Classical Zeno Effect Seen in the Coherence of Light Sources

In this notebook, we wish to repeat the results of [Soruce name blah blah]. In this paper, the authors perform an example of the Zeno effect in the classical limit, showing that use of slits to cause a distrubance on a beam of light can cause the measured intensity over a set distance to increase.

To do this, we have 4 equations we need to find:

1. $J(x_1, x_2) = \langle E(x_1)E^*(x_2)\rangle = \frac{1}{2a}exp\left[\frac{(x_1-x_2)^2}{d^2}\right]$
2. $P = \int_{-a}^{a} J(x,x)dx$
3. $\mu_g = \frac{1}{P}\cdot\left[\int\int_{-a}^{a}|J(x_1,x_2)|^2\right]^{1/2}$
4. $J(x_1', x_2') = \frac{k}{2\pi z}\cdot\int\int_{-a}^{a} J(x_1,x_2)\times K(x_1'-x_1)K^*(x_2'-x_2)dx_1dx_2$

Where:
$K(x) = exp\left[\frac{ikx^2}{2z}\right]$

We solve this set of equation numerically.

```
Main.workspace3.Data
```

```julia
"""
Struct containg all variables and data arrays needed for this calculation.
# Arguments
- n::Int : Dimensionality of the system
- zmax::Float : Distance from source slit to detector
- nr::Int : Number of slits the beam is to pass through, excluding souce and detector slit.
- dd::Float - 'transverse coherence length is proportional to dd'
"""
mutable struct Data
    n; zmax; nr; dd;

    a; b; c; d; e;

    x;

    z; step; ss;

    function Data(n, zmax, nr, dd)

        a = complex(zeros(n,n))
        b = complex(zeros(n,n))
        c = complex(zeros(n,n))
        d = complex(zeros(n,n))
        e = complex(zeros(n,n))

        x = zeros(n) * 0.0

        z = zmax/nr
        #step seems to be our 'dx'
        step = 2/(n-1)

        #This is the only
        ss = step * (1-1im)/(2*sqrt(pi * z))

        return new(n,zmax,nr,dd,a,b,c,d,e,x,z,step,ss)
    end
end
```

Main.workspace3.calculatePower

```julia
"""
Impliments equation (2)
"""
function calculatePower(dat::Data)
    power = 0

    for i in 1:dat.n-1
        power += dat.a[i,i] * dat.step
    end
    #This currently can return a complex number. Is this correct?
    return power
end
```

Main.workspace3.calculateCoherence

```julia
"""
Impliments equation (3)
"""
function calculateCoherence(dat::Data, power)
    coh = 0;

    for i in 1:dat.n
        for j in 1:dat.n
            #
            coh += abs(dat.a[i,j])^2 * 2 * dat.step^2
        end
    end

    coh = sqrt(coh)/power
    return coh;
end
```

getCentreIntensity (generic function with 1 method)

```julia
function getCentreIntensity(dat::Data)
    return dat.a[convert(Int64, (dat.n+1)/2), convert(Int64, (dat.n+1)/2)]
end
```

initExperimentValues (generic function with 1 method)

```julia
function initExperimentValues(dat::Data)
    for i in 1:dat.n
        dat.x[i] = -1 + 2 * (i-1)/(dat.n-1);
    end

    for i in 1:dat.n
        for j in 1:dat.n

            #dat.a = J(x1,x2) -> Equation 1 (a=1)
            dat.a[i,j] = exp(- (dat.x[i] - dat.x[j])^2 / dat.dd^2)/2

            #Equation for K(x) and K*(x)
            #We have also included the numberical
            dat.b[i,j] = exp(-1im * (dat.x[i]-dat.x[j])^2 / (2*dat.z)) * dat.ss
            dat.d[i,j] = conj(dat.b[i,j])
        end
    end
end
```

```
iterateOverSlits (generic function with 2 methods)
```

```julia
function iterateOverSlits(dat::Data, print_all=false)
    """
    Pretty sure this calculates over equation 4 ?
    """

    for m in 1:dat.nr
        # println("Iteration $m our of $dat.nr")

        #print_all && println("\tStarting first Quandrant")
        #Fill the first octant of e
        for i in 1:convert(Int64, (dat.n+1)/2)
            for j in 1:i
                dat.e[i,j] = 0
                for k in 1:dat.n
                    dat.c[k,j] = 0
                    for l in 1:dat.n
                        dat.c[k,j ] += dat.a[k,l] * dat.b[l,j]
                    end
                    dat.e[i,j] += dat.d[i,k] * dat.c[k,j]
                end
            end
        end

        #print_all && println("\tStarting below Quandrant")
        #Fill the octant below
        for i in convert(Int64, (dat.n+3)/2):dat.n
            for j in 1:(dat.n-i+1)
                dat.e[i,j] = 0
                for k in 1:dat.n
                    dat.c[k,j] = 0
                    for l in 1:dat.n
                        dat.c[k,j] += dat.a[k,l] * dat.b[l,j]
                    end
                    dat.e[i,j] += dat.d[i,k]* dat.c[k,j]
                end
            end
        end
        #print_all && println("\tStarting final Quandrants")
        for i in convert(Int64, (dat.n+3)/2):dat.n
            for j in (dat.n-i+2):i
                dat.e[i,j] = conj(dat.e[dat.n+1-j, dat.n+1-i])
            end
        end


        #print_all && println("\tFinalising data")
        for i in 1:dat.n
            for j in 1:dat.n
                if( j<= i)
                    dat.a[i,j] = dat.e[i,j]
                else
                    dat.a[i,j] = conj(dat.e[j,i])
                end
            end
        end
    end

    return
end

