# Report 2 - Grangier-Roger-Aspect experiment analysis

Francesco Lorenzi, October 2020

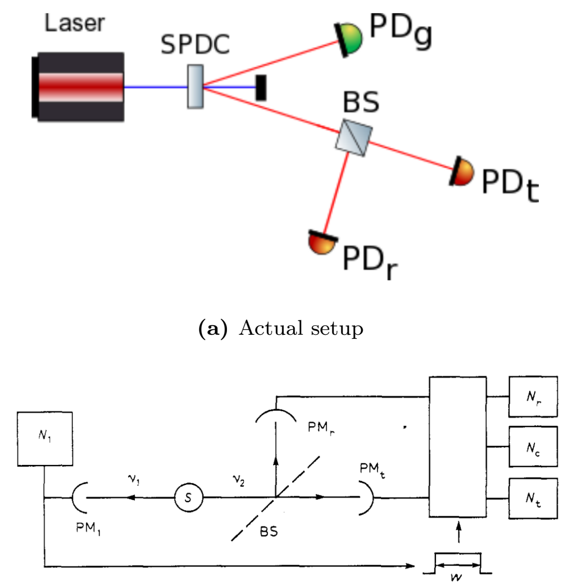
**Summary**

The purpose of this report is twofold: in a first part a statistical analysis will be carried out on a dataset collected from an experimental setup based on Grangier-Roger-Aspect experiment [1].

In a second part, an application of photon arrival statistics is also showed: using a dataset from coherent light photon detection, random integers are generated and analyzed using various visual techniques.

## 1 Photon indivisibility experiment

The experiment developed by Grangier, Roger and Aspect in 1986 consist in verifying, by using statistical methods on photomultipliers hits, that a single photon, after impinging on a beam splitter, is present in only one of the beams after, and so it is indivisible. From the theoretical point of view, this experiment confirms the quanized nature of radiation, as the classical model for photodetection, which predicts correlation between detection along the two branches, is completely contradicted by the data.




**(a)** Actual setup

**(b)** Original setup

**Figure 1:** Experimental setup scheme

### Experimental setup

Even if the description of the experiment with a single beam splitter and two detectors is straightfoward, an additional technique is needed to prevent detector noise from making the data unintelligible. So instead of a single source, a source which emits photon *in couples* is used, one is sent to a separate detector, and the other is sent to the setup described before. In that way the first photon triggers a *gate* signal that validate counts from the other detectors. Assuming a low rate emission from the source with respect to dark count rate of photodiodes, with that technique the noise is greatly reduced.

Altought in the original experiment this feature was hardwired with electronics, in our setup all events are collected regardless of their validation, and the *gate* signal is to be applied separately in post-processing.

In the setup is shown in Figure 1a: all the pulses from the photodiodes $PD_g, PD_t, PD_r$ are collected by a time-tagger on a common time scale in three different channels, respectively called *gate (G)* channel, *transmitted (T)* channel and *reflected (R)* channel.

In the experiment we will have that for every gate event which is associated with at least an event in the reflected or transmitted channel (and therefore could be considered not belonging to noise), there is a probability $p_r$ and $p_t$ to have an event in these channel. In fact, we can define two Bernoulli random variables $X_r \sim B(p_r)$ and $X_t \sim B(p_t)$ which represents the result of measurement associated with each gate event. In this sense all the data collected can be represented as a stochastic process of i.i.d. variables. For each measurement, we will call it *double coincidence* if only one of the two realizations is 1, and triple coincidence if they are both 1.

The physical problem is addressed by observing the correlation between the random variables.

$$\alpha = \frac{p_c}{p_r p_t} = \frac{\mathbb{E}[X_r X_t]}{\mathbb{E}[X_r]\,\mathbb{E}[X_t]} \tag{1}$$

If $\alpha > 0$ the variables are *correlated*, and the classical model is confirmed, indeed if $\alpha < 0$ the variables are *anti-correlated*, and the classical model is rejected in favour of the quantum model. Needless to say, we can only have an estimate of this parameter from experimental data: this is carried out as usual using sample mean estimators. If

$N_1$ is the number of valid gate events,

$$\widehat{\alpha} = \frac{\widehat{p_c}}{\widehat{p_r}\widehat{p_t}} = \frac{N_1 \sum_{i=1}^{N_1} X_r X_t}{\sum_{i=1}^{N_1} X_r \sum_{i=1}^{N_1} X_t} \tag{2}$$

## Analysis

As anticipated, before counting double and triple coincidences, a preprocessing step is necessary. First of all, we reject events on the same channel which are distanced by less than 3900 machine time units defined by $1MTU = 80.955ps$. In fact, events whose difference in time is less than $\approx 0.315\mu s$ could be *afterpulses*, artifacts of the detection electronics. After this passage the interarrival time of the three channels is plotted in Figure 2, from which we can confirm that the light used is of coherent type.

