Brute Force Simulation

Simulation Configuration

Classical field parameters

636

Number of emitting atoms

293

Temperature (K)

10

Time resolution (ps)

5

Maximum time (ns)

f0 (GHz) = 456811.0

Δf (GHz) = 1.0

σ (GHz) = 7.58

Photon count parameters

10

Expected counts in time tmax

1000

Total photon counts overall (approximate)

Number of classical simulations: 100

Plotting options

Available Plots:

- Single instance of classical calculations ☑
- Average of all classical calculations ☑
- Total counts from all time windows ☑
- Autocorrelation of photon count time series ☑

Make Plots [☑]

Plots

Window:

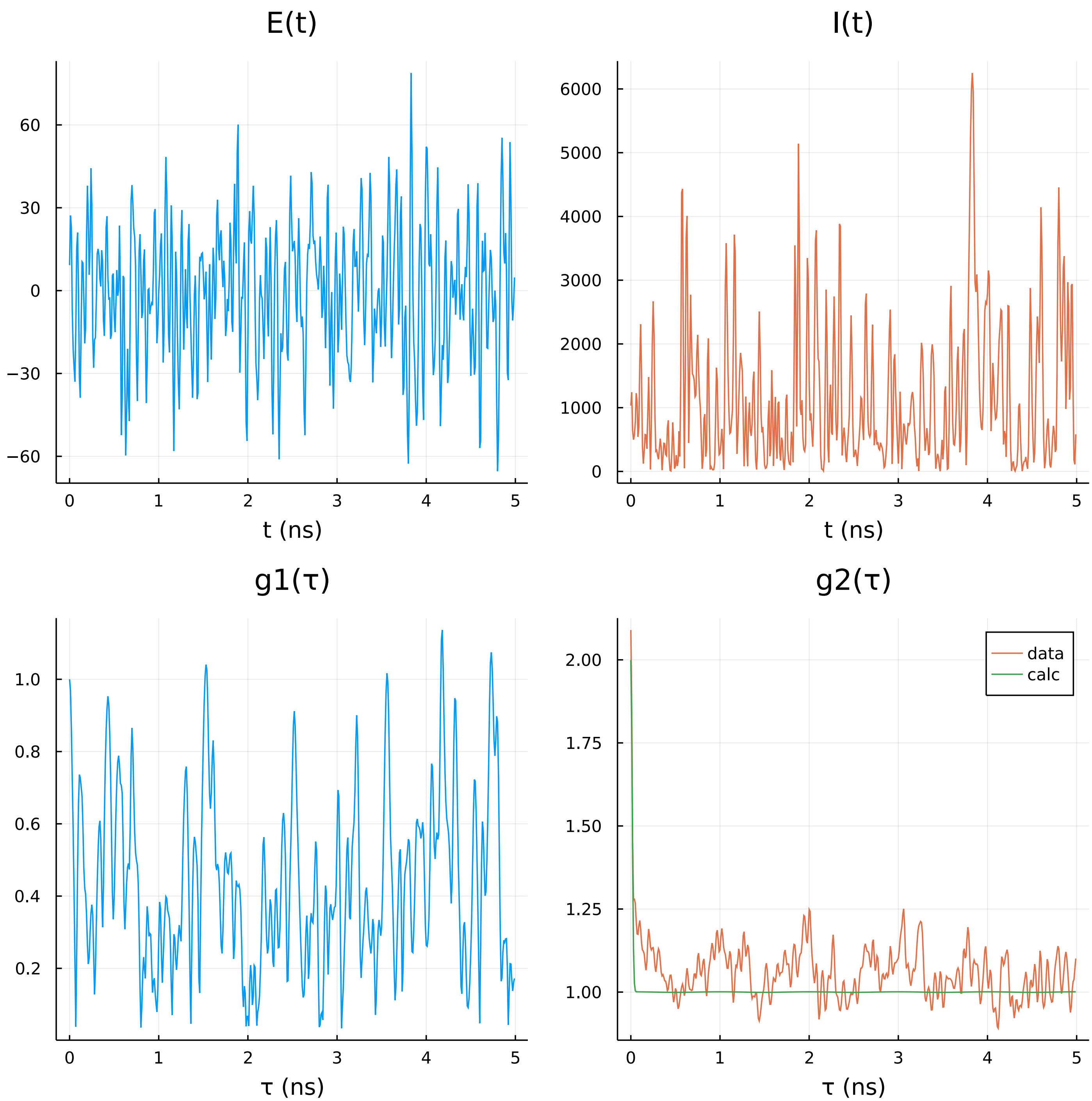
Window size (ns) = 5.0

Window position (ns) = 0.0

Size:

Position:

Single instance of classical calculations:

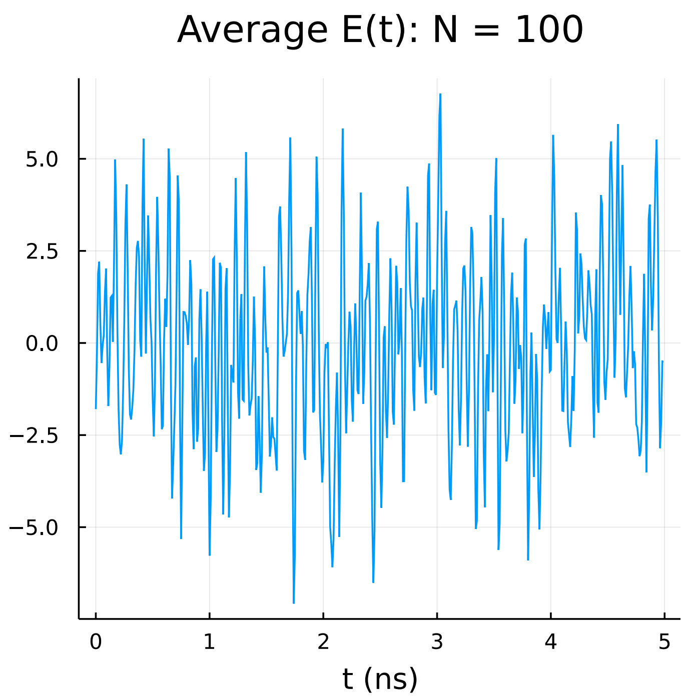


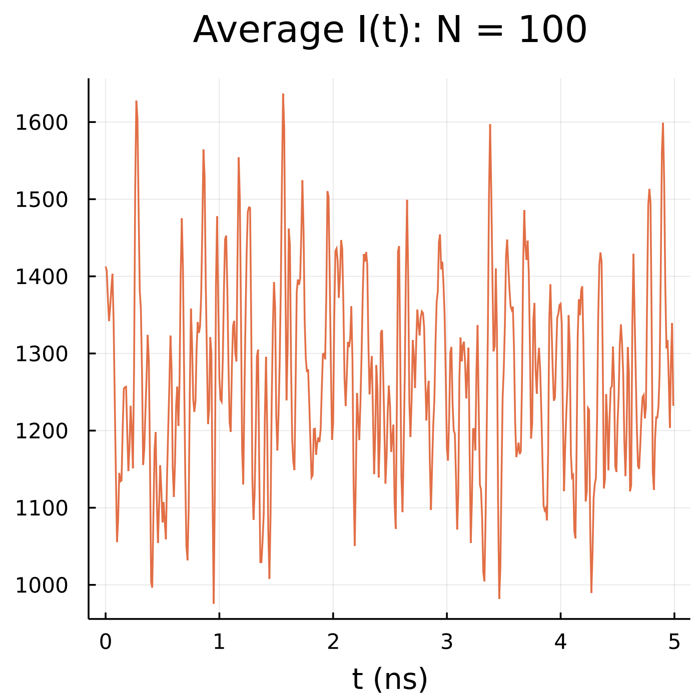
Download data for classical field and correlation plots:

