The most exciting roller coaster

Team number 404

Problem selected: B

Abstract: In this paper we discuss the design of a roller coaster. Firstly we define what 'exciting roller coaster' means. Secondly, we will discuss safety of people being subjected to enormous accelerations, which occur during ride. Then we present our model of the dynamics of the car during roller coaster ride. The paper explains used assumptions and our reasoning behind them.

In order to simulate car moving along the tracks, and to calculate it's acceleration, velocity, rotation in every single point, we've made a program using ROOT - an open source scientific software made by CERN. We will provide detailed explanation of how our simulation works, as well as present all assumptions we've made. Later we will describe elements used to construct the track, talking about functions we used to design them. We will emphasize usage of clothoid functions, also known as Euler's spiral, explaining why we've used it in elements of a roller coaster.

We will then present the result of our program - a track that is safe as well as exciting. In the end we will discuss what further work and research could be done to improve roller coasters even more.

1 Introduction

Although most of people associate the roller coasters with pure fun, there is also a more complicated side of the subject. The dynamics of a roller coaster car and the architecture of the track pose troubles in front of many engineers. Our paper is an attempt to design and present a safe and, most importantly, exciting roller coaster ride. Our team created a model of dynamics of a roller coaster ride and used it to simulate the parameters of the motion of the the car.

2 Measuring excitement, enjoyment and safeness of a ride

2.1 Positive factors

When it comes to designing most exciting roller coaster, one has to ask: what excitement is, and how to measure it?

Four main sources of enjoyment while riding roller coaster were considered:
- speed
- acceleration
- duration of the ride
- diversity of the track.

2.2 Negative factors

On the other hand there are some major restrictions on the shape of the track to assure safety. The one most important task is controlling the acceleration.

In the following paragraphs we will refer to the acceleration in units of g with $1g$ being equivalent of a standard gravity.

What kind of acceleration human body can withstand depends on factors such as rate of its changes, time of exposure and even its direction[1]. So, if g-force is applied parallel to the spine, directed downwards, $5g$ is a maximum a human can take without fainting, but in practice in might be lower. G-force applied upwards presents much greater risk, since all the fluids flow into upper parts of the body, including brain. The limit for this kind of force is $1g$.

When it comes to horizontal forces, human body has much higher tolerance. During experiments, it was shown that a person can withstand up to $20g$ for the duration of $10s$, or up to $10g$ for 1 minute. This means that cars accelerating along the track present no danger to people inside.

Negative impact on the enjoyment have been determined to be caused also by the forces perpendicular to the velocity which happen to not be countered by the the force of reaction of the track. It is of great importance to minimize that effect while designing a roller coaster track.

3 Model of the physics of the roller coaster used

3.1 Assumption made

The model used in the paper is based on following assumptions:

Firstly, the cars are assumed to have no moment of inertia, therefore no rotational energy is taken into account. We can make such assumption due to rotational velocity being low.

The main source of energy losses is rolling resistance. We use basic model of rolling resistance, given by the equation[2]:

$$F_{rol} = C_{rol} N \tag{1}$$

Where C_{rol} is a rolling resistance coefficient dependant on the materials witch collide and the size of used wheels. N is ground reaction force.

Another source of energy loss is air resistance. The used model of air resistance was a simple force proportional to the square of velocity.

$$F_{air} = -\kappa v^2 \tag{2}$$

Where v stands for velocity and κ is a coefficient of air resistance dependant on the aerodynamics of the moving car.

Our model also assumed that effects caused by geometry of cars are negligible. It treated cars as points.

We also neglected differences of normal acceleration of different cars caused by their different velocities at the same point in space. (full acceleration normal to track presented on graphs and used for computations of friction was calculated for first car)

3.2 Dynamics of the motion

In our paper, the movement of a car is deterministic, dependant only of the beginning conditions (starting height and velocity) and shape of the track. There are only four acting forces: gravitational, centripetal force given by the reactant force of the track and two resistance forces described above. The friction forces are not conservative, therefore the principal of energy conservation of the system can be written as:

$$mg\Delta h_1 + mg\Delta h_2 + ... = \Delta s * (a_n + g_n) + \frac{v_1^2 \cdot M}{2} - \frac{v_0^2 \cdot M}{2} - v^2 \alpha \Delta s \quad (3)$$

Where Δh_1, Δh_2...are changes of height for cars, m is weigh of one car, M is weight of all cars, a_n is normal component of acceleration, g_n is normal component of gravity acceleration, v_0 is velocity of roller coaster on step before, v_1 is velocity of roller coaster on next step.

4 Simulating roller coaster

We've used ROOT made by CERN to simulate car travelling on track. We've created tracks using various functions, which are described later on, in section 5. Our program can be used to calculate dynamics roller coaster for any track. In order to work it needs some parameters of roller coaster like number of cars, distance between center of mass of cars, friction coefficient and aerodynamics friction coefficient.

We numerically calculated step by step velocity using principal of energy conservation.

$$mg\Delta h_1 + mg\Delta h_2 + ... = \Delta s * (a_n + g_n) + \frac{v_1^2 \cdot M}{2} - \frac{v_0^2 \cdot M}{2} - v^2 \alpha \Delta s \quad (4)$$

Where Δh_1,Δh_2...are changes of height for cars, m is weigh of one car, M is weight of all cars, a_n is normal component of acceleration, g_n is normal component of gravity acceleration, v_0 is velocity of roller coaster on step before, v_1 is velocity of roller coaster on next step, Δs was length of piece of curve. We have decided to use this way of computing velocity because it allowed us to use different parametrization on different parts of curve. Only additional assumption needed was that steps are small for each fragment of curve. When we had values of velocity at all Δs we could calculate Δt (time needed to travel Δs).

Value of a_n was calculated by computing radius of curvature. It was made by calculating area of triangle defined by three points in space using Heron's formula. (position of first car one step before, current position and position at the next step). Then comparing area of this triangle with another formula:

$$S = \frac{abc}{4R} \tag{5}$$

We calculated radius of curvature at current point. Then using this radius and velocity using formula for centrifugal acceleration it was possible to calculate value of normal acceleration to the curve.

In simulation we also calculate vector of reaction force of track. It is calculated in few steps. Firstly, vector of normal acceleration is calculated. It is done by numerically finding vector which is normal to the curve and lays in plain of triangle described above. Then this vector is normalized and multiplied by value of a_n. Then we find normal to the curve part of gravitational acceleration and these two are added. The result is vector of reaction force divided by mass of roller coaster.

