

# 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 shown: 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 P. Grangier, G. Roger and A. Aspect in 1986 consist in verifying, by using statistical methods on photomultiplier 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. The statistical outcome is expressed in terms of *correlation* between events occurrences in the transmitted and reflected channels: the result highlight how a very strong *anticorrelation* between events is found.

From the theoretical point of view, this experiment confirms the quantized nature of radiation, as the classical model for photodetection, which predicts correlation between detection along the two branches, is completely contradicted by the data.

## Experimental and statistical setup

Even if the description of the experiment with a single beam splitter and two detectors is straightforward, 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.

Althought 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.

The optical bench 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.

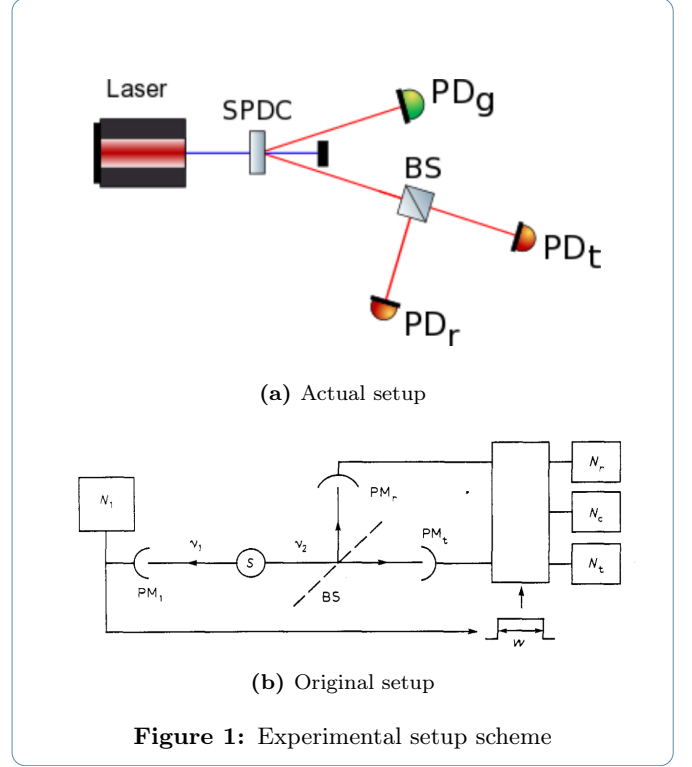


Figure 1: Experimental setup scheme

For every gate event which is associated with at least an event in the reflected or transmitted channel (and therefore can be considered as an event not belonging to noise of the gate detector), there is a constant 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. Each measurement will be called a *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, using the following definition of correlation coefficient:

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

If  $\alpha > 1$  the variables are *correlated*, and the classical model is confirmed, indeed if  $\alpha < 1$  the variables are *anticorrelated*, and the classical model is rejected in favour of the quantum model. This follows naturally from the concept of a photon indivisibility: if the photon can't be splitted we expect to never find triple coincidences. Needless to say, we can only have an estimate of this parameter from experimental data: this is carried out as usual

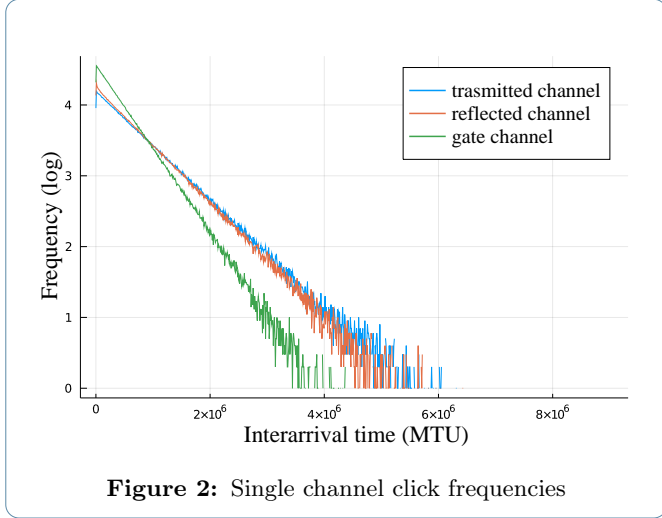
extracting a sample mean estimate using the following estimator: if  $N_1$  is the number of valid gate events,

$$\hat{\alpha} = \frac{\hat{p}_c}{\hat{p}_r \hat{p}_t} = \frac{N_1 \sum_{i=1}^{N_1} X_r X_t}{\sum_{i=1}^{N_1} X_r \sum_{i=1}^{N_1} X_t}.$$

Further comments on this estimator will be presented in the interpretation of result section.

## Analysis

As anticipated, before counting double and triple coincidences, a preprocessing step is necessary to remove noise. First of all, we reject events on the same channel which are distanced by less than 3900 machine time units, or  $MTU$  (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 devices. 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, as it follows a Poisson arrival statistics.