```

```
runExperiment (generic function with 1 method)
```

```
function runExperiment(dat::Data)
    initExperimentValues(dat)

    startPower = calculatePower(dat)
    startCoh = calculateCoherence(dat, startPower)
    startIntensity = getCentreIntensity(dat)
    iterateOverSlits(dat)

    endPower = calculatePower(dat)
    endCoh = calculateCoherence(dat, endPower)
    endIntensity = getCentreIntensity(dat)

    return startPower, startCoh, startIntensity, endPower, endCoh, endIntensity

end
```

```
Main.workspace3.runMultipleExperiments
```

```
"""
Runs a set of different experiments, with all varaibles constant except
the total number of slits 'nr'.
# Arguments
- 'n::Integer' : The dimensionality to solve over
- 'zmax::Float' : The distance between soruce slit and detector
- 'dd::Float' : Not really sure
"""
function runMultipleExperiments(n, zmax, dd, min_nr, max_nr)

    results = zeros(max_nr-min_nr + 1, 7) * 1im

    for nr in min_nr:max_nr
        dat = Data(n,zmax, nr, dd)
        sPow, sCoh, sInt, ePow, eCoh, eInt = runExperiment(dat)
#       println(nr - min_nr + 1)
        results[nr - min_nr + 1, 1] = nr
        results[nr - min_nr + 1, 2] = sPow;
        results[nr - min_nr + 1, 3] = sCoh;
        results[nr - min_nr + 1, 4] = sInt;

        results[nr - min_nr + 1, 5] = ePow;
        results[nr - min_nr + 1, 6] = eCoh;
        results[nr - min_nr + 1, 7] = eInt;

    end
    return results

end
```

```
0.1
```

```
begin
    #currently using a small n=51, this should be increased to 201
    n = 51
    zmax = 0.5
    dd = 0.1
end

```

```
"Experiment run succesfully for slit counts between 1 and 20"
```

```
begin
    results = runMultipleExperiments(n, zmax, dd, 1, 20);
    "Experiment run succesfully for slit counts between 1 and 20"
    # we wish z_d = 0.7/ka^2

end
```

We have generated and stored our results as a 2d array. Each row represents a different experiment, with the collomns representing (from left to right):

1. NR - The number of slits for this experiment
2. Start Power (Complex?)
3. Start Coherence
4. Start Intensity
5. End Power (Complex)
6. End Coherence
7. End Intensity

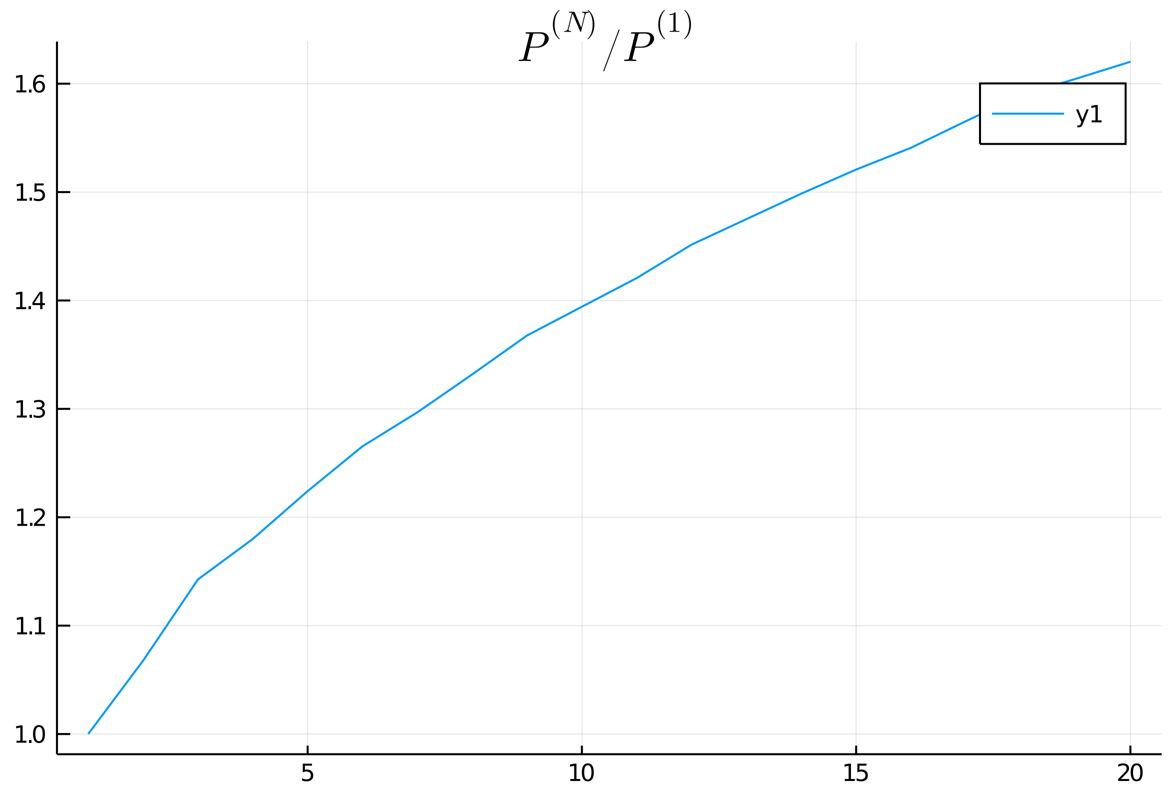
We can now try and plot these, such that we can compare them to the original paper.

