mctracer
photon propagation
in complex geometries

mindset and how to use

by
Sebastian Achim Müller
sebmuell@phys.ethz.ch

Institute for Particle Physics
ETH Zurich
2015-2017

Contents

Chapter 1

Abstract

The mctracer is a simulation for geometrical optics. It can propagate photons in a complex 3D environment. The mctracer simulates reflection, refraction and absorbtion. It does not cover diffraction. For the investigation of optical devices or phenomena, mctracer records the full photon's trajectory starting with the production, through all the photon's interactions until its final absorbtion. A small set of primitiv surfaces is provided in mctracer to form simple optical devices, such as lenses, imaging mirrors, light concentrators and aperture stops. Further, complex objects can be simulated using triangular meshes. To produce photons, mctracer comes with a set of light sources to illuminate your scenery. For more complex light sources, photons can be read from external files. mctracer can handle very complex geometry while beeing fast and accurate on a scientific level. You can feed your geometry into mctracer using common CAD files for triangular meshes and a custom mctracer file which describes the primitives provided by mctracer itself. When the provided tools do not cover your demand you have the chance to implement the features yourself. The mctracer was originally created for simulations in Astro Particle Physics. Imaging Atmospheric Cherenkov Telescopes like FACT and the CTA MST made use of mctracer to investigate and improve their performance.

Chapter 2

How to set up a scenery in source code

We will build a little scenery of a house with a roof and chimney as well as a simple tree. Further we add a small telescope with a reflective imaging mirror. First we will define the geometry and their surfaces, second we will declare the relations between them. Third and finally we will update all frames relation w.r.t. the root frame to enable fast tracing (post initializing). First we define the main frame of our scenery. The main frame, often called world, will be the root of the scenery tree **??**.

```
Frame world;
world.set_name_pos_rot("World", Vec3::null, Rot3::null);
```

Second we define frames that hold individual structures like a tree which will be composed from several objects. The tree will be placed in $x = 5\,\mathrm{m}$ w.r.t. its later mother frame, i.e. the wolrd.

```
Vec3 tree_pos(5.0, 0.0, 0.0);
Frame* tree = world.append<Frame>();
tree->set_name_pos_rot("My_Tree", tree_pos, Rot3::null);

Color leaf_green(0, 128, 0);
Sphere* leaf_ball = tree->append<Sphere>();
leaf_ball->set_name_pos_rot("leaf_ball", Vec3(0.0, 0.0, 2.0), Rot3::null);
leaf_ball->set_outer_color(&leaf_green);
leaf_ball->set_radius(0.5);

Color wood_brown(64, 64, 0);
Cylinder* tree_pole = tree->append<Cylinder>();
tree_pole->set_name_pos_rot("tree_pole", Vec3(0.0, 0.0, 0.5), Rot3::null);
tree_pole->set_outer_color(&wood_brown);
tree_pole->set_radius_and_length(0.1, 1.0);
```

Also part of the tree is the wooden pole.

and the rest of the source...

Chapter 3

Atmospheric Cherenkov Plenoscope (ACP)

The mctracer can simulate ACPs. An ACP consists out of two main parts. First, an imaging system like e.g. a segmented imaging reflector as it is used for classic Imaging Atmospheric Cherenkov Telescopes (IACTs). Second, a light field sensor.

3.1 Create the ACP scenery

The scenery, with the ACP in it, is described in a folder. The folder must contain all the resources needed to describe the scenery of the ACP and its sourroundings.

3.1.1 create a scenery folder

3.1.2 create a scenery.xml file

3.1.3 copy all resources into the scenery folder

Chapter 4

1D functions

The Function::Func1D class provides 1D mapping for floating numbers.

$$y = f(x) \tag{4.1}$$

$$x \in X \tag{4.2}$$

All functions have limits which need to be respected. Any call of a function $f(x)$ with $x \notin X$ will throw an exception. We are strict about this behaviour to enforce that no propagation passes silently where e.g. your mirror's reflective index is only defined up to 600 nm but you shoot 800 nm photons onto it. Functions live in their own namespace.