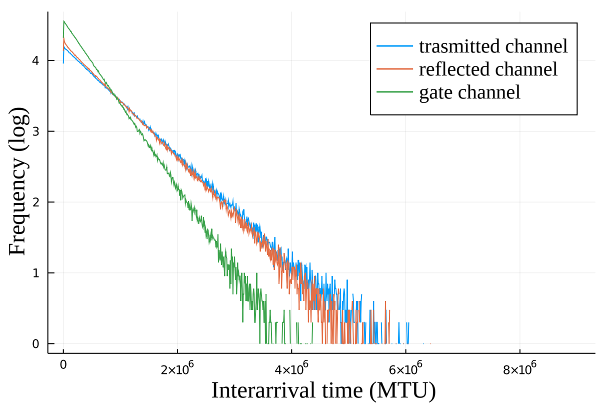
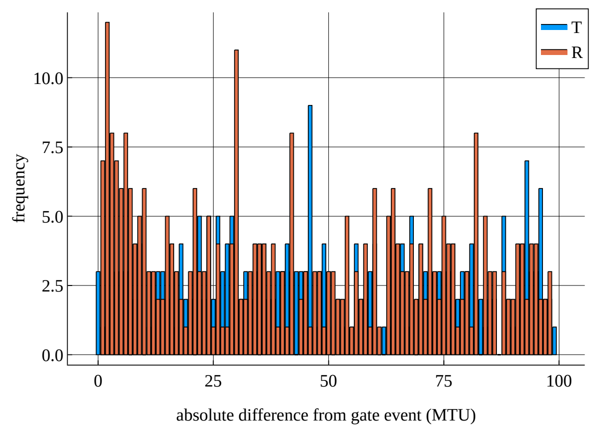



**Figure 2:** Single channel click frequencies

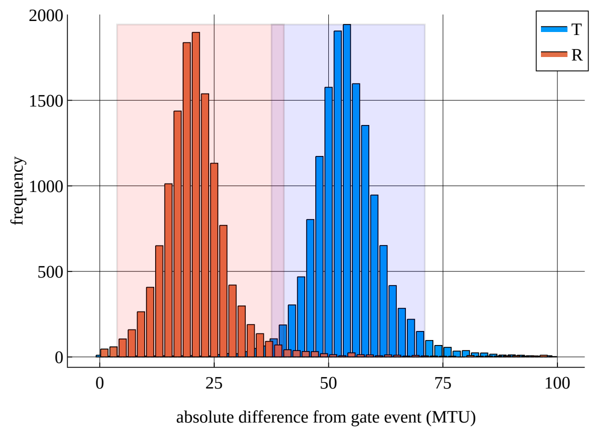
In a second step, we must filter the data with a suitable *gate* function, which will be triggered by the events on the gate channel, and will detect events from the other channels which are included in a well defined *window*.

In order to build a correct gate function, we notice that the free air path of photons in the three branches of the optical setup, as well as the difference in the coaxial cable length, can induce a differential time delay of events linked to the same photon pair. So we must observe all the occurences of reflection and transmission, before and after each gate event. By analyzing the distributions of the counts, it will be possible to obtain a mean and a variance of the delay of R and T channels.

To facilitate further the recognition of delayed events, we filter the data of the transmitted and reflected channels to be in a. In fact, if we have two photons belonging to the same pair, the differential delay of two arbitrary detections must be less that a given time. This can be said for *physical reasons*: the pulses reach the time-tagger's front end after the propagation of each photon along the optical bench, and of the signal along the RG cable. Considering the order of magnitude of the lenghts in laboratory, we deduce that the maximum time delay must be in the order of $\sim 10ns$, so we filter only events around $\pm 100MTU$ away from the nearest gate event. The outcome of this filtering is shown in Figure 3a for the interval $[-100, 0]$ (only noise) and in Figure 3b for the interval $[0, +100]$ (bell-shaped delay curves).




