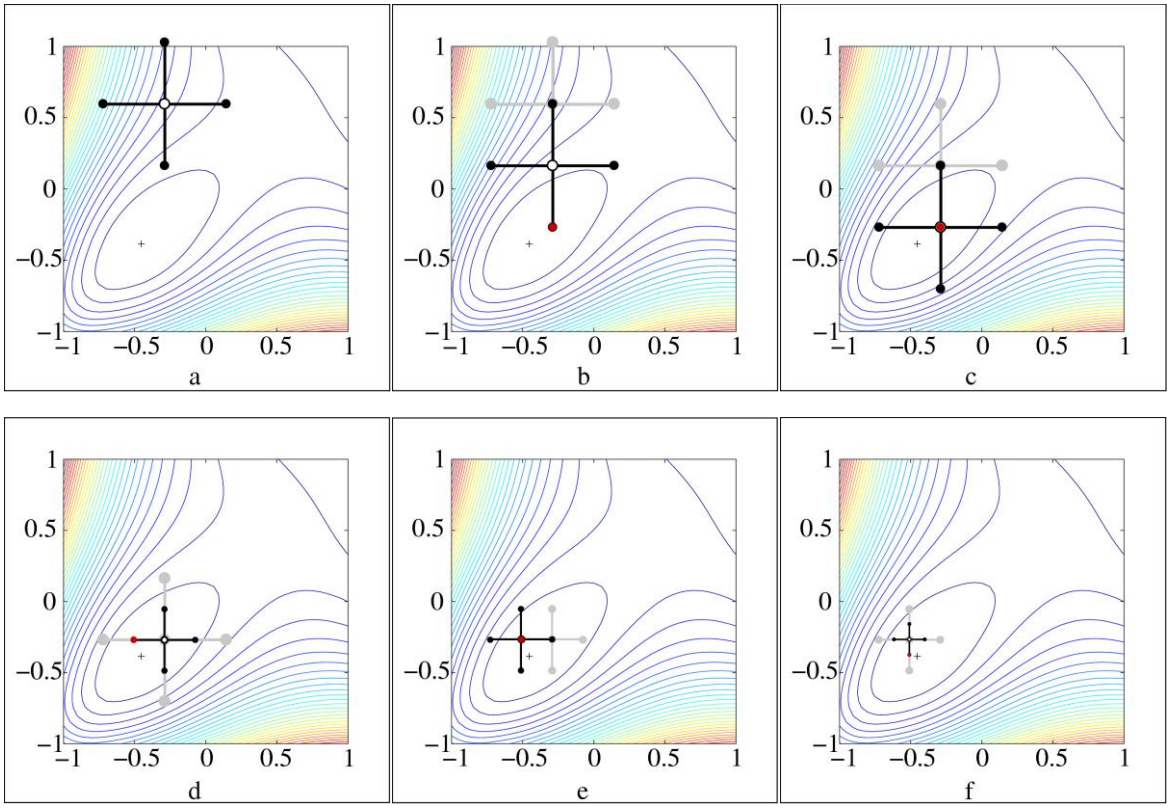
The function typically represents a desirable quantity (in which case you want to maximise), or an undesirable one, which you want to minimise. In machine learning you are maximising accuracy or performance but with regression you are minimising the error.

For a single argument function, the process is simple. All you have to do is measure the function at N points and then select the minimum. This can even be done for 2 or 3 dimensions, however as you go up in dimensions, then the number of measurements or evaluations increases exponentially as $E = M^n$ where n is the number of dimensions and M is the number of measurements.

In the case of wavefront optimisation where you want to find which combination of voltages will produce the optimal deformation of the mirror, you can have up to 60 or 70 voltage channels. If you were to measure 10 voltage values per actuator, the number of evaluations would be 10^{60}. Assuming the response time of the mirror was 10ms, the process would take 2.3×10^{40} times the current age of the universe. Also note that a resolution of 10 volts is way too low.

This brings us to our first algorithm. Note that all algorithms will be written in MATLAB and as a standard, we are trying to minimise the function whose value represents "cost".

Pattern search [1]



(Images from Wikipedia https://en.wikipedia.org/wiki/Pattern_search_(optimization))

The function begins at the origin, then creates a set of neighbouring points around it called *dr*. Beginning from the first dimension, creates a point at distance *vsize* (vertex size) orthogonal to the other dimensions. This is done for the other dimensions and after that, mirrors the points in the negative direction too. Pattern search then determines which neighbouring point is the best and compares it with the current position. If a neighbouring point is better, it moves there. Otherwise it shrinks.

```
function r = pattern_search(ndims,fhandle,epsilon)
    shrinkrate = 0.5;
    vsize = 0.5;
    r = [0;0];
    while vsize >= epsilon
        dr = vsize*[eye(ndims) -eye(ndims)] + r;
        cost = fhandle(dr);
        [mincost,idx] = min(cost);
        if fhandle(r) < mincost
            vsize = shrinkrate*vsize;
        else
            r = dr(:,idx);
        end
    end
end
```

Implementation

```
r = pattern_search(2,@f,1e-4);

function z = f(r)
    x = r(1,:);
    y = r(2,:);
    z = (x - 2.3334).^2 + (y + 1.2213).^2 - 1;
end
```

Defining a function for optimisation is fine and has some advantages, however doing this using object-oriented programming can be more useful. Defining a solver object allows us to encapsulate all its properties and is more flexible.

A class is basically a blueprint for an object. You define its properties in the “properties” section, and these are stored in memory as long as the object exists. It is the same as a struct in MATLAB. You access its elements the same way as such “object.property”. To create the object, you must call the constructor function which is the first function in the methods section and must have the same name as the class. The methods section encapsulates a set of functions that can be used on the object. Or other functions that do not necessarily have to do anything with the object (static methods).

The code above is implemented in a class as such

```
classdef pattern_search < handle
    properties
        numdims
        position
        poscost
        neighbours
        neighbourscost
        costfun
        vertexsize
        shrinkrate
    end
    methods
        function obj = pattern_search(ndims,fhandle)
            obj.numdims = ndims;
            obj.costfun = fhandle;
            obj.position = zeros(ndims,1);
            obj.neighbourscost = zeros(2*ndims,1);
            obj.vertexsize = 0.5;
            obj.shrinkrate = 0.5;
        end
        function step(obj)
            dr = obj.vertexsize*[eye(obj.numdims) -eye(obj.numdims)];
            obj.neighbours = dr + obj.position;
            obj.neighbourscost = obj.costfun(obj.neighbours);
            obj.poscost = obj.costfun(obj.position);
            [mincost,idx] = min(obj.neighbourscost);
            if obj.poscost < mincost
                obj.vertexsize = obj.shrinkrate*obj.vertexsize;
            else
                obj.position = obj.neighbours(:,idx);
            end
        end
    end
end
```

Implementation

```
solver = pattern_search(2,@f);

while solver.vertexsize > 1e-4
    step(solver)
end
disp(solver.position);

function z = f(r)
    x = r(1,:);
    y = r(2,:);
    z = (x - 2.3334).^2 + (y + 1.2213).^2 - 1;
end
```

Performance

This algorithm is relatively slow, as the number of evaluations per iteration is $E = 2n + 1$ where n : number of dimensions.

Nelder-Mead Simplex [2]

This method is using a simplex, which is the simplest shape you can have for a given number of dimensions. This way you are using the minimal number of points for your search algorithm which means you have fewer evaluations too. The method is a bit complicated, but the pseudocode can be seen below. The MATLAB code can be found at the page before the farfield simulation

Pseudocode

Create an initial simplex $s = \{x_1, \dots, x_{n+1}\}$

Begin optimisation:

1. Sort the simplex such that $f(x_1) \leq f(x_2) \leq \cdots \leq f(x_{n+1})$
   Check whether algorithm should terminate
2. Calculate the centroid of all points except x_{n+1}
3. Reflect worst point from centroid
   if $f(x_1) \leq f(x_{\text{reflected}}) \leq f(x_n)$:
   Replace worst point with the reflected point $x_{n+1} \coloneqq x_{\text{reflected}}$
   Go to step 1.
4. if $f(x_{\text{reflected}}) < f(x_1)$:
   Calculate expanded point
   if $f(x_{\text{expanded}}) < f(x_{\text{reflected}})$:
   Replace worst point with expanded point $x_{n+1} \coloneqq x_{\text{expanded}}$
   Go to step 1.
   else:
   Replace worst point with reflected point $x_{n+1} \coloneqq x_{\text{reflected}}$
   Go to step 1.
5. if $f(x_{\text{reflected}}) \geq f(x_n)$:
   Calculate contracted point
   if $f(x_{\text{contracted}}) < f(x_{n+1})$:
   Replace worst point with contracted point $x_{n+1} \coloneqq x_{\text{contracted}}$
   Go to step 1.
   else:
6. Shrink simplex
   Go to step 1.

The equations to calculate the reflected, expanded, contracted and the shrunk simplex can be seen below. Note that some adjustable parameters are introduced to control how the simplex acts for a given search space and should be adjusted if the default parameters are not good enough.

Reflection, expansion, contraction equations

$$x_r = x_o + \alpha(x_o - x_{n+1})$$
$$x_e = x_o + \gamma(x_r - x_o)$$
$$x_c = x_o + \rho(x_{n+1} - x_o)$$

Shrink

Replace all points except the best (x_1)

$$x_i = x_1 + \sigma(x_i - x_1)$$

Default values

$$\alpha = 1, \gamma = 2, \rho = 0.5, \sigma = 0.5$$

Performance

about $E = n + 2.5$

Gradient Descent with momentum [3]

The theory is quite simple. As mentioned in the deep learning section, with gradient descent you begin at some initial point in your parameter space. Then you perform a numerical calculation of the gradient. Then the point moves by some amount proportional to the magnitude of the gradient vector, in the direction opposite to the gradient. This ensures that the function is minimised. To prevent the point from converging to local minima, a momentum term is introduced that is essentially the resistance of the point's motion against the gradient vector. The higher the momentum the less it is affected by the gradient. The momentum value ranges from 0 to 1. When 0 the point moves in the direction of the gradient. When the value is 1, it completely ignores the gradient (which also means it will never minimise the function).

```
classdef gdm_solver < handle
    properties
        numdims
        costfun
        cost
        position
        momentum = 0.5
        gradient = inf
        gradstep = 0.1
        weightedgrad = 0
    end
    methods
        function obj = gdm_solver(dims,costfun)
            obj.numdims = dims;
            obj.costfun = costfun;
            obj.position = zeros(dims,1);
        end
        function step(obj)
            obj.cost = obj.costfun(obj.position);
            h = 1e-5;
            pos2 = h*eye(obj.numdims) + obj.position;
            cost2 = obj.costfun(pos2)';
            obj.gradient = (cost2-obj.cost)/h;
            obj.weightedgrad = obj.momentum*obj.weightedgrad +...
                (1-obj.momentum)*obj.gradient;
            obj.position = obj.position - obj.gradstep*obj.weightedgrad;
        end
    end
end
```

Implementation

Same as above.

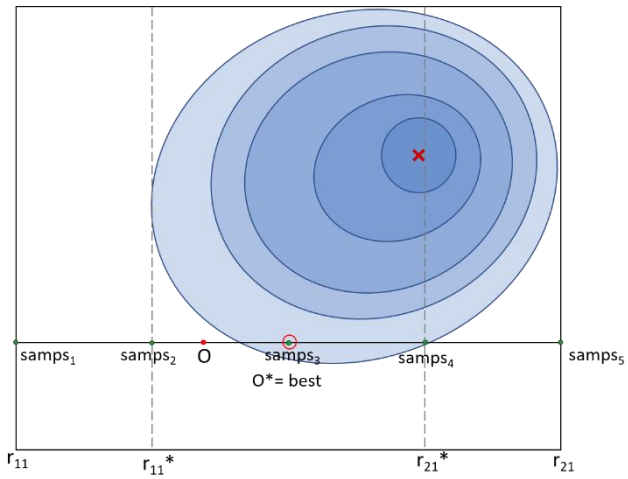
Performance

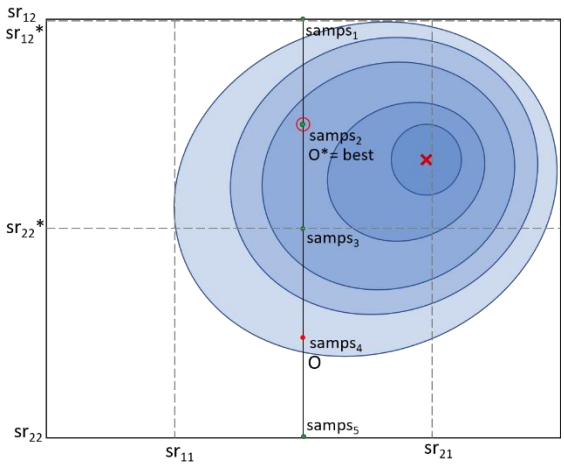
Number of evaluations per iteration $E = n + 1$ however due to the small step size it takes overall more iterations to converge. It is not very usable for noisy measurements of the cost function. Unless a higher gradient step is used for the gradient calculation, which also makes the gradient less accurate. The gradient step is the variable 'h' above.

