UNIVERSITY OF WATERLOO | Department of Mechanical and Mechatronics Engineering

ME 303 - Collapsing Vacuum Bubble ODEs

Austin W. Milne
Japmeet Brar
Joshua Selvanayagam
Kevin Chu

March 16, 2022

Abstract

Project 1 report for ME 303 in Winter 2022. Using numerical ODE solutions to model the collapse and rebound of underwater vacuum bubbles. Project source code and report available at: https://github.com/Awbmilne/vacuum_bubble_numerical_ODE.

Contents

List of Tables

List of Figures

List of Code Listings

1 Problem Overview

Differential calculus provides a wonderful insight into the way that systems operate in the real world. For some systems, these differential equation models can be quite complex, complex enough that they cannot be solved using standard analytical methods. One such system is the collapsing of vacuum bubbles underwater. While this may not seem like a common occurrence, there is a surprising number of cases where this system applies. For example, many impellers/propellers create cavitation bubbles under operation. This can be anything from a boat propeller to a jet pump for well water. In these cases, the cavitation bubbles collapse and cause what is known as “cavitation damage” [2] which appears as pitting and corrosion to the exposed surfaces. In more extreme situations, these vacuum bubbles can be caused by underwater explosions, such as those of torpedoes or nuclear bombs. These bubbles are usually significantly larger and can create not only massive shock waves, but also more interesting effects like sonoluminescence [3] and cavitation-ignition [7].

In this report, the Rayleigh-Plesset equation (equation 1) [5] will be studied and solved using numerical methods. The first half of the report addresses a simplified version of the Rayleigh-Plesset equation (equation 2) [6]. This simplified version is used to experiment with numerical solutions for ODEs. Due to the simplification, this equation can be solved analytically and used as a reference point for the numerical solution set. The second half of the report focuses directly on the Rayleigh-Plesset equation, modeling the collapse, and rebound of the bubble over a longer time span.

2 Simplified Solution

The Rayleigh-Plesset equation [5] models the behaviour of a vacuum bubble inside a fluid. This equation takes into account the pressure difference, the viscosity of the fluid, and the surface tension of the fluid-vacuum interface. Some simplifying assumptions are made, but this model is fairly accurate in modeling the displacement of a vacuum bubble.

$$\rho_l \left(R\dot{R} + \frac{3}{2}\dot{R}^2 \right) = -P_0 - 4\mu\frac{\dot{R}}{R} - 2\frac{\sigma}{R} \tag{1}$$

Unfortunately, this model has not yet been solved analytically, so numerical methods are the only option. The simplified version below found by [6] is a far more approachable model of the frequency and general size of the bubble. While this model is not accurate in that it represents the oscillation as a sin/cos function instead of hard impact at the collapse event, it maintains a pretty close relationship to the Rayleigh-Plesset model.

$$\ddot{R} + \lambda^2 (R - R_0) = \frac{-3}{2}\frac{P_0}{\rho_l R_0}, \quad \text{where } \lambda^2 = \frac{3P_0}{\rho_l R_0^2} \tag{2}$$

This simplified equation relies on relatively few parameters to model the system, namely the initial pressure (P_0), maximum radius (R_0), and the liquid viscosity (ρ_l).

The simplified model is solved as a system having initial conditions of $R_0 = 2\,[mm]$ and $\rho_l = 1000\,[kg/m^3]$ at a depth of 10 cm. The initial velocity and radius are $\dot{R}(t) = 0\,[m/s]$ and $R(t) = R_0 = 0.002\,[m]$.

2.1 Pressure at Depth

From the simplified equation 2 above, the only value not directly provided is the P_0 pressure. This pressure can be easily found using the pressure variance over depth equation. Below is said equation with values substituted for the conditions used in the first model.

$$\begin{aligned} \Delta P &= \rho\, g\, h \\ \Delta P &= 1000\,[kg/m^3]\ * 9.81\,[m/s^2]\ * 0.10\,[m] \\ \Delta P &= 981\,[Pa] \end{aligned} \tag{3}$$

In this case, ρ is the fluid density, g is gravity, and h is the vertical displacement below the surface. With an assumed atmospheric pressure of 10^5 Pa, the pressure at the 10 cm depth of the bubble is 100981 Pa.

2.2 Analytical Solution

The primary advantage of this simplified equation is that it can be easily solved analytically as a second order, linear, constant coefficient, inhomogeneous ODE. Equation 4 shows the general (4b) and specific (4c) solution to the simplified ODE. The full derivation can be found as equation 14 in the Equations section of the Appendix.

$$\ddot{R} + \lambda^2 (R - R_0) = \frac{-3}{2}\frac{P_0}{\rho_l R_0}, \quad \text{where } \lambda^2 = \frac{3P_0}{\rho_l R_0^2} \tag{4a}$$

$$\Downarrow$$

$$R(t) = c_a \cos\left(t\sqrt{\frac{3P_0}{\rho_l R_0^2}}\right) + c_b \sin\left(t\sqrt{\frac{3P_0}{\rho_l R_0^2}}\right) + \frac{1}{2}R_0 \tag{4b}$$

$$R(t) = 0.001\,\cos(8702.629t) + 0.001 \tag{4c}$$

This solution is used as a reference point for the numerical solutions.

