

On the quantum Schwarzschild gravitational theories consisting of field lines, for C++ programmers

S. Halayka*

Saturday 10th January, 2026 10:07

Abstract

This paper contains a short introduction to isotropic Schwarzschild gravitation. It is found that with non-normal (e.g. cosine weighted) messenger particle emission comes increased gravitational strength – matching that from the Schwarzschild solution. It is also found that there can be repulsive gravitation.

1 Introduction

First see [1] for a short tutorial for C++ programmers on isotropic Newtonian gravitation. In [1] we build an isotropic gravitational field through the use of pseudorandomly generated field lines. In [1] we use a sphere as the receiver.

In this paper, we find a match between the numerical gravitation and the gravitational time dilation from Schwarzschild's general relativity [2]. Here, we use an axis-aligned bounding box (AABB) as the receiver.

In this paper we use the natural Planck units, where $c = G = \hbar = k = 1$.

2 Method

Where r_e is the emitter's Schwarzschild radius, r_r is the receiver AABB radius (e.g. half of the AABB side length), and 1e11 and 0.01 are arbitrary constants:

$$r_e = \sqrt{\frac{1e11 \log(2)}{\pi}}, \quad (1)$$

$$r_r = r_e \times 0.01. \quad (2)$$

The event horizon area is:

$$A_e = 4\pi r_e^2. \quad (3)$$

The entropy (e.g. field line count) is:

$$n_e = \frac{A_e}{4 \log(2)} = 1e11. \quad (4)$$

*sjhalayka@gmail.com

Where R is the distance from the emitter's centre, the derivative is:

$$\alpha = \frac{\beta(R + \epsilon) - \beta(R)}{\epsilon}. \quad (5)$$

Here β is the get intersecting line density function. The gradient strength is:

$$g = \frac{-\alpha}{2r_r^3}. \quad (6)$$

From this we can get the Newtonian acceleration a_N , where $r_e \ll R$:

$$a_N = \frac{gR \log 2}{8M_e} = \sqrt{\frac{n_e \log 2}{4\pi R^4}} = \frac{M_e}{R^2}. \quad (7)$$

We can also get a general relativistic acceleration a_S , where $r_e < R$. In this case, the same gradient-based expression is applied, but is not restricted by the weak-field condition:

$$a_S = \frac{gR \log 2}{8M_e}. \quad (8)$$

At close proximity, where $r_e \approx R$, the metric produced is related to the Schwarzschild metric – curved space, curved time:

$$t = \sqrt{1 - \frac{r_e}{R}}, \quad (9)$$

$$\frac{\partial t}{\partial R} = \frac{r_e}{2tR^2}. \quad (10)$$

$$a_S \approx \frac{\partial t}{\partial R} \frac{2}{\pi} = \frac{r_e}{\pi t R^2}. \quad (11)$$

At far proximity, where $r_e \ll R$, $t \approx 1$, and $\frac{\partial t}{\partial R} \approx 0$, the metric produced is Newtonian – curved space, practically flat time.

In general relativity, acceleration is also dependent on the kinematic time dilation of the gravitated body – internal process forms a resistance to gravitation. Where the speed is $v \approx c$, like that for neutrinos, the gravitational attraction is twice of that predicted by Newtonian gravity because of a lack of said resistance.

3 C++ code

The following code uses the Schwarzschild gravitation, where the random unit vector generally points in a different direction than the normal, using cosine weighting:

```
// Beta function, for Schwarzschild gravitation
double get_intersecting_line_density(
    const long long unsigned int n,
    const double emitter_radius,
    const double receiver_distance,
    const double receiver_radius)
{
    double count = 0;

    generator.seed(static_cast<unsigned>(0));
```

```

52
53     for (long long unsigned int i = 0; i < n; i++)
54     {
55         vector_3 location = random_unit_vector();
56
57         location.x *= emitter_radius;
58         location.y *= emitter_radius;
59         location.z *= emitter_radius;
60
61         vector_3 surface_normal = location;
62         surface_normal.normalize();
63
64         vector_3 normal =
65             random_cosine_weighted_hemisphere(
66                 surface_normal);
67
68         std::optional<double> i_hit = intersect(
69             location, normal,
70             receiver_distance, receiver_radius);
71
72         if (i_hit)
73             count += *i_hit;
74     }
75
76     return count;
77 }