**(a)** T, R events before trigger




**(b)** T, R events after trigger

**Figure 3:** Analysis of inter-channel delay

In the hypotesis that the delay follow a normal distribution, the *gate* signal is tailored to be a *window* in time of the form $[(t_{Gi} + \mu) - n\sigma;\ (t_{Gi} + \mu) + n\sigma]$ where $t_{Gi}$ is the time of the $i$-th gate event, and $n$ is choosen arbitrarily.

## Interpretation of results

By filtering the events with such gate function, we are able to measure the occurences of our random variables $X_r$, $X_t$. The result is shown in Table **??**.

| | | |
|---|---|---|
| $\widehat{p_t}(\mathbf{x})$ | $\frac{13371}{27451} = 0.4871$ | |
| $\widehat{p_r}(\mathbf{x})$ | $\frac{14081}{27451} = 0.5130$ | |
| $\widehat{p_c}(\mathbf{x})$ | $\frac{1}{27451} = 3.6429 \cdot 10^{-5}$ | |
| $\widehat{\alpha}(\mathbf{x})$ | $1.458 \cdot 10^{-4}$ | |

# 2 Random Number Generation

In this report two methods of obtaining random bits, i.e. processes of independent and identically distributed Bernoulli ($p = 0.5$) random variables, are presented. They are different mainly in the possibility of their employment. The first one can operate on a continuous stream of photon arrivals, whereas the second needs to scan all the sequence of arrivals before generating the random bits. We will call the first one *Local-difference* method, and the second *Median*, for reasons that will be presented soon.