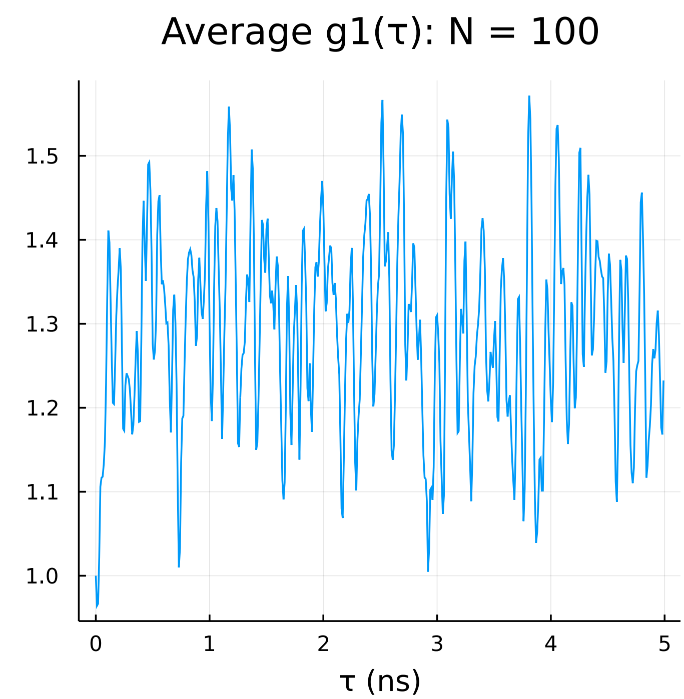
E-field/Intensity: Download... classical_field_single.csv

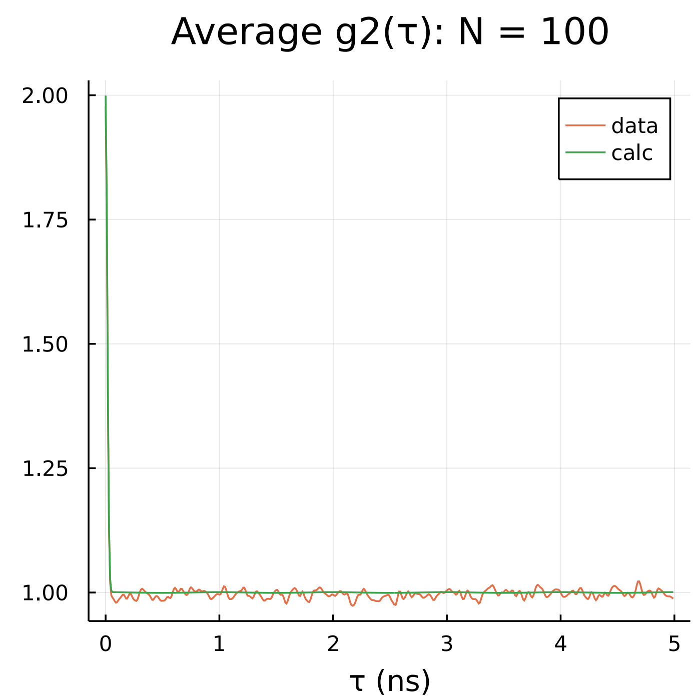
Correlations: Download... classical_correlations_single.csv

Average of all classical calculations:





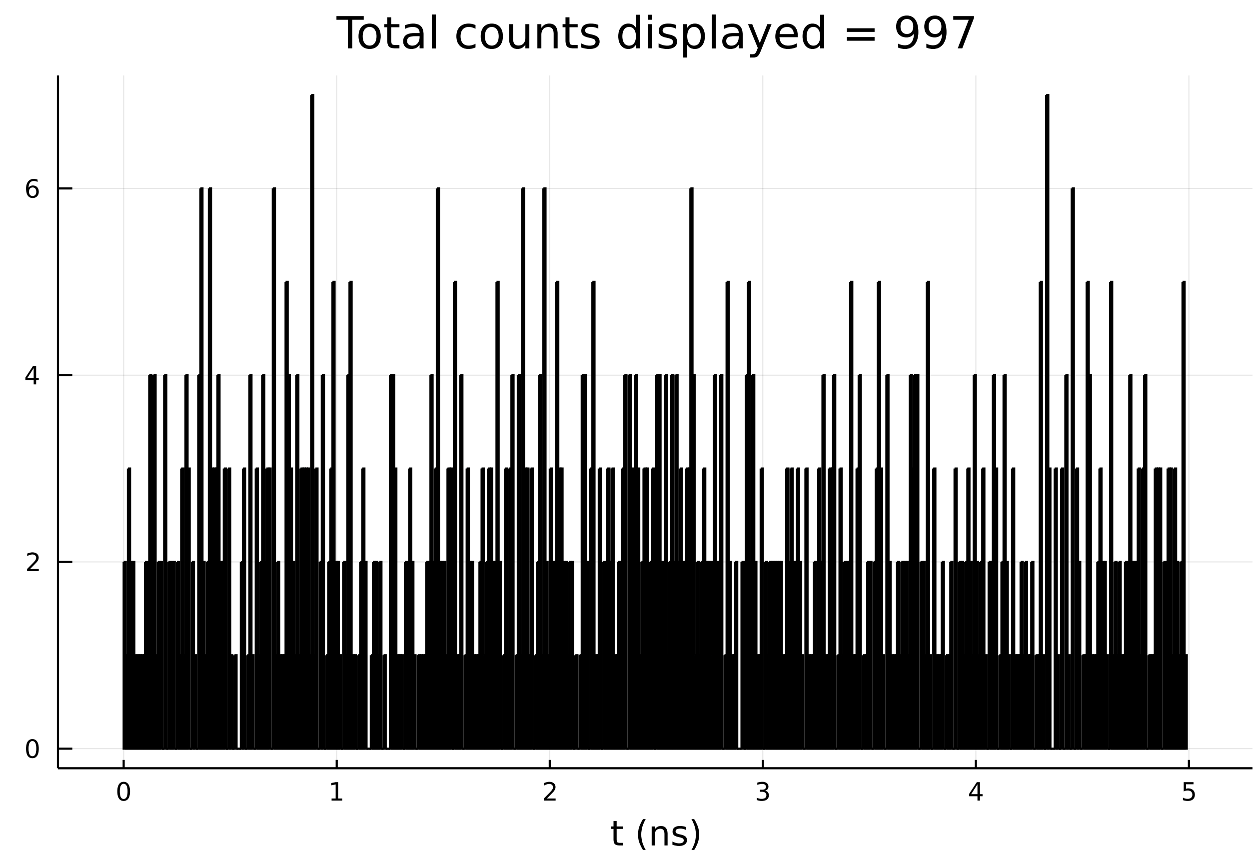




E-field/intensity: Download... classical_field_avg.csv

Correlations: Download... classical_correlations_avg.csv

Total counts:



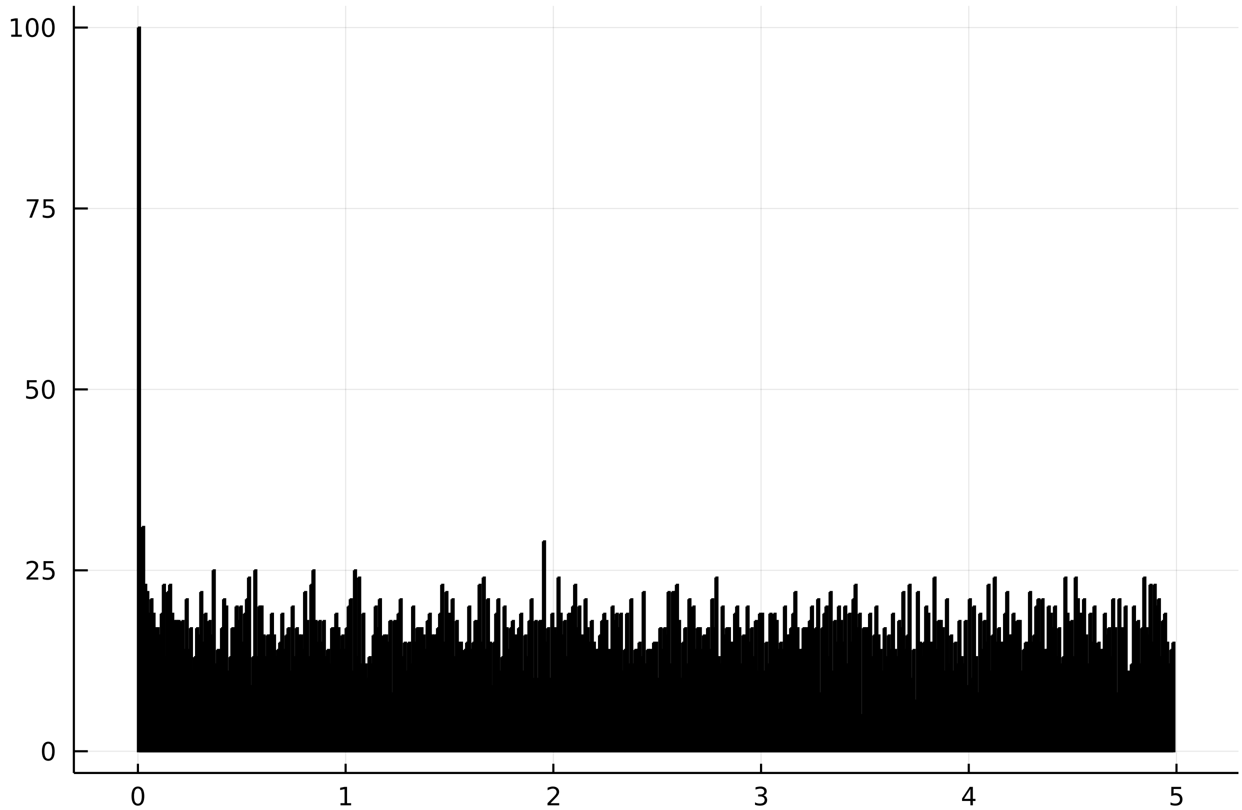
Bin edges: Download... photon-count_histogram_bin-edges.csv

Bin weights: Download... photon-count_histogram_bin-weights.csv

Arrival times: Download... photon-arrival-time_data.csv

Autocorrelation of photon counts

Note that the $\tau = 0$ bin matches the number of simulations. This only happens because we are using the autocorrelation for a single beam. This will not be true when I add a beam splitter.



Bin edges: Download... photon-correlation_histogram_bin-edges.csv

Bin weights: Download... photon-correlation_histogram_bin-weights.csv

Correlation times: Download... photon-correlation-tau_data.csv

Calculations

Classical calculations

```julia
# these parameters are needed for all simulations
begin
    # Balmer-α lines specified here
    ωM = 2*π*[456811.0, 456812.0]
    # magnitude of each line
    mag = convert(Vector{ComplexF64},ones(length(ωM)))
    # calculate line differences
    ΔM = ωM .- ωM[1]
    # generate times in tres ps intervals up to 2*tmax
    times = collect(0:tres*1e-3:2*tmax);
    # limit the window to tmax to avoid correlation cutoff
    window = convert(Integer,floor(length(times)/2));
    # τ is just the times up to our window
    τ = times[1:window];
end;
```

```julia
if makePlots
    # our calculated g2τ
    g2τCalc = 1
    Emag2 = real.(mag .* conj(mag))
    Emag4 = Emag2 .* Emag2
    sumEmag2 = sum(Emag2)
    sumEmag4 = sum(Emag4)
    term2 = sumEmag4/(bigN*sumEmag2^2)
    g2τCalc -= term2

    term3 = sum(Emag2 .* exp.(-im*ΔM .* transpose(τ)),dims=1)/sumEmag2
    term3 = real.(term3 .* conj(term3))

    kbOverMhC2 = 9.178e-14;
    σ = sqrt(kbOverMhC2*temp)*ωM[1]
    stauAvg = transpose(bigN .+ bigN*(bigN-1)*exp.(-σ^2*τ .^2))
    term3 = term3 .* stauAvg/bigN^2

    g2τCalc =  g2τCalc .+ term3
end;
```