4.1 Domains and their limits

First we define limits for the domains of our functions.

```
Limits limits(0.0, 1.0);
```

The limits here include the lower bound 0.0 and exclude the upper one 1.0. A limit can assert that a given argument is in its domain. If not, it will throw an exception.

```
EXPECT_THROW( limits.assert_contains(-0.1), Limits::OutOfRange );
EXPECT_NO_THROW( limits.assert_contains(0.0) );
EXPECT_NO_THROW( limits.assert_contains(0.5) );
EXPECT_THROW( limits.assert_contains(1.0), Limits::OutOfRange );
```

All functions have a domain within their limits. The limits are given to the funcions during construction.

```
Constant con(1.337, limits);
```

Functions assert, the argument to be inside their domain.

```
EXPECT_THROW( con(-0.1), Limits::OutOfRange );
EXPECT_NO_THROW( con(0.0) );
EXPECT_NO_THROW( con(0.5) );
EXPECT_THROW( con(1.0), Limits::OutOfRange );
```

4.2 Constant

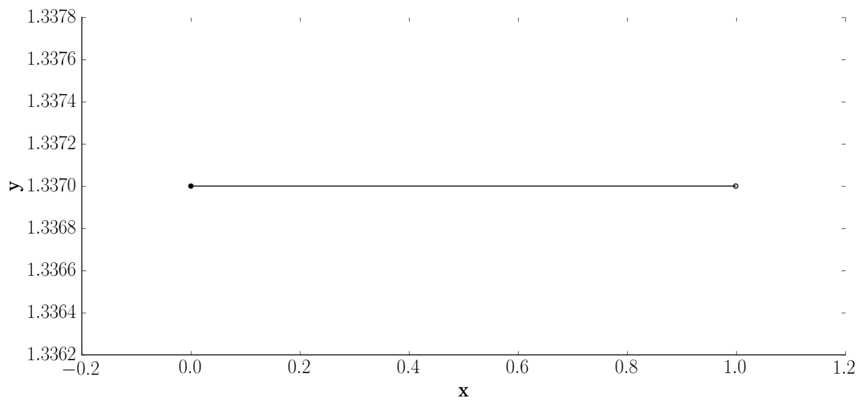
Sometimes it needs a constant function which will return the same value for any argument inside their domain limits.

$$y = f(x) = c \tag{4.3}$$

A constant function is created given its single constant value e.g. 1.337 and its domain limits.

```
Constant c(1.337, Limits(0.0, 1.0));
```

When called, within the limits, it will always return its constant value.



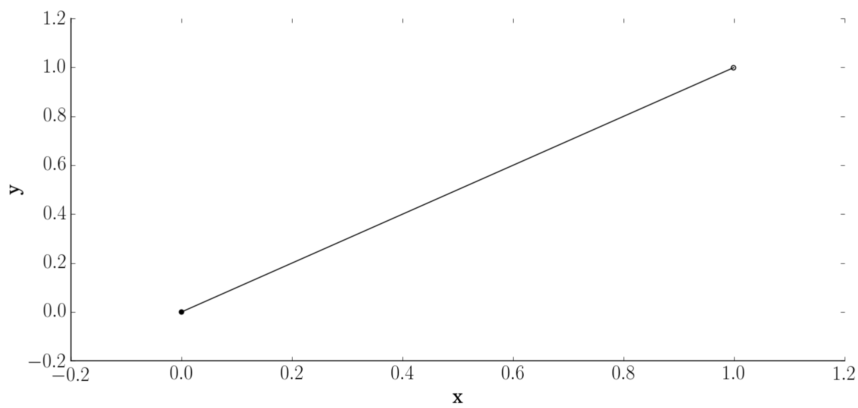
4.3 Polynom3

The versatile polynom to the power of 3 is defined by its four parameters a, b, c and d.

$$y = f(x) = ax^3 + bx^2 + cx^1 + dx^0 \tag{4.4}$$

We initialize the Polynom3 using a, b, c, d and the limits. By setting the higer orders to zero, we create e.g. a linear mapping.

