Malus' law photon by photon, a deterministic method.

Pierre Leroy

Abstract: This document proposes a method of calculation to define the direction of passage of a single photon in a polarizer. By accumulation of measures, law of Malus is generated. The method has the property of being completely deterministic. It allows the simulation of an alignment of polarizers.

Introduction.

Malus' law applied to a light intensity is a statistical law applied to a large number of photons.
It is interpreted by the classical electromagnetic model as a macroscopic law.

However, it is no longer applicable when transmitting photon by photon, because these cannot be partially transmitted.

The document presented here describes a method of polarization photon by photon, making it possible to produce Malus' law by accumulation.

The approach is algorithmic, and describes only the operations performed on locally available information.

It shows, however, that the Malus's law can be obtained using a deterministic method not using random variables.

It allows to give a local realistic explanation to the Dirac experiment with 3 polarizers [1].

---

1. How the model works.

The operation is done by simulating a polarizer with two outputs.
Outputs are named **o** and **e** for “ordinary” and “extraordinary”.
The simulation has 3 stages.

Three basic variables are used for operation.

- **p**, attached to the photon, is the polarization angle of the photon between [0..PI].
- **q**, attached to the photon, represents a physical quantity between [-1..1].
  The physical nature of this value is currently undefined.
  Assumptions are made later in "Possible interpretation of the variable q".
- **a_pol**, is the orientation angle of the polarizer between [0..PI].

Note: Angles outside the interval [0..PI] must be brought back into the interval using a modulus PI function.

Operating steps.

First step:

It consists in calculating a variable noted **s** (as **s**elect), with

$$s = \frac{\pi - \operatorname{acos}(q)}{2} + ad$$

With:
ad = p - a_pol [-PI..PI]

The variable **ad** represents the difference in angle between the polarization of the photon and the angle of the polarizer.

The variable **s** alone determines the output e or o which will be taken by the photon.

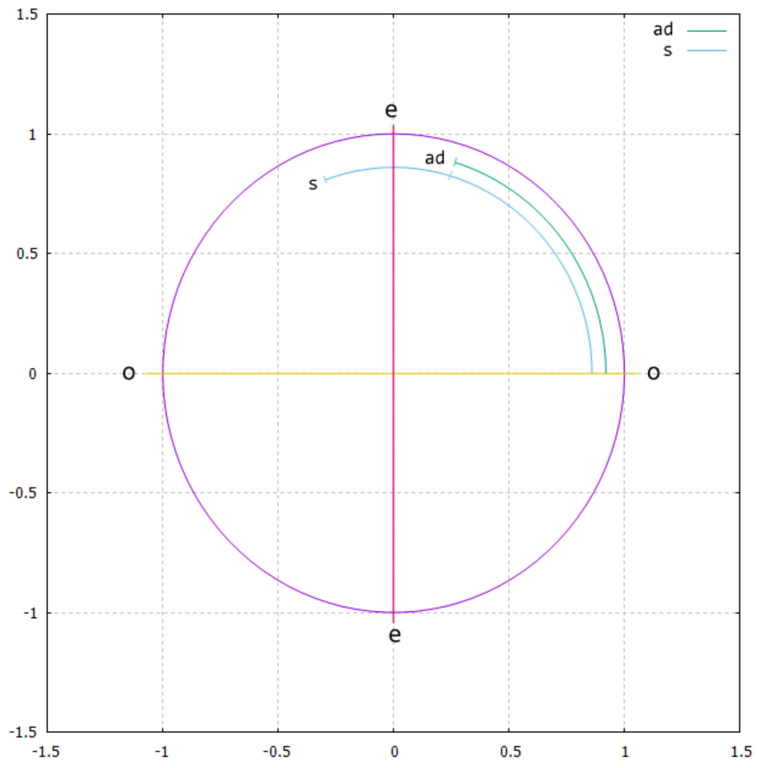
Second step:

Determine the output o or e of the polarizer taken by the photon.
This is defined by comparing the variable **s** with some threshold values associated with the outputs e and o.