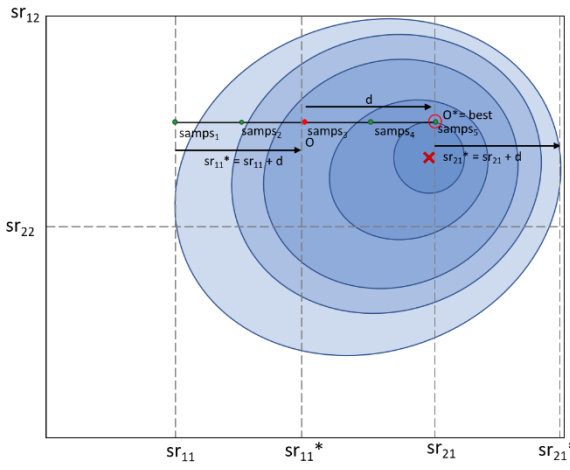
Coordinate Search

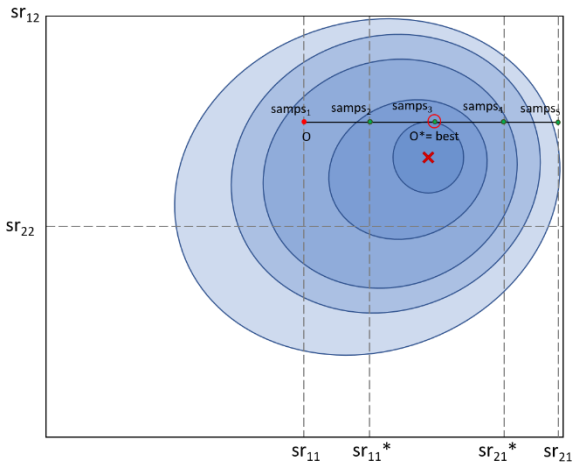
This algorithm was developed by the author. Pseudocode below. [Link to bigger image (GitHub)]

```
select origin: O = [x1,…,xN]'
initialise search range: sr = [r1,…,rN] where ri = [-1,1]'
while condition not met
    for i = 1 to N
        M equally spaced points within ri: xvals = linspace(sr1i,sr2i,M)
        mincost = inf
        for j = 1 to M
            set point equal to origin: p = O
            move along ith dimension to the jth coordinate: pj = xvalsj
            save point to samples matrix: sampsj: = p
            measure cost at current point: costj = f(p)
            if costj ≤ mincost
                mincost = costj
                save index of best so far: idx = j
            end
        end
        chose best p point and save index: best,j → min(cost)
        if j == N or j == 0
            shift search range towards best point: sr = sr + (best - O)
        else
            update search range: sri: = [sampsj-1,sampsj+1]'
        end
        set new origin: O = best
    end
end
```









ADAM [4]

The same as gradient descent with momentum however there is two weighted gradients to be considered, a first and second order one. Adam is a hybrid between Adagrad and RMS-prop. I will not be getting into the details of how it works, but it is the go-to optimiser for deep learning. Extremely fast under suitable action space. Not the best with highly noisy measurements of the cost function.

Define the same object as gradient descent but add the properties V and S and initialise as 0 in the properties section. Next define the property "epsilon" and keep track of the iterations in a property named "iteration". Replace the position update with code below.

```
obj.V = obj.beta1*obj.V + (1-obj.beta1)*obj.grad;
obj.S = obj.beta2*obj.S + (1-obj.beta2)*obj.grad.^2;
obj.Vcorr = obj.V/(1-obj.beta2^obj.iteration);
obj.Scorr = obj.S/(1-obj.beta2^obj.iteration);
obj.position = obj.position - obj.learningRate*(obj.Vcorr./sqrt(obj.Scorr + obj.epsilon))
```

Application in phase optimisation

Until now I explained optimisation using an arbitrary 2-dimensional function and cartesian coordinates. However, the coordinate system does not matter. All the algorithms care about, is how the cost changes when some parameters change.

When optimising the wavefront, we need to know what the function we are optimising is, and what our up this point arbitrary coordinates, represent. The cost function has to be some metric of the wavefront quality. If we had a direct measurement of the wavefront we could use the peak to valley value, or the RMS. We could even decompose it into its Zernike/Legendre basis, and then the use the magnitude of our coefficients vector. As for wavefront sensor-less optimisation, the far field quality is commonly used.

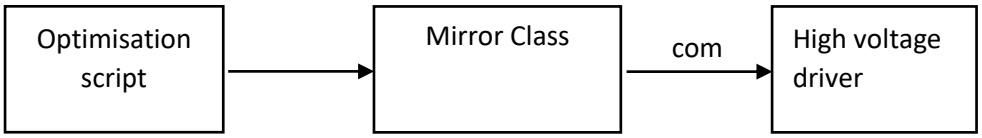
The deformable mirror has n number of actuators that are connected to n channels. So, the metric might be a function of voltages as $M = f(\boldsymbol{v})$. Additionally, we could measure the influence matrix of the DM and get a mapping between Zernike/Legendre modes into their corresponding voltages via a linear transformation. So maybe an easier function to optimise would be $M = g(\boldsymbol{a})$ where $\boldsymbol{a} = \boldsymbol{C}\boldsymbol{v}$ and $\boldsymbol{C}$ is the influence matrix. There are different advantages for choosing either coordinates. For example, the Zernike/Legendre coordinates are orthogonal to each other and $g(\boldsymbol{a})$ is overall less bumpy (fewer local minima) in my experience. Also, each coordinate has a clear and sometimes predictable change in the far field. The main problem with that, is that it requires the influence matrix to be computed beforehand. Since we are trying to completely remove the wavefront sensor this method makes less sense than using the voltages as our coordinates.

Defining cost function or Metric

Due to convention, the algorithms are trying to minimise the function. So, if the metric was wavefront quality, then we would like to maximise that. This can be easily done by flipping the metric function by using a negative sign because $\operatorname{argmax} f(x) = \operatorname{argmin} -f(x)$. Next, we formulate the function such that it takes a vector of the voltages and deforms the mirror accordingly. Then it measures the far-field and based on our quality metric, outputs a cost. We then output the negative of that value. Below is a MATLAB pseudocode of how that process might be.

```
function cost = metric_function(volts)
    set_mirror_channels(volts);
    wait_for_mirror;
    image = capture_farfield();
    cost = -get_metric(image);
end
```

To implement this, we require the MATLAB image acquisition toolbox and either some premade code we can use to interface with the mirror voltage driver or define our own code to communicate with it. Thankfully, Rebecca Harding worked on that task and sent me some code to communicate with the driver via com port. With it I created a mirror class which contained a method to connect to the driver box. Using a class in not necessary, but it adds a layer between a real mirror and our MATLAB script. If a different mirror is used, the optimisation script will not have to be changed, only the mirror's communication methods. Below is the Mirror class code



```
classdef mirror < handle
    properties
        serial                      % Serial connection
        port
        baud
        voltages = zeros(64,1);     % voltages log
        max_voltage = 50;           % default value
        channels = 64;              % number of channels
        isconnected = 0;
    end
    methods
        function obj = mirror(port,baud)
            % When mirror object is defined, a connection is established
            if ~isstring(obj.port)
                error('Port must be a string. eg. "COM4"')
            end
            obj.port = port;
            obj.baud = baud;
            initialiseDriver(obj)
        end
        function setChannel(obj,volt,chan)
            check_connection(obj)
            if abs(volt) > obj.max_voltage
                error('Too much voltage')
            elseif chan > obj.channels || chan < 1
                error('Incorrect Channel')
            end
            volt = volt*1e+3; % milivolts to volts
            msg = sprintf('%d %d set_output',volt,chan);
            writeline(obj.serial,msg)
            obj.voltages(chan) = volt;  % Logging the voltage.
        end
        function setChannels(obj,volts)
            check_connection(obj)
            if isscalar(volts)
                volts = volts*ones(64,1);
            end
            for i = 1:64
                setChannel(obj,volts(i),i)
            end
            obj.voltages = volts;
            pause(0.1) % Delay is used to give the actuators time to move.
        end
        function degaus(obj)
            for i = 1:64
                for v = linspace(-30,30,5)
                    setChannel(obj,v,i)
                end
            end
            setChannels(obj,0)
        end
        function disconnect(obj)
            check_connection(obj)
            obj.serial = [];
            obj.isconnected = 0;
            fprintf('Connection closed\n');
        end
```

```
        function initialiseDriver(obj)
            if obj.isconnected
                warning('Driver is already initialised. Skipping initialisation...')
                return
            end
            % Open serial port
            obj.serial = serialport(obj.port,obj.baud);
            % carriage return terminator
            configureTerminator(obj.serial,"CR");

            % initial command sent to drop out of the device's legacy mode
            writeline(obj.serial,"~~~");
            readString = readline(obj.serial);

            % send some carriage returns
            write(obj.serial,13,'char');
            readString = readline(obj.serial);
            write(obj.serial,13,'char');
            readString = readline(obj.serial);
            write(obj.serial,13,'char');
            readString = readline(obj.serial);

            % print to console the received response
            fprintf(readString);

            stringCheck = strfind(readString, 'ok');

            if stringCheck > 0
                fprintf("\r\nDevice now ready to receive commands\r\n");
                obj.isconnected = 1;
            else
                fprintf("\r\nCommunication Error\r\n");
                obj.serial = [];
                obj.isconnected = 0;
                fprintf('Connection closed\n');
            end
        end
        function check_connection(obj)
            if ~obj.isconnected
                error('Driver not connected')
            end
        end
    end
end
```

Far-field camera

Setting up the camera and taking images is much easier. The image acquisition toolbox provides us with all our necessary functions. All we need to do is define a camera object and call the “snapshot” method to save a picture. We can also have a live preview while the program is taking pictures, which is nice.

```
cam = gigecam('IP or SN here'); % establishes connection with camera
preview(cam);                   % shows a live preview of the camera
image = snapshot(cam);          % captures a simgle image
```

Extra Details

When using gradient descent, the coordinates must be withing the range of $[-1,1]$. It is also useful to normalise the coordinates for other functions too, and simply apply a scaling factor or a transformation between the solver coordinates and the actual voltages. Luckily both coordinate systems are centred around zero and are symmetric. That is, $v \in [-V_{\max}, V_{\max}]$ and $r \in [-1,1]$. Thus, we can simply scale the solver position by the maximum voltage that can be applied to the actuators.

Final Script

Considering all the above the final scrip is

```
clear

% Connecting to mirror
mirror = deformable_mirror('COM4',115200);
mirror.max_voltage = 100;
mirror.channels = 64;

% Connecting to camera
cam = gigecam('192.168.0.72');
cam.ExposureTimeAbs = 5000;
cam.Timeout = 2;
preview(cam)

fhandle = @(pos) cost_function(pos,cam,mirror);
solver = pattern_search(mirror.channels,fhandle);

% Main loop
for i = 1:100

    step(solver);

    % Plotting channels status
    figure(1);
    plot(mirror.voltages)
    title("Channels status")
    xlabel("Channel")
    ylabel("Voltage (V)")
    xlim([0 65])
    ax = gca;
    ax.XGrid = 1;
end

% Closing connections
clear cam mirror

% Cost function to be optimised
function cost = cost_function(pos,cam,mirror)
    volts = mirror.max_Voltage * pos;
    set_voltages(mirror,volts);
    image = single(snapshot(cam));
    maxI = max(image,'all');
    cost = -mean(image(image >= 0.6*maxI),'all');
end
```

The metric is defined to be the mean of the top 60% pixel values

Final notes

The camera settings might have to be adjusted for the metric to be accurate. For pattern search it does not matter much. But for gradient descent or more sensitive gradient methods, then the bit mode can be changed from 8bit to 12bit or even 32bit. The more pixel values however, the more noise is measurable. This is highly dependent on the camera of choice.

The exposure time should be low enough so that the image does not saturate when fully optimised. If it does, then the algorithm should be stopped, exposure time lowered, and then algorithm restarted. When saturated, the optimiser will see no improvement in metric, and will effectively perform a random walk near the optimum, or simply converge to the current point.

The above notes should be considered for any metric, but intensity-based metrics are especially susceptible.