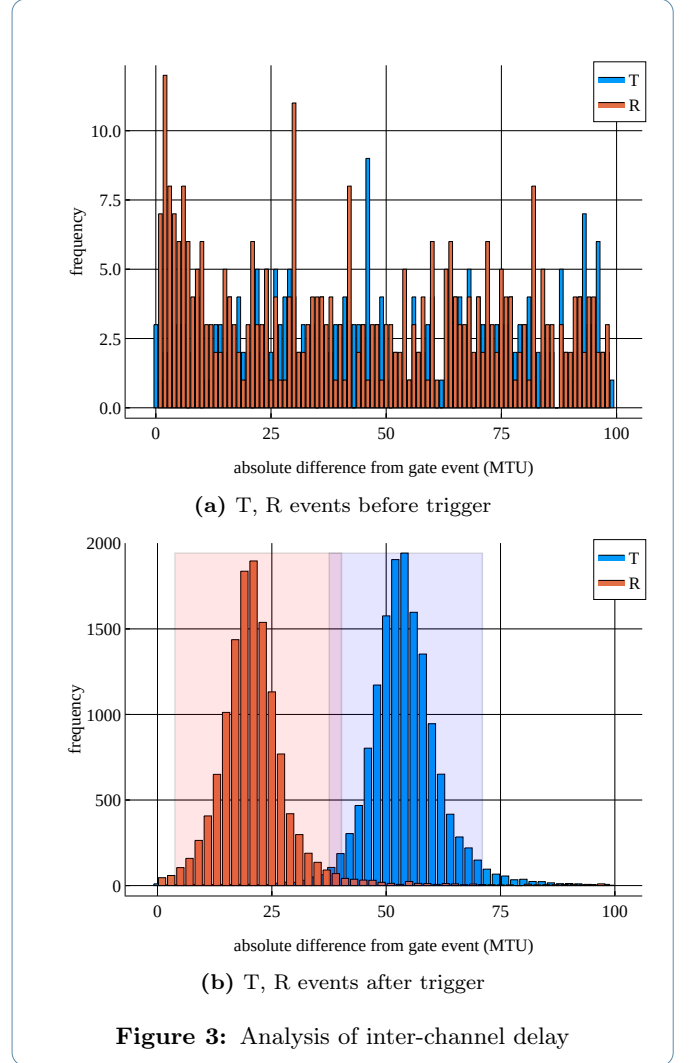
**Figure 2:** Single channel click frequencies

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 window, we notice that the free air path of photons in the three branches of the optical setup, as well as the difference in the length of the coaxial cables connecting the detector to the time-tagger, can induce a differential time delay of events linked to the same photon pair. So we shall not limit ourselves in looking of equality of time tags as coincidences, as the reflection and/or transmission events related may occur a little time before or after each gate event. By analyzing the total distribution of every channel counts, it will be possible to deduce an estimate of the differential delay of the T and R channels with respect to the nearest gate event.

To facilitate further the recognition of delayed events, we filter the data of the transmitted and reflected channels to be in a strict interval near each gate event. In general, if we have two photons belonging to the same pair, the

differential delay of two detections, on an arbitrary couple of detectors, must be less than 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 lengths in laboratory, we deduce that the maximum time delay must be in the order of  $\sim 10ns$ , so we select only events around  $\pm 100MTU$  away from the nearest gate event. The outcome of this filtering is shown in Figure 3a for the interval  $[-100, 0]$ , which show only noise, and in Figure 3b for the interval  $[0, +100]$  in which bell-shaped delay curves indicates the real delayed occurrences.



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

So we deduce that, in mean, the significant events in the channel R follows the gate event by  $\approx 22MTU$ , and the ones in the T channel follow the gate event by  $\approx 54MTU$ . In conclusion, using the hypothesis that the differential delay follows approximatively a normal distribution (which can be justified using the Central Limit Theorem), the standard deviations for each bell can be computed using the sample variance, and therefore the *gate* signal is tailored to be a *window* centered in the mean, and of width equal to a desired number of. That windows are shown in Figure 3b, and are of the form  $[(t_{Gi} + \mu) - 2\sigma; (t_{Gi} + \mu) + 2\sigma]$  where  $t_{Gi}$  is the time of the  $i$ -th gate event. Using the  $\pm 2\sigma$  interval, we expect to include in that way the 95%

of events. This construction of the gate window considers all the event belonging to the external of the windows as *noise*.

## Interpretation of results and conclusions

By filtering the events with the gate function described in the previous paragraph, we are able to measure the realization frequencies associated to the random variables  $X_r$ ,  $X_t$ . The results are shown in Table 1, along with estimated errors. In order to compute the errors, the sample variance is used to deduce the standard deviation  $\hat{\sigma}_p$ . For a Bernoulli random variable of estimated probability  $\hat{p}$  it is calculated as

$$\hat{\sigma}_p = \sqrt{\frac{\hat{p}(1-\hat{p})}{N_1 - 1}}$$

using the Bessel correction at the denominator.