By representing the polarizer with graph 1, the values of the output thresholds **o** and **e** are found at 0, PI/2, PI, 3PI/2.
That is to say 4 quadrants of size PI/2.

Angle 0 is located on the right o output. The rotations are counted positive in the counterclockwise direction.
The two outputs o and e are merged in a real polarizer, but the graph makes it possible to represent all the values that the variable **ad** can take between [–PI..PI]

Graph 1 :



The rule for defining the output is as follows:

The selected output is the closest one that is in the PI/2 interval containing s whose angle is less than s.

For example, on the graph 1, **s** is between PI/2 and PI, and the closest output whose angle is less than **s** is the output e with angle PI/2.
The selected output for the photon will be e.

Third step:

Update the two variables p and q of the photon.
This makes it possible to propagate the effect produced by the polarizer on the photon.

Update of p.

It consists in aligning the polarization of the photon on the angle of the polarizer according to the output taken.
That is :

p = angle of the polarizer for the output o.
p = angle of the polarizer + PI/2 modulus PI for the output e.

Note: for the simulation of a blocking polarizer (ex polaroid), only one of the outputs is used, the other is supposed to block the photon.

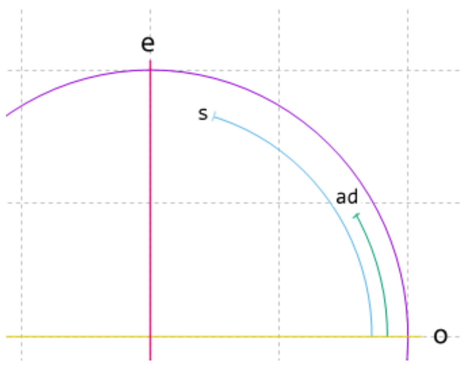
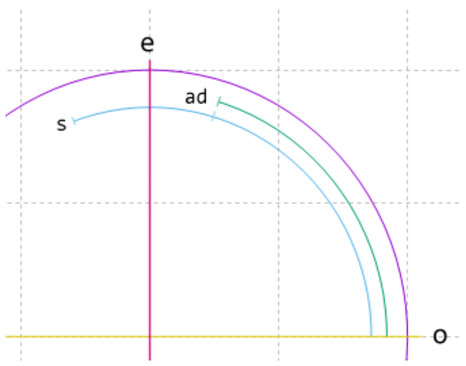
Update of q.

This update will condition, depending on **p** and the angle of the next polarizer, the output taken in the next polarizer. This requires the calculation of a variable noted **rp** (repolarization), and is defined as:

rp = **ad -** (angle of the e/o output selected in step 2), which is in the range]-PI/2..PI/2[

rp defines the adjustment value of the polarization **p** that the photon undergoes while passing through the polarizer. This variable is used to update **q**, according to two possible options depending on rp sign.

Graph 2 :

Condition 1 : **rp >= 0**	Condition 2 : **rp < 0**
	
This happens if the value of **ad** and **s** are in the same PI/2 quadrant. The output taken here is o (Closest less than s) The update of q (q2) is done as follows: $q2=\frac{q+1}{\cos(rp)^2}-1$	This happens if the value of **ad** and **s** are then in different PI/2 quadrants. The output taken here is e (Closest less than s) Updating q is done as follows: $q2=\frac{q-1}{\cos(rp)^2}+1$

This mechanism has the effect of increasing or reducing the value of q.

If we consider the passage of a photon in an alignment of polarizers,

Condition 1 statistically increases the value of **q** with a ratio of **4/3** and tends to increase the probability of producing condition 2 upon interaction with the next polarizer.
Condition 2 statistically reduces the value of **q** with a ratio of **4/5** and tends to increase the probability of producing condition 1 when interacting with the next polarizer.

The two situations (conditions 1 and 2) occur with the same ½ proportion and the two effects produced on **q** are balanced.