Additionally, we might consider adding postprocessing to the far-field image, such as median filters to reduce the noise, which is especially visible in low exposure times. Ideally the exposure time should be high with a physical intensity filter placed in front of the camera lens to reduce the image brightness. However, exposure time should not be too high, otherwise the algorithm will run considerably slower. The bottleneck should be the actuator response time.

```
classdef simplex_optimiser < handle
    properties
        dimensions
        simplex
        cost_function
        simplex_cost
        init_size = 0.25;

        r_coeff = 1
        e_coeff = 2
        c_coeff = .5
        s_coeff = .5
    end
    methods
        function self = simplex_optimiser(dims,costfun)
            self.dimensions = dims;
            self.cost_function = costfun;
            self.initialise_simplex();
        end
        function initialise_simplex(self)
            N = self.dimensions;
            init_position = zeros(N,1);
            nms = self.init_size*eye(N) + init_position;
            nms = cat(2,nms,init_position);
            self.simplex = nms;
        end
        function sort_simplex(self)
            % Sorting in ascending order: lower cost first (best < worst)
            self.simplex_cost = self.evaluate(self.simplex);
            [self.simplex_cost,idx] = sort(self.simplex_cost);
            self.simplex = self.simplex(:,idx);
        end
        function cost = evaluate(self,points)
            % Evaluates a single point or set of points
            % Must be N by M matrix where M is number of points
            num_points = size(points,2);
            cost = zeros(1,num_points);
            for idx = 1:num_points
                cost(idx) = self.cost_function(points(:,idx));
            end
        end
        function centroid = get_centroid(self)
            centroid = mean(self.simplex(:,1:end-1),2);
        end
        function reflected = get_reflected(self,centroid)
            worst = self.simplex(:,end);
            reflected = centroid + self.r_coeff*(centroid-worst);
        end
        function expanded = get_expanded(self,centroid,reflected)
            expanded = centroid + self.e_coeff*(reflected-centroid);
        end
        function contracted = get_contracted(self,centroid)
            worst = self.simplex(:,end);
            contracted = centroid + self.c_coeff*(worst - centroid);
        end
        function shrink_simplex(self)
            best = self.simplex(:,1);
            for i = 2:self.dimensions
                current = self.simplex(:,i);
                current = best + self.s_coeff*(current - best);
                self.simplex(:,i) = current;
            end
        end
```

```
        function step(self,varargin)
            % Usage, obj.step(num_steps). If left empty, steps = 1
            steps = 1;
            if ~isempty(varargin)
                steps = varargin{1};
            end
            for i = 1:steps
                self.sort_simplex();
                sec_to_worst_cost = self.simplex_cost(end-1);
                best_cost = self.simplex_cost(1);
                centroid = self.get_centroid();
                % centroid_cost = self.evaluate(centroid);
                reflected = self.get_reflected(centroid);
                reflected_cost = self.evaluate(reflected);
                if reflected_cost < sec_to_worst_cost && reflected_cost >= best_cost
                    self.simplex(:,end) = reflected;
                    self.simplex_cost(end) = reflected_cost;
                elseif reflected_cost < best_cost
                    expanded = self.get_expanded(centroid,reflected);
                    expanded_cost = self.evaluate(expanded);
                    if expanded_cost < reflected_cost
                        self.simplex(:,end) = expanded;
                        self.simplex(end) = expanded_cost;
                    else
                        self.simplex(:,end) = reflected;
                        self.simplex_cost(end) = reflected_cost;
                    end
                elseif reflected_cost >= sec_to_worst_cost
                    contracted = self.get_contracted(centroid);
                    contracted_cost = self.evaluate(contracted);
                    worst_cost = self.simplex_cost(end);
                    if contracted_cost < worst_cost
                        self.simplex(:,end) = contracted;
                    end
                else
                    self.shrink_simplex();
                end
            end
            self.sort_simplex();
        end
    end
end
```

Note: Since this was the first version, I added a method for all the simplex manipulations, but this can be done inside the step() method. This could speed up the process a tiny bit and have fewer lines of code.

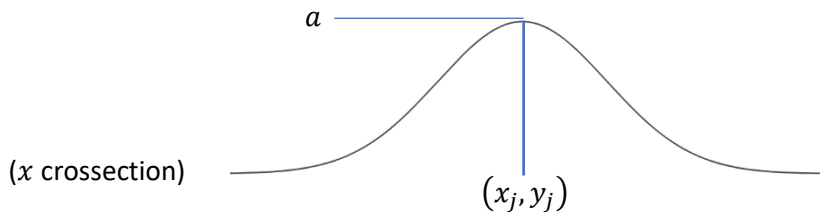
DM modelling and far-field simulation

This section is regarding my initial testing of WSAO algorithms before I had access to an AO setup. To test the algorithms, I created a very basic model of a deformable mirror and a basic far-field diffraction simulation.

Deformable Mirror

To model the mirror, I assumed each actuator to have a bell-shaped influence on the mirror surface as shown below

$$z(x,y) = a\exp\left(\frac{-(x-x_j)^2-(y-y_j)^2}{c}\right)$$



Next, I assumed a very simple mirror with N actuators where every actuator is placed in an $n \times n$ grid ($N = n^2$). Then for each actuator I generated a bell function in the location of the actuator.

To simulate the movement of the actuators, I simply changed the amplitude a of the exponential which I assumed to be proportional to the voltage. For a mirror that can correct a $\pm 30\mu\text{m}$ aberration with a maximum voltage of $\pm 100\text{V}$, the proportionality constant V_{const} becomes $0.3\mu\text{mV}^{-1}$.

$$a = V_{\text{const}} \cdot v$$

The resolution of my wavefront and thus mirror surface is 50×50 in rows×colums, or pixels when converted to an image.

For the spread of the function, I fitted Legendre polynomials until the error reduced considerably. There is an optimal value around 0.3 that gives on average the best Legendre approximation for each influence.

To do that, I generated some random voltages and then calculated the Legendre coefficients of the mirror surface by using the Fourier-Legendre transform. Then I defined a metric equal to the RMS error between the Legendre fit and the mirror shape as a function of the influence spread c. The value of c converges every time close to 0.3. So, to get the best Legendre representation of my influence functions, I selected that spread value.

The final equation of my influence functions is defined as

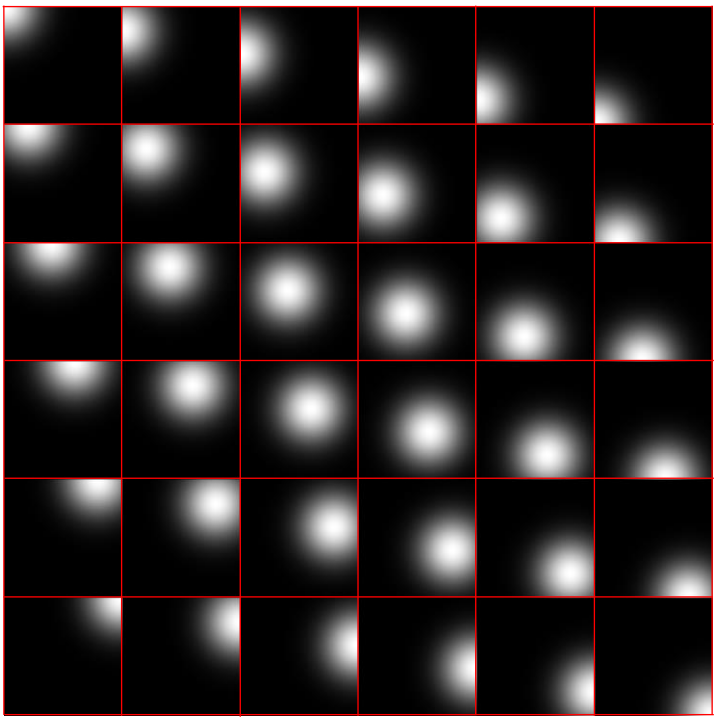
$$S_j(x,y) = 0.3\exp\left(\frac{-\left(x-x_j\right)^2-\left(y-y_j\right)^2}{0.3}\right)$$

Where $x_j,\ y_j$ are the coordinates of the jth actuator

Deformable mirror model in MATLAB

The class contains two methods. One is used to generate the influence of every actuator and are stored as a surface of a resolution given by the user $S_i = \text{resol} \times \text{resol}$. The second simply calculates the surface of the mirror when voltages are applied to it. The mirror surface is defined within the unit square $x, y \in [-1,1]$ and so are the influences. Then the x and y coordinates for every entry in the surface matrix are defined using *linspace(-1,1,resol)*. To define the position of the actuators, *linspace(-1,1,N)* is used, which effectively divides the region in N equally spaced points.

The resulting influences can be seen below. Note that the red square is not the wavefront region. It is just a display of all the mirror influence functions.



Usage

Define a mirror object given the number of actuators, the resolution of the surface and the spread of the influences. Then use the method to set the channel voltages and get the resulting surface by accessing the property 'shape' of the mirror object.

```
% 6x6 grid actuators, resolution of 100x100 and spread = 0.1
mirror = mirror_model(36,100,0.1);
mirror.set_channels(volts);
surface = mirror.shape;
```

The code for the mirror model can be seen below

```
classdef mirror_model < handle
    properties
        shape               % shape of mirror as resol x resol matrix
        channels            % number of channels
        voltages            % voltages vector
        influences          % influence of every actuator
        resolution          % width and height of every surface in rows & cols
        spread              % how much the influence curve spreads
        volt_const = 0.1    % default value
    end
    methods
        function obj = mirror_model(chan, resol, spread)
            % Checking user inputs
            if sqrt(chan) ~= round(sqrt(chan))
                error("Number of actuators must be the square of a natural number")
            elseif spread <= 0 || resol <= 0
                error("Invalid resolution or spread value. Must be positive")
            end
            obj.channels = chan;
            obj.spread = spread;
            obj.resolution = resol;
            generate_influences(obj);
            set_channels(obj,1);
        end
        function generate_influences(obj)
            % Actuator placement is NxN grid such that NxN = # acts
            N = sqrt(obj.channels);
            % Create N equally spaced points in region [-1,1]
            act_positions = linspace(-1,1,N);
            % Create a meshgrid of 'resol' number of points in region [-1,1]
            [X, Y] = meshgrid(linspace(-1,1,obj.resolution));
            obj.influences = zeros(obj.resolution,obj.resolution,obj.channels);
            k = 1;
            for i_pos = act_positions
                for j_pos = act_positions
                    % Creating gaussian curve at (i_pos,j_pos)
                    obj.influences(:,:,k) = obj.gaussian(X,Y,i_pos,j_pos,1,obj.spread);
                    k = k + 1;
                end
            end
        end
        function set_channels(obj,volts)
            % Sets channels to given voltages
            if isscalar(volts)
                % For ease of use, single value means all channels
                volts = ones(obj.channels,1)*volts;
            elseif isvector(volts)
                % if vector is row, transpose to column. Needed for permute
                % later
                if isrow(volts)
                    volts = volts';
                end
            else
                error('Argument must be a vector')
            end
            % Transforming vec from 'channels'x1 to 1x1x'channels'. Shape =
            % linear combination of all influences*volt then scaled by
            % Vconst. Influences are of size 'resol'x'resol'x'channels'
            volts = permute(volts,[3,2,1]);
            obj.shape = obj.volt_const*sum(volts.*obj.influences,3);
            obj.voltages = volts;
        end
    end
    methods (Static)
        function z = gaussian(x,y,i,j,a,c)
            % Used to generate the influences
            z = a * exp((-(x-i).^2-(y-j).^2)/c);
        end
    end
end
```

Far-Field simulation [6]

The image in the focal plane of a diffracted beam

$$U(r) = \iint_{\text{Aperture}} A(r)P(r)\exp\big(i\varphi(r)\big)\,dr$$

Where P is the aperture shape function, A is the amplitude of the wave and φ is the phase. This can be done in MATLAB using the furrier2 function. The aperture shape can be ignored when using a square beam, however it must be used for any other shape. In the case of Zernike surfaces, P is 1 within the unit circle and 0 outside.

To use the far-field simulation, all you must do is call the generate_farfield method from the FarField class. There is no constructor, so if you use `FarField.generate_farfield(nf,wf)` then the default noise, aperture size and phase gain values will be used. To change those, then you must define an object and change its properties. Like so:

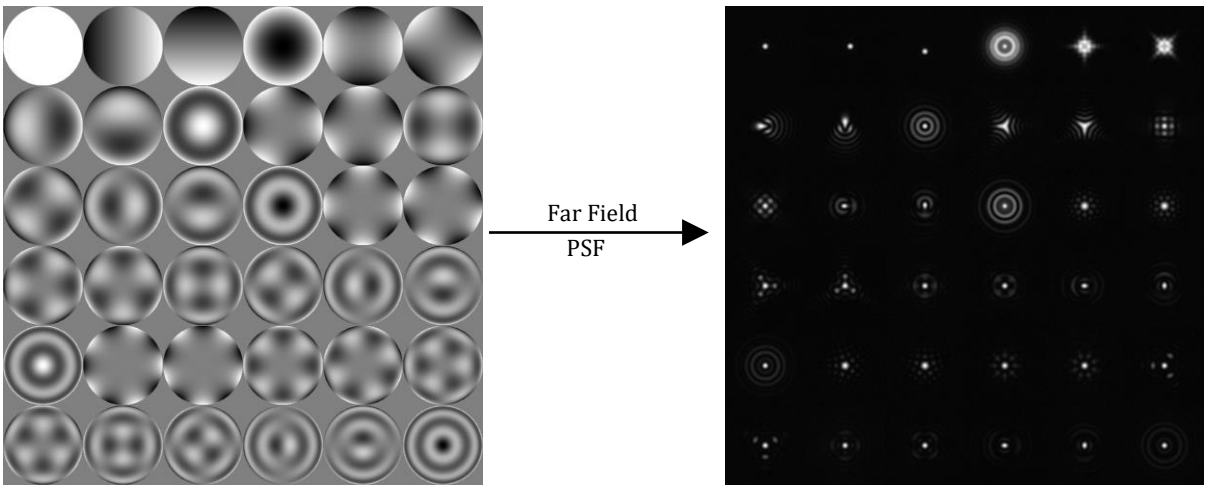
```
FF = FarField;
FF.settings(phase_gain,[],noise_dev,noise_mean)
img = FF.generate_farfield(nf,wf); % Note: Simulation does not apply aperture.
```

As the above comment suggests, when using circular apertures, make sure that in your near field and wavefront are 0 outside the unit circle. This means that you cannot use a constant for the near field because the function assumes rectangular aperture. If your Zernike PSFs are not "round" or wrong, it probably means that either the wavefront or near field are not circular

Instead of 1: `FF.generate_farfield(1,wf)`
use `FF.generate_farfield(nf,wf)` where nf(r>1) = 0

The final far-field simulation can be seen in the next page.