### Local-difference method

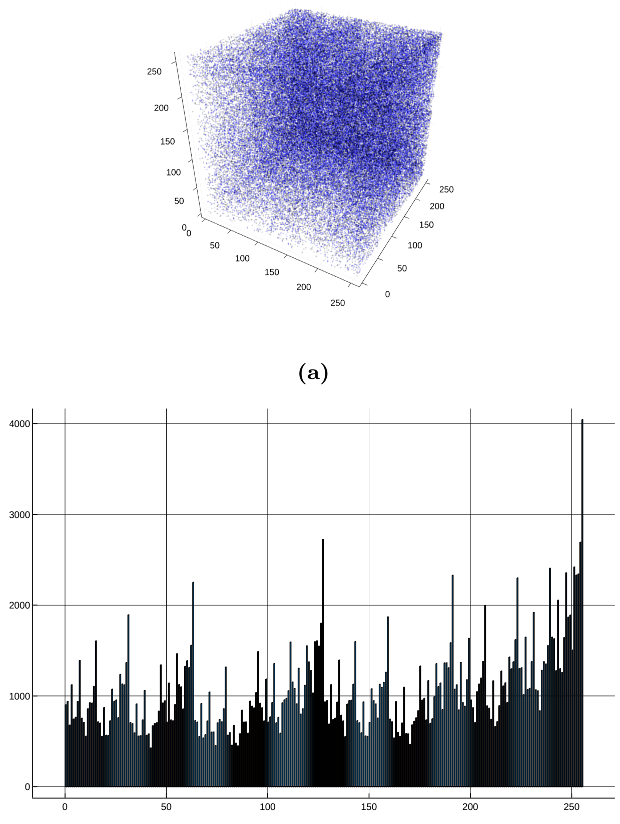



**Figure 4:** Local-difference

### Median method

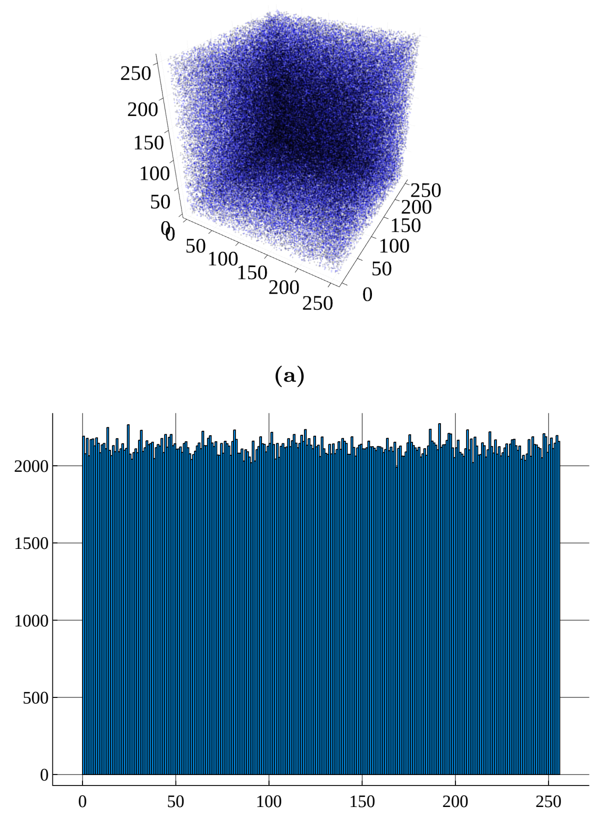



**Figure 5:** Local-difference

### A higher bit-rate alternative

If a complete statistical description of the Poisson arrival process is provided, a much higher bit-rate method is easily developed: We use the theorem of *Universality of uniform r.v.* to obtain a continuous Uniform r.v. $\sim U([0,1])$ from the Exponential. Once we have obtained that, we can divide the interval $[0,1]$ in, for example, 256 sections, and consider for each hit in that section, the corresponding 8-bit number. The efficiency in this way can be scaled of a factor of 8 or higher, keeping in mind that the time-tagger has not infinite resolution, and saves the events in a fundamentally discrete space of time tags. The obvious problem in this method is that the statistical description of the incoming process can only be estimated, so the random bits will be unreliable.