2.3 Numerical Solution

The numerical solution method used in this report focuses on discretization over a specified time range. In order use this style of numerical analysis with equation 2, the equation needs to be reduced to a set of first order differential statement. Using the system of first order ODEs, the slope can be solved at each discretization and applied over the specified time step.

2.3.1 Order Reduction

The simplified ODE from equation 2 can be broken down by substituting the $\ddot{R}$ value with $\dot{P}$. $\dot{R}$ can then be equal to P and the simplified equation can be rearranged to solve for $\dot{P}$. Equation 5 shows the substitution and final ODE system.

$$\begin{aligned}
\ddot{R} + \lambda^2(R - R_0) &= \frac{-3}{2}\frac{P_0}{\rho_l R_0} \\
\ddot{R} + \lambda^2 R &= \frac{-3}{2}\frac{P_0}{\rho_l R_0} + \lambda^2 R_0 \\
\text{Substitute:} \quad \dot{P} = \ddot{R}\,, &\quad P = \dot{R} \qquad (5a) \\
\dot{P} + \lambda^2 R &= \frac{-3}{2}\frac{P_0}{\rho_l R_0} + \lambda^2 R_0 \\
\therefore \qquad R &= \begin{cases} \dot{P} = \frac{-3}{2}\frac{P_0}{\rho_l R_0} + \lambda^2 R_0 - \lambda^2 R \\ \dot{R} = P \end{cases} \qquad (5b)
\end{aligned}$$

2.3.2 Solution Methods

A selection of explicit solution methods where used for this report, namely Euler's Method and 2 variations of the Runge-Kutta Method: 2nd order Heun's Method (RK2) and 4th order Runge-Kutta Method (RK4). Each of these methods provides different techniques to find the f_i value in equation 6.

$$y_{i+1} = y_i + \Delta\, t\, f_i \qquad (6)$$

Using the Euler Method, the f_i value is defined as the slope at i, such that $f_i = y_i'$. For the system outlined in equation 5b, this means that at each time step, the slopes of the LHS of the system can be used as the f_i values for their respective $R\,/\,P$ value. Equation 7 shows this system.

$$\begin{aligned}
y_{i+1} &= y_i + \Delta\, t\, y_i' \\
\text{Where:} \quad y_i' &= f(y_i, t_i)
\end{aligned} \qquad (7)$$

The 2nd order Heun's Method (RK2) focuses on improving the estimate of the f_i value by averaging the slope at y_i and an estimated slope at y_{i+1}. In order to do this, the y_{i+1} value is estimated using Euler's Method, and then the y_i' and y_{i+1}' values are averaged for f_i. This is shown by equation 8

$$
\begin{aligned}
& y_{i+1} = y_i + \Delta\, t\, f_i \\
\text{Where:}\quad & f_i = \frac{1}{2}\left({}^{*}y'_{i+1} + y'_i\right) \\
\text{and:}\quad & {}^{*}y'_{i+1} = f({}^{*}y_{i+1}, t_{i+1})\,, \quad y'_i = f(y_i, t_i) \\
\text{and:}\quad & {}^{*}y_{i+1} = y_i + \Delta\, t\, y'_i
\end{aligned}
\tag{8}
$$

The 4th order Runge-Kutta Method (RK4) goes another step past the RK2 Method to average a larger set of estimated y' values. These y' values, labeled as k_{1-4}, are solved over quarters of the time delta and lead to a more accurate estimate for the effective slope over the time step. Equation 9 shows how this method is applied.

$$
\begin{aligned}
& y_{i+1} = y_i + \Delta\, t\, f_i \\
\text{Where:}\quad & f_i = \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\
\text{and:}\quad & k_1 = f\left(y_i, t_i\right) \\
& k_2 = f\left(y_i + \Delta\, t\, \frac{k_1}{2}\right) \\
& k_3 = f\left(y_i + \Delta\, t\, \frac{k_2}{2}\right) \\
& k_4 = f\left(y_i + \Delta\, t\, k_3\right)
\end{aligned}
\tag{9}
$$

2.3.3 Software Implementation

In order to solve using these methods, they where implemented in Python using symbolic math and an object oriented design. The primary class used is the `Solvy_boi` class. This class stores the ODE system, variables, and state. It also provides the necessary methods for iterative solving over a specified time frame and time delta. The basic initializer is shown in listing 1 with the object definition for the ODE system in equation 5b shown in listing 2.