Simulating Zernike polynomials (Wyant indexing) farfields (point spread functions)



As you can see, even if the simulation is a simple Fourier transform of the pupil function, the results are accurate. Note that the PSFs have been normalised

```
classdef FarField < handle
    properties
        image
        wavefront
        near_field
        phase_gain    = 1
        aperture_size = 0.2   % final size = initial size / aperture_size
        noise_dev     = 0.05  % Range of noise as % of max value
        noise_mean    = 0.05  % Background value as % of max value
    end
    methods
        function generate_farfield(obj,nf,wf)
            rows = size(wf,1);
            cols  = size(wf,2);
            if isscalar(nf)
                nf = ones(rows,cols)*nf;
            end
            % Calculating padding amount from apert_size and then padding
            sz2 = [rows,cols] / obj.aperture_size;
            padwidth = floor(sz2-[rows,cols])/2;
            padded_nf = padarray(nf,padwidth,0,'both');
            padded_wf = padarray(wf,padwidth,0,'both');
            % Calculating farfield image
            pupil = padded_nf.*exp(1i.*padded_wf.*obj.phase_gain);
            obj.image = fftshift(fft2(pupil,sz2(1),sz2(2)));
            obj.image = obj.image.*conj(obj.image);
            %cropping to remove extra space created by padding
            obj.image = obj.image(padwidth(1)+1:end-padwidth(1),...
                padwidth(2)+1:end-padwidth(2));
            % Adding gaussian noise to image, simulating a real camera
            deviation = obj.noise_dev*max(obj.image,[],'all')/8;
            mean = obj.noise_mean*max(obj.image,[],'all');
            obj.image = obj.image + deviation*randn(rows,cols) + mean;

            obj.wavefront = wf;
            obj.near_field = nf;
        end
        function settings(obj,varargin)
            % Set simmulation settings, eg. obj.settings(0.1,[],0.2,0.3)
            % Read object properties
            prop_array = [
                obj.phase_gain;
                obj.aperture_size;
                obj.noise_dev;
                obj.noise_mean
                ];
            % Check for non-empty elements. Non-empty: idx = 1. Else 0
            idx = ~cellfun(@isempty,varargin);
            for i = 1:nargin-1
                % If not empty, overwrite
                if idx(i)
                    prop_array(i) = varargin{i};
                end
            end
            % Change properties
            obj.phase_gain      = prop_array(1);
            obj.aperture_size   = prop_array(2);
            obj.noise_dev       = prop_array(3);
            obj.noise_mean      = prop_array(4);
        end
    end
end
```

Legendre Polynomials [5]

The Legendre polynomials can be handy when generating random low frequency surfaces. They are orthogonal to the unit square, so they are convenient for fitting a square beam wavefront. Any surface in the region $x, y \in [-1,1]$ can be expressed as a linear combination of Legendre polynomials, known as Legendre series

$$f(x) \approx \sum_{i=0}^{N-1} \alpha_i L_i(r)$$

$$\alpha_n = \frac{2n+1}{2} \int_{-1}^{1} W(r) L_n(r)\, dx$$

In two dimensions, the Legendre polynomials are the product of the 1D polynomials

$$L_{ij}(x,y) = L_i(x) L_j(y)$$

The above equation makes defining polynomials manually, viable. It does not require an iterative method, nor their formal definition which requires heavy computation. By defining n polynomials we can get n^2 2D polynomials. I made a Legendre polynomials "library" which contains the first 10 1D polynomials which means 100 2D polynomials are available with little computation. The polynomials themselves are defined by a function handle (or in python) a lambda function. For example, the third order polynomial is $f(x) = \frac{1}{2}(5x^3 - 3x)$. This is easily represented in MATLAB as

```
P3 = @(x) 1/2*(5*x.^3 - 3*x)
```

Usage: `y = P3(x)`

All 10 are saved in a cell array. Then using 2 nested loops, another cell array is defined which contains the 100 2D polynomials as

```
Lnm = @(x,y) Pn(x).*Pm(y).
```

Usage: `y = Lnm(x,y)`

where x is a row vector and y is a column vector, a scalar, or they can be a mesh-grid.

The library also contains a method to generate a surface of given a coefficient vector and a resolution. Another method is used to get the coefficient values or "moments", given a 2D surface matrix. The higher the resolution, the more accurate the prediction. For efficiency, there is a final method which generates the 2D polynomials up to a given order and a given resolution. The polynomials can be used with all previous methods to speed up the process, as they will only be generated once. This is highly recommended.

Indexing Note: The single indexing of the orders is as such

k	1	2	…	N	N+1	N+2	…	2N	…	N^2
n	0	1	…	N	0	1	…	N	…	N
m	0	0	…	0	1	1	…	2	…	N

Below is the code

```
classdef LegendreLib
    % Library contains 1D and 2D Legendre functions and methods to
    % implement surface decomposition and reconstruction. Call the
    % constructor with a scalar input to define maximum order.
    % The methods are static and will create their
    % own library object to access the Legendre functions. For best
    % performance, generate all the polynomials beforehand and use them for
    % all other computations.
    %
    % Licence: Use at your will.
    %
    % Example: Decompose surface and find coefficient error
    %   L = LegendreLib;
    %   coeffs = randn(36,1);
    %   surface = L.surface(coeffs,100);
    %   predcoeffs = L.moments(surface,36);
    %   rms = sqrt(sum((coeffs-predcoeffs).^2,'all')/36);
    %   fprintf('RMS Error: %.4f\n',rms);
    %   plot(coeffs);
    %   hold on
    %   plot(predcoeffs)
    %   hold off
    %
    % Author: Petros Oratiou 27/03/2020
    properties
        L = cell(36,1);
        P = cell(6,1);
        maxordidx = 10
        orders
    end
    methods
        function obj = LegendreLib(varargin)
            % First 6 Legendre functions. To expand library, define other
            % functions below and change maxordidx to the new number of
            % functions. The total number of 2D polys will be maxordidx^2.
            if nargin == 1
                maxord = sqrt(varargin{1});
                if maxord ~= round(maxord)
                    error('Number of orders must be the square of a natural number')
                end
                obj.maxordidx = maxord;
            end
            obj.P{1} = @(x) 0.*x + 1;
            obj.P{2} = @(x) x;
            obj.P{3} = @(x) 1/2*(3*x.^2 - 1);
            obj.P{4} = @(x) 1/2*(5*x.^3 - 3*x);
            obj.P{5} = @(x) 1/8*(35*x.^4 - 30*x.^2 + 3);
            obj.P{6} = @(x) 1/8*(63*x.^5 - 70*x.^3 + 15*x);
            obj.P{7} = @(x) 1/16*(231*x.^6 - 315*x.^4 + 105*x.^2 - 5);
            obj.P{8} = @(x) 1/16*(429*x.^7 - 693*x.^5 + 315*x.^3 - 35*x);
            obj.P{9} = @(x) 1/128*(6435*x.^8 - 12012*x.^6 + 6930*x.^4 - 1260*x.^2 + 35);
            obj.P{10} = @(x) 1/128*(12155*x.^9 - 25740*x.^7 + 18018*x.^5 - 4620*x.^3 + 315*x);
            % Generating 2D functions. L_nm(x,y) = L_n(x)×L_m(y) and
            % storing corresponding n,m pairs
            obj.orders = zeros(obj.maxordidx^2,2);
            k = 1;
            for m = 1:obj.maxordidx
                for n = 1:obj.maxordidx
                    obj.L{k} = @(x,y) obj.P{n}(x).*obj.P{m}(y);
                    obj.orders(k,:) = [n,m] - 1;
                    k = k + 1;
                end
            end
        end
    end
```

```
    methods (Static)
        function polys = polys(numords,sz)
            % Generates Legendre polynomials up to n,m = numords-1
            obj = LegendreLib(numords);
            if isscalar(sz)
                sz = [sz,sz];
            end
            rows = sz(1);
            cols = sz(2);
            x = (linspace(-1,1,cols));
            y = (linspace(-1,1,rows));
            [X,Y] = meshgrid(x,y);
            polys = (zeros(rows,cols,numords));
            for i = 1:numords
                polys(:,:,i) = obj.L{i}(X,Y);
            end
        end
        function surface = surface(coeffs,arg2)
            % Generates surface given the coefficients vector. Second
            % argument can either be resolution or the polynomials.
            if isvector(arg2)
                numords = length(coeffs);
                obj = LegendreLib(numords);
                polys = obj.polys(numords,arg2);
            else
                polys = arg2;
            end
            if isrow(coeffs)
                coeffs = coeffs';
            end
            coeffs = permute(coeffs,[3,2,1]);
            surface = sum(coeffs.*polys,3);
        end
        function coeffs = moments(surface,arg2)
            % Decomposes surface into Legendre polynomials. Returns
            % Legendre coefficients vector. Second argument can either be
            % number of orders or the polynomials.
            sz = size(surface);
            if isscalar(arg2)
                maxorder = arg2;
                obj = LegendreLib(maxorder);
                polys = obj.polys(maxorder,sz);
            else
                polys = arg2;
                maxorder = size(polys,3);
                obj = LegendreLib(maxorder);
            end
            dx = 2/(sz(1)-1);
            dy = 2/(sz(2)-1);
            n = obj.orders(1:maxorder,1);
            m = obj.orders(1:maxorder,2);
            C = (2*m+1).*(2*n+1)/4;
            coeffs = C.*permute(sum(surface.*polys,[1,2]),[3,2,1])*dx*dy;
        end
    end
end
```

Note: The static methods create an instance of this class in order to generate the order pairs. I could have defined them manually, but I prefer it this way, although it would seem like a blasphemy amongst programmers to do what I am doing. Good thing I am a physics student not a programmer.

UPDATE: Do not use this class. Use leg_polys from the repository and coeff_surface.

WSAO Simulation

Simulating a simple adaptive optics system is now possible. With the mirror model, farfield simulation, the optimisation algorithm, and the Legendre polynomials we should be able to do a demonstration of wavefront sensor-less adaptive optics.

Below is the pseudocode

```
Define mirror_model object
Define cost function
Define solver object and parse a handle to the cost function

Generate input wavefront from random coefficients using Legendre library

while condition not satisfied
    step solver

cost function (solver position):
    map solver position from [-1,1] to voltage values [-100,100]
    set mirror channels to voltage values
    add the mirror influence to the input wavefront
    simulate farfield image from net wavefront
    get a metric value for the image so that smaller value = better
    return metric
```

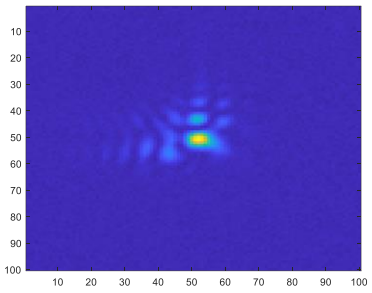
MATLAB code in the next page

```
L = LegendreLib(36);
polys = L.polys(36,100);

mirror = mirror_model(36,100,0.1);
mirror.set_channels(0);

FF = FarField;
FF.settings(0.5,[],[],[]);

coeffs = 5*randn(36,1);
inwf = L.surface(coeffs,polys);              % input wavefront
inff = FF.generate_farfield(1,inwf)*1e-7;    % initial far-field

flatW = L.surface(ones(36,1),polys);
idealff = FF.generate_farfield(1,flatW);     % used for strehl ratio

fhandle = @(r) costfun(r,mirror,FF,inwf,idealff);
solver = coordinate_search(36,fhandle,5);

figure(1)
imagesc(inff)
colorbar

figure(2)
surf(inwf)
shading interp
colorbar

iters = 12;
for i = 1:iters
    solver.step();
    frpintf('Iteration: %i\n',i);
end

volts = solver.current_position*100;
mirror.set_channels(volts);
img = FF.generate_farfield(1,mirror.shape+inwf)*1e-7;

figure(3)
imagesc(img)
colorbar

figure(4)
surf(mirror.shape+inwf)
shading interp
colorbar

function cost = costfun(pos,dm,ff,wf,idealff)
    volts = pos*100;
    dm.set_channels(volts);
    img = ff.generate_farfield(1,wf+dm.shape);
    cost = -strehl(idealff,img);
    % Other Metrics below
    % maxI = max(img,[],'all');
    % cost = -mean(img(img >= 0.6*maxI),'all'); % Goes with line below
    % cost = -mean(full_width_half_max(img));
End
```