We can notice that the new value of q (q2) seems to be able to take values < -1 or > 1 if the value of cos(rp)^2 is close to 0.
This would then produce an error when using the acos(q) function used to define **s**, this one requiring an argument between [-1..1].
However, this never happens and q2 always remains in the interval [-1..1]
Note also that the value of cos(rp)^2 never takes the value 0, because rp is always included in the interval]-PI/2..PI/2[(+/- PI/2 excluded)
We can also notice that the variable q is not modified if the photon passes through two polarizers aligned with the same angle.

This mechanism makes it possible to propagate the photon through an alignment of polarizers without requiring the intervention of a random factor.
The initial pair of variables p and q at the input of the first polarizer then determines the sequence of the o/e passages in the alignment of the following polarizers in an entirely deterministic process.
The accumulation of photons then produces exactly Malus' law.
An example of code showing this operation is available in the appendix.

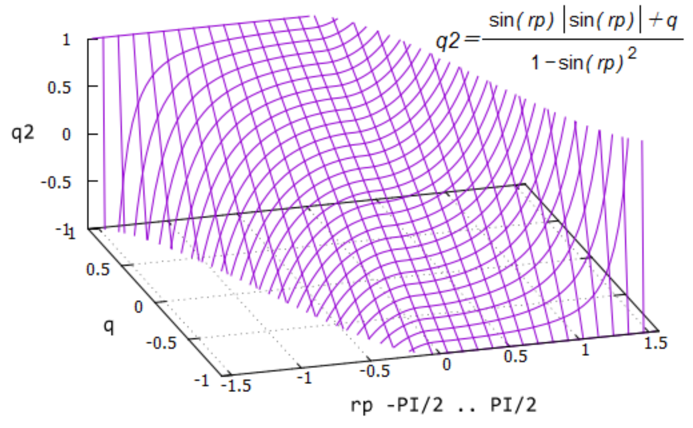
Plot of the curve q2 = f (q, rp)

The curve q2 = f (q, rp) can be plotted with gnuplot [3] using the following commands:

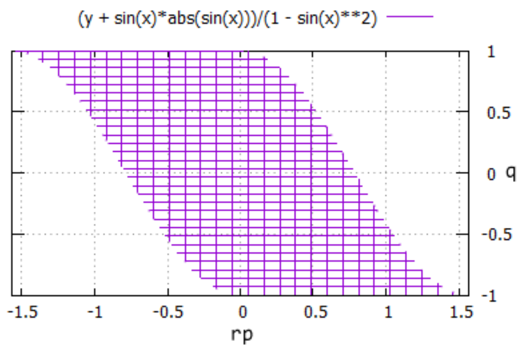
```
set xrange [-pi/2:pi/2]
set yrange [-1:1]
set zrange [-1:1]
set grid
set xyplane at -1
set isosamples 30
splot (y + sin(x)*abs(sin(x)))/(1 - sin(x)**2)
```

The function used combines the two updating functions of q into one, with x = rp, y = q, z = q2.
The graph produced is as follows.