Listing 1: Solver Initializer Function — Solvy_boi.py

```python
class Solvy_boi:  #
    """
    ODE numerical solution class.
    This Class takes a system of first order ODEs and provides a number of
    methods available for solving the system numerically.
    """
    def __init__(self, symbol, symbols, functions, s_lim, analytical):
        # Store the ODE system
        self.symbol = symbol
        self.symbols = symbols
        self.functions = functions
        self.s_lim = s_lim
        self.analytical = analytical #
```

Listing 2: Solver Object Definition — Question_1.py

```
    # Create the necessary symbols
    P, R, t = symbols("P R t") # sympy.symbol objects for symbolic formula

    # Create the ODE object
    system = Solvy_boi(
        R, # Variable of consequence. Used to determine error.
        [P, R], # List of variables, The slopes of which are the LHS of the below equations
        [-(3/2) * (p_0/(rho*r_0)) + lmda_sqr*r_0 - lmda_sqr*R, P], # List of functions, RHS of system
        slope_lim, # Limiting value for slope (absolute value)
        [lambda t: None, lambda t : 0.001 * cos(8702.629 * t) + 0.001] # Lambda functions for analytical solution
    )
    state_0 = Matrix([0,0.002]) # Initial state of system
```

The `Solvy_boi` class implements each solution method using an interator function. These functions implement the methods outlined in section 2.3.2 using matrix operations. Due to the volatility of the ODE in Part 2, the `Solvy_boi` class takes a `s_lim` argument to limit the slope of the ODE at any time step. This puts an absolute value limit on the f_i value from equation 6. Listings 3, 4, and 5 show the implementation of the Euler, RK2, and RK4 methods, respectively.

Listing 3: Exclusive Euler Interation Function — Solvy_boi.py

```
    # Exclusive Euler solution methodology
    def e_eul(self, state, dt, t):
        # Increment the positions using Explicit Euler method
        v_subs_dict = list(zip(self.symbols, state)) # Dictionary for value substitution
        slopes = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions]) # Solve for slopes
        new_state = state + dt * slopes # Apply slopes to state
        # Check for sign inversion
        return self.sign_inversion_correction(state, new_state, t) # Return the corrected state
```

Listing 4: Exclusive RK2 Interation Function — Solvy_boi.py

```
    # Exclusive Runge-Kutta (RK2) solution methodology
    def e_rk2(self, state, dt, t):
        # Increment the state using Runge-Kutta (RK2) method
        state_0 = copy.copy(state)
        # Solve for k1
        v_subs_dict = list(zip(self.symbols, state))
        k1 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions])
        # Solve for k2 using k1
        new_state = state_0 + 0.5 * dt * k1
        v_subs_dict = list(zip(self.symbols, state))
        k2 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions])
        # Apply the k2 slope to the function
        new_state = state_0 + dt * k2
        # Check for sign inversion
        return self.sign_inversion_correction(state, new_state, t) # Return the corrected state
```

Listing 5: Exclusive RK4 Interation Function — Solvy_boi.py

```
    # Exclusive Runge-Kutta (RK4) solution methodology
    def e_rk4(self, state, dt, t):
        # Increment the state using Runge-Kutta (RK4) method
        state_0 = copy.copy(state)
```

```python
        # Solve for k1
        v_subs_dict = list(zip(self.symbols, state))
        k1 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions]) #
        # Solve for k2 using k1
        new_state = state_0 + 0.5 * dt * k1
        v_subs_dict = list(zip(self.symbols, state))
        k2 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions])
        # Solve for k3 using k2
        new_state = state_0 + 0.5 * dt * k2
        v_subs_dict = list(zip(self.symbols, state))
        k3 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions])
        # Solve for k4 using k3
        new_state = state_0 + dt * k3
        v_subs_dict = list(zip(self.symbols, state))
        k4 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions])
        # Apply the k slopes to the function
        new_state = state_0 + dt * (1.0/6.0) * (k1 + 2*k2 + 2*k3 + k4)
        # Check for sign
        return self.sign_inversion_correction(state, new_state, t) # Return the corrected state
```

The `Solvy_boi` class then provides an incrementor function that takes a solution method as an arguement. Listing 6 shows the incrementor function and listing 7 shows how this function is called for the set of specified methods.

Listing 6: Incrementor Function — Solvy_boi.py

```python
    # Solution incrementor function
    def run_solution(self, method, state_0, d_t ,t):
        state = state_0 # Not necessary, just cleanliness
        data_set = [[0] + [v for v in state_0]] # Store the 0 initial data point
        for i in range(int(t / d_t)): # For every incremental step
            time = d_t*(i+1)
            state = method(self, state, d_t, time) # Update the state using the specified method
            data_set.append([d_t*(i+1)] + [v for v in state]) # Append the data point to the array
        return data_set # Return the list of data
```

Listing 7: Call to Incrementor — Question_1.py

```python
    # Run the computation using each method and collect data
    data = {} # Empty data dictionary
    time = {} # Empty time dictionary
    methods = [["Eulers",     Solvy_boi.e_eul],
               ["RK2",        Solvy_boi.e_rk2],
               ["RK4",        Solvy_boi.e_rk4],
               ["Analytical", Solvy_boi.anl]]
    for method in methods:
        start = timeit.default_timer() # Start a timer
        data[method[0]] = system.run_solution(method[1], state_0, delta_t, T_star)
        time[method[0]] = timeit.default_timer() - start # Store the time required to solve
```

2.4 Solution Analysis

The system was solved using a range of time steps. The largest being 1×10^{-5} $[sec]$ and the smallest being 9.9765625×10^{-9} $[sec]$. These time steps where selected over a logarithmic range, with each step being half of the previous. This ensured that certain time steps are always present in the output. This is useful for quantifying error in figure 3 further down. The system was solved over the time range of $0 < t < T^*$ where T^* is the time of first collapse. T^* was found to be $0.0003609935174\,[sec]$ by solving equation 4c for $R(t) = 0$.

