

Mooneyes

PhD

submitted by

Sebastian Achim Müller

born in Dortmund Germany, April 5, 1988

sebastian8.mueller@tu-dortmund.de

Institute for Particle Physics

Chair Experimental Physics 5

Astro Particle Physics

ETH Zurich

2015-2017

1st Examiner	Prof. Dr. Felicitas Pauss
2nd Examiner	Prof. Dr. Adrian Biland
Submission date

Contents

Contents	I
1 abstract	1
2 The scenery tree	2
2.1 The root of the scenery tree	6
3 How to set up a scenery in source code	7
4 1D functions	8
4.1 Domains and their limits	8
4.2 Constant	9
4.3 Polynom3	9
4.4 Linear interpolation look up table	10
4.5 Concatenation	11
4.6 Access	11
References	12

Chapter 1

abstract

In γ ray and cosmic ray astronomy it needs dedicated simulations of detectors to develop and run the instruments which observe particle interactions far beyond any energy accessible in the lab. The Monte Carlo Tracer exists since we do what we must, because we can.

Chapter 2

The scenery tree

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2.1 The root of the scenery tree

The frame at the root of the tree structure represents the whole scenery. Before ray tracing is performed on the scenery tree, all frames in the tree estimate threir position and orientation w.r.t. the root frame. This way rays can easily and fast be transformed back and forth from the root tree to an individual object frame.

Chapter 3

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 2.1.

```
32 |   Frame world;  
33 |   world.set_name_pos_rot("World", Vector3D::null, Rotation3D::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 $x = 5\text{ m}$ w.r.t. its later mother frame, i.e. the world.

```
35 |   Vector3D tree_pos(5.0, 0.0, 0.0);  
36 |   Frame tree;  
37 |   tree.set_name_pos_rot("My_Tree", tree_pos, Rotation3D::null);  
38 |  
39 |   Color leaf_green(0, 128, 0);  
40 |   Sphere leaf_ball;  
41 |   leaf_ball.set_name_pos_rot("leaf_ball", Vector3D(0.0, 0.0, 2.0), Rotation3D::null);  
42 |   leaf_ball.set_outer_color(&leaf_green);  
43 |   leaf_ball.set_sphere_radius(0.5);  
44 |  
45 |   Color wood_brown(64, 64, 0);  
46 |   Cylinder tree_pole;  
47 |   tree_pole.set_name_pos_rot("tree_pole", Vector3D(0.0, 0.0, 0.5), Rotation3D::null);  
48 |   tree_pole.set_outer_color(&wood_brown);  
49 |   tree_pole.set_radius_and_length(0.1, 1.0);  
50 |  
51 |   tree.set_mother_and_child(&leaf_ball);  
52 |   tree.set_mother_and_child(&tree_pole);
```

Also part of the tree is the wooden pole.

and the rest of the source...

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.

```
14 | 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.

```
16 | EXPECT_THROW( limits.assert_contains(-0.1), Limits::OutOfRange );
17 | EXPECT_NO_THROW( limits.assert_contains(0.0) );
18 | EXPECT_NO_THROW( limits.assert_contains(0.5) );
19 | EXPECT_THROW( limits.assert_contains(1.0), Limits::OutOfRange );
```

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

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

Functions assert, the argument to be inside their domain.

```
23 | EXPECT_THROW( con(-0.1), Limits::OutOfRange );
24 | EXPECT_NO_THROW( con(0.0) );
25 | EXPECT_NO_THROW( con(0.5) );
26 | EXPECT_THROW( con(1.0), Limits::OutOfRange );
```

4.2 Constant

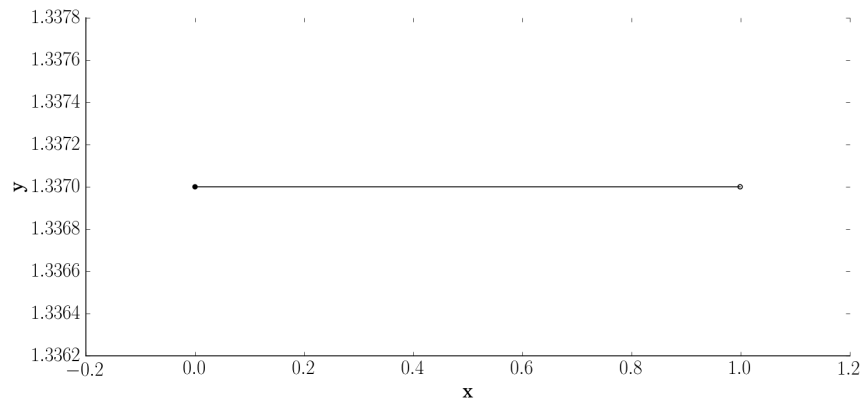
Sometimes it needs a constant function which will return the same value for any argument inside their domain limits.