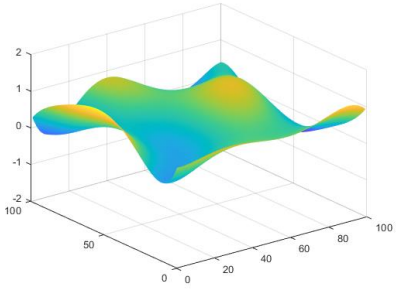
Below are some good results

Example 1

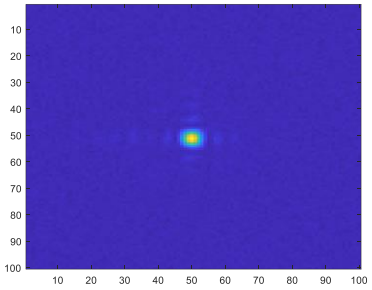
Initial far-field



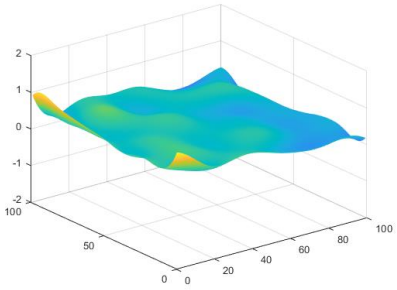
Input wavefront



Final far-field

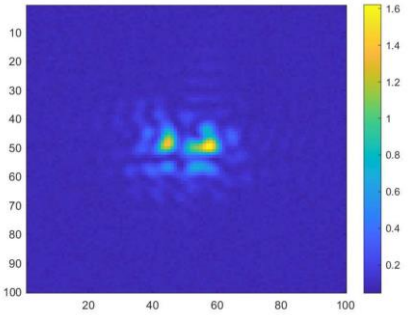


Output wavefront

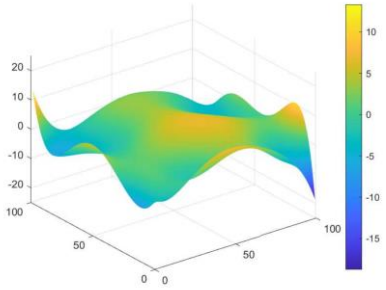


Example 2

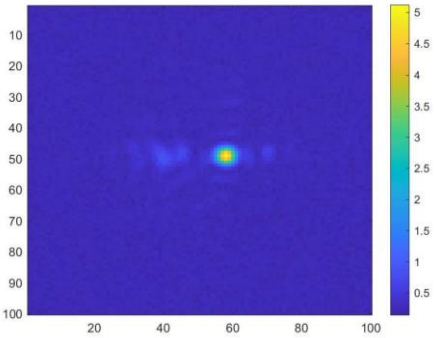
Initial far-field



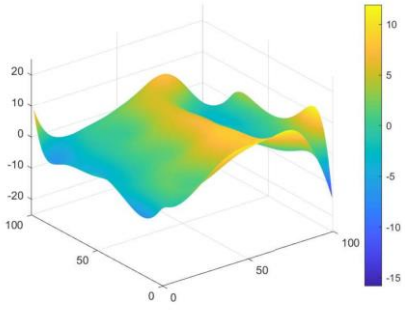
Input wavefront



Final far-field



Output wavefront



Updates:

This section is about all updates after the report was written.

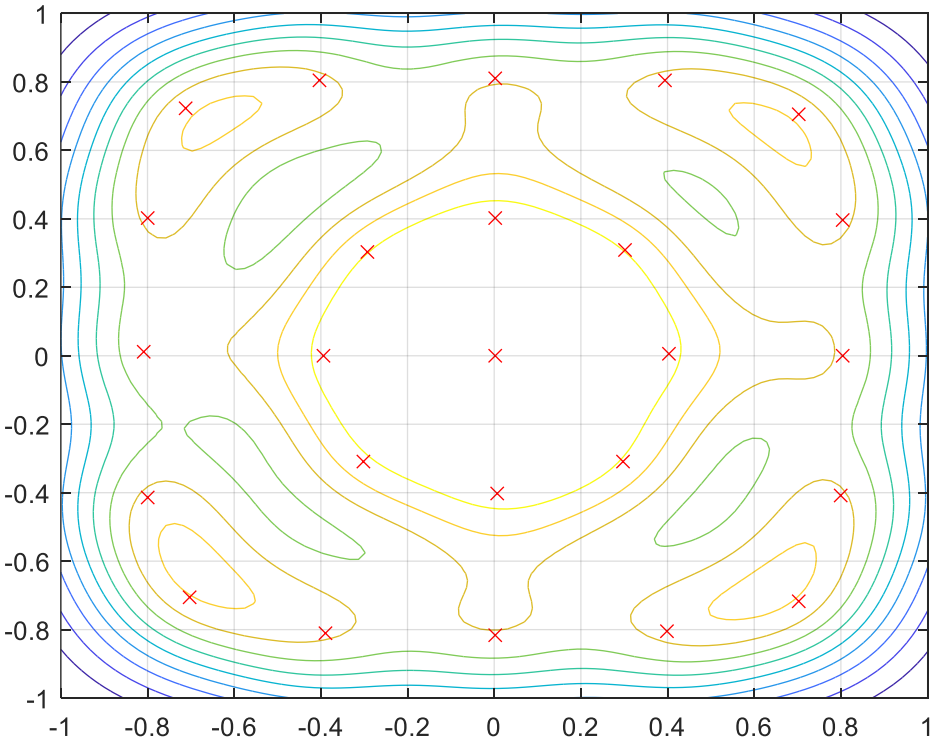
Update 1.
The wavefront is cropped centrally to remove the pulling effect that is caused by the Legendre polynomials. This effectively makes the wavefront smaller, In the real world the beam has to be smaller than the mirror so that the mirror is able to correct the edges more effectively. Cropping the pulling effect not only makes the wavefront more realistic, but also makes it more possible for the mirror to correct it. I increased the resolution in x and y by 20 pixels and then cropped 10pixels in every side, effectively having the same resolution as before. The mirror resolution was also increased by 20 pixels in each dimension.

Update 2.

I found out that matlab has its own legendre polynomials function which is much faster than my "LegendreLib" polynomials generator method. For that reason I wrapped the MATLAB function such that it returns the polynomials in the format LegendreLib.polys used to, and then created a function called "coeff_surface" which will generate a surface given the polynomials array and a coefficient vector.

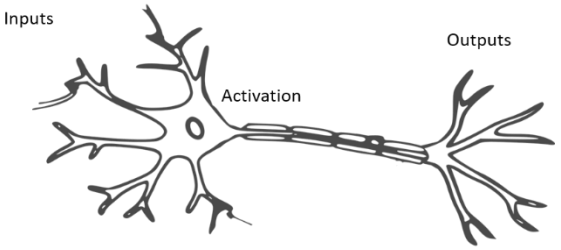
Update 3.

This is a huge update. Generating custom mirrors is now possible (other than the N by N grid actuator distribution). I have tested it with a new circular mirror whose actuator diagram is shown below. Unfortunately, it is not a very good mirror.

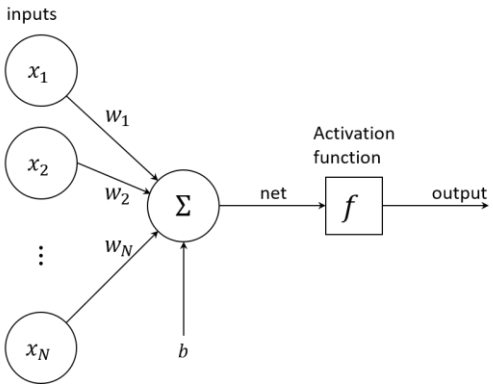


Deep Learning with MATLAB

Deep learning is a subfield of machine learning. Machine learning deals with neural networks, mathematical models that "learn" in a similar way to biological brains. Deep learning takes it a step further by making neural networks deeper and more complex. The field of machine learning is quite deep for this report; however a short introduction will follow.



This is an example of a human brain neuron. The cell receives electrical signals from its neighbours as an input. The amount of the signal that is received from a single neighbour is proportional to the "strength" of the connection between them. If the total contribution from the neighbouring cells reaches a threshold, then the neuron activates and propagates the signal to other cells connected to its outputs. In a real brain, signal propagation never stops and there are billions and billions of connections, let alone number of pathways. However even a simple brain can do fairly complicated tasks. This is what we attempt to do by modelling this process in a similar manner.



This is a visualisation of the neuron model. It is a primitive neural network with only one neuron, also called a perceptron. Confusingly enough, the term perceptron is also used for a neural network with multiple perceptrons. The net input is the weighted sum of the individual inputs to the neuron. Then the activation function decides whether to return a one or a zero based on the net input. This is shown mathematically like so.

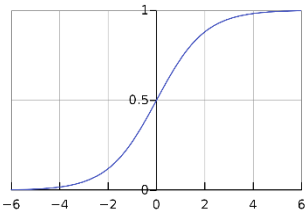
$$\text{net} = \sum_{i=1}^{N} w_i x_i + b \qquad \text{output} = f(\text{net})$$

We also introduce a bias term. In some cases the neuron will never train if the initial weight values are low, causing it to never activate which is why we need the bias.

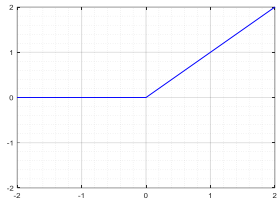
The activation function however does not have to take a binary state. A sigmoid function can be used, that tends to 1 when the net input is very high, or a 0 when it is highly negative. When the net input is 0 it will output a 0.5, which can represent uncertainty. The concept that a neuron can take a real value is a powerful one, which in theory makes artificial networks more capable than biological ones.

This simple neuron in essence is a function approximation. It takes an input and returns an output. This also extends to deep neural networks that are made of thousands of neurons stacked in lots of layers. They are all trying to approximate an unknown function from just a "few" data samples. If the function we are trying to approximate is non-linear, then we cannot hope to approximate it with linear equations. We need to introduce nonlinearities. This is the reason we need a nonlinear activation function such as the sigmoid or "tanh". Below are a few of the most commonly used functions.

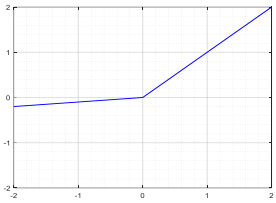
tanh



relu



Leaky relu



Note that from those 3 only $\tanh$ is nonlinear.

Training the perceptron: Theory

For the network to be able to make accurate predictions, the optimal weight and bias values must be determined. This is an optimisation problem that can be solved by many algorithms. The most commonly used algorithms for training are based on gradient descent. They are used to minimise "cost function" which is a metric of the network accuracy. A common choice for the cost function is the mean squared error which is defined as

$$E = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Where y_i is the predicted value of the i^{th} sample and $\hat{y}_i$ is the corresponding observed or target value.

We can see that the cost E is some unknown unknown function of y, which itself a function of the weights and the bias. We are trying to find the combination of weights and the bias that will give the minimum possible error. The problem is phrased mathematically as

$$\underset{w,b}{\operatorname{argmin}} E(w,b)$$

To actually train the network, we will employ the most basic of all the training algorithms. A simple gradient descent. To visualise the process only a single weight and the bias will be considered. The cost will be a two-dimensional surface that can have any shape.

We will start at some random position in the weight-bias plane and calculate the gradient of the cost on that spot. The gradient vector always points towards highest increase of the function, so if we desire to get to the minimum, we simply have to move opposite to the gradient vector. The amount that we move by, will be proportional to the gradient. So we define a constant by which the gradient is multiplied.

$$\Delta \boldsymbol{r} = -\eta \nabla E, \qquad \nabla E = \left(\frac{\partial E}{\partial w}\widehat{\boldsymbol{w}} + \frac{\partial E}{\partial b}\widehat{\boldsymbol{b}}\right)$$

Thus, the new weight and bias values are updated as such

$$w \coloneqq w - \eta\frac{\partial E}{\partial w}, \qquad b \coloneqq b - \eta\frac{\partial E}{\partial b}$$

To actually calculate the gradient you can simply apply a small change to a parameter and measure the change of the cost, or go through the actual calculation that depends of the response. This will be done later.

Gradient descent does not guarantee a convergence to a global minimum. There are some cases where the training parameters converge to a suboptimal value, a local minimum. To overcome local minima, a "momentum" term is introduced, that is proportional to the previous change of the parameter. As the name suggests, it prevents the algorithm from getting stuck to shallow local minima by continuing to move in the parameter space, even when the gradient is zero.

$$\Delta \boldsymbol{r} \coloneqq \beta \Delta \boldsymbol{r} - (1-\beta)\boldsymbol{\nabla} E$$

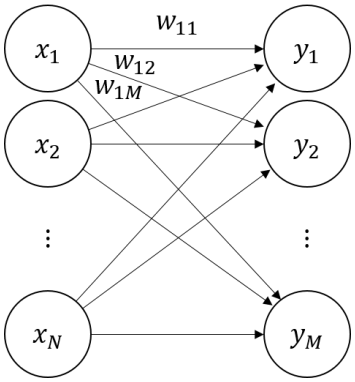
The values are updated the same way as previously

$$\Delta w \coloneqq \beta \Delta w + (1-\beta)\frac{\partial E}{\partial w}, \qquad \Delta b \coloneqq \beta \Delta b + (1-\beta)\frac{\partial E}{\partial b}$$

$$w \coloneqq w - \eta \Delta w, \qquad b \coloneqq b - \eta \Delta b$$

Multiple Neurons

For more than one neuron, the process is the same. The main difference being how we define the weights and how we calculate the gradient. The new network looks like this



For every input neuron we have M output neurons. So the weights are now represented as a matrix. The outputs are calculated the same way as before, repeated for every output neuron.

$$S_j = \sum_{i=1}^{N} w_{i,j} x_i + b_j, \qquad y_j = f(S_j)$$

Or simply

$$\mathrm{S} = \mathrm{w}^T \mathrm{x}, \quad \mathrm{y} = f(\mathrm{S})$$

The calculation of the cost changes slightly. The total cost is still the mean error of all samples and the sample cost is the square magnitude of the difference between the observed values and the responses.

$$E = \frac{1}{S}\sum_{s=1}^{S} E_s$$

$$E_s = \frac{1}{2}\|\mathrm{y} - \hat{\mathrm{y}}\|^2 = \frac{1}{2}\sum_{j}^{M}\left(y_j - \hat{y}_j\right)^2$$