- Our calculated version

$$g^{(2)}(\tau) = 1 - \frac{1}{N}\frac{\sum_{m=1}^{M}|\mathcal{E}|_m^4}{\left(\sum_{m=1}^{M}|\mathcal{E}_m|^2\right)^2} + \left|\frac{\sum_{m=1}^{M}|\mathcal{E}_m|^2 e^{-i\Delta_m\tau}}{\sum_{m=1}^{M}|\mathcal{E}|_m^2}\right|^2 \frac{\langle S(\tau)\rangle_\omega}{N^2}$$

```
# if ONLY the classical plots are desired, then this calculates a single instance
of the classical fields and correlations
if makePlots && (classicalPlots && !(classicalAvgPlots || countsPlot || corrPlot))
    # generate N doppler broadened frequencies
    ω1Doppler = ωnDoppler(ωM[1],bigN,temp)
    # generate N*M random phases
    ϕ1nm = 2*π*rand(Float64,(length(ωM),bigN))
    # construct field parameter object
    testParams1 = eFieldParams(mag,ΔM,ω1Doppler,ϕ1nm)

    # calculate electric field vs time
    e1fieldt = map(t->electricField(t,testParams1),times)

    # calculate g1τ
    g1τ1Norm = correlate(e1fieldt,conj.(e1fieldt),0,window);
    g1τ1 = abs.(map(i->correlate(e1fieldt,conj.
(e1fieldt),i,window),collect(0:window-1))/g1τ1Norm)

    # calculate the intensity vs time
    intensity1t = real.( e1fieldt .* conj(e1fieldt))

    # calculate g2τ
    g2τ1Norm = mean(intensity1t)^2
    g2τ1 = map(i->autocorrelate(intensity1t,i,window),collect(0:window-1))/g2τ1Norm

end;
```

```
# the classical average plots and photon-based plots require multiple instances of the classical field calculations
if makePlots && (classicalAvgPlots || countsPlot || corrPlot)
    # calculate number of trials from the total desired photon counts and the average photon count per trial
    nTrials = convert(Integer,ceil(ntot/nbar))
    # make an array of the average photon counts per trial for array broadcasting
    nPerTrial = bigN*ones(Integer,nTrials)
    # calculate nTrials instances of doppler broadened frequencies
    ωDoppler = ωnDoppler.(ωM[1],nPerTrial,temp);
    # calculate nTrials instances of random phases
    ϕnm = map(n->2*π*rand(Float64,(length(ωM),n)),nPerTrial);
    # generate nTrials instances of field parameters
    testParams = map((x,y)->eFieldParams(mag,ΔM,x,y),ωDoppler,ϕnm);

    # calculate nTrials instances of the time dependent electric field
    efieldt = map(x->map(t->electricField(t,x),times),testParams);

    # calculate nTrials instances of g1τ
    g1τNorm = map(eft->correlate(eft,conj.(eft),0,window),efieldt);
    g1τ = map((eft,normG1τ)->abs.(map(i->correlate(eft,conj.(eft),i,window),collect(0:window-1))/normG1τ),efieldt,g1τNorm);

    # calculate nTrials instances of the time dependent intensity
    intensityt = map(eft->real.( eft .* conj(eft)),efieldt);

    # calculate nTrials instances of g2τ
    g2τNorm = map(intens->mean(intens)^2,intensityt)
    g2τ = map((intens,normG2τ)->map(i->autocorrelate(intens,i,window),collect(0:window-1))/normG2τ,intensityt,g2τNorm)

    # pick out one instance of everything for plotting
    e1fieldtM = efieldt[1]
    g1τ1M = g1τ[1]
    intensity1tM = intensityt[1]
    g2τ1M = g2τ[1]
end;
```

```
if makePlots && classicalAvgPlots
    efieldtAvg = vectorAvg(efieldt)
    g1τAvg = vectorAvg(g1τ)
    intensitytAvg = vectorAvg(intensityt)
    g2τAvg = vectorAvg(g2τ)
end;
```