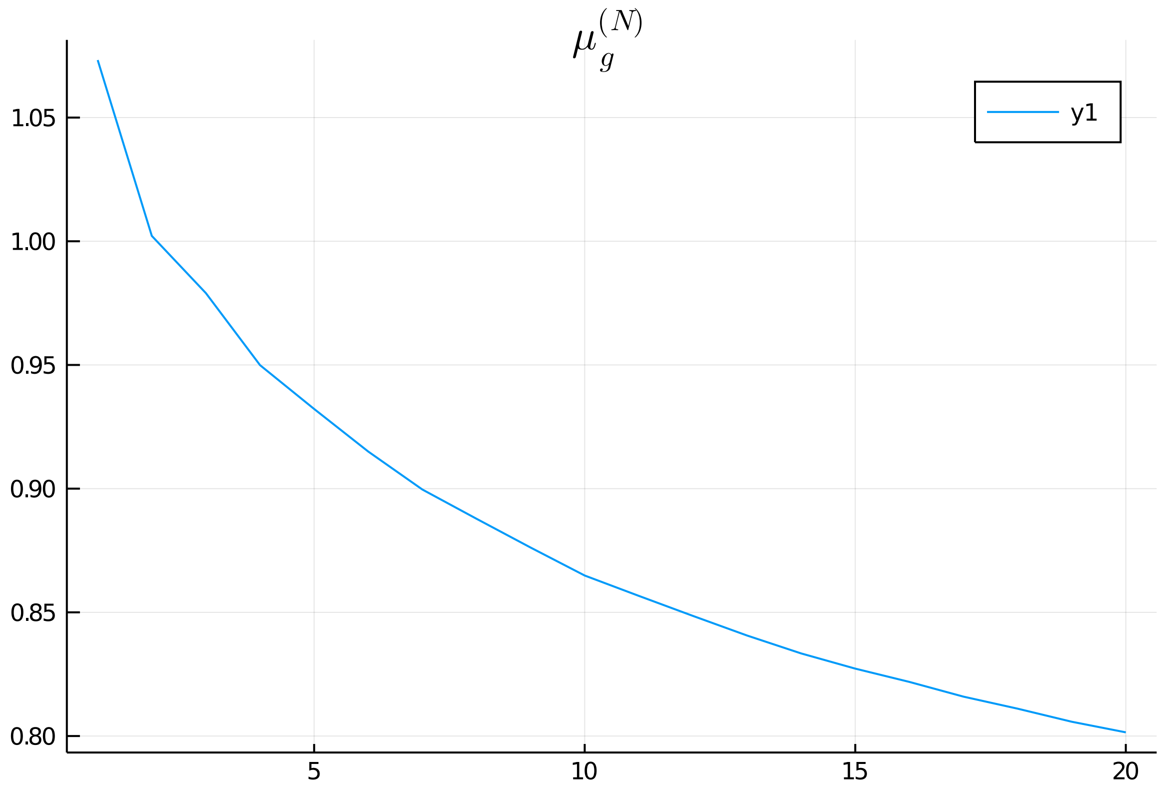
```
md"""
We have generated and stored our results as a 2d array. Each row represents a
different experiment, with the collomns representing (from left to right):\

1.    NR - The number of slits for this experiment
2.    Start Power (Complex?)
3.    Start Coherence
4.    Start Intensity
5.    End Power (Complex)
6.    End Coherence
7.    End Intensity

We can now try and plot these, such that we can compare them to the original paper.
"""
```

```
"Plot values succesfully extracted"
```




$\mu_g^{(N)}$
y1
1.05
1.00
0.95
0.90
0.85
0.80
5
10
15
20