For larger $\Delta\,t$ values, the system was surprisingly stable and resolved with decent accuracy. Figure 1 shows how the solution methods compare to analytical solution. In the figure, the Euler, RK2, and RK4 solutions matched so closely that they are hard to distinguish.

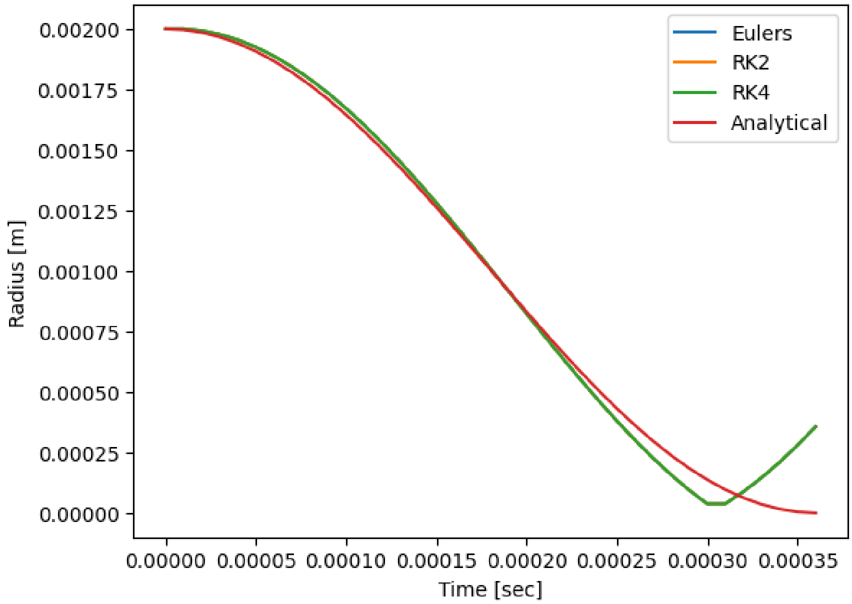
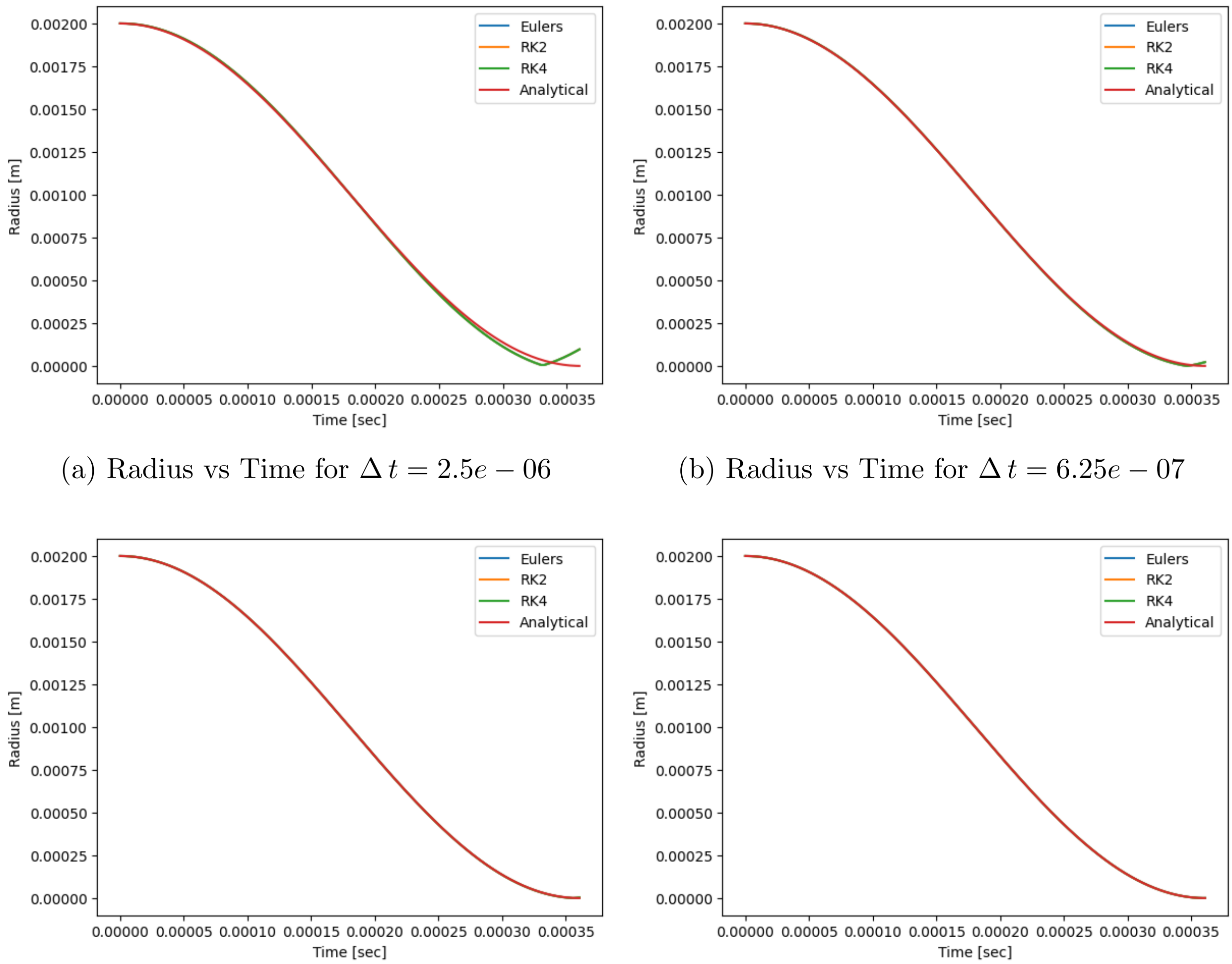