4.1 Track segments

We represented one segment of a track by parametrized function: $X(\tau)$, $Y(\tau)$, $Z(\tau)$. Each segment was a different, continuous and differentiable function. We were choosing functions on different segments in such a way that transitions between them were also continuous and differentiable. We were testing our assumptions by running simulation and checking whether velocity and acceleration are continuous functions.

Each function was only evaluated for a certain interval of τ, effectively making them finite curves.

We used our algorithm to move and rotate each segment, so they fit together. We ensured that adjacent curves have one point in common, so our calculations did not loose accuracy.

5 Track elements

5.1 Loop

In this subsection we describe possible loops' shapes and we select the ones that provide exciting, yet safe experience.

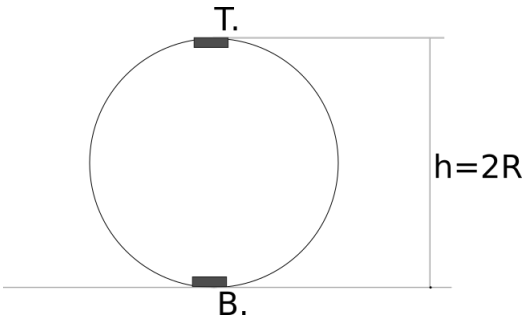

Figure 1: Circular loop. B. means car's position at the bottom of a loop, while T. - being at the top

5.1.1 Circular loop

The most obvious shape of a loop is a circle. However, this shape comes with few problems, that disqualify it from being used.

Figure 1 shows a simple diagram of a circular loop. Let v_b, v_t be car's velocity at the bottom and top of the loop, and a_b, a_t - its centripetal acceleration. Thus:

$$a_b = \frac{v_b^2}{R} \tag{6}$$

$$a_t = \frac{v_t^2}{R} \tag{7}$$

Then, from principle of energy conservation (friction and air resistance are neglected, since we only want to get the general idea about acceleration values)

$$\frac{v_b^2}{2} = \frac{v_t^2}{2} + gh \tag{8}$$

$$v_b^2 = v_t^2 + 4gR \tag{9}$$

Then, from 6, 7 and 9:

$$a_b = a_t + 4g \tag{10}$$

Since the lowest possible centripetal acceleration at the top of a loop is g, if we want the force of gravity to act as centripetal force, not allowing the car to fall down, then the lowest possible centripetal acceleration at the bottom is $5g$. Which is something rather to be avoided, because it might pose a

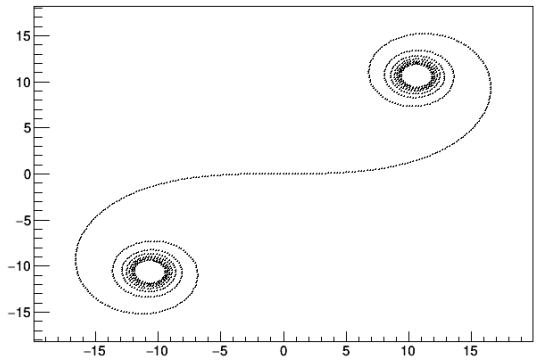

Figure 2: Euler's spiral

danger. However, if we attach the car to the tracks, making it unable to fall off, then the car can go as slowly as we want, even barely making it through the loop's top point. In this kind of scenario, the centripetal acceleration at the bottom is around $4g$, while almost 0 at the top. Although, hanging upside - down is unpleasant, and if done for a long time - might be dangerous.

Secondly, when a car approaches circular loop, it has no centripetal acceleration (since it's riding on a plain track), then it suddenly is subjected to 4 or $5g$. It is not pleasant feeling at all, we believe.

Hence, vertical circular loops should be avoided.

5.1.2 Clothoid loop

The solution of problems stated above is to use a loop that gradually decreases its radius, allowing passengers for a more gentle onset of additional acceleration[3]. Such shape can be obtained by using Euler's spiral, given by equations:

$$y(t) = \int_0^t sin(\frac{x^2}{2})dx \tag{11}$$

$$x(t) = \int_0^t cos(\frac{x^2}{2})dx \tag{12}$$

This spiral is presented on figure 2. When we use only part of it, with t varying from 0 to ... (those numbers were found using ROOT numerical methods to find maximum of a function), then combine it with other spiral's part, mirrored and shifted, we achieve shape of a loop (figure 3.

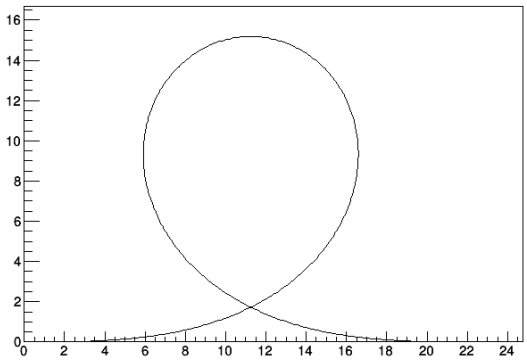

Figure 3: Loop made from two parts of Euler's spiral

Although, we can use other equations than 11 and 12 to obtain loop shape, for example $y(t) = \int_0^t sin(\frac{x^4}{4})$. It is hard to say which one is better, because the acceleration along the loop varies, when different initial velocity is concerned. So, we've decided that in order to see which from those two and many other shapes fits our needs the best, computer simulation is needed.

5.2 Turn

To ensure that there are no rapid onsets of acceleration felt by the passengers, turns are designed using clothoid function. It also allows the track to rotate gently in order to compensate centrifugal force. We were using the first clothoid until it was rotated by 45^o, and then we were adding mirrored one to complete a full 90^o turn.

5.3 Hill

Hills we've made were also designed using Euler's spiral, allowing for continuous change of normal (perpendicular to velocity) acceleration.

6 Results

The final results were calculated via simulation done by the program we have written. The track consists of 8 segments, which are as follows:

1. Short downhill ride (to gain speed)
2. First loop
3. A turnaround

4. Second loop
5. Double loop
6. Second short downhill ride
7. A second turnaround
8. The end (or the beginning) made out of line followed by an elevator (to "charge" the car with the potential energy)

All used segments were constructed out of clothoid functions connected in a proper way to assure continuity of the change of acceleration. Simulated ride of one car provided following results: $33.08s$ on a track of length $520.34m$. Used coefficients were as follows: 0.5 - air resistance coefficient, 0.0004 - friction coefficient.