Where S is the number of samples and M is the number of output neurons. The gradient equation is now

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{S}\sum_{s=1}^{S}\frac{\partial E_s}{\partial w_{ij}}$$

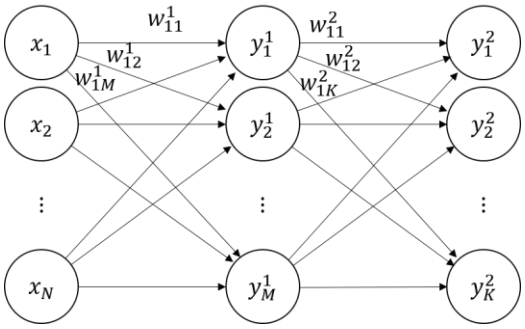
$$\frac{\partial E_s}{\partial w_{ij}} = \left(y_j - \hat{y}_j\right)\frac{\partial y_j}{\partial w_{ij}}$$
$$= \left(y_j - \hat{y}_j\right) f'(S_j)\frac{\partial S_j}{\partial w_{ij}}$$
$$= \left(y_j - \hat{y}_j\right) f'(S_j) x_i$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{S}\sum_{s=1}^{S}\left(y_j - \hat{y}_j\right) f'(S_j) x_i$$

Define $\delta_j = \left(y_j - \hat{y}_j\right) f'(S_j)$,

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{S}\sum_{s=1}^{S}\delta_j x_i$$

Multiple Layers



Having more layers makes everything more complicated. The superscript denotes the layer and is not an exponent. The goal once again is find the gradient in respect of the hidden layer weights. Note that previously the index j was used for the output but now is used for the hidden layer. The new index k now corresponds to the output layer.

$$\frac{\partial E}{\partial w_{ij}^1} = \frac{1}{S}\sum_{s=1}^{S}\frac{\partial E_s}{\partial w_{ij}}$$

$$\frac{\partial E_s}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\frac{1}{2}\sum_{k}^{K}(y_k - \hat{y}_k)^2\right)$$

$$= \sum_{k}^{K}(y_k - \hat{y}_k)\frac{\partial y_k}{\partial w_{ij}}$$

$$\frac{\partial y_k^2}{\partial w_{ij}^1} = \frac{\partial y_k^2}{\partial S_k^2}\frac{\partial S_k^2}{\partial y_j^1}\frac{\partial y_j^1}{\partial S_j^1}\frac{\partial S_j^1}{\partial w_{ij}^1}$$

$$\frac{\partial y_k^2}{\partial S_k^2}\frac{\partial S_k^2}{\partial y_j^1} = f^{2\prime}(S_k^2)w_{jk}^2$$

$$\frac{\partial y_j^1}{\partial S_j^1}\frac{\partial S_j^1}{\partial w_{ij}^1} = f^{1\prime}(S_j^1)x_i$$

$$\frac{\partial E_s}{\partial w_{ij}} = \sum_{k}^{K}(y_k - \hat{y}_k)f^{2\prime}(S_k^2)w_{jk}^2\ f^{1\prime}(S_j^1)x_i$$

$$= \sum_{k}^{K}\delta_k^2 w_{jk}^2 f^{1\prime}(S_j^1)x_i$$

$$\delta_j^1 = \sum_{k}^{K}\delta_k^2 w_{jk}^2 f^{1\prime}(S_j^1)$$

$$\frac{\partial E}{\partial w_{ij}^1} = \frac{1}{S}\sum_{s=1}^{S}\delta_j^1 x_i$$

This is the same equation as previously, however with a different delta. It turns out that this relationship can be used for any number of hidden layers, with delta being the net weighted delta of the next layer. This process propagates the initial δ, sometimes called error, from the output layer towards the input layer and is named “backpropagation”.

To make the process cleaner and easier to implement in code, the final equations are turned in vector form.

Forward propagation:

$$\mathrm{S}^1 = \mathrm{w}^{1^T}\mathrm{x}, \qquad \mathrm{y}^1 = f^1(\mathrm{S}^1)$$

$$\mathrm{S}^2 = \mathrm{w}^{2^T}\mathrm{y}^1, \qquad \mathrm{y}^2 = f^2(\mathrm{S}^2)$$

Back propagation

$$\delta^2 = (\mathrm{y}^2 - \hat{\mathrm{y}})f^{2\prime}(\mathrm{S}^2), \qquad \nabla_{\mathrm{w}^2}E = \frac{1}{S}\sum_{s=1}^{S}\mathrm{y}^1(\delta^2)^T$$

$$\delta^1 = \mathrm{w}^2\delta^2 f^{1\prime}(\mathrm{S}^1), \qquad \nabla_{\mathrm{w}^1}E = \frac{1}{\mathrm{S}}\sum_{s=1}^{S}\mathrm{x}(\delta^1)^T$$

Equations reference > Neural Networks: Algorithms and Applications > ISBN: 9781842651315

Implementation

The network seems to work, however it takes too many epochs and adding momentum makes it less effective.

```
clc; clear;

% Defining a known function
fhandle = @(x) 0.285*(-4.7*x.^3 - 4*x.^2 +
1.9*x + 3.3);
samples = 100;

% Adding noise and taking samples
X = linspace(-1,1,samples)';
Y = fhandle(X) + 0.2*randn(samples,1);

% Layer sizes
sizeIn = 1;
sizeL1 = 30;
sizeL2 = 30;
sizeL3 = 1;

% Initialising weights and biases manually
w{1} = sqrt(2/sizeIn)*randn(sizeIn,sizeL1);
w{2} = sqrt(1/sizeL1)*randn(sizeL1,sizeL2);
w{3} = sqrt(1/sizeL2)*randn(sizeL2,sizeL3);
b{1} = ones(sizeL1,1);
b{2} = ones(sizeL2,1);
b{3} = ones(sizeL3,1);

% Defining activation function and
% derivatives
f{1} = @(x) max(0,x);
f{2} = @(x) 1./(exp(-x)+1);
f{3} = @(x) x;
actderiv{1} = @(x) max(0,x)./(x+1e-20);
actderiv{2} = @(x) f{2}(x).*(1-f{2}(x));
actderiv{3} = @(x) 1;

% Pre-allocating
dEdw = cell(3,1);
dw = cell(3,1);
dEdb = cell(3,1);
db = cell(3,1);
S = cell(3,1);
delta = cell(3,1);
y = cell(4,1); % activations include inputs

for i = 1:3
  dEdw{i} = 0;
  dEdb{i} = 0;
  dw{i} = 0;
  db{i} = 0;
end

learnRate = 0.001;
m = 0.2; % momentum

for i = 1:1000
  for l = 1:3
    dEdw{l} = 0;
    dEdb{l} = 0;
    dw{l} = 0;
    db{l} = 0;
  end
  for s = 1:samples
    % Forward propagation
    y{1} = X(s);
    for l = 1:3
      S{l} = w{l}'*y{l} + b{l};
      y{l+1} = f{l}(S{l});
    end
    delta{end} = (y{end} -
Y(s)).*actderiv{end}(S{end});
    for l = 2:-1:1
      delta{l} =
w{l+1}*delta{l+1}.*actderiv{l}(S{l});
    end
    for l = 3:-1:1
      dEdw{l} = dEdw{l} + y{l}*delta{l}';
      dEdb{l} = dEdb{l} + delta{l};
    end
  end
  % Back propagation (updating parameters)
  for l = 1:3
    dw{l} = m*dw{l} + (1-m)*dEdw{l};
    db{l} = m*db{l} + (1-m)*dEdb{l};
    w{l} = w{l} - learnRate*dw{l};
    b{l} = b{l} - learnRate*db{l};
  end
end

% Generating predictions
pred = zeros(sizeIn,1);
for s = 1:samples
    y{1} = X(s);
    for l = 1:3
      S{l} = w{l}'*y{l} + b{l};
      y{l+1} = f{l}(S{l});
    end
    pred(s) = y{end};
end

figure(1)
plot(X,Y);
hold on
plot(X,pred);
hold off
```

Implementation using MATLAB's Deep Learning toolbox

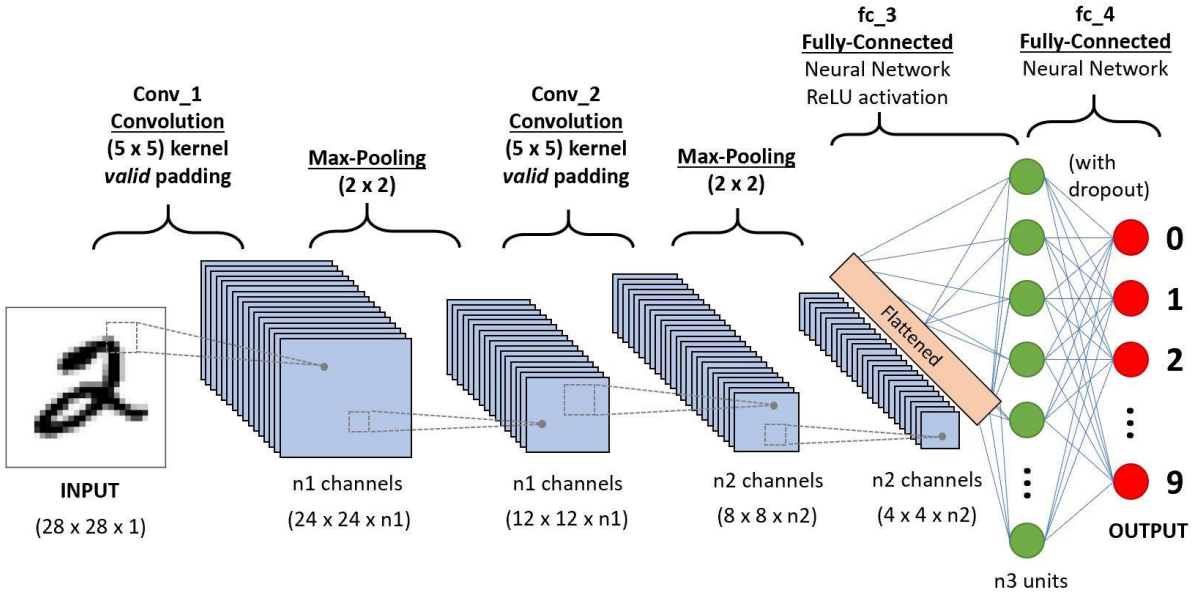
In this case the weight initialisation is different. You can select other options, however glorot is the preferred method. While glorot generates random numbers from a normal distribution with a deviation of $\sqrt{1/(\text{sizeIn} + \text{sizeOut})}$, I used $\sqrt{1/\text{sizeIn}}$ which helped it better. Unfortunately, you cannot use a custom distribution (or can you?).

```
clc; clear;

% Defining the same function
fhandle = @(x) 0.285*(-4.7*x.^3 - 4*x.^2 + 1.9*x + 3.3);
samples = 100;

% Adding noise and taking samples
X = linspace(-1,1,samples)';
Y = fhandle(X) + 0.2*randn(samples,1);
X = permute(X,[4,3,2,1]);

% Defining network architecture
layers = [
  imageInputLayer([1 1 1],"Name","imageinput","Normalization","none")
  fullyConnectedLayer(30,"Name","fc_1")
  reluLayer("Name","relu")
  fullyConnectedLayer(30,"Name","fc_2")
  tanhLayer("Name","tanh")
  fullyConnectedLayer(1,"Name","fc_3")
  regressionLayer("Name","regressionoutput")];

% Defining training options
options = trainingOptions('sgdm','Plots','training-progress',...
  'MaxEpochs',1000,'MiniBatchSize',100,'InitialLearnRate',0.001,...
  'Momentum',0.2);

% Training the network
net = trainNetwork(X,Y,layers_2,options);

% Getting predictions and plotting
pred = predict(net,X);
figure(1)
x = permute(X,[4,3,2,1]);
plot(x,pred);
hold on
plot(x,Y);
hold off
```

Note: The input layer is called "imageInputLayer", however that does not mean that it is only meant for images. An array of any size can be used if you format it so that the size of the last dimension is the number of training samples. Normally the input layer takes $W \times H \times C$ where C, is the number of colour channels.

This network is not very effective for image classification/regression, but it does provide the basis for deep and complex networks. On their own they are called Feedforward networks, but most of the time they are combined with convolutional layers. These provide effective feature extraction in images, which is also rotationally invariant. The fully connected layers that make up a feedforward network are used as the actual "brain" of the network.

While a single neuron in a FFN is “observing” a single pixel, a convolutional filter in convolutional neural networks (CNNs) is swiped across the image. In CNNs, the filters are equivalent to weights. When fully trained, a single filter can learn to detect a certain feature of an image such as edges, colour, contrast, etc. In deeper convolutional layers, the filters learn to detect more specific features. Depending on the dataset, these can be complete objects such as body parts, vehicles, flowers and more.



For my purposes, it is easier to use the DL toolbox for anything more than a feedforward network. It uses the GPU to perform operations, whereas MATLAB scripts run on a single CPU thread. You could in theory use gpu arrays and parfor loops, but there is no reason to do that when a powerful toolbox is available for the same job. In addition to that I have not managed to understand or derive backpropagation in CNNs completely.

Implementation for sensor-less adaptive optics

The idea is that a CNN can be taught to detect the features of a far-field image when certain voltages are applied to the deformable mirror.

Below are some methods that utilise a CNN.