Graph 3a



Graphe 3b

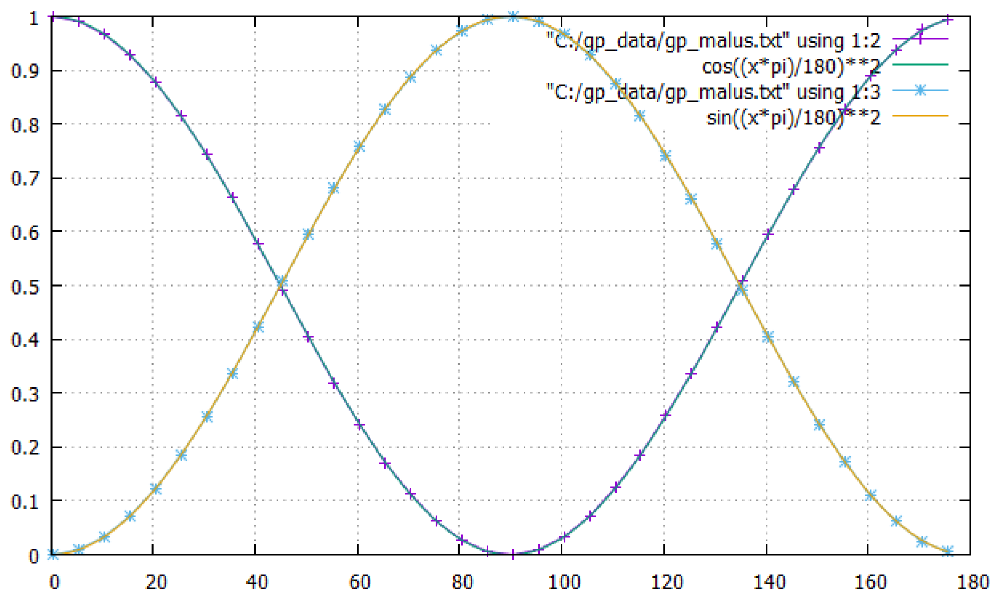


Top view, projection on the plane q,rp

We see on the plot that to produce a value of q2 between -1 and 1, we must respect a constraint between q and rp.
This is clearly visible on graph 3b, and has a cos² shape.
This one is produced automatically by the polarization algorithm.
Although a plot of the distribution produced by the value of q is uniformly random in shape, q is not random.

Law of Malus by accumulation.

Graph 4:



This graph shows that the curves produced by accumulation of photons on the o and e outputs comply with Malus' law.

2. Discussion.

The method produces a partial decoupling between the polarization of the photon and its orientation in the polarizer.

Although the variable p completely defines the polarization of the photon, the orientation taken does not depend only on this parameter, but also on the variable q.

The result of the orientation "decision" then depends on 2/3 of the polarization and 1/3 of the variable q.

The variable q produces a form of jitter that alters the orientation.
This jitter is however not random and does not depend on external parameters, but is updated by a continuous function during the process of repolarization of the photon.

The value of (PI - acos(q))/2 used to define s always specifies a positive quantity between [0..PI/2]

An alternative method generating a value varying between –PI/4..PI/4 is possible, however it produces situations where the variable rp (repolarization) is greater than +/- PI/2, which physically seems improbable.
In addition, this no longer makes it possible to update q with the simple method proposed. (A method may exist but has not been looked for)
It is therefore assumed that q must be positive.

We can also notice, by applying a principle of least action on the quantity of repolarization that the photon undergoes during the alignment of its polarization on an output e or o, and by considering a possible output at all the intervals PI/2, that maximum repolarization should not exceed PI/4.
Indeed, there is always a possible output e or o located at an angle difference less than or equal to this PI/4 value.
However, the principle does not apply, otherwise we would obtain a rectangular Malus law. This means that the “jitter” mechanism takes priority over this principle.

3. A possible interpretation of the variable **q**.

Note: This interpretation is that of the author who is a programmer and not a physicist. It is mentioned because it seems to agree well with the method, but may seem very “inappropriate” to a physicist.

The structure of the algorithm indicates that the output taken by the photon depends only on the variable **s**.
The value of **s** depends on the polarization deviation **ar** to which is added an always positive value dependent on **q**.

If we consider the photon to have a location and a volume, it seems appropriate to think that the variable s can define a positional offset of the photon during the interaction defining the output.

One option would be to consider q as a "stretch" or "dispersion" coefficient of the photon wave packet tending to shift the "center" of the photon towards the section corresponding to the output.
This is suggested by the method used to update **q** that produce a compress/stretch effect by dividing by $\cos(rp)^2$.

For a more conventional physical approach, a document by B Dalton [2] has similarities with the method described here.
His method uses random sources, but uses a distribution similar to that produced by acos(q).
Perhaps it is possible to replace these random sources with a method similar to that used to update the variable q.
This will make it possible to produce an entirely local and deterministic physical model of Malus' law.
If a physicist is interested in this project, I can try to help him with computer simulations.

4. Summary and conclusion.

The model presented shows that it is possible to produce Malus' law photon by photon using a local and deterministic process.

A random source is therefore not mandatory in order to apply a probability test to determine the output of the photon.

The model makes it possible to explain the “paradox” of the three polarizers at the level of a single photon.

To complete this, a future document will study the correlations that this model produces, and the results it generates in EPR-type experiments.