Time of a ride for all coefficients equal to zero: $30.66s$.

Time of a ride for only non-zero coefficient being friction of 0.0004: $30.83s$.

Time of a ride for only non-zero coefficient being air resistance of 0.5: $33.12s$.

The non-zero coefficients were approximated for a car of shape of cuboid and track made of steel with wheels of radius $0.1m$. The results show that for used approximations air resistance was a more dominant factor.

Figures 4 and 5 show our roller coaster in 3 dimensions. Figure 6 shows projection of the same track along y axis. Figure 7 shows velocity, figure 8 shows acceleration calculated for $\alpha = 0.5$ (air resistance coefficient), and $\beta = 0.0004$ (friction coefficient)

References

[1] Wikipedia. g-force.

[2] Wikipedia. Rolling resistance.

[3] Nick Berry. Why roller coaster loops are never circular. *Gizmodo.com*, 2014.

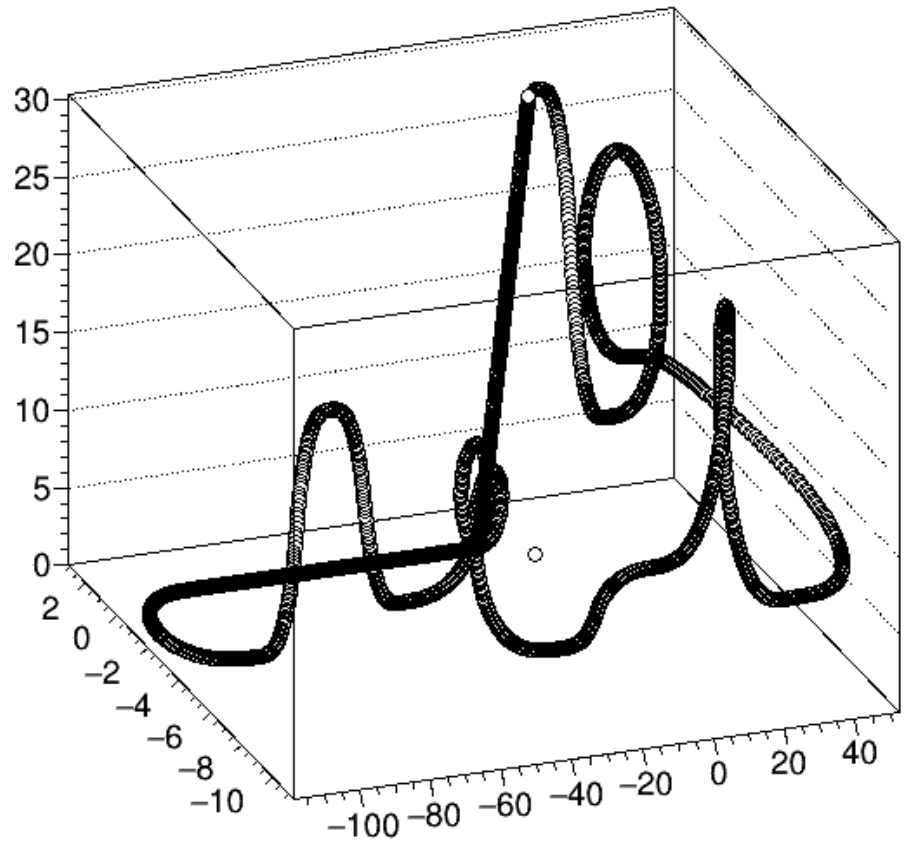

Figure 4: 3D model of the roller coaster

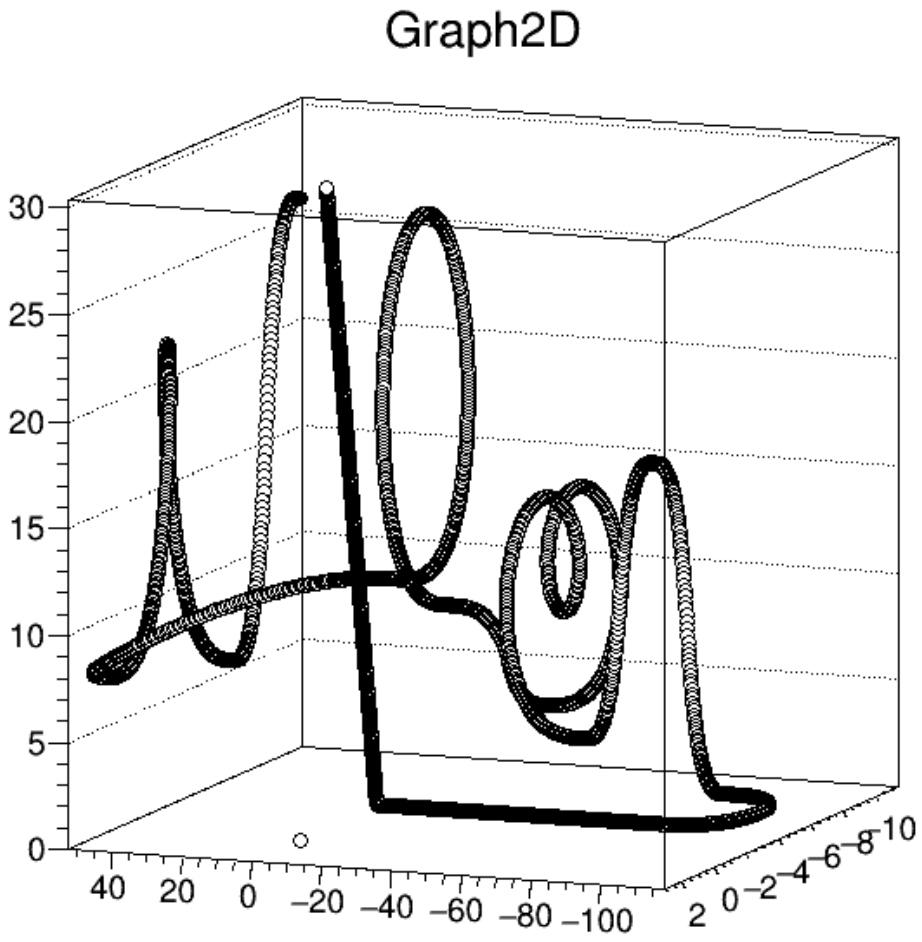

Figure 5: 3D model of the roller coaster from a different perspective

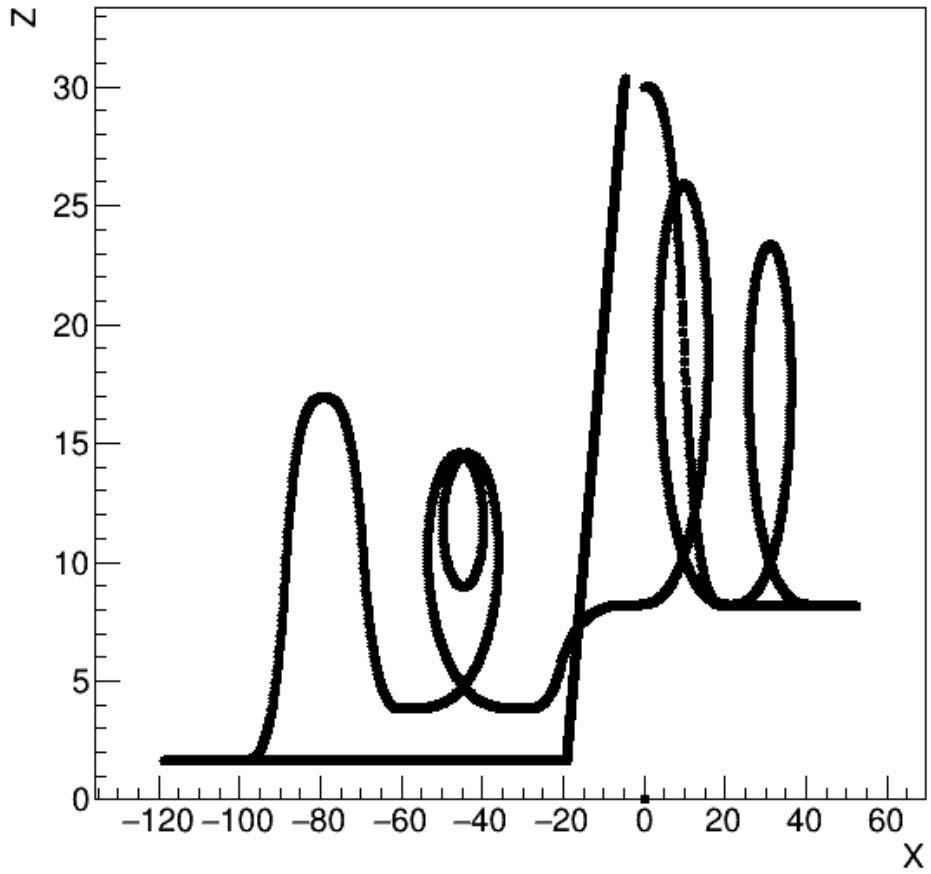

Figure 6: Projection of the roller coaster along y axis

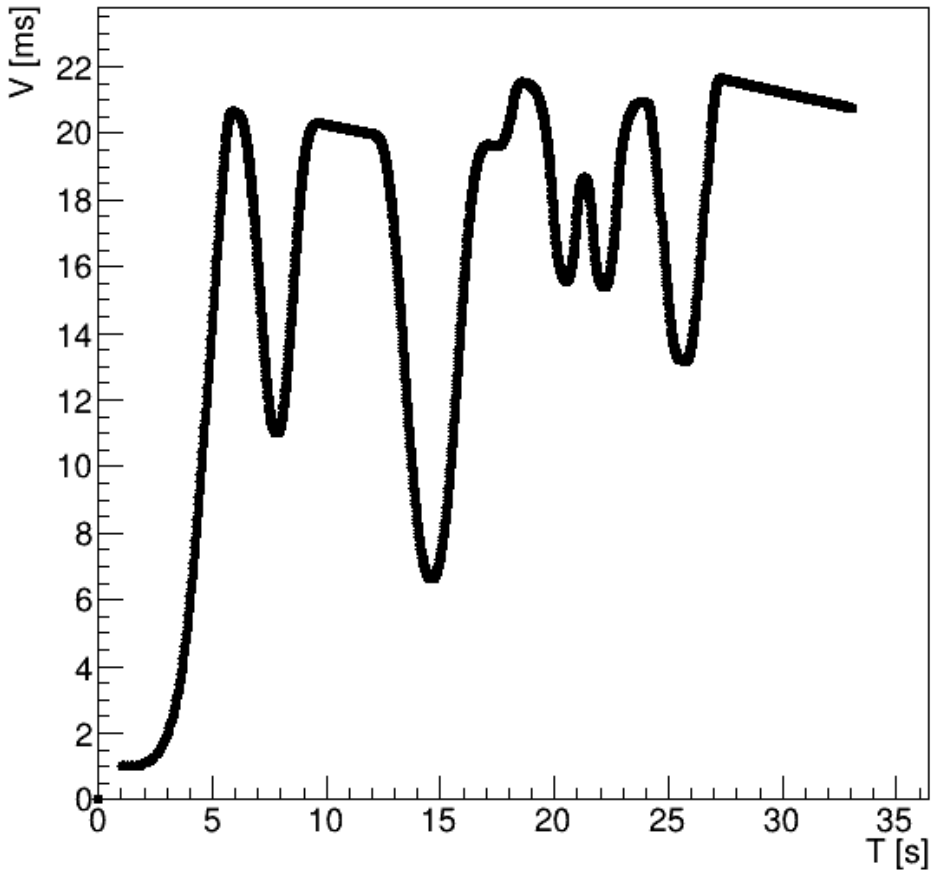

Figure 7: Velocity-time graph of the roller coaster ride for the first set of coefficients

