BPhO Computational Challenge 2023 Technical Report

Ademola Daboiku

Abstract

This report details the process of making a submission for the 2023 British Physics Olympiad Computational Challenge.

Introduction

1 Technologies used

In order to build my project, I used a combination of C++ and QML, a JavaScript-like language designed for creating graphical user interfaces (GUIs) with the Qt engine.

I mainly used C++ for all the data-intense and performance critical operations (e.g. integrating values and calculating orbits), focusing on creating optimised code while maintaining readability.

Before deciding to use the Qt engine to create the user interface (UI) for my project, I had considered several options. For example, the *Godot Engine* would allow me to create the UI in less time since it provided an intuitive visual editor. However, the lack of a graphing library with the features required for my project was the main factor in my decision not to use it.

Applications built using the Qt engine can be compiled for multiple platforms including PC, Android and WebAssembly. This allowed me to compile code to run natively on my computer for quick testing before compiling it to WebAssembly so it can be run in a web browser.

2 Task 5

Task 5 required evaluating the following equation:

$$t = P(1-\varepsilon^2)^{\frac{3}{2}} \frac{1}{2\pi} \int_{\theta_0}^{\theta} (1-\varepsilon\cos\theta)^{-2} d\theta$$

Where P is the period of the planet's orbit, ε is the eccentricity of the planet's orbit, and θ is the angle of the planet's orbit. This involved approximating a numerical solution using Simpson's rule.

2.1 1-D Interpolation

Task 5 involves plotting the value of θ against t for evenly spaced values of t. However, since the above gives t in terms of θ, linear interpolation is needed to approximate the value of θ for a given value of t. Since the array library used in my submission didn't provide any analogue to MATLAB's `interp1` function, this task required me to create a function that uses linear interpolation to approximate the value of a function $f(x)$ for a number of values of x given two arrays of corresponding values of $f(x)$ and x. I was also able to use the interpolation function in my solutions for Tasks 6 and 7.

3 Algorithms

3.1 Visvalingham-Whyatt

Sending data from the C++ back-end to the QML front-end proved to be a major bottleneck for the performance of the application, since data needed to be converted from xtensor-compatible format that allowed for quick processing to format that could be interpreted by QML. To reduce the performance impact of this I decided to implement the *Vivalingham-Whyatt* algorithm [1] to reduce the amount of data that needed to be transferred in order to produce an accurate visualisation of the data.

Before implementing this algorithm, in order to produce a graph of a function I had taken the naive approach of sampling the function at small, regular intervals. This was a simple approach that produced an accurate graph, however sampling at small intervals increased the amount of data that needed to be calculated, stored and plotted, leading to poor performance and even caused the app to crash at times - especially with task 5 where other expensive operations like integration and interpolation also needed to take place.

3.1.1 How it works

Given a set of points $P_1, P_2, \ldots, P_n$, the algorithm determines the "least important" point by calculating the area of the triangle with vertices P_{n-1}, P_n and P_{n+1} for every point P_n between the first and last point of the set. The point with the least area is selected and, if the area falls below a predefined tolerance, the point is removed. This process is repeated until no more points have an area that falls below the threshold.

3.1.2 Performance

The implementation of the Visvalingham-Whyatt algorithm in my submission calculates a threshold area based on the area of the rectangle covered by the curve so the algorithm performs similarly on curves with the same shape but different sizes without needing to use different threshold values for each curve. This is particularly useful for curves like the orbit paths generated by tasks 2, 3, and 4. However, as a result of this the percentage of points removed by the algorithm depends on how complex the curve is. The algorithm can remove up to 75% the points for smooth curves such as planetary orbits without a noticeable difference. However, for complex curves like the orbits generated by the Ptolemaic orbital model, the algorithm will have much a lower reduction in points without noticeable reduction in the quality of the curves produced.

4 Optimisations

4.1 Treating Vectors as Arrays

As mentioned previously, sending data between C++ and QML requires the data to be converted to a QML compatible format. Unfortunately, the only data types for collections of data items (numbers or vector2s) that QML accepts are dynamically sized vectors, which are often slower to append to as adding a new value may require the whole vector to be relocated in memory and can be less memory efficient as more space than is needed will often be allocated in case more values are added in the future. Since these vectors will always be created and then immediately passed to the QML front-end, the `reserve()` function can be used to manually set the size of the vector before populating it with value. This avoids the vector's data needing to be reallocated and any more memory than is needed being reserved.