## Conclusions

## References


[1] Grangier, Roger, Aspect

### Code

```
    module Analyzer
using Plots
using Printf
import Plotly
import PGFPlots
import Statistics
import ProgressMeter

export delay_estimator, main, loader, difference_info, gated_counter, single_chan_stat

Plots.gr()
default(show = true)
# PyPlot.clf()
# println(PyPlot.backend)
const machine_time = 80.955e-12

function loader(;aft_filter = true)
    println("Loading...")
    s = "./tags.txt"
    a = readlines(s)
    for y in a
        filter(x -> !isspace(x), y)
    end
    i=0
    b = Array{Int, 2}(undef, 2, length(a))

    b[1, :] = [parse(Int, split(x, ";")[1]) for x in a]
    b[2, :] = [parse(Int, split(x, ";")[2]) for x in a]


    tags = Array{Int, 2}(undef, 3, length(b))
    fill!(tags, 0)
    println(typeof(tags))
    k = Array{Int, 1}(undef, 3) # k[i] will be the total count of trigger events on channel i
    fill!(k, 1)
    i=0
    cnt = 0
    aft = Array{Int, 1}(undef, 3)
    fill!(aft, 0)
    if (aft_filter)
        aft_const = 3900
    else
        aft_const = 0
    end
    for i = 1:length(a)
         if (i<8 || tags[ b[2, i]-1, k[b[2, i]-1] - 1 ] + aft_const < b[1, i] )
            tags[ b[2, i]-1, k[b[2, i]-1] ] = b[1, i]
            k[b[2, i] - 1] += 1
        else
            # println("Afterpulse on CH-", b[2, i] - 1)
            aft[b[2, i] - 1] +=1
        end
    end

    println("Number of valid hits")
    @printf("\t n. of transmitted hits    : %6d \n", k[1])
    @printf("\t n. of reflected hits      : %6d \n", k[2])
    @printf("\t n. of gate hits           : %6d \n", k[3])
    println("T+R = ", k[1]+k[2], ", G = ", k[3])
    println("Number of afterpulses:")
    @printf("\t chan 1 - transmitted (2) : %6d \n", aft[1])
    @printf("\t chan 2 - reflected (3)   : %6d \n", aft[2])
    @printf("\t chan 3 - gate (4)        : %6d \n", aft[3])
    println("Percentage of afterpulses")
    @printf("\t chan 1 - transmitted (2) : %4.1f %% \n", aft[1]/k[1] * 100)
    @printf("\t chan 2 - reflected (3)   : %4.1f %% \n", aft[2]/k[2] * 100)
    @printf("\t chan 3 - gate (4)        : %4.1f %% \n", aft[3]/k[3] * 100)
    return (tags, k);
end

function delay_estimator((tags, k); mode = "gate_first")
```

```julia
    println("Analyzing...")
    machine_time = 80.955e-12
    diff1 = Array{Int, 1}(undef, k[1])
    diff2 = Array{Int, 1}(undef, k[2])
    fill!(diff1, 0)
    fill!(diff2, 0)
    if mode == "gate_last"
        g1 = -1          # BE CAREFUL : NOT A REAL GATE EVENT
        g2 = tags[3, 1]
        n = 1
        # Retarded gate method - positive diff
        for i = 2:k[3]
            while (tags[1, n]<g2 && n<k[1])
                diff1[n] = g2 - tags[1, n]
                n += 1
            end
            g2 = tags[3, i]
        end

        g1 = -1          # BE CAREFUL : NOT A REAL GATE EVENT
        g2 = tags[3, 1]
        n = 1
        for i = 2:k[3]
            while (tags[2, n]<g2 && n<k[2])
                diff2[n] = g2 - tags[2, n]
                n += 1
            end
            g2 = tags[3, i]
        end
    elseif mode == "gate_first"
        # Anticipated gate method - positive diff
        g1 = -1          # BE CAREFUL : NOT A REAL GATE EVENT
        g2 = tags[3, 1]
        n = 8
        for i = 2:k[3]
            while (tags[1, n]<g2 && n<k[1])
                diff1[n] = tags[1, n] - g1
                n += 1
            end
            g1 = g2
            g2 = tags[3, i]
        end
        diff1 = diff1[8:length(diff1)]

        g1 = -1          # BE CAREFUL : NOT A REAL GATE EVENT
        g2 = tags[3, 1]
        n = 8
        for i = 2:k[3]
            while (tags[2, n]<g2 && n<k[1])
                diff2[n] = tags[2, n] - g1
                n += 1
            end
            g1 = g2
            g2 = tags[3, i]
        end
        diff2 = diff2[8:length(diff2)]
    else
        # Minimum distance method
        g1 = -100000000         # BE CAREFUL : NOT A REAL GATE EVENT
        g2 = tags[3, 1]
        n = 1
        for i = 2:k[3]
            while (tags[1, n]<g2 && n<k[1])
                if ((tags[1, n] - g1) <  (g2 - tags[1, n]))
                    diff1[n] = tags[1, n] - g1
                else
                    diff1[n] = tags[1, n] - g2
                end
                n += 1
            end
            g1 = g2
            g2 = tags[3, i]
        end
        g1 = -100000000         # BE CAREFUL : NOT A REAL GATE EVENT
        g2 = tags[3, 1]
```

```
            n = 1
            for i = 2:k[3]
                while (tags[2, n]<g2 && n<k[1])
                    if ((tags[2, n] - g1) <  (g2 - tags[2, n]))
                        diff2[n] = tags[2, n] - g1
                    else
                        diff2[n] = tags[2, n] - g2