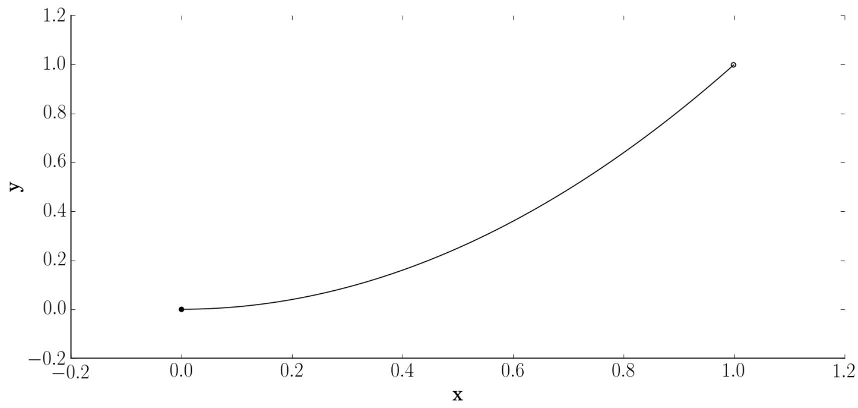
```
Polynom3 p3(0.0, 0.0, 1.0, 0.0, Limits(0.0, 1.0));
```



We can do a quadratic mapping.

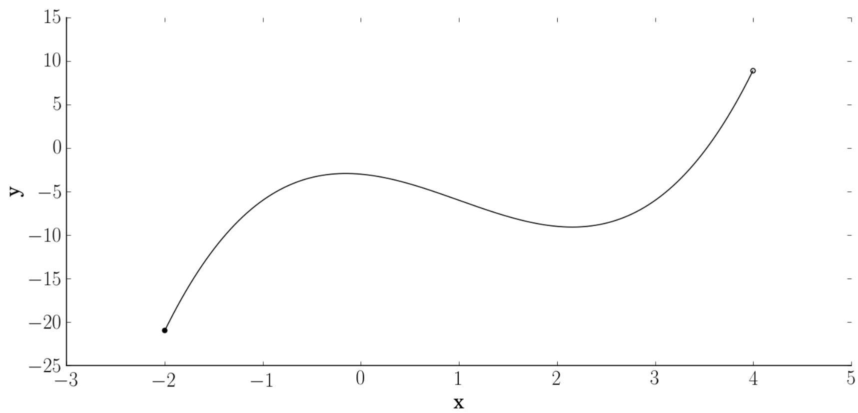
```
Polynom3 p3(0.0, 1.0, 0.0, 0.0, Limits(0.0, 1.0));

EXPECT_NEAR( 0.0, p3(0.0) ,1e-9);
EXPECT_NEAR( 0.25, p3(0.5) ,1e-9);
EXPECT_NEAR( 0.04, p3(0.2) ,1e-9);
```



The full polynom to the power of 3.

```
Polynom3 p3(1.0, -3.0, -1.0, -3.0, Limits(-2.0, 4.0));
```