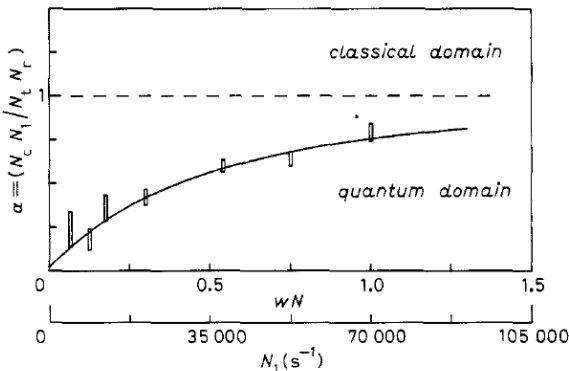
As for the error in  $\hat{\alpha}$ , a similar estimate is not readily applicable, as there is no trivial statistical description of this estimator. However, a basic error propagation estimate, using Taylor series, is carried out using the standard deviations of the probabilities.

$\hat{p}_t(\mathbf{x})$	$0.5130 \pm 0.003017$
$\hat{p}_r(\mathbf{x})$	$0.4871 \pm 0.003017$
$\hat{p}_c(\mathbf{x})$	$(3.6429 \pm 3.6429) \cdot 10^{-5}$
$\hat{\alpha}(\mathbf{x})$	$(1.458 \pm 1.476) \cdot 10^{-4}$

**Table 1:** Results

It may seem a problem that the estimated errors on  $\hat{p}_c$  and  $\hat{\alpha}$  are so large to be comparable to the actual magnitudes, but this is due to the poor statistics of the sample of triple coincidences: in fact, only 1 over 27451 gate events leads to a triple coincidence, and this greatly compromises the accuracy of the probability estimations.

Nonetheless this great uncertainty *does not invalidate the physical conclusions*: this value of  $\alpha$  shows a near-perfect anticorrelation, and is in complete contradiction with respect to the classical theory.



**Figure 4:** Original experiment results

In addition, the result on  $\alpha$  and its error seem to be compatible with the result obtained by Grangier et. al. shown in Figure 4. In the case of collected data, after the filtering, the average gate trigger rate is  $\approx 1$ , so it is extremely low with respect to the order of magnitude of the one of the original experiment. So the aim of the experiment is met: this experiment validate the indivisibility of the photons, and, in spite of the classical theory of EM fields, it shows a peculiar quantistic feature of reality.

## 2 Random Number Generation

In this report two methods of obtaining streams of random bits (i.e. processes of independent and identically distributed Bernoulli ( $p = 0.5$ ) random variables) are presented. Starting from a coherent source of light, the photon arrival times are collected by a time-tagger, so the difference between the arrivals tags represents a realization of a process of exponential random variables. The two methods differ mainly in the fact that their algorithms needs different resources: the first one can operate directly on a continuous stream of photon arrivals, whereas the second needs to scan all the sequence of arrivals before generating, in a second scan, the random bits. We will call the first one *Local difference* method, and the second *Median*, for reasons that will be clear soon.

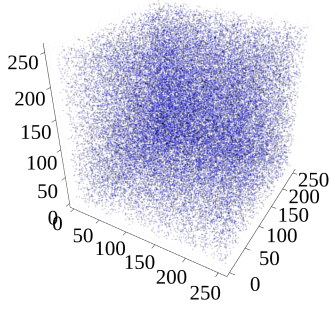
### Local difference method

This algorithm can operate with constant memory over a stream of subsequent time tags, and works as follows:

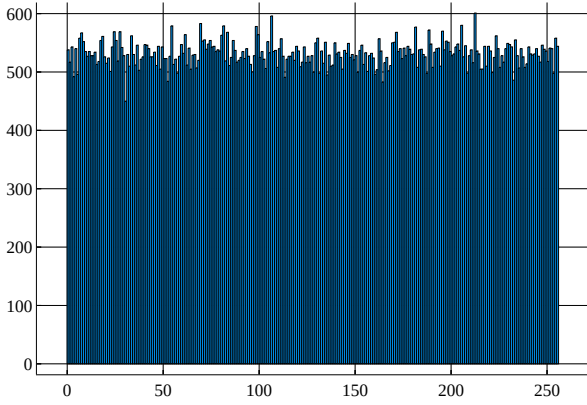
1. generate from the time tags the vector of the differences between each one with the previous one (skip the first tag),
2. select 4 subsequent differences and group them by pairs  $[(1, 2); (3, 4)]$ , and compute the difference between the second element of the pair and the first one: a sequence  $(\Delta_1, \Delta_2)$  of 2 values is obtained,
3. compare the values: if  $\Delta_1 \geq \Delta_2$  set the bit to 1, otherwise set the bit to 0,
4. iterate 2. to 3. over all the other subsequent group of 8 time tags, and save the deduced bits.

Using the vector of 4353849 time tags belonging to the Arecchi wheel experiment, we generate  $\lfloor 4353849 / (4 \cdot 8) \rfloor = 136057$  bytes which represents numbers from 0 to 255.

Using them as triplets of coordinates it is possible to build a 3D scatter plot to show their uniform distribution among space. In Figure 5 this representation is shown as well as the histogram of frequencies of numbers.

