The Monte Carlo Tracer

Sebastian Achim Mueller[a,b]

[a] *Institute for Particle Physics, ETH, Otto-Stern-Weg 5, 8093 Zuerich, Switzerland*
[b] *Experimental Physics 5b, TU Dortmund, Otto-Hahn-Strasse 4, 44227 Dortmund, Germany*

Abstract

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

Keywords: ray tracing, photon propagation

1. The scenery tree

1.1. 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.

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 1.1.

*Sebastian Achim Mueller
Email address: sebmuell@phys.ethz.ch (Sebastian Achim Mueller)

```
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 $x = 5\,\mathrm{m}$ w.r.t. its later mother frame, i.e. the wolrd.

```
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...

3. Numerical functions

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

$$y = f(x) \tag{1}$$

$$x \in X \tag{2}$$

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.

```
83 using namespace Function;
```

3.1. Function limits

First we define limits to our functions.

```
89     Limits our_limits(0.0, 1.0);
```

The limits here include the lower bound 0.0 and exclude the upper one 1.0. Let's see the acceptance of our limits.

```
91     EXPECT_THROW( our_limits.assert_contains(-0.1), Limits::OutOfRange );
92     EXPECT_NO_THROW( our_limits.assert_contains(0.0) );
93     EXPECT_NO_THROW( our_limits.assert_contains(0.5) );
94     EXPECT_THROW( our_limits.assert_contains(1.0), Limits::OutOfRange );
```

When we create functions with our limits like

```
96     Constant our_const(1.337, our_limits);
```

the function will also show the restrictive access:

```
98     EXPECT_THROW( our_const(-0.1), Limits::OutOfRange );
99     EXPECT_NO_THROW( our_const(0.0) );
100    EXPECT_NO_THROW( our_const(0.5) );
101    EXPECT_THROW( our_const(1.0), Limits::OutOfRange );
```

3.2. Constant

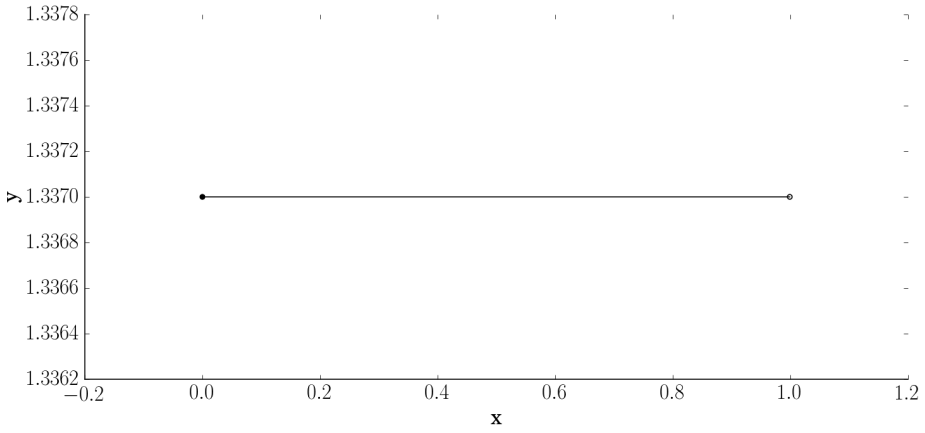
$$y \quad = \quad f(x) = c \tag{3}$$

A constant function takes its single constant value e.g. 1.337 and its limits.

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

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

```
110    EXPECT_EQ( 1.337, c(0.0) );
111    EXPECT_EQ( 1.337, c(0.2) );
112    EXPECT_EQ( 1.337, c(0.3) );
113    EXPECT_EQ( 1.337, c(0.43657657) );
114    EXPECT_EQ( 1.337, c(0.78) );
115    EXPECT_EQ( 1.337, c(0.9999) );
```



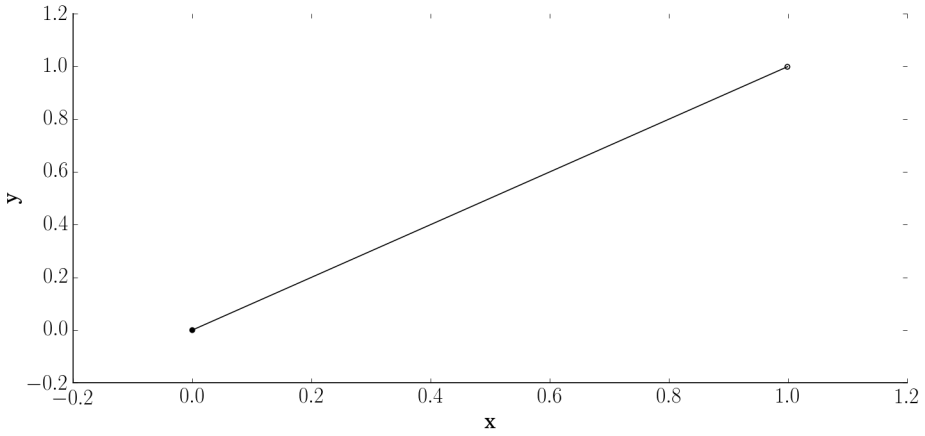
3.3. Polynom3

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

$$y \quad = \quad f(x) = ax^3 + bx^2 + cx^1 + dx^0 \tag{4}$$

We initialize it using a, b, c, d and the limits. Here we create a linear mapping.

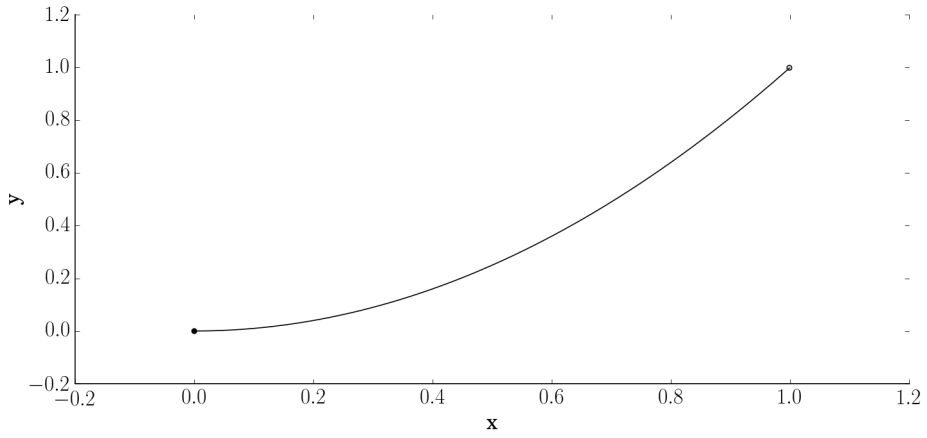
```
124 Polynom3 p3(0.0, 0.0, 1.0, 0.0, Limits(0.0, 1.0));
```



We can do a quadratic mapping:

```
136 Polynom3 p3(0.0, 1.0, 0.0, 0.0, Limits(0.0, 1.0));
137
```

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



3.4. Concatenation

```
148     Polynom3 f1(0.0, 1.0, 0.0, 0.0, Limits(-1.5, -0.5));
149     Polynom3 f2(0.0, 0.0, 2.0, 0.0, Limits(-0.5, 0.5));
150     Polynom3 f3(0.0, -1.0, 0.0, 0.0, Limits(0.5, 1.5));
151
152     std::vector<const Func1D*> funcs = {&f1, &f2, &f3};
153     Concat concat(funcs);
```