                    end
                    n += 1
                end
                g1 = g2
                g2 = tags[3, i]
            end
        end

        # max_delay = 7.5 # [ns]
        # max_clicks = max_delay * 1e-9/machine_time
        max_clicks = 80
        max_delay = max_clicks * machine_time / 1e-9
        @printf("PRE-filtering at max delay = %d ns \n ", max_delay)
        # unreal difference filter
        filter!(x-> (x< max_clicks), diff1)
        filter!(x-> (x< max_clicks), diff2)

        filter!(x -> (x>0), diff2)

        difference_info(diff1, diff2, k)
        Îij1 = Statistics.mean(diff1)
        Îij2 = Statistics.mean(diff2)
        ÏČ1 = sqrt(Statistics.var(diff1 .- Îij1))
        ÏČ2 = sqrt(Statistics.var(diff2 .- Îij2))

        return [Îij1, ÏČ1, Îij2, ÏČ2]
    end

    function single_chan_stat((tags, k); chan = 3)
        machine_time = 80.955e-12
        series = tags[chan, :]
        diff = Array{Int, 1}(undef, length(series)-1)
        for i = 1:length(series)-1
            diff[i] = series[i+1] - series[i]
        end
        filter!(z -> (z>0), diff)
        max_diff = maximum(diff)
        println("min: ", minimum(diff))
        bin_num = 1000

        bin_step =Int(ceil(max_diff / bin_num))+1
        println("max diff : ", max_diff, " bin step", bin_step)
        hist = Array{Int, 1}(undef, bin_num)
        fill!(hist, 0)
        i = 1
        for i = 1:length(diff)
            hist[Int(floor((diff[i]) / bin_step)) + 1] += 1
        end
        @printf("minimum difference between gate event: %10d clicks -> %5.2f ns \n",minimum(diff) , minimum(diff)*machine_time/1e-9
        )

        prob = hist / sum(hist)
        accum = 0
        i = 1
        for i = 1:bin_num
            accum += (i-1) * prob[i]
        end
        mu = accum
        var = 0
        sk_acc = 0
        kr_acc = 0
        for i = 1:bin_num
            var += (i-1 - mu)^2 * prob[i]
            sk_acc += (i-1 - mu)^3 * prob[i]
            kr_acc += (i-1 - mu)^4 * prob[i]
        end
        sigma = sqrt(var)
```

```
    sk = sk_acc
    kr = kr_acc
    theo_mom = poisson_moments(mu)
    @printf("Statistical analysis of gate events process:\n")
    @printf("\t Îť mean                  : %5.3f \n", mu  - theo_mom[1])
    @printf("\t Îť variance              : %5.3f \n", var - theo_mom[2])
    @printf("\t Îť skewness non std      : %5.3f \n", sk  - theo_mom[3])
    @printf("\t Îť kurtosis non std      : %5.3f \n", kr  - theo_mom[4])

    fig = Plots.plot((1:bin_num)*bin_step,
                     [log10(h) for h in hist],
                     show=true,
                     xlabel = "absolute difference between gate events (clicks)",
                     size = (1200, 800))
    savefig(fig, string("./images/", chan, "-single_chan.pdf"))
end

function poisson_moments(mu)
    return [mu, mu, 1/sqrt(mu), 1/mu]
end

function bose_ein_moments(mu)
    sigma = sqrt(mu + mu^2)
    return [mu,
            sigma^2,
            (mu + 3*mu^2 + 2*mu^3)/sigma^3,
            (mu + 10*mu^2 + 18*mu^3 + 9*mu^4)/sigma^4]
end

function difference_info(diff1, diff2, k)
    machine_time = 80.955e-12
    println("Difference Info...")
    max_diff1 = maximum(diff1)
    min_diff1 = minimum(diff1)
    max_diff2 = maximum(diff2)
    min_diff2 = minimum(diff2)
    @printf("1) maximum difference      : %10d \n", max_diff1)
    @printf("1) minimum difference      : %10d \n", min_diff1)
    @printf("1) maximum time difference  (ns)  : %10.4f \n", max_diff1 * machine_time * 1e9)
    @printf("1) minimum time difference  (ns)  : %10.4f \n", min_diff1 *machine_time * 1e9)

    @printf("2) maximum difference      : %10d \n", max_diff2)
    @printf("2) minimum difference      : %10d \n", min_diff2)
    @printf("2) maximum time difference  (ns)  : %10.4f \n", max_diff2*machine_time * 1e9)
    @printf("2) minimum time difference  (ns)  : %10.4f \n\n", min_diff2*machine_time * 1e9)

    @printf("1) Fraction of accepted hits : %d / %d = %4.2f \n", length(diff1), k[1], length(diff1)/k[1])
    @printf("2) Fraction of accepted hits : %d / %d = %4.2f\n", length(diff2), k[2], length(diff2)/k[2])

    # Want to show exactly 100 bins in histogram
    mod = Int(ceil(maximum([length(diff1), length(diff2)]) / 1e4)) # TO BE MODIFIED

    # plot clicks
    x_delays1 = (min_diff1:mod:max_diff1)
    x_delays2 = (min_diff2:mod:max_diff2)

    bin_num1 = Int(floor((max_diff1-min_diff1) / mod)) + 1
    println("bins 1: ", bin_num1)
    bias1 = Int(floor(-min_diff1/mod))
    hist1 = Array{Int, 1}(undef, bin_num1)
    fill!(hist1, 0)
    i = 1
    while (i<=length(diff1))
        hist1[Int(floor((diff1[i] - min_diff1) / mod))+1] += 1
        i += 1
    end