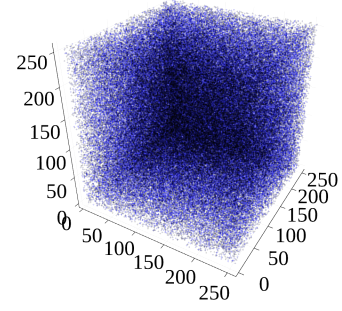


(a) Scatter 3D

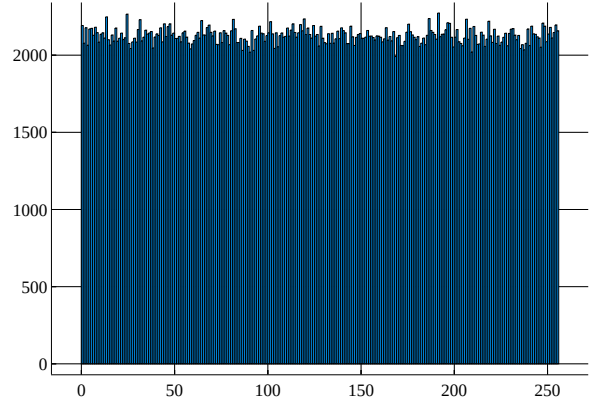


(b) Histogram

**Figure 5:** Local difference method



(a) Scatter 3D



(b) Histogram

**Figure 6:** Median method

## Median method

This algorithm can operate with a double examination of the stream of the time tags, and works as follows:

1. generate from the time tags the vector of the differences between each one with the previous one (skip the first tag),
2. compute the sample median  $\tilde{d}$  of the differences vector (or alternatively, assuming an exponential distribution, compute the sample mean and obtain the median multiplying it by  $\ln(2)$ ),
3. for each differences sample  $d_i$ , set a bit to 1 if  $d_i \geq \tilde{d}$ , otherwise set it to 0.

This algorithm provides a generation rate of 4 times the Local difference method, at the expense of a greater computing effort, as can be noticed by the plots in Figure 6, where there are  $\lfloor 4353849/8 \rfloor = 544231$  bytes.

## An even higher bit-rate proposal

If a complete statistical description of the Poisson arrival process is provided (or estimated observing the dataset), 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.

## References

- [1] P.Grangier, G.Roger, A.Aspect - *Experimental evidence for a photon anticorrelation effects on a beam splitter: a new light on single-photon interferences* (Europhys. Lett., 1 (4), pp. 173-179, 1986)
- [2] R.V.Hogg, J.W.McKean, A.T.Craig, *Introduction to mathematical statistics* (Pearson, 2019)

## Code

The code is written in Julia Programming Language.