Figure 1: Simplified System - Radius vs Time for $\Delta\,t = 1e - 05$

As the $\Delta\,t$ value was decreased, the accuracy increased. Figure 2 shows all the solution methods approaching the analytical solution, such that they cannot be seen behind it.



(a) Radius vs Time for $\Delta t = 2.5e-06$

(b) Radius vs Time for $\Delta t = 6.25e-07$

(c) Radius vs Time for $\Delta t = 7.8125e-08$

(d) Radius vs Time for $\Delta t = 9.765625e-09$

Figure 2: Simplified System - Solutions with different Δt values

From figure 2, we can assume that a smaller Δt leads to higher accuracy. In order to quantify the error for each method, the R value was taken at $t = 0.00034$ for each solution method. The R value was then compared to the analytical solution and the error was marked in figure 3. As similarly demonstrated in figure 2, figure 3 shows that all the solution methods have the same amount of error when compared to the analytical solution; they are all identical to the RK4 solution's error which is visible in the plot. With respect to the analytical solution, the maximum error for all three methods has a magnitude of $1 \times 10^{-4}\,[m]$ which occurs when $\Delta t = 1.5 \times 10^{-5}\,[sec]$.

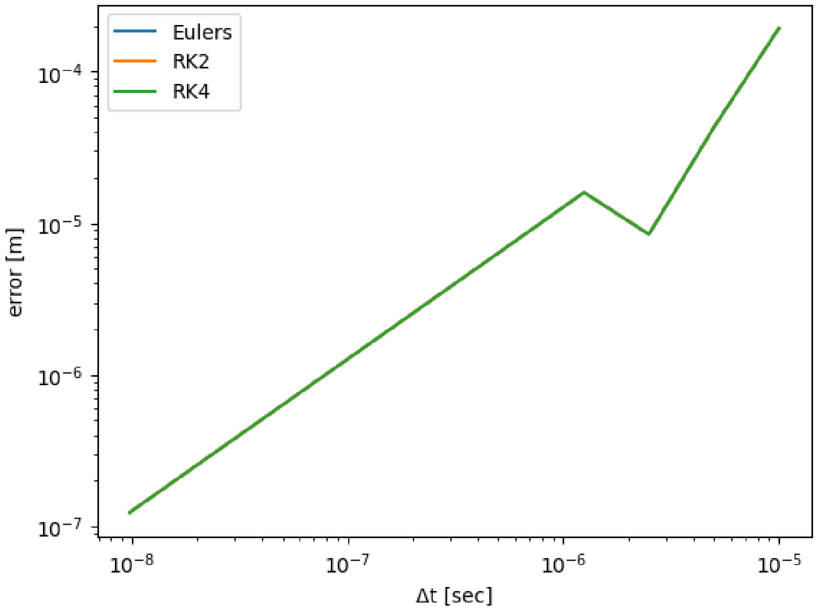


Figure 3: Simplified System - Error of Solving Methods

Furthermore, to quantify the accuracy of each solution method when solving the simplified ODE, the grid convergence of each method's results is marked in Figure 4. For each method, the R value was taken at $t = 0.00034$ using each of the time steps from $\Delta t = 1 \times 10^{-5}$ to $\Delta t = 9.765325 \times 10^{-9}$. Figure 4 shows that as Δt decreases, the accuracy increased. As concluded earlier, the solution methods are almost identical. We can use the grid convergence to verify the solution methods' results against the actual result from the simplified ODE. Therefore, because all solution methods converge to approximately $R = 0.00002\,[m]$ at $t = 0.00034\,[sec]$, the actual result of the simplified ODE will approach the same solution.

