

Newtonian gravitation for C++ programmers

S. Halayka*

Wednesday 27th November, 2024 18:32

Abstract

...

1 Numerical: integer field line count

```
long long unsigned int get_intersecting_line_count(  
    const vector<vector_3>& unit_vectors ,  
    const vector_3 sphere_location ,  
    const real_type sphere_radius)  
{  
    long long unsigned int count = 0;  
  
    vector_3 cross_section_edge_dir(sphere_location.x, sphere_radius , 0);  
    cross_section_edge_dir.normalize();  
  
    vector_3 receiver_dir(sphere_location.x, 0, 0);  
    receiver_dir.normalize();  
  
    const real_type min_dot = cross_section_edge_dir.dot(receiver_dir);  
  
    for (size_t i = 0; i < unit_vectors.size(); i++)  
        if (unit_vectors[i].dot(receiver_dir) >= min_dot)  
            count++;  
  
    return count;  
}  
  
int main(int argc, char** argv)  
{  
    // Field line count  
    const size_t n = 1000000000;  
  
    cout << "Allocating memory for field lines" << endl;  
    vector<vector_3> unit_vectors(n);
```

*sjhalayka@gmail.com

```

for (size_t i = 0; i < n; i++)
{
    unit_vectors[i] = RandomUnitVector();

    static const size_t output_mod = 10000;

    if (i % output_mod == 0)
        cout << "Getting pseudorandom locations: "
        << static_cast<float>(i) / n << endl;
}

string filename = "newton.txt";
ofstream out_file(filename.c_str());
out_file << setprecision(30);

const real_type start_distance = 10.0;
const real_type end_distance = 100.0;
const size_t distance_res = 1000.0;

const real_type distance_step_size =
    (end_distance - start_distance)
    / (distance_res - 1);

for (size_t step_index = 0; step_index < distance_res; step_index++)
{
    const real_type r =
        start_distance +
        step_index * distance_step_size;

    const vector_3 receiver_pos(r, 0, 0);
    const real_type receiver_radius = 1.0;

    const real_type epsilon = 1.0;

    vector_3 receiver_pos_plus = receiver_pos;
    receiver_pos_plus.x += epsilon;

    const long long signed int collision_count_plus =
        get_intersecting_line_count(
            unit_vectors,
            receiver_pos_plus,
            receiver_radius);

    const long long signed int collision_count =
        get_intersecting_line_count(
            unit_vectors,
            receiver_pos,
            receiver_radius);

    const real_type gradient =
        static_cast<real_type>
        (collision_count_plus - collision_count)
        / epsilon;
}

```

```

        const real_type gradient_strength =
            -gradient
            / (receiver_radius * receiver_radius);

        cout << "r: " << r << " gradient strength: "
        << gradient_strength << endl;

        out_file << r << " " << gradient_strength << endl;
    }

    out_file.close();

    return 0;
}

```

2 Analytical: real field line count

```

real_type get_intersecting_line_count(
    const real_type n,
    const vector_3 sphere_location,
    const real_type sphere_radius)
{
    const real_type big_area =
        4 * pi * sphere_location.x * sphere_location.x;

    const real_type small_area =
        pi * sphere_radius * sphere_radius;

    const real_type ratio =
        small_area / big_area;

    return n * ratio;
}

int main(int argc, char** argv)
{
    const real_type emitter_radius = 1.0;

    const real_type emitter_area =
        4.0 * pi * emitter_radius * emitter_radius;

    // Field line count
    // re: holographic principle:
    const real_type n =
        (c3 * emitter_area)
        / (log(2.0) * 4.0 * G * hbar);

    const real_type emitter_mass = c2 * emitter_radius / (2.0 * G);

    // 1.73502e+70 is the 't Hooft–Susskind constant:
    // the number of field lines for a black hole of
    // unit Schwarzschild radius

```

```

//
//const real_type G_ =
//      (c3 * pi)
//      / (log(2.0) * hbar * 1.73502e+70);

const string filename = "newton.txt";
ofstream out_file(filename.c_str());
out_file << setprecision(30);

const real_type start_distance = 10.0;
const real_type end_distance = 100.0;
const size_t distance_res = 1000;

const real_type distance_step_size =
      (end_distance - start_distance)
      / (distance_res - 1);

for (size_t step_index = 0; step_index < distance_res; step_index++)
{
    const real_type r =
        start_distance + step_index * distance_step_size;

    const vector_3 receiver_pos(r, 0, 0);
    const real_type receiver_radius = 1.0;

    const real_type epsilon = 1.0;

    vector_3 receiver_pos_plus = receiver_pos;
    receiver_pos_plus.x += epsilon;

    // https://en.wikipedia.org/wiki/Directional\_derivative
    const real_type collision_count_plus =
        get_intersecting_line_count(
            n,
            receiver_pos_plus,
            receiver_radius);

    const real_type collision_count =
        get_intersecting_line_count(
            n,
            receiver_pos,
            receiver_radius);

    const real_type gradient =
        (collision_count_plus - collision_count)
        / epsilon;

    real_type gradient_strength =
        -gradient
        / (receiver_radius * receiver_radius);

    const real_type gradient_strength_ =
        n / (2.0 * pow(receiver_pos.x, 3.0));
}

```

```

    const real_type newton_strength =
        n * c * hbar * log(2.0)
        /
        (pow(receiver_pos.x, 2.0)
         * emitter_mass * 4.0 * pi);

    const real_type newton_strength_ =
        c4 * emitter_area
        / (16.0 * pi * G
         * pow(receiver_pos.x, 2.0) * emitter_mass);

    const real_type newton_strength_ =
        G * emitter_mass / pow(receiver_pos.x, 2.0);

    //cout << newton_strength_ / newton_strength << endl;

    cout << "r: " << r << " gradient strength: "
         << gradient_strength << endl;

    out_file << r << " " << gradient_strength << endl;
}

out_file.close();

return 0;
}

```