### 1. Analysis of photon indivisibility experiment

```
1 module Analyzer
2 using Plots
3 using Printf
4 import Plotly
5 import PGFPlots
6 import Statistics
7 import ProgressMeter
8
9 export delay_estimator, loader, difference_info, gated_counter, ↵
   ↵ single_chan_stat, config
10
11 default(show = true)
12 const machine_time = 80.955e-12
13
14 function loader(;aft_filter = true)
15     println("Loading...")
16     s = "./tags.txt"
17     a = readlines(s)
18     for y in a
19         filter(x -> !isspace(x), y)
20     end
21     i=0
22     b = Array{Int, 2}(undef, 2, length(a))
23
24     b[1, :] = [parse(Int, split(x, ";")[1]) for x in a]
25     b[2, :] = [parse(Int, split(x, ";")[2]) for x in a]
26
27
28     tags = Array{Int, 2}(undef, 3, length(b))
29     fill!(tags, 0)
30     println(typeof(tags))
31     k = Array{Int, 1}(undef, 3) # k[i] will be the total count of trigger ↵
       ↵ events on channel i
32     fill!(k, 1)
33     i=0
34     cnt = 0
35     aft = Array{Int, 1}(undef, 3)
36     fill!(aft, 0)
37     if (aft_filter)
38         aft_const = 3900
39     else
40         aft_const = 0
41     end
42     for i = 1:length(a)
43         if (i<8 || tags[ b[2, i]-1, k[b[2, i]-1] - 1 ] + aft_const < b[1, i] )
44             tags[ b[2, i]-1, k[b[2, i]-1] ] = b[1, i]
45             k[b[2, i] - 1] += 1
46         else
47             aft[b[2, i] - 1] +=1
48         end
49     end
50
51     println("Number of valid hits")
52     @printf("\t n. of transmitted hits      : %6d \n", k[1])
53     @printf("\t n. of reflected hits       : %6d \n", k[2])
54     @printf("\t n. of gate hits                  : %6d \n", k[3])
55     println("T+R = ", k[1]+k[2], ", G = ", k[3])
56     println("Number of afterpulses:")
57     @printf("\t chan 1 - transmitted (2) : %6d \n", aft[1])
58     @printf("\t chan 2 - reflected (3)    : %6d \n", aft[2])
59     @printf("\t chan 3 - gate (4)           : %6d \n", aft[3])
60     println("Percentage of afterpulses")
61     @printf("\t chan 1 - transmitted (2) : %4.1f %% \n", aft[1]/k[1] * 100)
62     @printf("\t chan 2 - reflected (3)    : %4.1f %% \n", aft[2]/k[2] * 100)
63     @printf("\t chan 3 - gate (4)         : %4.1f %% \n", aft[3]/k[3] * 100)
64     return (tags, k);
65 end
66
67 function delay_estimator((tags, k); mode = "gate_first")
68     println("Analyzing...")
```

```

69 machine_time = 80.955e-12
70 diff1 = Array{Int, 1}(undef, k[1])
71 diff2 = Array{Int, 1}(undef, k[2])
72 fill!(diff1, 0)
73 fill!(diff2, 0)
74 if mode == "gate_last"
75     g1 = -1
76     g2 = tags[3, 1]
77     n = 1
78     # Retarded gate method - positive diff
79     for i = 2:k[3]
80         while (tags[1, n]<g2 && n<k[1])
81             diff1[n] = g2 - tags[1, n]
82             n += 1
83         end
84         g2 = tags[3, i]
85     end
86
87     g1 = -1
88     g2 = tags[3, 1]
89     n = 1
90     for i = 2:k[3]
91         while (tags[2, n]<g2 && n<k[2])
92             diff2[n] = g2 - tags[2, n]
93             n += 1
94         end
95         g2 = tags[3, i]
96     end
97 elseif mode == "gate_first"
98     # Anticipated gate method - positive diff
99     g1 = -1
100    g2 = tags[3, 1]
101    n = 8
102    for i = 2:k[3]
103        while (tags[1, n]<g2 && n<k[1])
104            diff1[n] = tags[1, n] - g1
105            n += 1
106        end
107        g1 = g2
108        g2 = tags[3, i]
109    end
110    diff1 = diff1[8:length(diff1)]
111
112    g1 = -1
113    g2 = tags[3, 1]
114    n = 8
115    for i = 2:k[3]
116        while (tags[2, n]<g2 && n<k[1])
117            diff2[n] = tags[2, n] - g1
118            n += 1
119        end
120        g1 = g2
121        g2 = tags[3, i]
122    end
123    diff2 = diff2[8:length(diff2)]
124 else
125     # Minimum distance method
126     g1 = -1000000000
127     g2 = tags[3, 1]
128     n = 1
129     for i = 2:k[3]
130         while (tags[1, n]<g2 && n<k[1])
131             if ((tags[1, n] - g1) < (g2 - tags[1, n]))
132                 diff1[n] = tags[1, n] - g1
133             else
134                 diff1[n] = tags[1, n] - g2
135             end
136             n += 1
137         end
138         g1 = g2
139         g2 = tags[3, i]
140     end
141     g1 = -1000000000
142     g2 = tags[3, 1]
143     n = 1

```



```

144     for i = 2:k[3]
145         while (tags[2, n]<g2 && n<k[1])
146             if ((tags[2, n] - g1) < (g2 - tags[2, n]))
147                 diff2[n] = tags[2, n] - g1
148             else
149                 diff2[n] = tags[2, n] - g2
150             end
151             n += 1
152         end
153         g1 = g2
154         g2 = tags[3, i]
155     end
156 end
157 max_clicks = 100
158 max_delay = max_clicks * machine_time / 1e-9
159 @printf("PRE-filtering at max delay = %d ns \n ", max_delay)
160 filter!(x-> (x< max_clicks), diff1)
161 filter!(x-> (x< max_clicks), diff2)
162
163 difference_info(diff1, diff2, k)
164 mu_1 = Statistics.mean(diff1)
165 mu_2 = Statistics.mean(diff2)
166 sigma_1 = sqrt(Statistics.var(diff1 .- mu_1))
167 sigma_2 = sqrt(Statistics.var(diff2 .- mu_2))
168
169 return [mu_1, sigma_1, mu_2, sigma_2]
170 end
171
172 function difference_info(diff1, diff2, k)
173     machine_time = 80.955e-12
174     println("Difference Info...")
175     max_diff1 = maximum(diff1)
176     min_diff1 = minimum(diff1)
177     max_diff2 = maximum(diff2)
178     min_diff2 = minimum(diff2)
179     @printf("1) maximum difference           : %10d \n", max_diff1)
180     @printf("1) minimum difference           : %10d \n", min_diff1)
181     @printf("1) maximum time difference (ns)   : %10.4f \n", max_diff1 * ↵
182         ↵ machine_time * 1e9)
183     @printf("1) minimum time difference (ns)   : %10.4f \n", min_diff1 ↵
184         ↵ *machine_time * 1e9)
185
186     @printf("2) maximum difference           : %10d \n", max_diff2)
187     @printf("2) minimum difference           : %10d \n", min_diff2)
188     @printf("2) maximum time difference (ns)   : %10.4f \n", ↵
189         ↵ max_diff2*machine_time * 1e9)
190     @printf("2) minimum time difference (ns)   : %10.4f \n\n", ↵
191         ↵ min_diff2*machine_time * 1e9)
192
193     @printf("1) Fraction of accepted hits : %d / %d = %4.2f \n", ↵
194         ↵ length(diff1), k[1], length(diff1)/k[1])
195     @printf("2) Fraction of accepted hits : %d / %d = %4.2f\n", ↵
196         ↵ length(diff2), k[2], length(diff2)/k[2])
197
198     mod = Int(ceil(maximum([length(diff1), length(diff2)]) / 1e4))
199
200     # plot clicks
201     x_delays1 = (min_diff1:mod:max_diff1)
202     x_delays2 = (min_diff2:mod:max_diff2)
203
204     bin_num1 = Int(floor((max_diff1-min_diff1) / mod)) + 1
205     println("bins 1: ", bin_num1)
206     bias1 = Int(floor(-min_diff1/mod))
207     hist1 = Array{Int, 1}(undef, bin_num1)
208     fill!(hist1, 0)
209     i = 1
210     while (i<=length(diff1))
211         hist1[Int(floor((diff1[i] - min_diff1) / mod))+1] += 1
212         i += 1
213     end
214
215     bin_num2 = Int(floor((max_diff2-min_diff2) / mod)) + 1
216     bias2 = Int(floor(-min_diff2/mod))
217     println("bins 2: ", bin_num2)
218     hist2 = Array{Int, 1}(undef, bin_num2)

