# Technical Report for KimboBPhO

Jamie Whiting, Nicholas Gregory

## Abstract

This report details the technical considerations and implementations that went into creating a website submission for the British Physics Olympiad Computational Challenge 2022, as laid out in (French, 2022) - particularly our approach to Task 3.

## Introduction

### Our aims for the project

As the 3 tasks prescribed for the challenge were straightforward, we decided to lean heavily into the suggested extensions and give ourself a technical challenge: write a fully cross-platform app that makes minimal performance concessions. We achieved this goal through our use of Progressive Web App and WebAssembly technology.

### How we wrote our code

We wrote our web app using TypeScript, with a WebAssembly module written in Rust to handle the main intensive computations. In our TypeScript code we placed an emphasis on clean, maintainable and expandable code, whilst in our Rust code we focused purely on performance optimisation, forgoing certain niceties if it resulted in a slower compiled binary, even in cases where it resulted in verbose code.

### What this is not

This report does not seek to motivate or derive what is already covered in (French, 2022), as we do not see much value in senselessly copying Andrew French's work in order to extend our word count. In a similar vein, we do not intend to explain Euler's method or Runge-Kutta's 4th Order method, because we consider these *common knowledge* - in particular, knowledge shared by members of a particular field (MIT, n.d.) - and as such will not provide citations for them.

## 1 Progressive Web App

Upon considering the best graphical user interface to use, we kept in mind our goal of accessibility unto the largest number of people possible. We debated creating an iOS/Android App, a desktop application and a web application, and eventually decided on what we believe is the perfect combination of all three - a Progressive Web App. This shortened our development time by only having to create one application, whilst being installable as both a mobile and desktop app for users that so wish to use our software offline without an internet connection. Moreover, anyone can access our software without downloading anything just by visiting the website - `https://bpho.kim/`. Our site is blazingly fast to load, and the addition of offline caching with service workers only aids to further ameliorate this. We were very pleased with how this choice worked out, and it is definitely a technology we would use again in the appropriate circumstance.

## 2 Task 3

### 2.1 The Problem

Task 3 required us to solve the following set of equations:

$$\frac{dT}{dh} = -L(r, T_K)$$
$$\frac{dP}{dh} = -\frac{34.171}{T_K}\left(P - 0.37776 U E_S(T)\right)$$

Given

$$L(r, T_K) = 9.7734\frac{1 + 5420.3\frac{r}{T_K}}{1 + 8400955.5\frac{r}{T_K^2}}$$
$$r(P, T) = \frac{U E_S(T)}{P - U E_S(T)}$$
$$E_S(T) = 6.1121 e^{\left(18.678 - \frac{T}{234.5}\right)\left(\frac{T}{T + 257.14}\right)}$$

where $T_K$ is temperature in Kelvin, and $T$ is temperature in °C and for some parameter $U \in [0, 1]$, given the initial conditions $h_0 = 0$km, $P_0 = 1013.25$mbar and $T_0 = 15$°C, with step size $\Delta h = 0.01$km, up to $h_1 = 11$km.

### 2.2 Euler vs Runge-Kutta

See (French, 2022) for an explanation of how Euler's method is implemented, and see Appendix A for an example implementation.

The main reason for choosing Runge-Kutta 4th Order (RK4) over Euler's method is that, while Euler's method has global error $O(h)$, where $h$ is the step size, RK4 has global error $O(h^4)$, meaning that with $h = 0.1$, RK4 achieves a more accurate solution than Euler's method with $h = 0.01$. Conducting benchmarks, we were able to conclude that in our use case RK4 is over twice as fast as Euler's method, with an average runtime of 31 microseconds, compared to Euler's 78 microseconds, over 100,000 runs. Note this benchmark included the point pruning and data transform into WebAssembly-compatible format described later, as this gives an actual view of the practical benefit of using RK4, and as a result the raw compute times for the methods will be lower than the given values, and may not differ by the same factor as these results. The reduced step-size also has the effect of increasing memory efficiency, as only a tenth of the space is required to store the solution data. However, this advantage is rendered moot by storing only every tenth point in Euler's method.

## 3 Ramer-Douglas-Peucker

### 3.1 Motivation

Despite our extremely fast calculations, we found our most major bottleneck was simply plotting 111 points for each of 5 plots - 555 points total - using the JavaScript graphing library recharts, as the re-render time was unacceptably long, and while dragging sliders there was a noticeable latency. Unwilling to refactor all of our code with a faster library (which may still have not been enough), we

implemented the Ramer-Douglas-Peucker (RDP) algorithm (Ramer, 1972) into our WebAssebly module, pruning anywhere from 80% to 95% of points, and - after optimising it - incurring a runtime cost of *less than one microsecond* to our original module code, whilst resulting in a seamless dynamic graph experience that still maintains perfect visual clarity.