$$y = f(x) = c \quad (4.3)$$

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

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

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



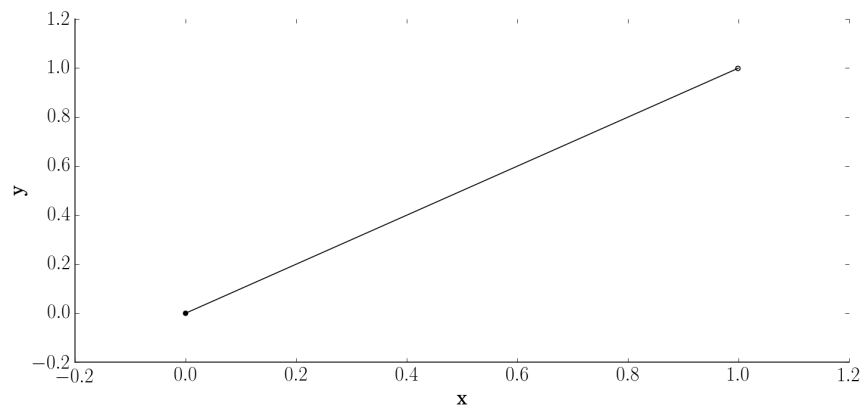
4.3 Polynom3

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

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

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

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

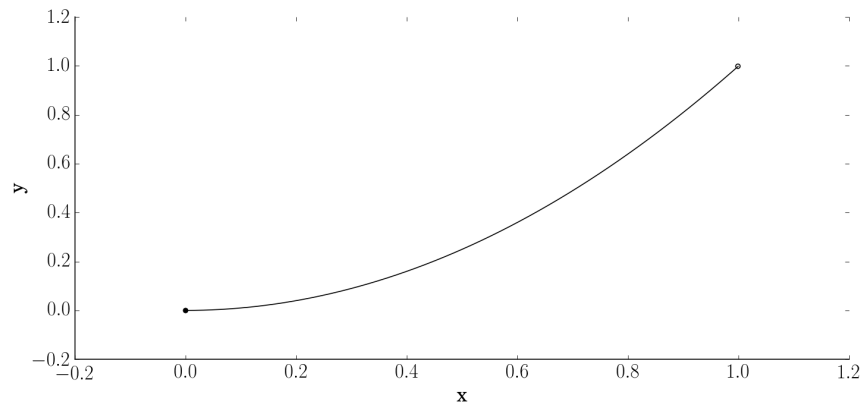


We can do a quadratic mapping.


```

61 | Polynom3 p3(0.0, 1.0, 0.0, 0.0, Limits(0.0, 1.0));
62 |
63 | EXPECT_NEAR( 0.0, p3(0.0) ,1e-9);
64 | EXPECT_NEAR( 0.25, p3(0.5) ,1e-9);
65 | EXPECT_NEAR( 0.04, p3(0.2) ,1e-9);

```

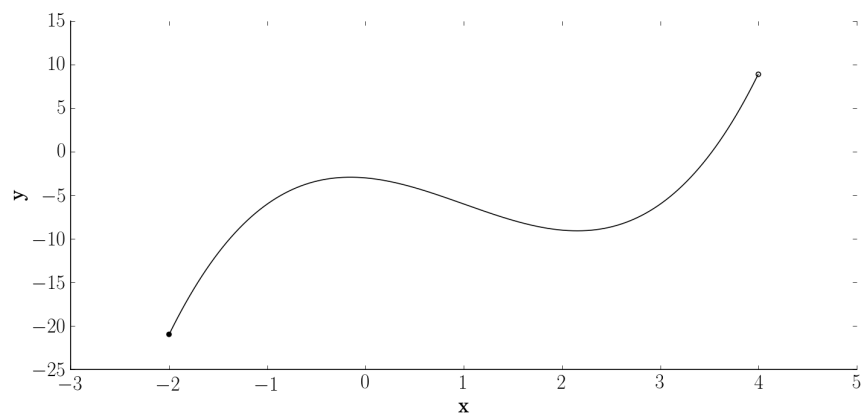


The full polynom to the power of 3.

```

73 | 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.