5. Annex.

Source code:

This first example tests Malus' law photon by photon by simulating an alignment of 15 blocking polarizers.
The Dirac test with three polarizers can be enabled by replacing the #if 1 line with #if 0 in the main() function.
It is very simplified in order to be included directly in this document. (A downloadable version is available at the end of the text).
It does not use any data structure or functions specific to the C language so that it can be easily transcribed into another language.

```c
// Test Malus's law accuracy through a series of polarizers
// Simulate blocking polarizer (PBS only o output used)
#include <stdio.h>                                // for print functions
#include <math.h>                                 // for math acos() function

#define PI 3.14159265358979323846
#define DEG_TO_RAD(d) (((d)*PI)/180.0)           // convert degree to radians

#define OUT_O 0                                   // coded value for o output
#define OUT_E 1                                   // coded value for e output

#define MAX_POL 100                               // max count of aligned polarizers for test

// arrays store polarizer angles settings, and count of photons passing through polarizers
double pol_angle[MAX_POL];                        // angles of aligned polarizers
int pol_ctr_o[MAX_POL] = { 0 };                   // counters for passing photons, initialized with 0

// ---------------------------------------------------------------------
// random generator, used to initialize photons incoming on first polarizer

unsigned long n_seed = -123;                      // default seed

void srand(int seed) { n_seed = seed; }           // change seed

double rand1(void)                                // return random value [0..1]
{
  n_seed = n_seed * 214013L + 2531011L;
  return ((n_seed >> 16) & 0x7fff)*(1.0/0x7fff);
}

// ---------------------------------------------------------------------
// count of photons used for simulation (results more accurate if high value used)
#define SIM_N_PHO (100*1000*1000)

// ------------------------------------------------
// main simulation

void main(void)
{
  int i;                                          // photons loop counter
  double res_ma;                                  // photons transmission ratio result

  // ---------------------------------------------
  // init polarizer alignment angles

#if 1                                             // change to #if 0 to test Dirac 3 pol.
  // define some random angles for polarizer
  int n_pol = 15;
  for (i=0; i<n_pol; i++)
    pol_angle[i] = rand1()*PI;
#else
  // define user angles (Dirac 3 polarizers test)
  int n_pol = 3;
  pol_angle[0] = DEG_TO_RAD(0);
  pol_angle[1] = DEG_TO_RAD(45);
  pol_angle[2] = DEG_TO_RAD(90);
#endif

  // ---------------------------------------------
  // simulate

  srand(-1234);
  printf("Simulate %d aligned blocking polarizers. Please wait.. (may require 1 minute)\n", n_pol);

  for (i=0; i<SIM_N_PHO; i++)
  {
    // init photon p and q for first polarizer (random polarized light)
    double pho_p = rand1()*PI;                    // initial entering polarization
    double pho_q = 2*rand1() - 1;                 // initial q, -1..1 range
    int j;                                        // declare loop counter

    // pass photon through polarizer serie
    for (j=0; j<n_pol; j++)
    {
      double a_pol = pol_angle[j];                // get initialized polarizer angle

      // apply polarization method
      int o;                                      // declare output selected variable
      double rp, rp_cos, rp_cos2;                 // declare rp, cosinus, and cosinus squared
      double ad = pho_p - a_pol;                  // get polarization angle diff
      double s = ad + (PI - acos(pho_q))*0.5;     // get s to define out

      // define output and rp
           if (s >=  PI)    { o = OUT_O; rp = ad -   PI; }
      else if (s >=  PI/2)  { o = OUT_E; rp = ad - PI/2; }
      else if (s >=  0)     { o = OUT_O; rp = ad;        }
      else if (s >= -PI/2)  { o = OUT_E; rp = ad + PI/2; }
      else                  { o = OUT_O; rp = ad +   PI; }

      // if output is not o, photon is blocked (passed e), stop alignment simulation
      // for this photon (update of p/q then not required and not done)
      if (o != OUT_O)
        break;                                    // stop simulation (j loop only)

      // update p
      pho_p = a_pol;                              // o align polarization

      // update q
      rp_cos = cos(rp);                           // cos(rp)
      rp_cos2 = rp_cos*rp_cos;                    // cos(rp)^2
      if (rp >= 0)
        pho_q = (pho_q + 1)/rp_cos2 - 1;
      else
        pho_q = (pho_q - 1)/rp_cos2 + 1;

      pol_ctr_o[j]++;                             // increment passing photons counter
    }
  }

  // print simulation result, compare with theoretical macroscopic Malus's law
  res_ma = 0.5;                                 // define 1st polarizer ratio, must be 50% transmitted
  for (i=0; i<n_pol; i++)
  {
    double res_sim, diff;
    double a_pol = pol_angle[i];                  // get polarizer angle

    // def theoretical results
    if (i > 0)                                    // if not first polarizer, adjust transmitted ratio
      res_ma *= pow(cos(a_pol - pol_angle[i-1]), 2);  // cos²(angle with previous polarizer)

    // get simulated result
    res_sim = (double)pol_ctr_o[i]/SIM_N_PHO;     // passing photon ratio for polarizer[i]

    // get simulation/theoretical results difference
    diff = fabs(res_sim - res_ma);

    // print results
    printf("pol %d:  th:%.6f  sim:%.6f  diff:%.6f\n", i, res_ma, res_sim, diff);
  }
}
```