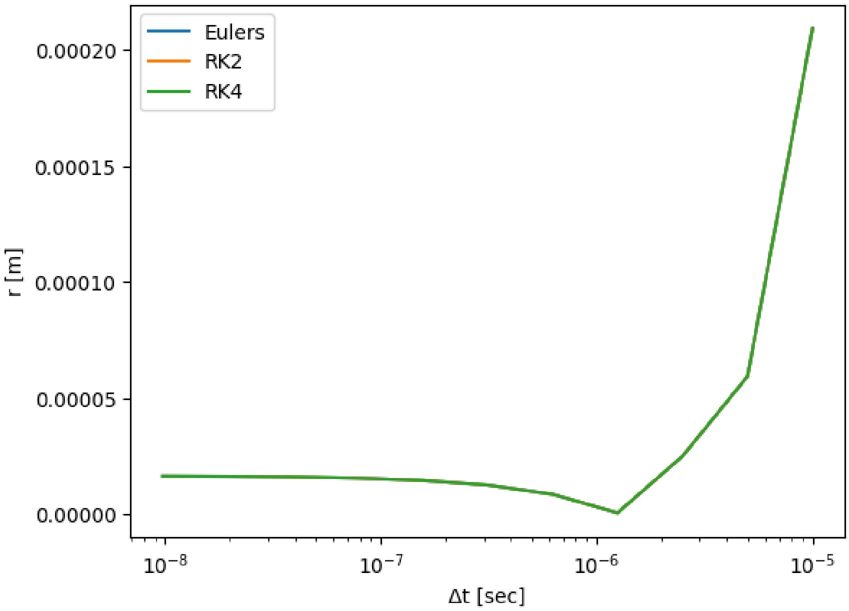


Figure 4: Simplified System - R values at $t = 0.00034$

3 Rayleigh-Plesset Equation

Now having a functional numerical solution method, the attention turns back to the original Rayleigh-Plesset equation [5]. While the simplified solution provides some insight into how the bubble system acts, it does not provide very accurate values near the collapse. For this reason, the R.P. equation is far preferable.

$$\rho_l \left(R\ddot{R} + \frac{3}{2}\dot{R}^2 \right) = -P_0 - 4\mu\frac{\dot{R}}{R} - 2\frac{\sigma}{R} \tag{10}$$

3.1 Bubble Energy

An interesting thing to consider is how much energy is required to create these bubbles. If it is assumed that the forces of surface tension and fluid viscosity are ignored, and assumed that the bubble does not contain any vapour, the energy of formation of the bubble can be treated as work by displacement.

$$W = \int_a^b P dV \tag{11}$$

Assuming that the bubble forms in a body of water large enough that the height of fluid does not significantly change due to displacement, the pressure will be constant at any given depth. Substituting the constant P value and the volume of a sphere, we arrive at equation 12a.

$$\begin{aligned}
W &= \int_a^b \underbrace{P}_{\text{const.}} dV \\
W &= P\int_0^r dV \\
W &= P\,\Delta V \\
W &= P\,\frac{4}{3}\pi r^3 && \text{(12a)} \\
4.5280822 \times 10^{13}\,[J] &= 10810000\,[Pa] \times \frac{4}{3}\pi\,(100\,[m])^3 && \text{(12b)}
\end{aligned}$$

As shown by equation 12b, The energy for creation of a 100 meter vacuum bubble at 1000 meters of depth is on the same order of magnitude as the orbital kinetic energy of the International Space Station [1][4] or the Little Boy nuclear bomb dropped on Hiroshima [8]. This is an incredible amount of energy. The only reasonable means of creating a vacuum bubble of this scale would be to detonate a nuclear warhead under the ocean.

3.2 Numerical Solution

In order to arrive at a numerical solution for the Rayleigh-Plesset equation, the same steps are followed as with the simplified ODE. The equation must first be converted to a system of first order ODEs and incremented over using a numerical method. Unfortunately, some aspects of this system make it more difficult to implement in software.

3.2.1 Order Reduction

Firstly, the system must be converted to a system of first order ODEs. Since the Rayleigh-Plesset equation is only second order, the same substitution can be used as is outlined in section 2.3.1 and equation 5. Equation 13 shows the substitution and construction of the system of equations.

$$\rho_l \left(R\ddot{R} + \frac{3}{2}\dot{R}^2 \right) = -P_0 - 4\mu\frac{\dot{R}}{R} - 2\frac{\sigma}{R} \tag{13a}$$

$$\text{Substitute:} \quad \dot{P} = \ddot{R}\,, \quad P = \dot{R} \tag{13b}$$

$$\begin{aligned}
\rho_l \left(R\dot{P} + \frac{3}{2}P^2 \right) &= -P_0 - 4\mu\frac{P}{R} - 2\frac{\sigma}{R} \\
\rho_l R\dot{P} + \frac{3}{2}\rho_l P^2 &= -P_0 - 4\mu\frac{P}{R} - 2\frac{\sigma}{R} \\
\dot{P} &= \frac{-P_0 - 4\mu\frac{P}{R} - 2\frac{\sigma}{R} - \frac{3}{2}\rho_l P^2}{\rho_l P}
\end{aligned}$$

$$\therefore \qquad R = \begin{cases} \dot{P} = \frac{-P_0 - 4\mu\frac{P}{R} - 2\frac{\sigma}{R} - \frac{3}{2}\rho_l P^2}{\rho_l P} \\ \dot{R} = P \end{cases} \tag{13c}$$

Notably, the $\dot{P}$ value is defined to be a function of a couple of terms, including $-4\mu\frac{P}{R}$ and $-2\frac{\rho}{R}$. These two terms present an issue in that they approach infinity as the bubble's radius approaches zero. These issues are handled more directly in section 3.2.3 with the software implementation.