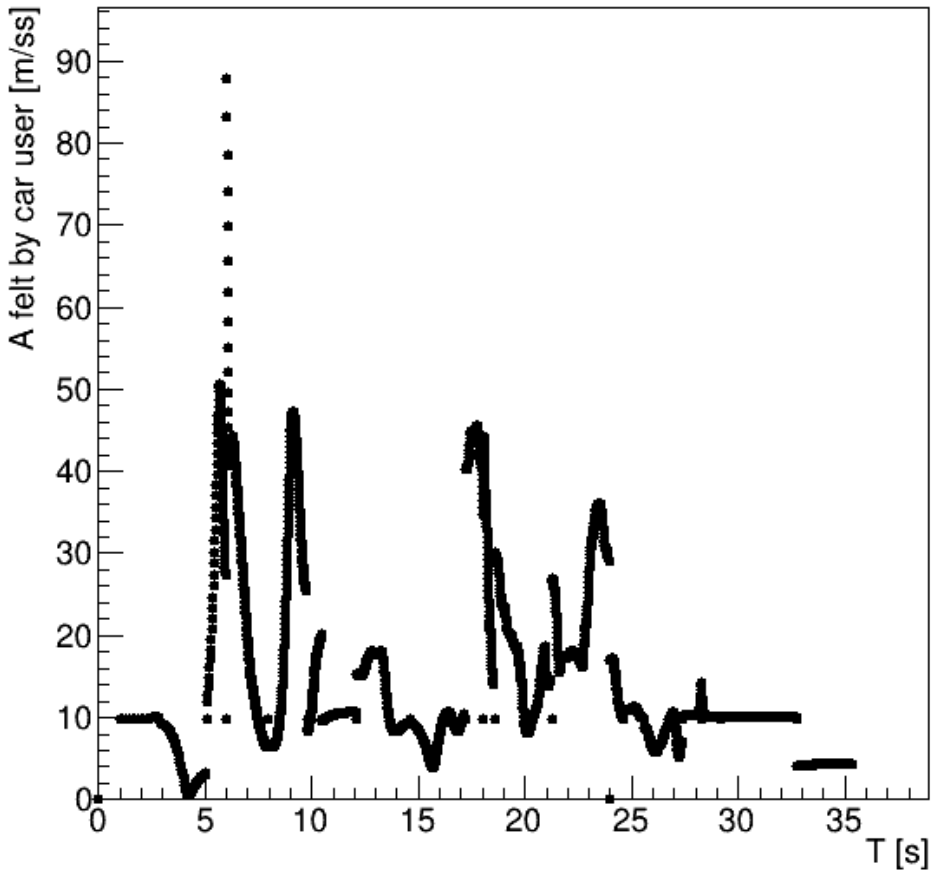

Figure 8: Acceleration-time graph of the roller coaster ride for the first set of coefficients