- Train on far-field image inputs and predict voltages.
- Train on far-field images and predict Legendre moments.
- Decompose far-field images into discrete polynomials and train on moment vectors to predict voltages/Legendre moments. Basically treating the moments as feature vectors that would normally be produced by convolutional layers.
- Define metrics to detect certain aberration orders and use them as feature vectors.
- Train network to be able to determine the order causes most aberration and reduce or increase value until new order is predicted.

This concludes the theory and implementation of Deep Learning in MATLAB.

Phase correction using machine learning

[8]

The idea is training a neural network to predict the required phase correction given a far-field image. Sounds simple enough, however if humans cannot tell what the wavefront shape is based in the far-field, then why should we expect a machine to do? In this section of my report I will be testing and coming up with ideas on how deep learning can be used to perform phase optimisation.

As explained in the Deep learning section of the report, a big dataset is required to effectively train a neural network.

Current ideas so far:

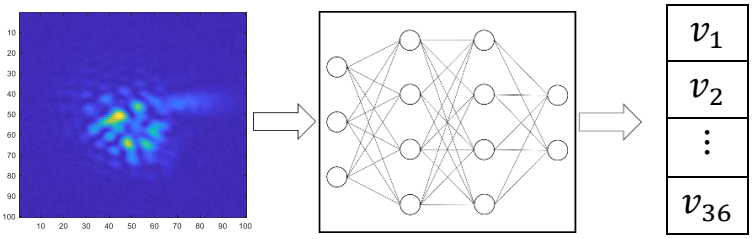
1. Input: Far-field image. Output: Actuator voltages
2. Input: Far-field image. Output: Wavefront Legendre coefficients
3. Input: Far-field Chebyshev moments Output: wavefront Legendre coefficients
4. Input: Far-field features. Output: Wavefront Legendre coefficients

Note that there is a linear transformation between voltages and wavefront coefficients. I believe using coefficients is easier, as a single order will have a clear effect on the far field, but multiple voltages are required for the same result.

Idea 1

This is the most straight forward plan and also the first I came up with. This is also the one that taught me neural networks are not magic. Simply generate thousands of wavefronts by applying random voltages to the mirror model. The input wavefront is 0, so the far-field is produced by the mirror deformation directly. Calculate the far-field for each and save it together with the voltage applied.

Then, train a convolutional network that takes as input a far-field image and returns the actuator voltages that would be required to flatten the wavefront. This means that the targets are opposite of the voltage values that caused the deformation initially.



Generating dataset

There are a few ways to store a dataset. For convolutional neural networks it is recommended that an image datastore is used, however this is only for image classification. For that reason, a file datastore will be created instead, where the input images are stored in a folder and the voltages arrays in another folder as csv files. To make the process even faster, we will parallelise the process with multiple CPU threads. Fortunately, I have 16 available threads. Note that this does not mean it will be 16 times faster, but the difference with just one or four threads is drastic.

In a published paper [insert reference] the researchers achieved the same thing I am trying to do. They used 80,000 far-field images. I trust that 80,000 images are more than enough. That is 5,000 images per thread.

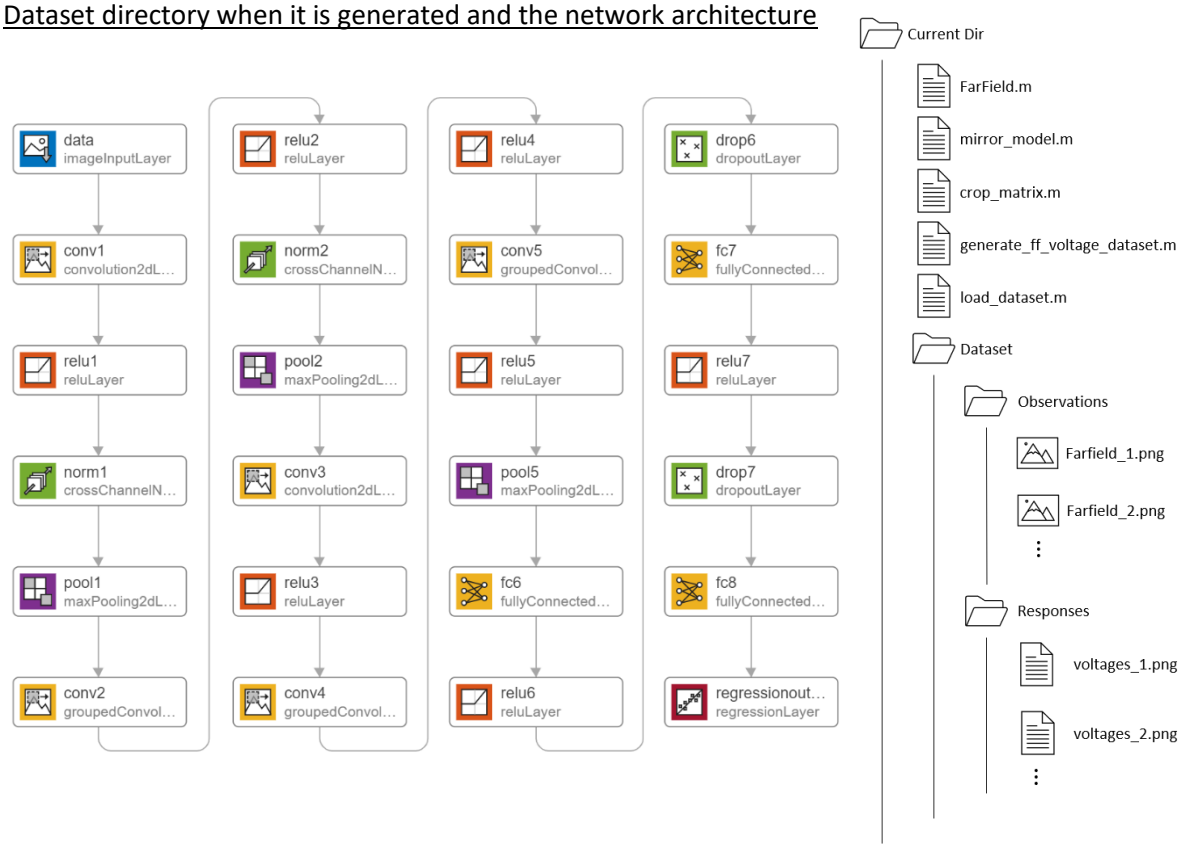
```
function generate_ff_voltage_dataset(dataset_size)
    % Generates dataset of farfields vs voltages
    % Requires mirror_model.m, FarField.m and crop_matrix.m
    tic
    poolobj = gcp;
    addAttachedFiles(poolobj,{'mirror_model.m','FarField.m','crop_matrix.m'})
    workers = poolobj.NumWorkers;

    mkdir('Dataset')
    mkdir(fullfile('Dataset','Responses'))
    mkdir(fullfile('Dataset','Observations'))

    files_per_worker = floor(dataset_size/workers);

    parfor worker = 0:workers-1
        mirror = mirror_model(36,71,0.1);
        FF = FarField;
        FF.settings(0.5,[],[],[]);

        start_idx = worker * files_per_worker + 1;
        final_idx = start_idx + files_per_worker - 1;

        for i = start_idx:final_idx
            voltages = 100*rand(36,1)-50;
            mirror.set_channels(voltages);
            farfield = FF.generate_farfield(1,crop_matrix(mirror.shape,10));
            farfield = uint8(mat2gray(farfield)*255);

            fname = sprintf('voltages_%i.csv',i);
            fdir = fullfile('Dataset','Responses',fname);
            writematrix(voltages,fdir);

            fname = sprintf("farfield_%i.png",i);
            fdir = fullfile('Dataset','Observations',fname);
            imwrite(farfield,fdir);
        end
    end
    end_time = toc;
    fprintf("Duration: %.2f\n",end_time);
end
```

The process took 100 seconds only. After generating the dataset, I realised the in total the data should not take more that 200MB of space, so I decided to load all the images and the targets into two variables, rather than point to their location. This is much faster, however for massive datasets that do not fit in the GPU's memory, then a datastore should be used.

Below is the code use to load all the data.

```
function dataset = load_dataset(start_idx,end_idx)
    dataset_size = end_idx - start_idx;
    observations = zeros(51,51,dataset_size);
    responses = zeros(dataset_size,36);
    for i = start_idx:end_idx
        fname = sprintf('voltages_%i.csv',i);
        fdir = fullfile('Dataset','Responses',fname);
        responses(i,:) = readmatrix(fdir)';

        fname = sprintf("farfield_%i.png",i);
        fdir = fullfile('Dataset','Observations',fname);
        observations(:,:,i) = imread(fdir);
    end
    dataset = {observations;responses};
end
```

Dataset directory when it is generated and the network architecture



Network architecture

The network model is identical to alexnet, a network developed by Alex Krizhevsky. It was the first fast GPU-implementation of a CNN to win an image recognition contest. In this case I will be modifying it to perform regression and will also be changing the size of the image inputs to make it work with my 51x51 far-fields.

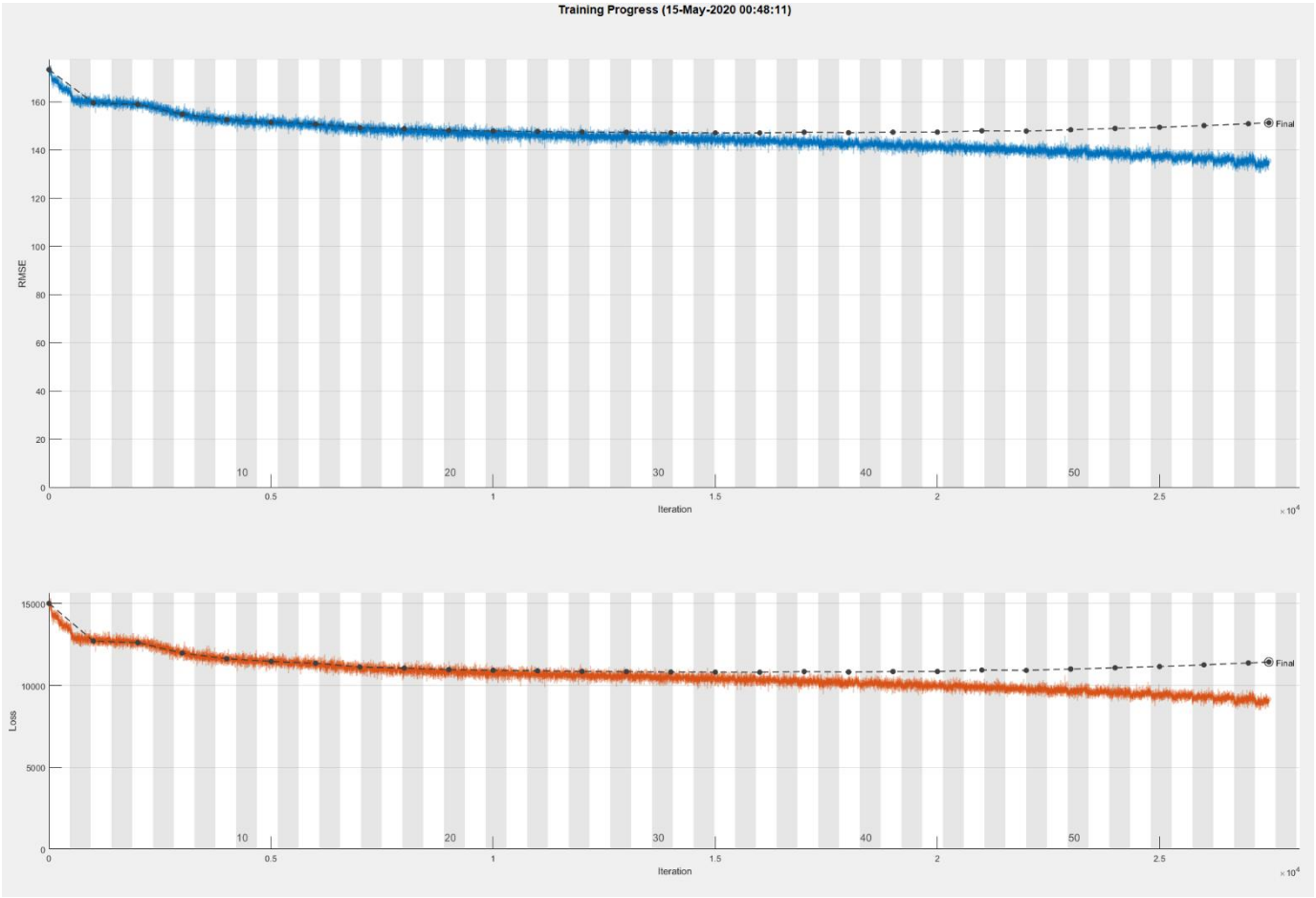
Below is the code on how to generate and modify the layers from alexnet

```
layers = alexnet('Weights','none');
layers(1) = imageInputLayer([51 51 1],'Name','data');
layers(16) = maxPooling2dLayer([3 3],'Stride',[2 2],...
    'Padding','same','Name','pool5');
layers(end-2) = fullyConnectedLayer(36,'Name','fc8');
layers(end-1) = regressionLayer('Name','regressionoutput');
layers(end) = [];
```

The max pooling layer had to be replaced with a new one that has its padding set to ‘same’. Without this, the output size does not quite add up since we changed the image input size. The SoftMax layer normalises the output so that it can be interpreted as a probability that a given input belongs to a certain category or label. This must only be used in classification networks, not regression.