```

```

213 fill!(hist2, 0)
214 i = 1
215 while (i<=length(diff2))
216     hist2[Int(floor((diff2[i] - min_diff2) / mod))+1] += 1
217     i += 1
218 end
219 mu_1 = Statistics.mean(diff1)
220 mu_2 = Statistics.mean(diff2)
221 sigma_1 = sqrt(Statistics.var(diff1 .- mu_1))
222 sigma_2 = sqrt(Statistics.var(diff2 .- mu_2))
223
224 if (length(hist1)<600 && length(hist2)<600)
225     println("Plotting...")
226     # fig = Plotly.figure()
227     n_sigma_ = 2
228     fig = Plots.bar(x_delays1,
229                     hist1,
230                     show=true,
231                     xlabel = "absolute difference from gate event (MTU)",
232                     ylabel = "frequency",
233                     label = "T",
234                     size = (600, 400))
235     Plots.bar!(x_delays2, hist2, label = "R")
236     rectangle(w, h, x, y) = Plots.Shape(x .+ [0,w,w,0], y .+ [0,0,h,h])
237
238     recr = rectangle(2*n_sigma_*sigma_1, maximum([maximum(hist1), ↵
239         ↵ maximum(hist2)]), mu_1-n_sigma_*sigma_1, 0)
240     rect = rectangle(2*n_sigma_*sigma_2, maximum([maximum(hist1), ↵
241         ↵ maximum(hist2)]), mu_2-n_sigma_*sigma_2, 0)
242     # Plots.plot!(recr, linewidth = 2, opacity = 0.1, color=:blue, ↵
243         ↵ label=nothing)
244     # Plots.plot!(rect, linewidth = 2, opacity = 0.1, color=:red, ↵
245         ↵ label=nothing)
246
247     display(fig)
248     savefig("./images/delays.pdf")
249 else
250     println("Too long to plot...")
251 end
252 end
253
254 function gated_counter((tags, k), params; mode = "confidence")
255     println("Gated counting...")
256     mu_1 = params[1]
257     sigma_1 = params[2]
258     mu_2 = params[3]
259     sigma_2 = params[4]
260
261     @printf("mean transmitted : %6.4f \n", params[1])
262     @printf("stddev transmitted : %6.4f \n", params[2])
263     @printf("mean reflected : %6.4f \n", params[3])
264     @printf("stddev reflected : %6.4f \n", params[4])
265     N_1 = 0
266     intervals = [2]
267     for n_sigma_ in intervals
268         max_clicks = 100
269         x = 1
270         r_hit = false
271         refl = 0
272         multiple_refl = 0
273         y = 1
274         t_hit = false
275         tran = 0
276         multiple_tran = 0
277         coincidences = 0
278         if (mode == "confidence")
279             for i=1:length(tags[3, :])-1
280                 r_hit = false
281                 t_hit = false
282                 while tags[1, x] < -n_sigma_*sigma_1 + tags[3, i] + mu_1
283                     x += 1
284                 end
285                 while -n_sigma_*sigma_1 + tags[3, i] + mu_1 <= tags[1, x] < ↵
286                     ↵ +n_sigma_*sigma_1 + tags[3, i] + mu_1 && tags[1, x] < ↵
287                     ↵ tags[3, i+1]