Our program

```
{
  int k = 0;

  double M = 1500.; // mass of roller coaster
  double g = 9.81; // acceleration of gravity
  const int N = 1e3; // number of iterations
  // double tauE = 100.; // ending curve parameter
  double alpha = 0.5; // aerodinamic friction coefficient
  double beta = 0.0005;
  // double Numberiter=10e5;

  const int ilev = 0; // number of cars
  double dist = 0.5; // distance between cars

  double tau0 = 0;
  double x0 = tabtor1[k].X->Eval(0);
  double y0_ = tabtor1[k].Y->Eval(0);
  double z0 = tabtor1[k].Z->Eval(0);
  double s0 = 0;
  double v0 = 20;
  double t0 = 0;
  double an0 = 0; // to do poprawki w zalenoci od toru
  double anwithg0[3] = {0, 0, g};

  struct dataline {
    double tau;
    double x;
    double y;
    double z;
    double s;
    double v;
    double t;
    double at;
    double an;
    double anwithg1;
    double anwithg2;
    double anwithg3;
  };
```

```
struct dataline Tab[N]; // our data in function of the
   ↪ parameter tau
struct dataline d0 {
  tau0, x0, y0_, z0, s0, v0, t0, an0, 0, 0, g
};
Tab[0] = d0;

double X[N];
double Y[N];
double Z[N];
double V[N];
double TAU[N];
double T[N];
double AT[N];
double AN[N];
double S[N];
double ANWITHG[N][3];
double ANGT[N];

T[0] = 0;

double tau = tabtor1[0].taub;

long double sigma_tau = 0;

for (int i = 0; i < Iranges; i++) {
  sigma_tau += tabtor1[i].taue - tabtor1[i].taub;
}

k = 0;
int i = 0;

double delta_x = 0, delta_y = 0, delta_z = 0;

double hx = 0; // tabtor1[k].X->Eval(tau);
double hy = 0; // tabtor1[k].Y->Eval(tau);
double hz = 0; // tabtor1[k].Z->Eval(tau);
double deg = tabtor1[0].deg;

while (k < Iranges) {
```

```
if (k < Iranges && tau > tabtor1[k].taue) {
  k++;

  if (k == Iranges) {
    n = i + 1;
    break;
  }

  tau = tabtor1[k].taub;

  hx = tabtor1[k].X->Eval(tau);
  hy = tabtor1[k].Y->Eval(tau);
  hz = tabtor1[k].Z->Eval(tau);
  deg = tabtor1[k].deg;

  double hx2 = hx;
  hx = hx * cos(deg) - hy * sin(deg);
  hy = hx2 * sin(deg) + hy * cos(deg);
  cout << cos(deg);

  delta_x = X[i] - hx;
  delta_y = Y[i] - hy;
  delta_z = Z[i] - hz;

  continue;
}

double tau1 = tau;

double x1 = tabtor1[k].X->Eval(tau);
double y1 = tabtor1[k].Y->Eval(tau);
double z1 = tabtor1[k].Z->Eval(tau);

double hx2 = x1;
x1 = x1 * cos(deg) - y1 * sin(deg);
y1 = hx2 * sin(deg) + y1 * cos(deg);
x1 += delta_x;
y1 += delta_y;
z1 += delta_z;

tau += sigma_tau / N;
```

```
i++;

if (i >= N)
  break;

double norm = sqrt((Tab[i - 1].x - x1) * (Tab[i - 1].x - x1
   ↪ ) +
                (Tab[i - 1].y - y1) * (Tab[i - 1].y - y1)
                   ↪ +
                (Tab[i - 1].z - z1) * (Tab[i - 1].z - z1))
                   ↪ ;
if (norm == 0)
  norm = 1;

double s1 = Tab[i - 1].s + norm;
double v1;

int nzcar[ilev];
for (int l = 0; l < ilev; l++) {
  nzcar[l] = 0;
}

if (s1 > ilev * dist) {
  for (int c = 1; c < ilev + 1; c++) {
    int j = i - 1;
    nzcar[c - 1] = j;
    // cout<<s1-Tab[j].s<<endl;

    while (s1 - Tab[nzcar[c - 1]].s < c * dist) {
      j--;
      nzcar[c - 1] = j;
    }
  }
}

double x0 = Tab[i - 1].x;
double y0 = Tab[i - 1].y;
double z0 = Tab[i - 1].z;
double anwithg1[3];
double x2 = 0;
double y2 = 0;
```

```
double z2 = 0;

if (i + 1 < N) {
  x2 = tabtor1[k].X->Eval((i + 1) * sigma_tau / N);
  y2 = tabtor1[k].Y->Eval((i + 1) * sigma_tau / N);
  z2 = tabtor1[k].Z->Eval((i + 1) * sigma_tau / N);
}

double q1 = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
   ↪ y2) +
                (z1 - z2) * (z1 - z2));
double q2 = sqrt((x2 - x0) * (x2 - x0) + (y2 - y0) * (y2 -
   ↪ y0) +
                (z2 - z0) * (z2 - z0));
double q3 = sqrt((x1 - x0) * (x1 - x0) + (y1 - y0) * (y1 -
   ↪ y0) +
                (z1 - z0) * (z1 - z0));
double p = (q1 + q2 + q3) / 2;

double P = sqrt(p * (p - q1) * (p - q2) * (p - q3));

double R = 0;
double an1;
if (P == 0) {
  an1 = 0;
}

else {
  R = q1 * q2 * q3 / (4 * P);
  an1 = Tab[i - 1].v * Tab[i - 1].v / R;
}

double b = 10;
double a = 0;

if (((x1 - x0) * (x2 - x0) + (y1 - y0) * (y2 - y0) +
     (z1 - z0) * (z2 - z0)) != 0)
  a = -b *
      ((x2 - x1) * (x2 - x0) + (y2 - y1) * (y2 - y0) +
       (z2 - z1) * (z2 - z0)) /
      ((x1 - x0) * (x2 - x0) + (y1 - y0) * (y2 - y0) +
```

```
      (z1 - z0) * (z2 - z0));

double anvector[3] = {a * (x1 - x0) + b * (x2 - x1),
                      a * (y1 - y0) + b * (y2 - y1),
                      a * (z1 - z0) + b * (z2 - z1)};
double normalization =
    sqrt(anvector[0] * anvector[0] + anvector[1] * anvector
       ↪ [1] +
        anvector[2] * anvector[2]);

double gvectortangent[3] = {g * (z1 - z0) * (x1 - x0) / (
   ↪ norm * norm),
                            g * (z1 - z0) * (y1 - y0) / (norm
                               ↪ * norm),
                            g * (z1 - z0) * (z1 - z0) / (norm
                               ↪ * norm)};
double gvectornormal[3] = {0 - gvectortangent[0], 0 -
   ↪ gvectortangent[1],
                           g - gvectortangent[2]};

if (normalization != 0) {
  anvector[0] = an1 * anvector[0] / normalization +
     ↪ gvectornormal[0];
  anvector[1] = an1 * anvector[1] / normalization +
     ↪ gvectornormal[1];
  anvector[2] = (an1 * anvector[2] / normalization +
     ↪ gvectornormal[2]);
} else {
  anvector[0] = gvectornormal[0];
  anvector[1] = gvectornormal[1];
  anvector[2] = gvectornormal[2];
}

double atangent =
    sqrt(anvector[0] * anvector[0] + anvector[1] * anvector
       ↪ [1] +
        anvector[2] * anvector[2]);

// cout<<atangent<<" "<<gvectornormal[2]<<endl;

double v1square = 0;
```

```
if (s1 < ilev * dist) {
  v1 = sqrt(Tab[i - 1].v * Tab[i - 1].v + g * 2 * (Tab[i -
     ↪ 1].z - z1) -
           2 * Tab[i - 1].v * Tab[i - 1].v * alpha * norm /
              ↪  M -
           2 * fabs(atangent * beta * norm));
}

else {
  double m = M / (ilev + 1);

  v1square =
      (g * 2 * m * (Tab[i - 1].z - z1) + M * Tab[i - 1].v *
         ↪ Tab[i - 1].v -
       2 * Tab[i - 1].v * Tab[i - 1].v * alpha * norm -
       2 * fabs(atangent * M * beta * norm)) /
      M;

  // cout<<i<<" "<<Tab[i-1].v<<" "<<v1square<<"
  // "<<Tab[i-1].v*alpha*norm-2*fabs(atangent*M*beta*norm)
     ↪ <<"\n";

  for (int c = 1; c < ilev + 1; c++) {
    v1square +=
        g * 2 * m * (Tab[nzcar[c - 1]].z - Tab[nzcar[c - 1]
           ↪ + 1].z) / M;
  }

  v1 = sqrt(v1square);
}

double t1 = Tab[i - 1].t + norm / v1;

struct dataline d1 = {tau1, x1, y1, z1, s1, v1,
                      t1, an1, anvector[0], anvector[1],
                         ↪ anvector[2]};
Tab[i] = d1;

V[i] = v1;
TAU[i] = tau1;
T[i] = t1;
```

```
  AN[i] = an1;
  S[i] = s1;
  X[i] = x1;
  Y[i] = y1;
  Z[i] = z1;
  ANWITHG[i][0] = anvector[0];
  ANWITHG[i][1] = anvector[1];
  ANWITHG[i][2] = anvector[2];
  ANGT[i] =
      sqrt(ANWITHG[i][0] * ANWITHG[i][0] + ANWITHG[i][1] *
         ↪ ANWITHG[i][1] +
          ANWITHG[i][2] * ANWITHG[i][2]);
}

auto c1 = new TCanvas("c1", "␣", 600, 600);
c1->cd();
auto *gr = new TGraph(N, T, ANGT);
gr->SetTitle();
gr->GetXaxis()->SetTitle("Time␣[s]");
gr->GetYaxis()->SetTitle("Normal␣acceleration␣[m/s^2]");

gr->SetMarkerSize(0.8);
gr->SetMarkerStyle(20);
gr->Draw("AP");
c1->Print("z1.png");

auto c2 = new TCanvas("c2", "␣", 600, 600);
c2->cd();
auto *gr2 = new TGraph(N, T, V);
gr2->SetTitle();
gr2->GetXaxis()->SetTitle("T");
gr2->GetYaxis()->SetTitle("V");

gr2->SetMarkerSize(0.8);
gr2->SetMarkerStyle(20);
gr2->Draw("AP");
c2->Print("z2.png");
cout << "Send␣" << S[N - 1] << "␣Time␣end␣" << T[N - 1] <<
   ↪ endl;

auto c3 = new TCanvas("c3", "␣", 600, 600);
```

```
  c3->cd();
  auto *gr3 = new TGraph(N, X, Z);
  gr3->SetTitle();
  gr3->GetXaxis()->SetTitle("X");
  gr3->GetYaxis()->SetTitle("Z");

  gr3->SetMarkerSize(0.8);
  gr3->SetMarkerStyle(20);
  gr3->Draw("AP");
  c3->Print("z3.png");

  TGraph2D graph2(N, X, Y, Z);

  graph2.Draw();

  cout << Tab[N - 1].t;
}
```