```

```
    bin_num2 = Int(floor((max_diff2-min_diff2) / mod)) + 1
    bias2 = Int(floor(-min_diff2/mod))
    println("bins 2: ", bin_num2)
    hist2 = Array{Int, 1}(undef, bin_num2)
    fill!(hist2, 0)
    i = 1
    while (i<=length(diff2))
        hist2[Int(floor((diff2[i] - min_diff2) / mod))+1] += 1
        i += 1
    end
    Îij1 = Statistics.mean(diff1)
    Îij2 = Statistics.mean(diff2)
    ÏČ1 = sqrt(Statistics.var(diff1 .- Îij1))
    ÏČ2 = sqrt(Statistics.var(diff2 .- Îij2))

    if (length(hist1)<600 && length(hist2)<600)
        println("Plotting...")
        # fig = Plotly.figure()
        n_ÏČ = 2
        fig = Plots.bar(x_delays1,
                        hist1,
                        show=true,
                        title = string("Event delay and Âś", n_ÏČ, "ÏČ decision region"),
                        xlabel = "absolute difference from gate event (clicks)",
                        ylabel = "Frequency",
                        label = "transmitted photon delay",
                        size = (1000, 600))
        Plots.bar!(x_delays2, hist2, label = "reflected photon delay")
        rectangle(w, h, x, y) = Plots.Shape(x .+ [0,w,w,0], y .+ [0,0,h,h])

        recr = rectangle(2*n_ÏČ*ÏČ1, maximum([maximum(hist1), maximum(hist2)]), Îij1-n_ÏČ*ÏČ1, 0)
        rect = rectangle(2*n_ÏČ*ÏČ2, maximum([maximum(hist1), maximum(hist2)]), Îij2-n_ÏČ*ÏČ2, 0)
        Plots.plot!(recr, linewidth = 2, opacity = 0.1, color=:blue, label="transmitted decision region")
        Plots.plot!(rect, linewidth = 2, opacity = 0.1, color=:red, label="reflected decision region")

        display(fig)
        savefig("./images/delays.pdf")
    else
        println("Too long to plot...")
    end
end

# need to decide what method to use -> we use  GATE -> REFLECTED -> TRANSMITTED
function gated_counter((tags, k), params; mode = "full-width")
    println("Gated counting...")
    Îij1 = params[1]
    ÏČ1 = params[2]
    Îij2 = params[3]
    ÏČ2 = params[4]

    @printf("mean reflected   : %6.4f \n", params[1])
    @printf("stdd reflected   : %6.4f \n", params[2])
    @printf("mean tramsmitted : %6.4f \n", params[3])
    @printf("stdd tramsmitted : %6.4f \n", params[4])
    N_1 = length(tags[3, :])
    intervals = [6]
    # Gate function (not counting with multiple hits)
    for n_ÏČ in intervals
        x = 1
        r_hit = false
        refl = 0
        multiple_refl = 0
        y = 1
        t_hit = false
        tran = 0
        multiple_tran = 0
        coincidences = 0
```

```
        if (mode == "confidence")
            for i=1:length(tags[3, :])-1
                r_hit = false
                t_hit = false
                while  tags[1, x] < -n_ÏČ*ÏČ1 + tags[3, i] + Îij1
                    x += 1
                end
                while -n_ÏČ*ÏČ1 + tags[3, i] + Îij1 <= tags[1, x] < +n_ÏČ*ÏČ1 + tags[3, i] + Îij1 && tags[1, x] < tags[3, i+1]
                    r_hit = true
                    x += 1
                end
                if r_hit
                    refl += 1
                end

                while  tags[2, y] < -n_ÏČ*ÏČ2 + tags[3, i] + Îij2
                    y += 1
                end
                while -n_ÏČ*ÏČ2 + tags[3, i] + Îij2 <= tags[2, y] < +n_ÏČ*ÏČ2 + tags[3, i] + Îij2  && tags[2, y] < tags[3, i+1]
                    t_hit = true
                    y += 1
                end
                if t_hit
                    tran += 1
                end
                if r_hit && t_hit
                    coincidences += 1
                end
            end
        else
            for i=1:length(tags[3, :])-1
                r_hit = false
                t_hit = false
                while tags[1, x] < tags[3, i]
                    x += 1
                end
                while tags[3, i] < tags[1, x] <= tags[3, i+1]
                    r_hit = true
                    x += 1
                end
                if r_hit
                    refl += 1
                end

                while tags[2, y] < tags[3, i]
                    y += 1
                end
                while tags[3, i] < tags[2, y] <= tags[3, i+1]
                    t_hit = true
                    y += 1
                end
                if t_hit
                    tran += 1
                end
                if r_hit && t_hit
                    coincidences += 1
                end
            end
        end
        @printf("Measurement with ciao confidence \n")
        prob_refl = refl / N_1
        prob_tran = tran / N_1
        prob_triple = coincidences / N_1
        Îś = prob_triple/ (prob_refl * prob_tran)
        @printf("\t gate          hits :  %9d \n", N_1)
        @printf("\t reflected     hits :  %9d \n", refl)
        @printf("\t transmitted   hits :  %9d \n", tran)
        @printf("\t coincidences  hits :  %9d \n", coincidences)
        @printf(" ----------------------\n")
        @printf("\t P[double]          : %9.8f \n", prob_refl + prob_tran)
        @printf("\t P[triple]          : %9.8f \n", prob_triple)
        @printf("\t Alpha              : %9.8f \n", Îś)
    end
end
```

```

function main()
    println("Nothing to do...")
end
end
```