Idea 1 Results

Training was unsuccessful, with a final rms of 150 and quite a bit of overfitting. It is clear that the network is overfitting because the cost is decreasing for the training dataset but is increasing for the validation dataset.



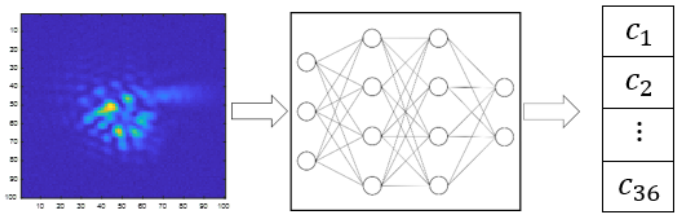
From this result I can conclude that due to either the parameters, the architecture or the complexity of the problem, the network is unable to find a pattern between the far-field images and the mirror control signals.

For now, I will keep the same network and parameters, and try to see for which method the network is able to get the best result. Then hopefully I can develop the network while using that method.

Idea 2

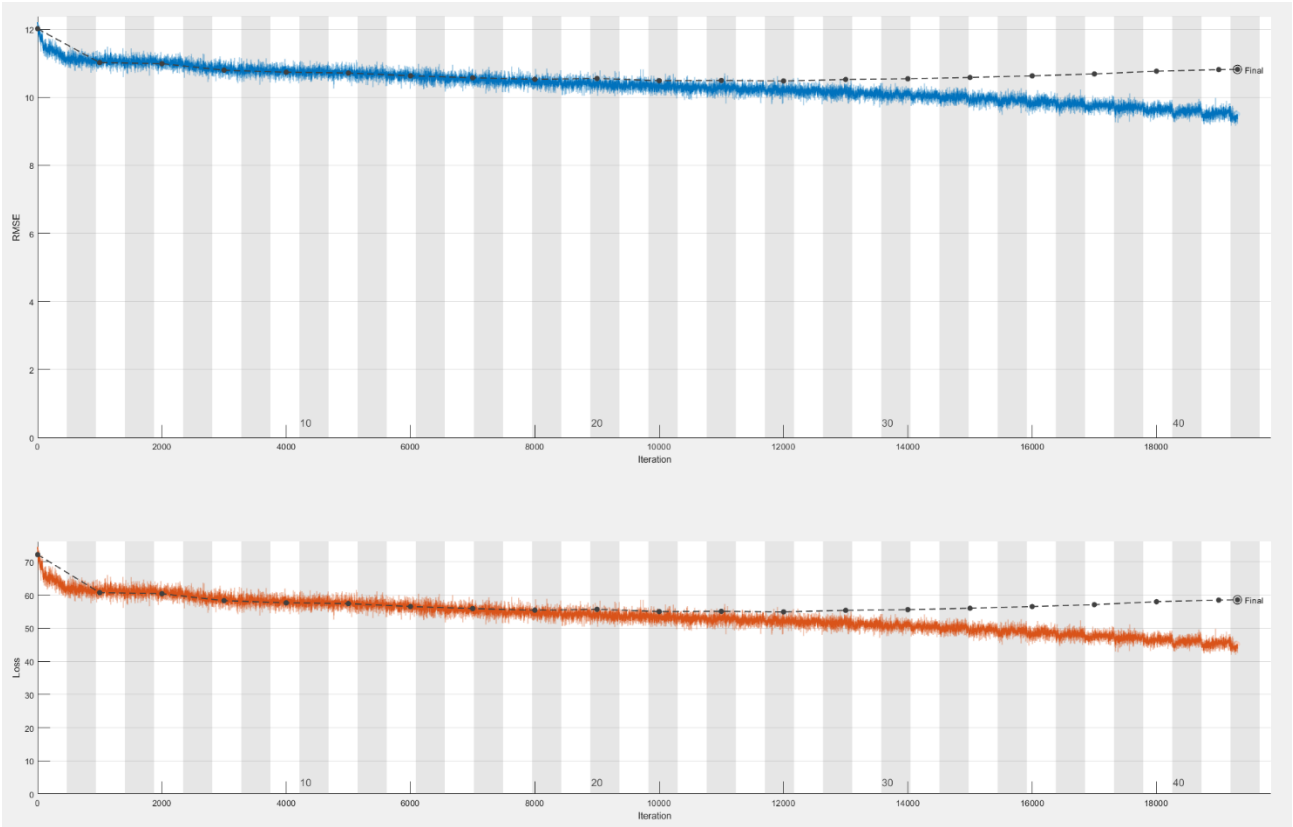
This time I will be generating random Legendre coefficients, and from those I will construct a wavefront for which I will simulate the resulting far-field. Then the far-field will be used as an input to the network and instead of trying to predict the voltages, it will try to estimate the coefficients. Effectively it will be learning how to predict the wavefront from the far-field. I expect this to be slightly more effective than the voltages because a single coefficient has a more predictable effect in the far-field. For example, the 2nd order will produce a tilt, the defocus term will produce defocusing etc. It should be easier for it to find a pattern.

The counter argument to this is that there is a linear transformation between voltages and coefficients, so in the end it might not even mater.



Idea 2 Results

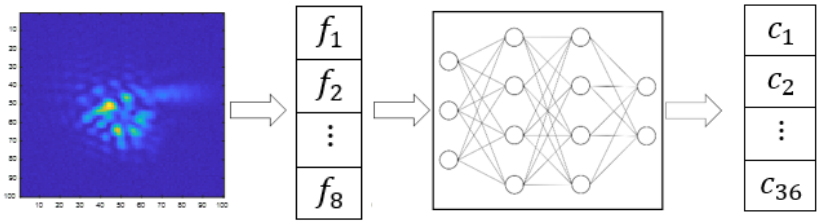
Similar to the first idea, the network overfitted the dataset after a while with a minimum rms of 10.48



For some reason, the rms started at a lower value than the previous experiment, which is surprising to me.

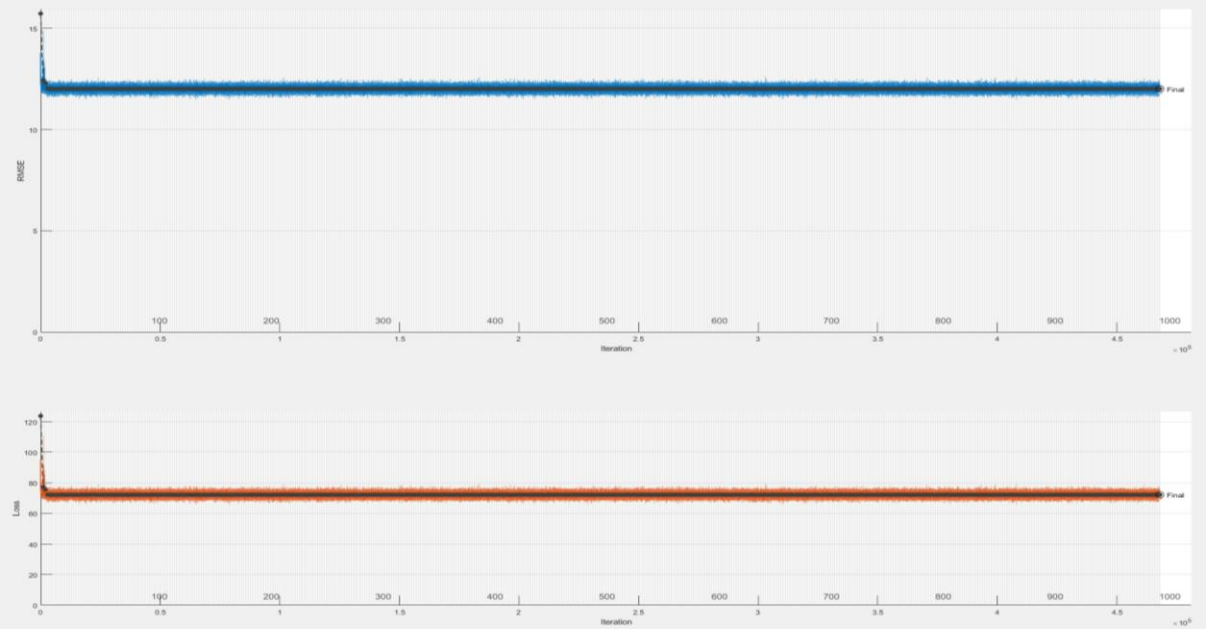
Idea 4

This time rather than defining image features as discrete moments, I defined 8 metrics for the far-field image. They are image brightness, image contrast, number of edge pixels using Sobel filter, mean of the top 60% pixel values and mean of the lower 60% values, the FWHM of x and y slices at the central spot and finally the Strehl integral.

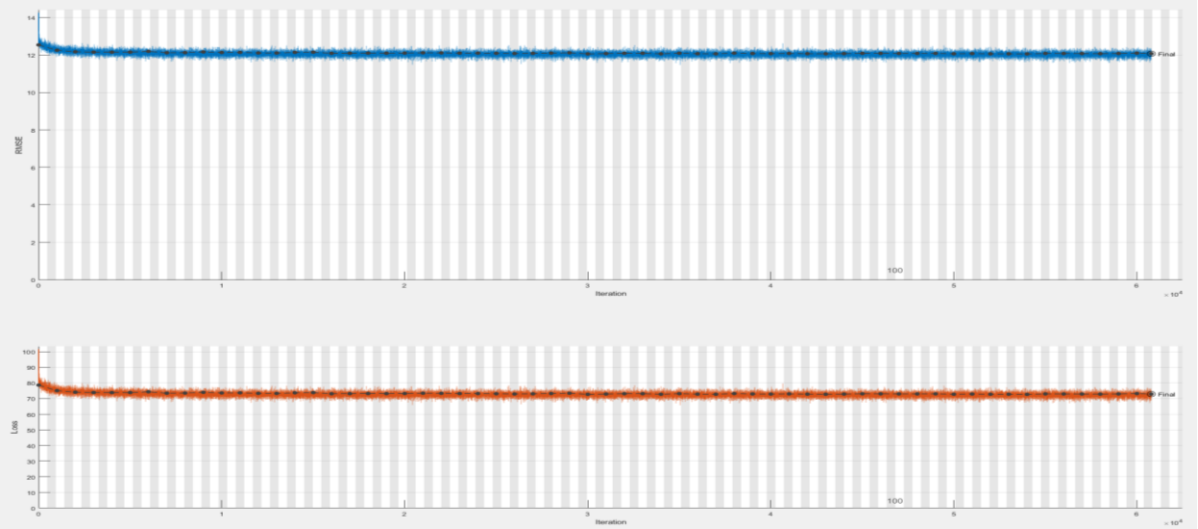


Idea 4 results

The network could not even train. The weights and biases barely changed much from their initial value which was 0. It seems that 8 image metrics are not enough to extract information about the wavefront. Maybe reducing the number of coefficients would help, but I have little hope that this will help.



Biases initialised with 1s: Because all outputs had a similar value, I tried again with a bias value of 1 rather than 0. Still no difference.



Idea 3

Unfortunately, I did not manage to create a functioning transform for the Chebyshev moments of an image so I could not test this idea.

Conclusions

The phase optimisation algorithms seem promising, especially coordinate search. The algorithm should be updated so that it converges slower near the minimum. A possible solution would be to change the way the new search range limits are calculated. You can select a range slightly bigger than the range bounded by the samples next to the optimum.

Deep learning is not out of the picture yet, as the Deep learning model was shown to work (see [7]). Either the network is not deep enough, or the training parameters are not ideal (or both). Ideally a real dataset should be gathered from a real mirror and farfield. Then the different architectures should be tested to see which produces the best results and then fine tune it.

Final Notes

The code included in this report can be found in my GitHub repository
https://github.com/PetrosOratiou/WSAO-Project-STFC-Placement

References

[1] Hooke, R.; Jeeves, T.A. (1961). ""Direct search" solution of numerical and statistical problems". Journal of the ACM. 8 (2): 212–229. doi:10.1145/321062.321069.

[2] J. A. Nelder, R. Mead, A Simplex Method for Function Minimization, *The Computer Journal*, Volume 7, Issue 4, January 1965, Pages 308–313, https://doi.org/10.1093/comjnl/7.4.308

[3] Curry, Haskell B. (1944). "The Method of Steepest Descent for Non-linear Minimization Problems". Quart. Appl. Math. 2 (3): 258–261. doi:10.1090/qam/10667

[4] Diederik P. Kingma, Jimmy Ba “Adam: A Method for Stochastic Optimization” conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015 arXiv:1412.6980

[5] Legendre Polynomials (Wikipedia) https://en.wikipedia.org/wiki/Legendre_polynomials

[6] Goodman, Joseph (2005). Introduction to Fourier Optics (3rd ed, ed.). Roberts & Co Publishers. ISBN 0-9747077-2-4. Book: section 6.4.1 page 145 “The Generalized Pupil Function”

[7] Qinghua Tian, Chenda Lu, Bo Liu, Lei Zhu, Xiaolong Pan, Qi Zhang, Leijing Yang, Feng Tian, and Xiangjun Xin, "DNN-based aberration correction in a wavefront sensorless adaptive optics system," Opt. Express 27, 10765-10776 (2019)

[7] Guohao Ju, Xin Qi, Hongcai Ma, and Changxiang Yan, "Feature-based phase retrieval wavefront sensing approach using machine learning," Opt. Express 26, 31767-31783 (2018)

[8] Matlab Deep learning courses https://matlabacademy.mathworks.com