```

78 The cosine weighting is given by:

```

79 vector_3 random_cosine_weighted_hemisphere(const vector_3& normal)
80 {
81     double u1 = dis(generator);
82     double u2 = dis(generator);
83
84     double r = sqrt(u1);
85     double theta = 2.0 * pi * u2;
86
87     double x = r * cos(theta);
88     double y = r * sin(theta);
89
90     double z = sqrt(1.0 - u1);
91
92     vector_3 n = normal;
93     n.normalize();
94
95     vector_3 arbitrary;
96     if (fabs(n.x) > 0.9)
97         arbitrary = vector_3(0, 1, 0);
98     else
99         arbitrary = vector_3(1, 0, 0);
100
101     vector_3 tangent = n.cross(arbitrary);
102     tangent.normalize();
103
104     vector_3 bitangent = n.cross(tangent);
105     bitangent.normalize();
106

```

```

107     vector_3 result;
108     result.x = tangent.x * x +
109             bitangent.x * y + n.x * z;
110
111     result.y = tangent.y * x +
112             bitangent.y * y + n.y * z;
113
114     result.z = tangent.z * x +
115             bitangent.z * y + n.z * z;
116
117     return result.normalize();
118 }

```

119 To contrast, the following code uses the Newtonian gravitation, where the random unit vector
120 points in the same direction as the accompanying normal:

```

121 // Beta function, for Newtonian gravitation
122 double get_intersecting_line_density(
123     const long long unsigned int n,
124     const double emitter_radius,
125     const double receiver_distance,
126     const double receiver_radius)
127 {
128     double count = 0;
129
130     generator.seed(static_cast<unsigned>(0));
131
132     for (long long unsigned int i = 0; i < n; i++)
133     {
134         const vector_3 p = random_unit_vector();
135
136         vector_3 normal = p;
137         vector_3 location = normal;
138
139         location.x *= emitter_radius;
140         location.y *= emitter_radius;
141         location.z *= emitter_radius;
142
143         std::optional<double> i_hit = intersect(
144             location,
145             normal,
146             receiver_distance,
147             receiver_radius);
148
149         if (i_hit)
150             count += *i_hit;
151     }
152
153     return count;
154 }

```

155 4 Conclusion

156 It's worth noting that gravitation, if cosine weighted, is a diffuse process similar to electromagnetic
157 lighting in path tracing from computer graphics.

158 Experimentation with the receiver size is necessary to understand the intricacies of the code.
159 For instance, where $r_r = r_e \times 0.01$, it is found that gravitation can be repulsive.

160 The full code for this paper can be found at:

161 https://github.com/sjhalayka/schwarzschild_falloff_field_lines

162 It should be noted that repulsive gravitation plagues all non-normal (e.g. cosine weighted)
163 theories. It should also be noted that getting rid of the field lines altogether, by replacing them
164 with randomly emitted gravitons, does not solve the problem of repulsive gravitation.

165 5 Conflict of interest

166 There are no conflicts of interest to report.

167 6 Data availability

168 The C++ code is freely available, providing a way to reproduce the data.

169 References

170 [1] Halayka. Newtonian gravitation from scratch, for C++ programmers. (2024)

171 [2] ‘t Hooft. Dimensional reduction in quantum gravity. (1993)



Figure 1: This figure shows an axis-aligned bounding box and an isotropic emitter, looking from slightly above. An example field line (red) and intersecting line segment (green) are given. The bounding box is filled with these green intersecting line segments. It is the gradient of the density of these line segments that forms the gravitational acceleration. Note that the field line is normal to the surface.