Code for track

```
  {
struct segment {
  TF1 *X; // cartesian coordinates as functions
  TF1 *Y;
  TF1 *Z;
  double taub; // begining curve parameter
  double taue; // ending curve parameter
  double deg;
};

const int Iranges = 20;
segment tabtor1[Iranges];

double pi = TMath::Pi();

TF1 sine("sine", "sin(x^2)");
TF1 cosi("cosi", "cos(x^2)");

//-----------------------------------------------
//---------START -------------------------------
//-----------------------------------------------

double limit = 0.88622;

TF1 start_z1("start_z1", "-50*sine.Integral(0,x)+30", 0,
   ↪ limit);
TF1 start_x1("start_x1", "12*cosi.Integral(0,x)", 0, limit);
// cout << start_z1.Eval();

TF1 start_y("start_y", "0");

segment start_first_half = {&start_x1, &start_y, &start_z1,
   ↪ 0, limit, 0};

TF1 start_z2("start_z2", "50*sine.Integral(0,␣0.88622-␣x)",
   ↪ 0, limit);
TF1 start_x2("start_x2", "12*-1*cosi.Integral(0,0.88622-␣x)",
   ↪  0, limit);
```

```
segment start_second_half = {&start_x2, &start_y, &start_z2,
   ↪ 0, limit, 0};

tabtor1[0] = start_first_half;
tabtor1[1] = start_second_half;

//-------------------------------------------
//---------loop ----------------------------
//-------------------------------------------

double obrot = -0.218669 / 2.55;
limit = 2.5066283;

TF1 sinee("sine", "sin(x^2/2)");
TF1 cosii("cosi", "cos(x^2/2)");

TF1 loop_z1("loop_z1", "12*sinee.Integral(0,x)", 0, limit);
TF1 loop_x1("loop_x1", "12*cosii.Integral(0,x)", 0, limit);

TF1 loop_y("loop_y", "x*5/(␣2.5066283+␣2.5066283)");

TF1 loop_z2("loop_z2", "12*sinee.Integral(0,␣␣2.5066283+␣
   ↪ 2.5066283-x)",
            limit, 2 * limit);
TF1 loop_x2("loop_x2",
            "-12*cosii.Integral(0,␣2.5066283+␣2.5066283-x)␣+␣
               ↪ "
            "2*loop_x1.Eval(loop_z1.GetMaximumX())",
            limit, 2 * limit);

segment segment1_loop, segment2_loop;
segment1_loop = {&loop_x1, &loop_y, &loop_z1, 0., 2.5066283,
   ↪ obrot};
segment2_loop = {
    &loop_x2, &loop_y, &loop_z2, 2.5066283, 2.5066283 +
       ↪ 2.5066283, obrot};
// segment tabtor1[2];
tabtor1[2] = segment1_loop;
tabtor1[3] = segment2_loop;

//-------------------------------------------
```

```
//---------turn -----------------------------
//-------------------------------------------

limit = 1.253314137;

TF1 curve_y1("curve_y1", "-13*sine.Integral(0,x)", 0, limit);
TF1 curve_x1("curve_x1", "10*cosi.Integral(0,x)", 0, limit);
// cout << curve_z1.Eval();

TF1 curve_z("curve_z", "0");

TF1 curve_y2("curve_y2",
            "13*(sine.Integral(0,␣1.253314137-␣(x␣-␣
               ↪ 1.253314137))␣+␣2*␣"
            "curve_y1.Eval(1.253314137))",
            limit, 2 * limit);
TF1 curve_x2("curve_x2", "30*cosi.Integral(0,1.253314137-(x␣-
   ↪ ␣1.253314137))",
            limit, 2 * limit);

/*TF1 curve_z2("curve_z2", "sine.Integral(0, x)", limit, 2 *
   ↪  limit);
   TF1 curve_x2("curve_x2",
            "-cosi.Integral(0, 2*0.88622-x) + "
            "curve_x1.Eval(curve_z1.GetMaximumX())",
            limit, 2 * limit);*/

segment segment1_curve, segment2_curve;
segment1_curve = {&curve_x1, &curve_y1, &curve_z, 0., limit,
   ↪ 0};
segment2_curve = {&curve_x2, &curve_y2, &curve_z, limit, 2 *
   ↪ limit, 0};

tabtor1[4] = segment1_curve;
tabtor1[5] = segment2_curve;

//-------------------------------------------
//---------loop2 -----------------------------
//-------------------------------------------

obrot = 0.218669 / 2.55;
```

```
limit = 2.5066283;

TF1 loop2_z1("loop2_z1", "14*sinee.Integral(0,x)", 0, limit);
TF1 loop2_x1("loop2_x1", "-14*cosii.Integral(0,x)", 0, limit)
   ↪ ;

TF1 loop2_y("loop2_y", "x*5/(␣2.5066283+␣2.5066283)");

TF1 loop2_z2("loop2_z2", "14*sinee.Integral(0,␣␣2.5066283+␣
   ↪ 2.5066283-x)",
            limit, 2 * limit);
TF1 loop2_x2("loop2_x2",
            "14*cosii.Integral(0,␣2.5066283+␣2.5066283-x)␣+␣
               ↪ "
            "2*loop2_x1.Eval(loop2_z1.GetMaximumX())",
            limit, 2 * limit);

segment segment1_loop2, segment2_loop2;
segment1_loop2 = {&loop2_x1, &loop2_y, &loop2_z1, 0.,
   ↪ 2.5066283, obrot};
segment2_loop2 = {
    &loop2_x2, &loop2_y, &loop2_z2, 2.5066283, 2.5066283 +
       ↪ 2.5066283,
    obrot};

tabtor1[6] = segment1_loop2;
tabtor1[7] = segment2_loop2;

//-------------------------------------------
//--------- ----------------------------
//-------------------------------------------

limit = 0.88622;

TF1 start2_z1("start2_z1", "-10*sine.Integral(0,x)+30", 0,
   ↪ limit);