3.2.2 Solution Methods

Looking at the results in section 2.4, It is clear that, while it requires more computational time, the RK4 method is the most accurate of the solution methods. For this reason, the RK4 method is used to solve the system. In order to observe the behaviour of the other solution methods, both Eulers Method and the RK2 method are also tested.

3.2.3 Software Implementation

As mentioned at the end of section 3.2.1, the $-4\mu\frac{P}{R}$ and $-2\frac{\rho}{R}$ terms from the ODE system in equation 13c present an issue as R approaches zero. The solution to this problem was to limit the absolute values of $\dot{R}$ and $\dot{P}$. This allows the system to operate close enough to the true mathematical model without requiring incredibly large float values to remain accurate. While this change does effect the slope greatly, the range of t values for which $\dot{R}$ and $\dot{P}$ are this large is very very small. Through experimentation, it was determined that a reasonable slope limiting value was 10^6. This value seemed to provide the most consistent results. The slope value was limited using the `numpy.clip` function in Python; The placement of which can be found in listing 8 on line 65 in the solver class.

Listing 8: Slope Limiting with `numpy.clip` in line 65 — Solvy_boi.py

```
59     # Exclusive Runge-Kutta (RK4) solution methodology
60     def e_rk4(self, state, dt, t):
61         # Increment the state using Runge-Kutta (RK4) method
62         state_0 = copy.copy(state)
63         # Solve for k1
64         v_subs_dict = list(zip(self.symbols, state))
65         k1 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.
               functions]) #
66         # Solve for k2 using k1
67         new_state = state_0 + 0.5 * dt * k1
68         v_subs_dict = list(zip(self.symbols, state))
69         k2 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.
               functions])
70         # Solve for k3 using k2
71         new_state = state_0 + 0.5 * dt * k2
72         v_subs_dict = list(zip(self.symbols, state))
73         k3 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.
               functions])
74         # Solve for k4 using k3
75         new_state = state_0 + dt * k3
76         v_subs_dict = list(zip(self.symbols, state))
77         k4 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.
               functions])
78         # Apply the k slopes to the function
79         new_state = state_0 + dt * (1.0/6.0) * (k1 + 2*k2 + 2*k3 + k4)
80         # Check for sign
81         return self.sign_inversion_correction(state, new_state, t) # Return the corrected
               state
```

The Rayleigh-Plesset equation created another issue. As the system approaches zero, the collapse speed increases and eventually the system arrives at $R = 0$ at this point, referred to as the singularity, there is an incredible pressure spike. For simplification in this report, it can be assumed that at the singularity, all the inward pointed velocities instantly reverse and point outward. While this effect could be well estimated in software using a more complex Time Of Impact (TOI) algorithm, a more simple approach was used for this model. The iteration functions each make use of sign inversion correction. After the $i+1$ values are calculated for each iteration, the variable of consequence (in this case, R) is checked for if the sign has changed (+/-). If the sign has changed, the values for i are restored and the other values (P) are multiplied by -1. This forces R to always be a positive value, but reduces the precision. The maximum error for this case is a function of the time step of the iteration. As the time step decreases, so does the potential error near $R = 0$. The implementation of the sign inversion correction is shown in listing 9.

Listing 9: Sign Inversion Correction — Solvy_boi.py

```
    # Sign inversion correction for vacuum bubble collapse
    def sign_inversion_correction(self, state, new_state, t):
```

```python
        i = self.symbols.index(self.symbol)
        if math.copysign(1, state[i]) != math.copysign(1, new_state[i]):
            if (state[i] != 0) and (new_state[i] != 0):
                # print(f"Collapse @ t={t}") # Debug output
                state[i] = -state[i] # Invert the value of consequence
                return -state # Invert the entire state (reverts value of consquence)
        return new_state # Return the new state if no sign change
```

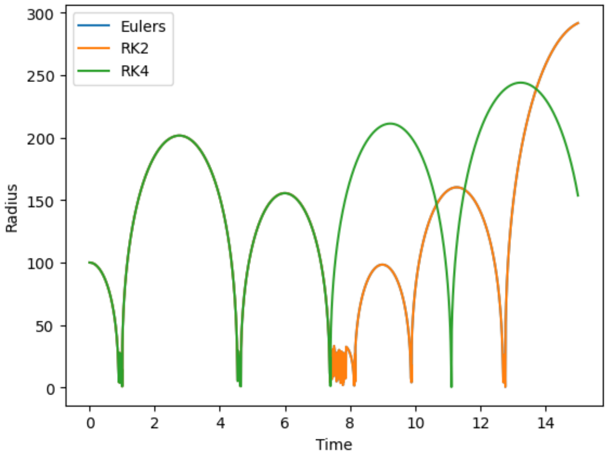
The implementation of the solver class is shared between the solutions of the simplified ODE and the Rayleigh-Plesset equation. These functions are used in both cases, but only really effect the outcome of the Rayleigh-Plesset equation, where the slope has the potential to exceed 10^6 and the radius is liable to dip below 0.