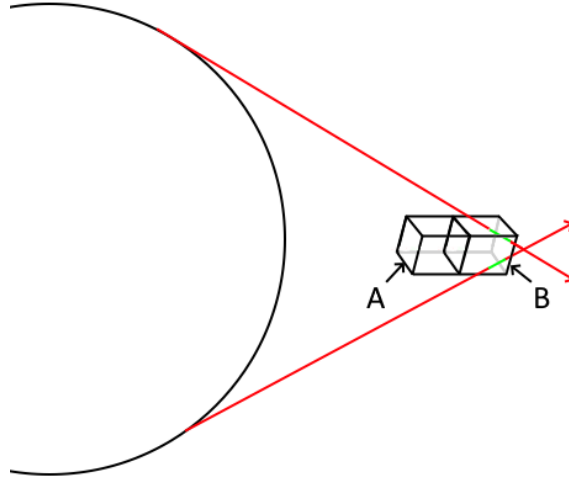


Figure 2: Axis-aligned bounding box A and box B are separated by the value epsilon $\epsilon = 2r_r$. Box B contains more field lines than box A, resulting in repulsive gravitation. Note that the field lines are not normal to the surface.

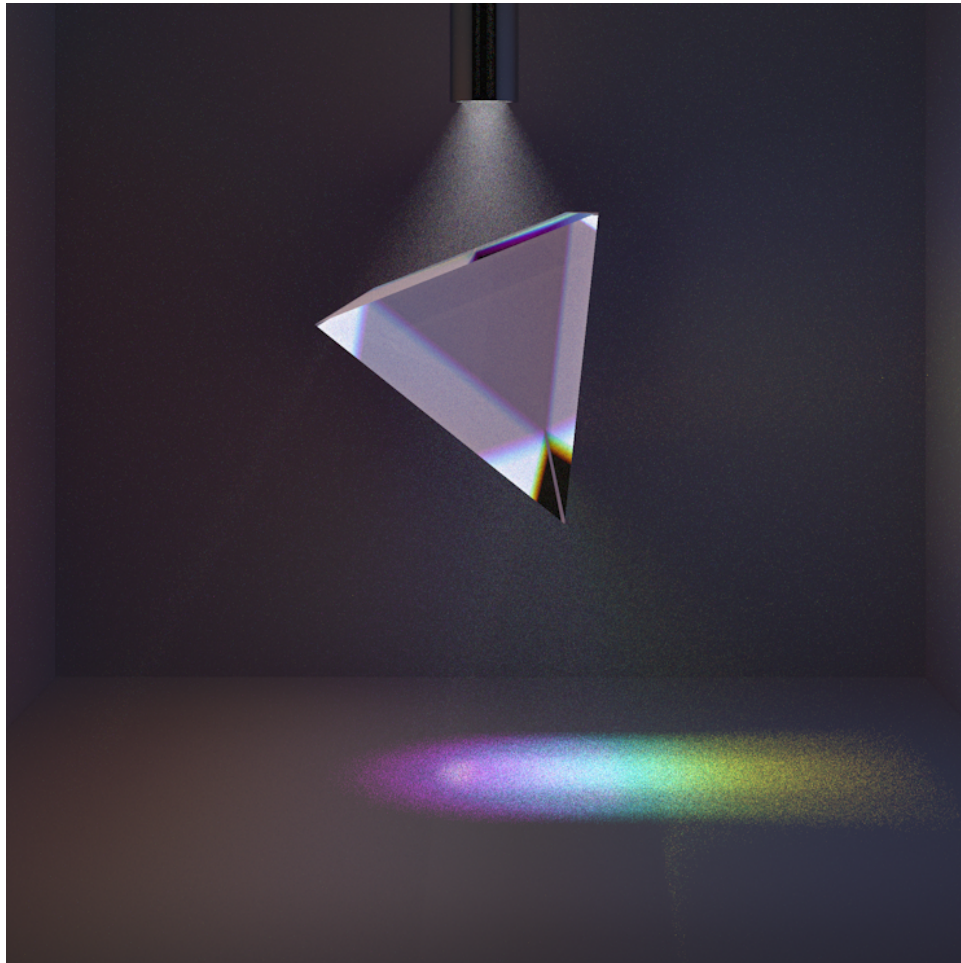


Figure 3: Backward path tracing in the Vulkan graphics API. The scene has a white barrel light that is dispersed by a prism into a spectrum of colours. There is some fog.