TF1 start2_x1("start2_x1", "-20*cosi.Integral(0,x)", 0, limit
   ↪ );
// cout << start2_z1.Eval();

TF1 start2_y("start2_y", "0");
```

```
segment start2_first_half = {&start2_x1, &start2_y, &
   ↪ start2_z1, 0, limit, 0};

TF1 start2_z2("start2_z2", "10*sine.Integral(0,␣0.88622-␣x)",
   ↪  0, limit);
TF1 start2_x2("start2_x2", "-12*-1*cosi.Integral(0,0.88622-␣x
   ↪ )", 0, limit);

segment start2_second_half = {&start2_x2, &start2_y, &
   ↪ start2_z2, 0, limit, 0};

tabtor1[8] = start2_first_half;
tabtor1[9] = start2_second_half;

//-----------------------------------------
//---------loop3 ----------------------------
//-----------------------------------------

obrot = 0.218669 / 2.55 / 2;
limit = 2.5066283;

TF1 loop3_z1("loop3_z1", "12*sine.Integral(0,x)", 0, limit);
TF1 loop3_x1("loop3_x1", "-24*cosi.Integral(0,x)", 0, limit);

TF1 loop3_y("loop3_y", "x*5/(␣2.5066283+␣2.5066283)");

TF1 loop3_z2("loop3_z2", "12*sine.Integral(0,␣␣2.5066283+␣
   ↪ 2.5066283-x)",
            limit, 2 * limit);
TF1 loop3_x2("loop3_x2",
            "24*cosi.Integral(0,␣2.5066283+␣2.5066283-x)␣+␣"
            "2*loop3_x1.Eval(loop3_z1.GetMaximumX())",
            limit, 2 * limit);

segment segment1_loop3, segment2_loop3;
segment1_loop3 = {&loop3_x1, &loop3_y, &loop3_z1, 0.,
   ↪ 2.5066283, obrot};
segment2_loop3 = {
    &loop3_x2, &loop3_y, &loop3_z2, 2.5066283, 2.5066283 +
       ↪ 2.5066283,
```

```
    obrot};

tabtor1[10] = segment1_loop3;
tabtor1[11] = segment2_loop3;

//-----------------------------------------------
//---------hill -------------------------------
//-----------------------------------------------

limit = 0.88622;

TF1 hill1_z1("hill1_z1", "30*sine.Integral(0,x)+30", 0, limit
   ↪ );
TF1 hill1_x1("hill1_x1", "-12*cosi.Integral(0,x)", 0, limit);
// cout << hill1_z1.Eval();

TF1 hill1_y("hill1_y", "0");

segment hill1_first_half = {&hill1_x1, &hill1_y, &hill1_z1,
   ↪ 0, limit, 0};

TF1 hill1_z2("hill1_z2", "-30*sine.Integral(0,␣0.88622-␣x)",
   ↪ 0, limit);
TF1 hill1_x2("hill1_x2", "12*cosi.Integral(0,0.88622-␣x)", 0,
   ↪  limit);

segment hill1_second_half = {&hill1_x2, &hill1_y, &hill1_z2,
   ↪ 0, limit, 0};

tabtor1[12] = hill1_first_half;
tabtor1[13] = hill1_second_half;

limit = 0.88622;

TF1 hill2_z1("hill2_z1", "-40*sine.Integral(0,x)+30", 0,
   ↪ limit);
TF1 hill2_x1("hill2_x1", "-12*cosi.Integral(0,x)", 0, limit);
// cout << hill2_z1.Eval();

TF1 hill2_y("hill2_y", "0");
```

```
segment hill2_first_half = {&hill2_x1, &hill2_y, &hill2_z1,
   ↪ 0, limit, 0};

TF1 hill2_z2("hill2_z2", "30*sine.Integral(0,␣0.88622-␣x)",
   ↪ 0, limit);
TF1 hill2_x2("hill2_x2", "12*1*cosi.Integral(0,0.88622-␣x)",
   ↪ 0, limit);

segment hill2_second_half = {&hill2_x2, &hill2_y, &hill2_z2,
   ↪ 0, limit, 0};

tabtor1[14] = hill2_first_half;
tabtor1[15] = hill2_second_half;

//-----------------------------------------------
//---------turn around -------------------------------
//-----------------------------------------------

limit = 1.253314137;

TF1 curve2_y1("curve2_y1", "3.6*sine.Integral(0,x)", 0, limit
   ↪ );
TF1 curve2_x1("curve2_x1", "-20*cosi.Integral(0,x)", 0, limit
   ↪ );
// cout << curve2_z1.Eval();

TF1 curve2_z("curve2_z", "0");

TF1 curve2_y2("curve2_y2",
              "-4*(sine.Integral(0,␣1.253314137-␣(x␣-␣
                 ↪ 1.253314137))␣+␣2*␣"
              "curve2_y1.Eval(1.253314137))",
              limit, 2 * limit);
TF1 curve2_x2("curve2_x2",
              "-20*cosi.Integral(0,1.253314137-(x␣-␣
                 ↪ 1.253314137))", limit,
              2 * limit);

segment segment1_curve2, segment2_curve2;
segment1_curve2 = {&curve2_x1, &curve2_y1, &curve2_z, 0.,
   ↪ limit, 0};
```

```
  segment2_curve2 = {&curve2_x2, &curve2_y2, &curve2_z, limit,
     ↪ 2 * limit, 0};

  tabtor1[16] = segment1_curve2;
  tabtor1[17] = segment2_curve2;

  //-------------------------------------------
  //---------straight -----------------------------
  //-------------------------------------------
  limit = 80;

  TF1 line_y1("line_y1", "0", 0, limit);
  TF1 line_x1("line_x1", "x", 0, limit);

  TF1 line_z("line_z", "0");

  segment segment1_line;
  segment1_line = {&line_x1, &line_y1, &line_z, 0., limit, 0};
  tabtor1[18] = segment1_line;

  limit = 14.5;

  TF1 line2_y1("line2_y1", "0", 0, limit);
  TF1 line2_x1("line2_x1", "x", 0, limit);

  TF1 line2_z("line2_z", "30/13*x");

  segment segment1_line2;
  segment1_line2 = {&line2_x1, &line2_y1, &line2_z, 0., limit,
     ↪ 0};
  tabtor1[19] = segment1_line2;
}
```