```



```

282         t_hit = true
283         x += 1
284     end
285     if t_hit
286         tran += 1
287     end
288
289     while tags[2, y] < -n_sigma*sigma_2 + tags[3, i] + mu_2
290         y += 1
291     end
292     while -n_sigma*sigma_2 + tags[3, i] + mu_2 <= tags[2, y] < ↵
293         ↵ +n_sigma*sigma_2 + tags[3, i] + mu_2 && tags[2, y] < ↵
294         ↵ tags[3, i+1]
295         r_hit = true
296         y += 1
297     end
298     if r_hit
299         refl += 1
300     end
301     if r_hit && t_hit
302         coincidences += 1
303     end
304     if r_hit || t_hit
305         N_1 += 1
306     end
307 end
308 else
309     for i=1:length(tags[3, :])-1
310         r_hit = false
311         t_hit = false
312         while tags[1, x] < tags[3, i]
313             x += 1
314         end
315         while tags[3, i] <= tags[1, x] < tags[3, i] + max_clicks
316             t_hit = true
317             x += 1
318         end
319         if t_hit
320             tran += 1
321         end
322         while tags[2, y] < tags[3, i]
323             y += 1
324         end
325         while tags[3, i] <= tags[2, y] < tags[3, i] + max_clicks
326             r_hit = true
327             y += 1
328         end
329         if r_hit
330             refl += 1
331         end
332         if r_hit && t_hit
333             coincidences += 1
334         end
335         if r_hit || t_hit
336             N_1 += 1
337         end
338     end
339 end
340 @printf("Measurement with Âs sigma_ confidence \n")
341 println("sigma = ", n_sigma_)
342 prob_refl = refl / N_1
343 prob_tran = tran / N_1
344 prob_triple = coincidences / N_1
345 alpha = prob_triple / (prob_refl * prob_tran)
346 @printf("\t gate hits : %9d \n", N_1)
347 @printf("\t reflected hits : %9d \n", refl)
348 @printf("\t transmitted hits : %9d \n", tran)
349 @printf("\t coincidences hits : %9d \n", coincidences)
350 @printf(" ----- \n")
351 @printf("\t P[double] : %9.8f \n", prob_refl + prob_tran - 2 ↵
352         ↵ *prob_triple)
353 @printf("\t P[triple] : %9.8f \n", prob_triple)
354 @printf("\t Alpha : %9.8f \n", alpha)
355 sigma_r = sqrt(prob_refl*(1-prob_refl)/(N_1-1))

```

```

355     sigma_t = sqrt(prob_tran*(1-prob_tran)/(N_1-1))
356     sigma_c = sqrt(prob_triple*(1-prob_triple)/(N_1-1))
357     @printf("p_r variance: %9.8f \n", sigma_r)
358     @printf("p_t variance: %9.8f \n", sigma_t)
359     @printf("p_c variance: %9.8f \n", sigma_c)
360     @printf(" variance: %9.8f \n", sigma_c/(prob_refl*prob_tran) +
361           sigma_r * prob_triple/(prob_refl^2*prob_tran) +
362           sigma_t * prob_triple/(prob_refl*prob_tran^2))
363   end
364 end
365
366 function config()
367   Plots.plotly()
368   Plots.default(size=(600, 400),
369   guidefont=("times", 14),
370   legendfont=("times", 14),
371   tickfont=("times", 14)
372   )
373 end
374
375 function single_chan_stat((tags, k))
376   machine_time = 80.955e-12
377   bin_num = 1000
378   hist = Array{Int, 2}(undef, 3, bin_num)
379   fill!(hist, 0)
380   bin_step = Array{Int}(undef, 3)
381   diff = Array{Int, 2}(undef, 3, maximum(k)-1)
382   fill!(diff, 0)
383   println(length(tags[3, :]), k)
384   maxx = 0
385   for chan in [1, 2, 3]
386     series = tags[chan, :]
387     for i = 1:k[chan]-1
388       diff[chan, i] = series[i+1] - series[i]
389     end
390     filter!(z -> (z>0), diff[chan, :])
391     max_diff = maximum(diff[chan, :])
392     if max_diff>maxx
393       maxx = max_diff
394     end
395   end
396   x_axis = 0:bin_num:maxx
397   bin_size = maxx/bin_num
398   i = 1
399   for chan = [1, 2, 3]
400     filter!(z -> (z>0), diff[chan, :])
401     for i = 1:k[chan]-2
402       hist[chan, Int(ceil(diff[chan, i]/bin_size))] += 1
403     end
404   end
405
406   fig = Plots.plot((0:bin_num-1)*bin_size,
407     [log10(x) for x in hist[1, :]] ,
408     label = string("transmitted channel"),
409     show=true,
410     xlabel = "Interarrival time (MTU)",
411     ylabel = "Frequency (log)",
412     size = (600, 400))
413
414   Plots.plot!((0:bin_num-1)*bin_size,
415     [log10(x) for x in hist[2, :]],
416     label = string("reflected channel"))
417
418   Plots.plot!((0:bin_num-1)*bin_size,
419     [log10(x) for x in hist[3, :]],
420     label = string("gate channel"))
421   @printf("Sum 1 : %5.4f \n", sum(hist[1, :]/sum(hist[1, :])))
422   @printf("Sum 2 : %5.4f \n", sum(hist[2, :]/sum(hist[2, :])))
423   @printf("Sum 3 : %5.4f \n", sum(hist[3, :]/sum(hist[3, :])))
424
425   savefig(string("./images/single_chan.pdf"))
426 end
427
428 end