### 3.2 How it works

The RDP algorithm works on a set of points $P_1, P_2, ..., P_n$, by taking the the straight line $P_1P_n$ between the first and last points, $P_1$ and $P_n$, and finding the point in the collection that is furthest away from it, $Q$. If this farthest point is less than a predefined tolerance then just $P_1$ and $P_n$ are returned, but otherwise, if the tolerance is exceeded, then $Q$ is included in the return, and the algorithm recursively calls itself on the the sub-sets $P_1, ..., Q$ and $Q, ..., P_n$ to add more points by the same process where relevant. See appendix B for an implementation that takes two slices, one of $x$ coordinates and one of $y$ coordinates, rather than the usual single slice of points.

### 3.3 Optimisations

**Pre-Pruning** As already noted in section 2.2, we store only every tenth iteration of Euler's method, giving an output step size of $h = 0.1$, the same as RK4. This has an additional benefit to those already described, as RDP runs in $O(n^2)$ time in the worst case, meaning that operating on ten times the points results in the pruning taking one hundred times longer. Currently a run of RDP in our hyper-optimised implementation takes less than a microsecond on average, on our 111 step solutions, however neglecting this pre-pruning for Euler's method would cause it to take up to 100 microseconds, longer than the method calculation takes itself.

**Indexing** The sample wikipedia implementation uses list slicing in its recursive calls, but this is massively inefficient compared to passing indexes. This is because it allows us to simply pass a reference to the original solution array, which requires no memory allocations or copies whilst generating slices.

**Arrays** We know that our solution arrays will always have a length of 111, meaning the RDP algorithm will return at most a list of 111 points, and so we can initialise an array of 111 points, and simply fill them as we go, increasing a 'count' variable as we go to index into the array (as well as to return at the end). This avoids the **significant** problem of re-allocating memory when using dynamically sized objects such as vectors, and allows a truly extraordinary performance boost, one only available in low level languages like Rust.

## 4 WebAssembly

### 4.1 How memory works in WASM

WebAssembly executes code in a sandboxed environment, and works with *linear memory*, which is just a mutable contiguous array of raw bytes. As the WebAssembly standard is new, there is no official support for returning many types, most notably nested arrays, which is rather inconvenient when trying to return a list of 2d points, for 5 different plots. Due to this, we had to think of an efficient format to return our data in.

### 4.2 Other considerations

Firstly, WebAssembly only has a 32-bit runtime, and JavaScript, while technically 64-bit, handles 64-bit values by converting them to two 32-bit values. Because of this we used singe-precision, 32-bit floating point values to represent our data (we do not need the extra precision anyway) to avoid needless overhead.

Additionally, WebAssembly does not currently allow for the returning of arrays, instead one must return a raw pointer to the memory address of the array and index into wasm's allocated memory to retrieve values. Thankfully this can be avoided via Rust's *Box* type, a pointer type that abstracts away most of the hassle of dealing with raw pointers and potentially unsafe accessing of data. In this case we return a boxed slice, Box<[f32]>.

### 4.3 Our memory format

After RDP pruning of points, every variable we are tracking has a different set of alititudes, and a different number of points. To allow our TypeScript to correctly index into our slice we must return not only the coordinates but also the number of points, and we settled on the following form:

[num points, ...h vals..., ...variable values...]

And the final return format is simply this form for all the variables concatenated together. This allows us to perfectly index into the slice because we can read the first value (at index zero), $n_1$, extract the $2n_1$ values ($n_1$ altitudes and corresponding variable values), and then read the value at index $2n_1 + 1$ to get the number of points for the next variable, and repeat until all variables have been extracted (we know there are 5 and the order we return them in).

## References


French, A. (2022). A standard atmosphere. Retrieved from `https://www.bpho.org.uk/bpho/computational-challenge/`

MIT. (n.d.). What is common knowledge? *Academic Integrity at MIT*. Retrieved from `https://integrity.mit.edu/handbook/citing-your-sources/what-common-knowledge`

Ramer, U. (1972). An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, *1*(3), 244-256. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0146664X72800170` doi: https://doi.org/10.1016/S0146-664X(72)80017-0

# A Euler Scheme Code

The following declarations are used extensively through both our Euler and RK4 schemes.

```
// step size for euler
const DHE: f32 = 0.01;
// for celcius <-> kelvin
const KELVIN: f32 = 273.15;
// August-Roche-Magnus approximation constants
const A: f32 = 17.625;
const B: f32 = 243.04;

// calculating relative humidity
#[inline(always)]
fn calc_es(t: f32) -> f32 {
    6.1121 * ((18.678 - t / 234.5) * (t / (t +
    257.14))).exp()
}
// calculating dp/dh
fn calc_dp(p: f32, ues: f32, tk: f32) -> f32 {
    -34.171 * (p - 0.37776 * ues) / tk
}
// calculating lapse rate
fn calc_l(p: f32, ues: f32, tk: f32) -> f32 {
    let r = ues / (p - ues);
    9.7734 * (tk + 5420.3 * r) / (tk * tk +
    8400955.5 * r) * tk
}
// calculating dew-point temperature
fn calc_dew(t: f32, u: f32) -> f32 {
    B * (u.ln() + A * t / (B + t)) / (A - u.ln() -
     A * t / (B + t))
}
// calculating boiling temperature
fn calc_boil(p: f32) -> f32 {
    1.0 / (1.0 / (100.0 + KELVIN) - 8.314 /
    45070.0 * (p / 1013.25).ln()) - KELVIN
}
```