5 Algorithm Implementations

5.1 Visvalingham-Whyatt

```cpp
xt::xarray<double> LineSimplify::vwReduce(xt::xarray<double> points, double epsilon)
{
    /*
    Implementation of the Visvalingham-Whyatt line simplification algorithm
    points - Nx2 array of points
    epsilon - minimum area for each triangle as a percentage of the total area.
              Controls the level of detail produced
    */
    xt::xarray<double> out = points;
    int length = points.shape(0);

    //Calculate area covered by the curve
    auto max = xt::amax(out, {0});
    auto min = xt::amin(out, {0});
    double epsArea = (max(0) - min(0)) * (max(1) - min(1)) * epsilon;

    while(length > 3)
    {
        //Store the index & area of the point with the lowest efective area
        int minIndex = -1;
        double minArea = epsArea;

        for (int i = 1; i < length - 1; i++)
        {
            //Calulate the area of the triangle
            double area = std::abs((out(i - 1, 0) - out(i + 1, 0)) * (out(i, 1) - out(i - 1, 1))
                                   - (out(i - 1, 0) - out(i, 0)) * (out(i + 1, 1) - out(i - 1, 1)))
                                   * 0.5;

            if (area < minArea)
            {
                minIndex = i;
                minArea = area;
            }
        }

        if (minIndex != -1)
        {
            //exclude the point
            out = xt::view(out, xt::drop(minIndex), xt::all());
            length -= 1;
        }
        else
        {
            break;
        }
    }
    return out;
}
```

This implementation of the Visvalingham-Whyatt algorithm uses `xt::view` functionality provided by the xtensor library to remove to remove points from the array `out`. This is also used in my solution for task 5 to select the even and odd numbered points in an array.

5.2 Simpson Integration

```cpp
xt::xarray<double> AngleIntegrator::integrate(double period, double ecc, int n)
{
    xt::xtensor<double, 1>::shape_type shape = {n * periods};
    auto interpolatedTheta = xt::xtensor<double, 1>::from_shape(shape);
    auto t = xt::xtensor<double, 1>::from_shape(shape);

    xt::xtensor<double, 1> theta = xt::arange(0.0, TAU, sampleSize);
    //calculate the integrand for each value of theta
    xt::xtensor<double, 1> integral = angleFunction(ecc, theta);

    // get all the even-numbered values (excluding the last value)
    auto evens = xt::view(integral, xt::range(2, integral.size() - 1, 2));
    // get all the odd-numbered values (excluding the fist & last value)
    auto odds = xt::view(integral, xt::range(1, integral.size() - 1, 2));
    evens *= 2;
```

```
    odds *= 4;

    // Calculate the cumulative sum to get the sum at each t
    integral = xt::cumsum(integral);

    //Calculate the value of t for each value of theta
    xt::xtensor<double, 1> calculatedTime = period * qPow(1 - SQUARED(ecc), 1.5) * 1 / (TAU * 3) *
    sampleSize * integral;
    //Generate n evenly spaced times to sample theta from
    xt::xtensor<double, 1> t = xt::linspace(xt::amin(calculatedTime), xt::amax(calculatedTime), n);
    xt::xtensor<double, 1> interpolatedTheta = interpolate(calculatedTime, theta, t);
    ...
```

This function also takes feature of the xtensor library: user-defined vectorised functions. Without using any loops, the integrand of the equation in task 5 can be calculated for a single eccentricity over a range of values of theta.

```
static auto angleFunction = xt::vectorize([](double ecc, double angle) -> double {
    return qPow(1 - ecc * qCos(angle), -2);
});
```

5.2.1 Interpolation Function

```
xt::xtensor<double, 1> AngleIntegrator::interpolate(
    xt::xtensor<double, 1> &x,
    xt::xtensor<double, 1> &y,
    xt::xtensor<double, 1> &samplePoints
)
{
    xt::xtensor<double, 1> out = xt::empty_like(samplePoints);

    for (int i = 0; i < samplePoints.size(); i++)
    {
        // Get the first number greater than the sample value
        auto upperBound = x.begin();
        while (*upperBound < samplePoints[i] && upperBound != x.end())
        {
            upperBound++;
        }

        if (upperBound == x.end())
        {
            out[i] = *upperBound;
        }
        else if (upperBound == x.begin())
        {
            out[i] = *upperBound;
        }
        else
        {
            // Get the last number less than the sample value
            auto lowerBound = upperBound; lowerBound--;
            // Get floating-point index of the value
            int index = std::distance(x.begin(), lowerBound);
            double remainder = (samplePoints[i] - *lowerBound) / (*upperBound - *lowerBound);
            // Interpolate between the lower and upper bound based on the remainder
            out[i] = y[index] + (y[qCeil(index + remainder)] - y[index]) * (remainder);
        }

    }
    return out;
}
```

References

[1] J. D Whyatt M. Visvalingham. *Line generalisation by repeated elimination of the smallest area.* Tech. rep. University of Hull, 1992.