Random number section

```
function random(tags; mode="naif")
data = diff(tags[1])
data_diff = diff(data)
println("Generating with ", mode, " rule...")
Îij =  Statistics.mean(data)
Îż = 1/Îij
median = Statistics.median(data)
ÏČ = sqrt(Statistics.var(data))
byte_stream = Array{UInt8}(undef, Int(floor(length(data)/8))-1)
if mode == "naif"
    stream = BitArray(undef, length(data))
    for i = 1:length(data)
        if data[i]<median
            stream[i] = 1
        else
            stream[i] = 0
        end
    end
    byte_stream = Array{UInt8}(undef, Int(floor(length(stream)/8)))
    for i = 1:length(byte_stream)-1
        byte_stream[i] =
            1*stream[8*i]+2*stream[8*i+1]+4*stream[8*i+2]+8*stream[8*i+3]+
                 16*stream[8*i+4]+32*stream[8*i+5]+64*stream[8*i+6]+128*
    end
elseif mode == "high-rate"
    uniform_events = Array{Float64}(undef, length(data))
    for i = 1:length(data)
        uniform_events[i] = exp(-data[i] * Îż) - 1
    end
    # fig = Plots.histogram(uniform_events, bin= 10000)
    # display(fig)
    byte_stream = Array{UInt8}(undef, length(data))
    for i = 1:length(byte_stream)
        byte_stream[i] = -Int(floor(255 * uniform_events[i]))
    end
elseif mode == "diff"
    stream = BitArray(undef, length(data_diff))
    for i =1:length(data_diff)-1
        if data_diff[i]<data_diff[i+1]
            stream[i] = 1
        else
            stream[i] = 0
        end
    end
    byte_stream = Array{UInt8}(undef, Int(ceil(length(stream)/8)))
    for i = 1:length(byte_stream)
        byte_stream[i] =
            1*stream[8*i]+2*stream[8*i+1]+4*stream[8*i+2]+8*stream[8*i+3]+
                 16*stream[8*i+4]+32*stream[8*i+5]+64*stream[8*i+6]+128*
    end
elseif mode == "diff-2"
    stream = BitArray(undef, Int(ceil(length(data))/2))
    fill!(stream, 0)
    k = 1
    for i =1:2:length(data_diff)-2
        if data_diff[i]<data_diff[i+1]
            stream[k] = 1
        else
            stream[k] = 0
        end
        k+=1
    end
    byte_stream = Array{UInt8}(undef, Int(ceil(length(stream)/8)))
    fill!(byte_stream, 0)
```

```
            for i = 1:length(byte_stream)-8
                byte_stream[i] = 1*stream[8*i]+2*stream[8*i+1]+4*stream[8*i+2]+8*stream[8*i+3]+
                                  16*stream[8*i+4]+32*stream[8*i+5]+64*stream[8*i+6]+128*
            end
        end
        println("Byte stream generated")
        println("mean: ", Statistics.mean(byte_stream))

        rnd_tester(byte_stream)
        return byte_stream
    end

    function rnd_tester(byte_stream)
        samples = length(byte_stream)-9
        while samples % 3 != 0
            samples -= 1
        end
        println("Analyzing ", samples, " UInt8 numbers...")
        # 3D Scatter
        fig1 = Plots.scatter3d(byte_stream[1:3:samples], byte_stream[2:3:samples], byte_stream[3:3:samples],
                               markercolor = :blue,
                               markershape = :cross,
                               markersize  = 0.5,
                               opacity = 0.1,
                               label= nothing,
                               tickfont=("times", 12),
                               size = (1024, 768)
                               )
        # Plots.xaxis!(lims=(0, 255))
        # Plots.yaxis!(lims=(0, 255))
        # Plots.zaxis!(lims=(0, 255))
        # Histogram
        fig2 = Plots.histogram(byte_stream, bins = 256, label= nothing)
        # Random walk
        x = Array{Float64}(undef, samples)
        fill!(x, 0)
        Îij = Statistics.mean(byte_stream)
        println("Statistical mean of byte_stream: ", Îij)
        for i=2:samples
            x[i] = x[i-1] + byte_stream[i] - Îij
        end
        fig3 = Plots.plot(1:samples, x, label= nothing, size = (1024, 768))

        display(fig1)
        savefig(fig1, "./random_img/scatter3d.pdf")
        display(fig2)
        savefig(fig2, "./random_img/histogram.pdf")
        display(fig3)
        savefig(fig3, "./random_img/random_walk.pdf")
    end
```