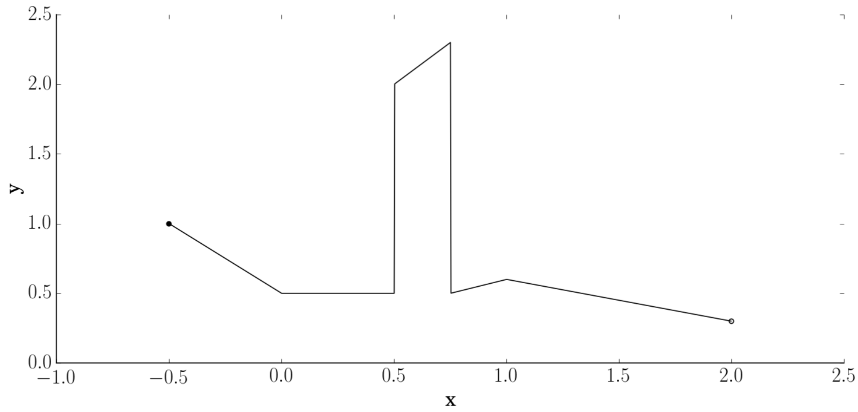


4.4 Linear interpolation look up table

In some cases, it can be tough to model an analytic $1D$ function. In these cases one can still use the a look up table with linear interpolation. The input table also defines the domain limits, so no limits have to be given during construction.

```
std::vector<std::vector<double>> table = {
    {-0.5, 1.0},
    { 0.0, 0.5},
    { 0.5, 0.5},
    { 0.50001, 2.0},
    { 0.75, 2.3},
    { 0.75001, 0.5},
    { 1.0, 0.6},
    { 2.0, 0.3}
};
```

```
LinInterpol ip(table);
```

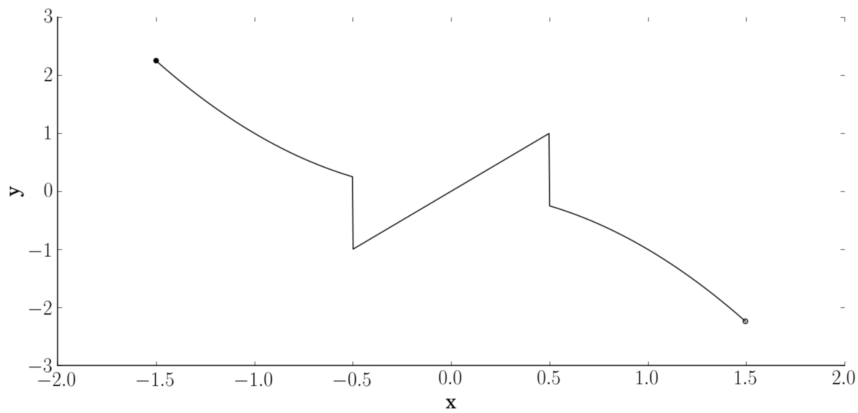


4.5 Concatenation

Functions can be concatenated when their domain limits match. The functions to be concatenated can be of any kind, even concatenated functions themselve. Since the concatenated function can deduce its domain limits from the input functions, no limit has to be given during construction.

```
Polynom3 f1(0.0, 1.0, 0.0, 0.0, Limits(-1.5, -0.5));
Polynom3 f2(0.0, 0.0, 2.0, 0.0, Limits(-0.5, 0.5));
Polynom3 f3(0.0, -1.0, 0.0, 0.0, Limits(0.5, 1.5));

std::vector<const Func1D*> funcs = {&f1, &f2, &f3};
Concat concat(funcs);
```



4.6 Access

Access to the values of a function is done using the bracket operator.

```
Polynom3 p3(1.0, -3.0, -1.0, -3.0, Limits(-2.0, 4.0));

double value1 = p3(-1.0);
double value2 = p3(-0.0);
```

Also a function can provide a table of both argument and value. The number of samples along the domain limits of the function can be specified.

```
    std::vector<std::vector<double>> table = p3.get_samples(1000);
    EXPECT_EQ(1000u, table.size());
```

Using the ascci io, a function can be exported into a text file.

```
    AsciiIo::write_table_to_file(p3.get_samples(7), "Examples/Out/my_p3.txt");
```

The output text file is a two column matrix. First column is the argument x, second is the function value $f(x)$.

```
-2 -21
-1.142857143 -7.268221574
-0.2857142857 -2.982507289
0.5714285714 -4.364431487
1.428571429 -7.635568513
2.285714286 -9.017492711
3.142857143 -4.731778426
```