The code produces the following output: (may vary slightly if RNG seed is changed)

```
Simulate 15 aligned blocking polarizers. Please wait.. (may require 1 minute)
pol 0:  th:0.500000  sim:0.500097  diff:0.000097
pol 1:  th:0.452976  sim:0.453067  diff:0.000091
pol 2:  th:0.159733  sim:0.159764  diff:0.000031
pol 3:  th:0.124032  sim:0.124053  diff:0.000022
pol 4:  th:0.082649  sim:0.082669  diff:0.000021
pol 5:  th:0.078977  sim:0.078993  diff:0.000017
pol 6:  th:0.077181  sim:0.077197  diff:0.000016
pol 7:  th:0.007218  sim:0.007223  diff:0.000006
pol 8:  th:0.001194  sim:0.001198  diff:0.000004
pol 9:  th:0.000316  sim:0.000317  diff:0.000001
pol 10:  th:0.000130  sim:0.000131  diff:0.000000
pol 11:  th:0.000128  sim:0.000128  diff:0.000000
pol 12:  th:0.000068  sim:0.000068  diff:0.000000
pol 13:  th:0.000067  sim:0.000067  diff:0.000000
pol 14:  th:0.000004  sim:0.000004  diff:0.000000
```

Note: The diff error tends to 0 when SIM_N_PHO tends to infinity.

The following examples are available as a downloadable link.

test_malus_block.c :
Downloadable version of the example contained in this document.
Compilation with GCC can be done with the following command:
  gcc –O2 test_malus_block.c –o test_malus_block.exe

The following sources use a separate function for the polarization method.

pol_malus_pbs.c:
This example tests Malus' law photon by photon in an alignment of PBS polarizers.
It tests an alignment of 15 polarizers with o and e outputs.
The Dirac test with three polarizers can be enabled by replacing the #if 1 line with #if 0 in the main() function.
Compilation with GCC can be done with the following command:
  gcc –O2 pol_malus_pbs.c –o pol_malus_pbs.exe

malus_crv.c:
This code simulates the accumulation Malus'law and produces a data file for gnuplot [3] in order to plot graph 4 used in this document.
The command line for gnuplot is defined as comment in the source code.
Compilation with GCC can be done with the following command:
  gcc –O2 malus_crv .c –o malus_crv .exe

6. References.

[1] Dirac Three Polarizers Experiment:
https://www.informationphilosopher.com/solutions/experiments/dirac_3-polarizers/

[2] B Dalton (2001) « Law of Malus and Photon-Photon Correlations: A Quasi-Deterministic Analyzer Model »
https://arxiv.org/pdf/quant-ph/0101127.pdf

[3] gnuplot, allows to draw the graph 3.
http://www.gnuplot.info/

Email : pierrel5@free.fr

Initial version.