Calculate photon counts

Photon counts are calculated by treating the intensity in each time bin as the average photon count rate, then sampling from a poisson distribution with that average count rate.

```julia
if makePlots && (countsPlot || corrPlot)
    γCounts = map(intens->poissonCount.(γIntensity(intens,nbar*2)),intensityt)
    γcountTimes = map(γCt->countTimes(times,γCt),γCounts)
    flatγCountTimes = vcat(γcountTimes...)
end;
```

Calculate autocorrelation of photon counts

Note that I only look at whether two bins both have counts or not when calculating the autocorrelation. I **do not** look at *how many* counts there are in each bin.

```julia
if makePlots && corrPlot
    correlationTimes = map(γCt->singleDeltaTimes(τ,γCt),γCounts)
    flatCorrelationTimes = vcat(correlationTimes...)
end;
```

Functions

```
Main.workspace2558.singleDeltaTimes
```

```julia
"""
    function singleDeltaTimes(τ::Vector,γCounts::Vector)

Returns an array of τ values for which the γCounts autocorrelation is non-zero.
"""
function singleDeltaTimes(τ::Vector,γCounts::Vector)
    return τ[map(i->autocorrelate(γCounts,i,length(τ)) > 0 ? true :
false,collect(0:length(τ)-1)) ]
end

```

```
Main.workspace2568.countTimes
```

```julia
"""
    function countDeltaTimes(τ::Vector,γCounts::Vector)

Returns an array of τ values for which the γCounts autocorrelation is non-zero.
"""
function countTimes(times::Vector,γCounts::Vector)
    out = Vector{Real}(undef,0)
    for (i,counts) in enumerate(γCounts)
        if counts != 0
            countTimes = times[i]*ones(counts)
            out = vcat(out,countTimes)
        end
    end
    return out
end
```

Main.workspace3.eFieldParams

```julia
"""
    eFieldParams64(mag::Array{S},ΔM::Array{T},ωN::Array{T},ϕ::Array{S} ) where
{T<:Real, S<:Complex}

Static parameters for the electric field
"""
struct eFieldParams
    mag::Vector
    ΔM::Vector
    ωN::Vector
    ϕ::Matrix

    function eFieldParams(mag::Vector,ΔM::Vector,ωN::Vector,ϕ::Matrix )
        @assert length(mag) == length(ΔM) "Number of magnitudes must match number
of emission lines"
        @assert (size(ϕ)[1] == length(ΔM) && size(ϕ)[2] == length(ωN)) "Must have a
unique phase for each n and m"
        new(
            convert(Vector{Complex},mag),
            convert(Vector{Real},ΔM),
            convert(Vector{Real},ωN),
            convert(Matrix{Real},ϕ)
        )

    end
end
```

Main.workspace3.electricField

```julia
"""
    function electricField(t::Real,params::eFieldParams)

Returns the electric field value at time t
"""
function electricField(t::Real,params::eFieldParams)
    # generate frequencies
    ωNM = transpose(params.ωN) .+ params.ΔM
    # add the phase
    exponentnm = -im*(t*ωNM+params.ϕ)
    # put them in the exponent
    enm = exp.(exponentnm)
    # multiply by the field magnitude
    fieldnm = params.mag .* enm
    # add it all together
    return sum(ivec(fieldnm))
end
```