```

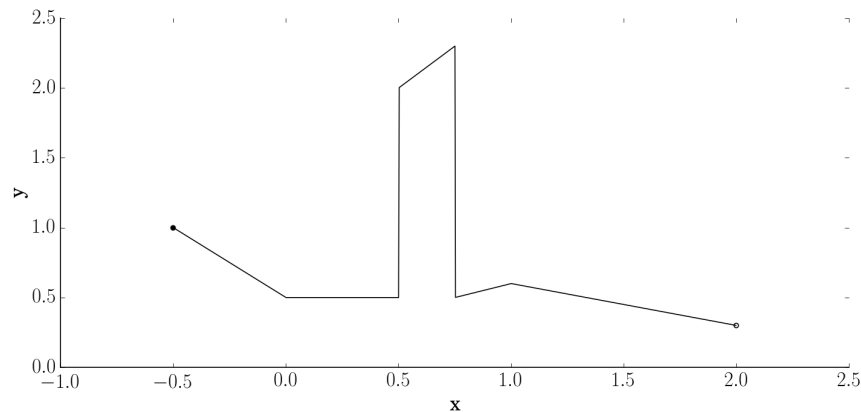
81 | std::vector<std::vector<double>> table = {
82 |     {-0.5, 1.0},
83 |     { 0.0, 0.5},
84 |     { 0.5, 0.5},
85 |     { 0.50001, 2.0},
86 |     { 0.75, 2.3},
87 |     { 0.75001, 0.5},
88 |     { 1.0, 0.6},
89 |     { 2.0, 0.3}
90 | };

```

```

91
92   LinInterpol ip(table);

```



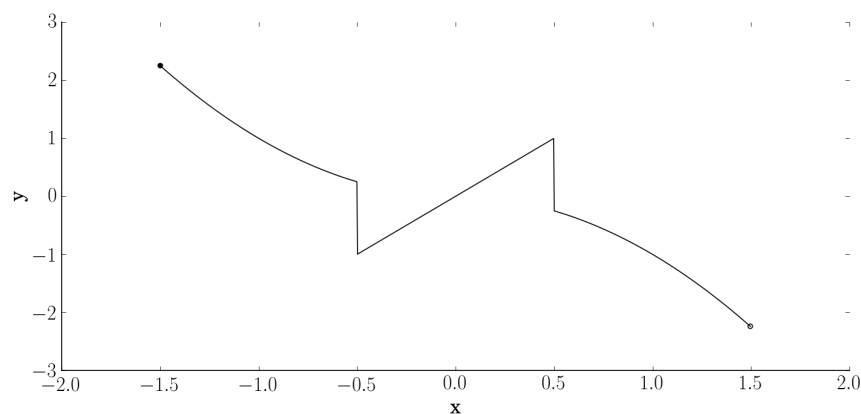
4.5 Concatenation

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

```

100   Polynom3 f1(0.0, 1.0, 0.0, 0.0, Limits(-1.5, -0.5));
101   Polynom3 f2(0.0, 0.0, 2.0, 0.0, Limits(-0.5, 0.5));
102   Polynom3 f3(0.0, -1.0, 0.0, 0.0, Limits(0.5, 1.5));
103
104   std::vector<const Func1D*> funcs = {&f1, &f2, &f3};
105   Concat concat(funcs);

```



4.6 Access

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

```

113   Polynom3 p3(1.0, -3.0, -1.0, -3.0, Limits(-2.0, 4.0));
114
115   double value1 = p3(-1.0);
116   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.

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

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

```
125 |     AsciiIo::write_table_to_file(p3.get_samples(7), "my_p3.txt");
```

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

```
1 | -2 -21  
2 | -1.142857143 -7.268221574  
3 | -0.2857142857 -2.982507289  
4 | 0.5714285714 -4.364431487  
5 | 1.428571429 -7.635568513  
6 | 2.285714286 -9.017492711  
7 | 3.142857143 -4.731778426
```