```

## 2. Functions involved in generation and analysis of random data

```

1  function random(tags; mode="naif")
2  data = diff(tags[1])
3  data_diff = diff(data)[1:2:length(data)]
4  println("Generating with ", mode, " rule, over ", length(tags[1]) , " ↵
    ↵ tags stream...")
5  println("Diffs length: ", length(data), " \nDiffs_of_diffs length: ", ↵
    ↵ length(data_diff))
6  mu = Statistics.mean(data)
7  lambda = 1/mu
8  median = Statistics.median(data)
9  sigma = sqrt(Statistics.var(data))
10 byte_stream = Array{UInt8}(undef, Int(floor(length(data)/8))-1)
11 if mode == "naif"
12     stream = BitArray(undef, length(data))
13     for i = 1:length(data)
14         if data[i]<median
15             stream[i] = 1
16         else
17             stream[i] = 0
18         end
19     end
20     byte_stream = Array{UInt8}(undef, Int(floor(length(stream)/8)))
21     for i = 1:length(byte_stream)-1
22         byte_stream[i] =
23             1*stream[8*i]+2*stream[8*i+1]+4*stream[8*i+2]+8*stream[8*i+3]+
24             16*stream[8*i+4]+32*stream[8*i+5]+64*s[8*i+6]+128*stream[8*i+7]
25     end
26 elseif mode == "high-rate"
27     uniform_events = Array{Float64}(undef, length(data))
28     for i = 1:length(data)
29         uniform_events[i] = exp(-data[i] * lambda) - 1
30     end
31
32     byte_stream = Array{UInt8}(undef, length(data))
33     for i = 1:length(byte_stream)
34         byte_stream[i] = -Int(floor(255 * uniform_events[i]))
35     end
36 elseif mode == "diff"
37     stream = BitArray(undef, length(data_diff))
38     for i = 1:length(data_diff)-1
39         if data_diff[i]<data_diff[i+1]
40             stream[i] = 1
41         else
42             stream[i] = 0
43         end
44     end
45     byte_stream = Array{UInt8}(undef, Int(ceil(length(stream)/8)))
46     for i = 1:length(byte_stream)-1
47         byte_stream[i] =
48             1*stream[8*i]+2*stream[8*i+1]+4*stream[8*i+2]+8*stream[8*i+3]+
49             16*stream[8*i+4]+32*stream[8*i+5]+64*s[8*i+6]+128*stream[8*i+7]
50     end
51 elseif mode == "diff-2"
52     stream = BitArray(undef, Int(ceil(length(data_diff))/2))
53     fill!(stream, 0)
54     k = 1
55     for i = 1:2:length(data_diff)-2
56         if data_diff[i]<data_diff[i+1]
57             stream[k] = 1
58         else
59             stream[k] = 0
60         end
61         k+=1
62     end
63     byte_stream = Array{UInt8}(undef, Int(ceil(length(stream)/8)))
64     fill!(byte_stream, 0)
65     for i = 1:length(byte_stream)-1
66         byte_stream[i] =
67             1*stream[8*i]+2*stream[8*i+1]+4*stream[8*i+2]+8*stream[8*i+3]+
68             16*stream[8*i+4]+32*stream[8*i+5]+64*s[8*i+6]+128*stream[8*i+7]
69     end
70 end
71 println("Byte stream generated")

```

```

72     println("mean: ", Statistics.mean(byte_stream))
73
74     rnd_tester(byte_stream)
75     return byte_stream
76 end
77
78 function rnd_tester(byte_stream)
79     samples = length(byte_stream)-9
80     while samples % 3 != 0
81         samples -= 1
82     end
83     println("Analyzing ", samples, " UInt8 numbers...")
84     # 3D Scatter
85     fig1 = Plots.scatter3d(byte_stream[1:3:samples], ↵
        ↵ byte_stream[2:3:samples], byte_stream[3:3:samples],
86         markercolor = :blue,
87         markershape = :cross,
88         markersize = 0.5,
89         opacity = 0.1,
90         label= nothing,
91         tickfont=("times", 12),
92         size = (1024, 768)
93     )
94     # Histogram
95     fig2 = Plots.histogram(byte_stream, bins = 256, label= nothing)
96     # Random walk
97     x = Array{Float64}(undef, samples)
98     fill!(x, 0)
99     mu = Statistics.mean(byte_stream)
100    println("Statistical mean of byte_stream: ", mu)
101    for i=2:samples
102        x[i] = x[i-1] + byte_stream[i] - 127.5
103    end
104    fig3 = Plots.plot(1:samples, x, label= nothing, size = (1024, 768))
105
106    display(fig1)
107    savefig(fig1, "./random_img/scatter3d.pdf")
108    display(fig2)
109    savefig(fig2, "./random_img/histogram.pdf")
110    display(fig3)
111    savefig(fig3, "./random_img/random_walk.pdf")
112 end

```