As already detailed, we only store every tenth point from Euler's method, hence the inner loop of length 10. To ensure locality, a single solution 'soln' array is initialised, and to prevent uselessly re-allocating memory the auxiliary variables are declared immediately. We return lists of the values, without accompanying altitudes because we know the altitude exactly based on the index.

```
// can return array here as only used internally
pub fn euler(u: f32, p0: f32, t0: f32) -> [[f32;
    111]; 5] {
    // initialising variables
    let mut soln = [[0.0; 111]; 5];
    let (mut t1, mut ues1, mut tk1, mut tm, mut pm
    , mut lm): (f32, f32, f32, f32, f32, f32);
    // initial conditions
    soln[0][0] = p0;
    soln[1][0] = t0;
    soln[2][0] = calc_l(p0, u * calc_es(t0), t0 +
    KELVIN);
    soln[3][0] = calc_dew(t0, u);
    soln[4][0] = calc_boil(p0);
    // stepping through soln array
    for i in 1..111 {
        // initialising values for inner recursion
        pm = soln[0][i - 1];
        tm = soln[1][i - 1];
        lm = soln[2][i - 1];
        // 10 steps per big step
        for _ in 0..10 {
            t1 = tm - lm * DHE;
            ues1 = u * calc_es(t1);
            tk1 = t1 + KELVIN;
            tm = t1;
            pm = pm + DHE * calc_dp(pm, ues1, tk1)
    ;
            lm = calc_l(pm, ues1, tk1);
        }
        // writing values to soln
        soln[0][i] = pm;
        soln[1][i] = tm;
        soln[2][i] = lm;
        soln[3][i] = calc_dew(tm, u);
        soln[4][i] = calc_boil(pm);
    }
    // returning soln
    soln
}
```

# B Ramer-Douglas-Peucker Code

This code is split into a root function and a step function, where the root calls the step, which then recursively calls itself until it terminates. Additionally this code takes separate $x, y$ coordinate lists, due to our scheme code, seen in appendix A, returning a set of value lists for each variable - this is in contrast to the typical list of points used.

```
pub fn ramer_douglas_peucker(x: &[f32], y: &[f32],
     epsilon: f32) -> Vec<f32> {
    let last = 110;
    // initialises arrays with length 111, as we
    know that is the length they will have
    let mut xnew = [0.0; 111];
    let mut ynew = [0.0; 111];
    // adds starting point for line
    xnew[0] = x[0];
    ynew[0] = y[0];
    // number of points count
    let mut count: usize = 1;
    // adds points in between first and last
    points
    rdp_step(x, y, 0, last, epsilon, &mut xnew, &
    mut ynew, &mut count);
    // adds ending point for line
    xnew[count] = x[last];
    ynew[count] = y[last];
    count += 1;
    [&[count as f32], &xnew[..count], &ynew[..
    count]].concat()
}
```

```
fn rdp_step(
    x: &[f32], y: &[f32],
    first: usize, last: usize,
    epsilon: f32,
    xnew: &mut [f32], ynew: &mut [f32],
    count: &mut usize,
) {
    // declaring variables
    let mut dmax: f32 = 0.0;
    let mut idx: usize = 0;
    let mut d: f32;
    let m = (y[first] - y[last]) / (x[first] - x[
    last]);
    let c = y[first] - m * x[first];
    let modulus = (1.0 + m.powi(2)).sqrt();
    // finding index of point furthest from the
    line between first and last points
    for i in (first + 1)..last {
        d = (y[i] - m * x[i] - c).abs() / modulus;
        if d > dmax {
            idx = i;
            dmax = d;
        }
    }
    // if distance of furthest point exceeds
    threshold, add it to the list to be returned
    if dmax > epsilon {
        // maintaining order of points here
        // adds points between first and furthest
        if idx > first + 1 {
            rdp_step(x, y, first, idx, epsilon,
    xnew, ynew, count)
        }
        // adds furthest point to list
        xnew[*count] = x[idx];
        ynew[*count] = y[idx];
        // increases list length
        *count += 1;
        // adds points between furthest and last
        if last > idx + 1 {
            rdp_step(x, y, idx, last, epsilon,
    xnew, ynew, count)
        }
    }
}
```