Main.workspace3.ωnDoppler

```julia
"""
    function ωnDoppler(ω0::Real,N::Integer,temp::Real,seed::Integer = -1)

Generates N doppler shifted frequencies around frequency ω0 for a source at
temperature temp. Seed optional for reproducible results.
"""
function ωnDoppler(ω0::Real,N::Integer,temp::Real,seed::Integer = -1)
    rng = MersenneTwister()
    if seed != -1
        rng = MersenneTwister(seed)
    end
    kbOverMhC2 = 9.178e-14;
    σ = sqrt(kbOverMhC2*temp)*ω0
    d = Normal(ω0,σ)
    return rand(rng,d,N)
end
```

Main.workspace3.correlate

```julia
"""
    function correlate(u::Vector{T},v::Vector{T},offset::Integer,window::Integer = -1) where {T<:Number}

Calculates correlation between vectors u and v with given offset. Specify averaging window to limit range of correlation. If the window extends beyond the end of one vector, it treats out-of-bounds indices as zero.
"""
function correlate(u::Vector{T},v::Vector{T},offset::Integer,window::Integer = -1) where {T<:Number}

    @assert offset <= length(u) "Offset out of bounds"
    @assert window <= length(u) && window <= length(v) "Window must be smaller than input vector lengths"

    if window == -1
        window = length(u)
    end

    v1 = view(u,1:window)
    v2 = view(v,1+offset:min(window+offset,length(v)))
    if window+offset > length(v)
        v2 = vcat(v2,zeros(window+offset-length(v)))
    end

    return dot(v1,v2)/window
end
```

Main.workspace3.autocorrelate

```julia
"""
    function autocorrelate(u::Vector{T},offset::Integer, window::Integer = -1) where {T<:Number}

Calculates correlation of vector u with itself.
"""
function autocorrelate(u::Vector{T},offset::Integer, window::Integer = -1) where {T<:Number}
    correlate(u,u,offset,window)
end
```

Main.workspace3.γIntensity

```julia
"""
    function γIntensity(intensity::Vector,nbar::Real)

Calculates the photon count rate in each bin of an intensity histogram
"""
function γIntensity(intensity::Vector,nbar::Real)
    nintensity = intensity/sum(intensity)
    return nbar*nintensity
end
```

Main.workspace3.poissonCount

```
"""
    function poissonCount(nbar::Real)

Returns Poisson distributed counts for average count rate nbar
"""
function poissonCount(nbar::Real)
    p = exp(-nbar)
    s = p
    r = rand()
    count = 0
    while r > s
        count += 1
        p *= nbar/count
        s += p
    end
    return count
end
```

Main.workspace3.beCount

```
"""
    function beCount(nbar::Real)

Returns Bose-Einstein distributed counts for average count rate nbar
"""
function beCount(nbar::Real)
    p = 1/(nbar+1)
    fnbar = p*nbar
    f = p*nbar
    s = p
    r = rand()
    count = 0
    while r>s
        count += 1
        p *= f
        s += p
    end
    return count
end
```

vectorAvg (generic function with 1 method)

```
function vectorAvg(someVector::Vector)
    return +(someVector...)/length(someVector)
end
```

Load Prerequisites

```julia
import Pkg
```

```julia
Pkg.add("Distributions")
```

```julia
Pkg.add("DataFrames")
```

```julia
Pkg.add("CSV")
```

```julia
Pkg.add("Plots")
```

```julia
Pkg.add("IterTools")
```

```julia
Pkg.add("PlutoUI")
```

```julia
Pkg.add("StatsBase")
```

```julia
using Random, Distributions, StatsBase
```

```julia
using LinearAlgebra
```

```julia
using Plots
```

```julia
using IterTools
```

```julia
using PlutoUI
```

```julia
using DataFrames
```

```julia
using CSV
```