3.3 Solution Analysis

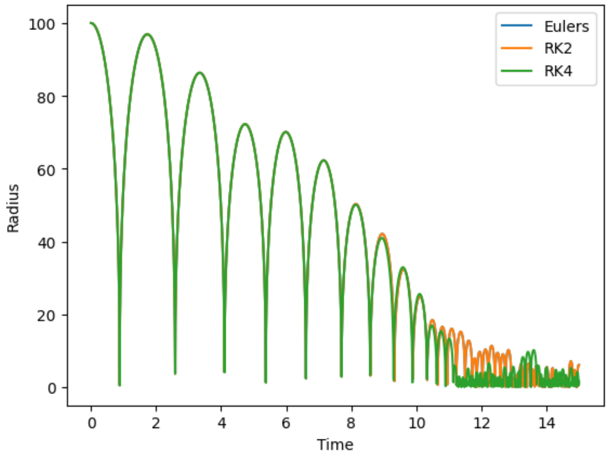
The R.P. system was again solved using a range of time steps: the largest being 1×10^{-2} and the smallest being 9.765625×10^{-6}. These time steps where selected over a logarithmic range, with each step being half of the previous. This ensured that certain time steps are always present in the output. This is useful for quantifying error in Figure 6 further down.

Unlike the simplified ODE, for larger Δt values, the R.P. system was less stable and resolved with lower accuracy when compared with the smaller time steps. Additionally, Figure 5 demonstrates that, at larger time steps, the RK4 solutions have a higher accuracy and stability than the RK2 solutions. This coincides with the results in section 2.4.

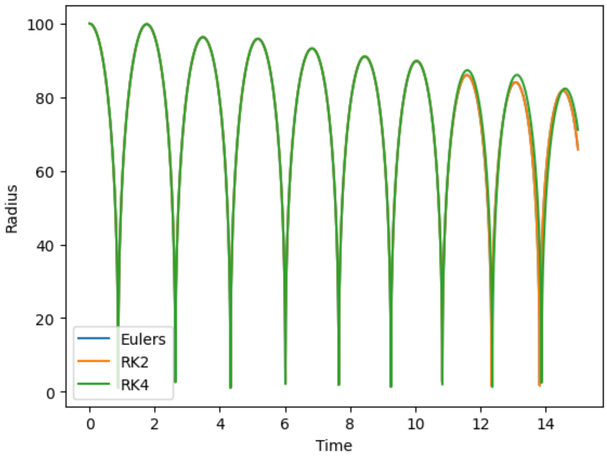
The data also highlights the relationship between the time step used to solve the R.P. equation and the time observed between two neighbouring collapses. For $\Delta t > 6.25 \times 10^{-4}$, the R.P. system was unstable and produced multiple collapses at a higher and sporadic frequency as time increased. However, as concluded from the simplified ODE, as Δt decreased, a higher accuracy was produced as both the number and frequency of bubble collapses over time becomes consistent. Figures 5c and 5d illustrate the high accuracy achieved by the RK4 solution once the time step is sufficiently small.



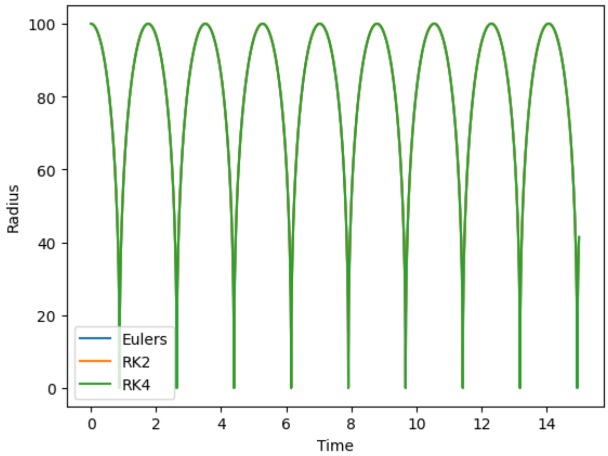
(a) Radius vs Time for $\Delta\, t = 0.005$



(b) Radius vs Time for $\Delta\, t = 0.00125$



(c) Radius vs Time for $\Delta\, t = 0.000625$



(d) Radius vs Time for $\Delta\, t = 9.765625e - 06$

Figure 5: Rayleigh-Plesset System - Solutions with different $\Delta\, t$ values

In order to quantify error (when solving the full R.P. equation) between the methods of Euler, RK2, and RK4, the R value was taken at $t = 14$ for each solution method. Each solution's respective R value was then compared to the Δt_{min} RK4 solution and the error was marked in Figure 6. It is interesting to note the slight differences of error between RK2 and RK4, which are more apparent when solving the full R.P. equation as opposed to the simplified ODE; in Section 2.4 both methods' error are nearly indistinguishable from each other. However, all the solution methods have a lower error for the R.P. equation at the same respective $\Delta\, t$ than they had with solving the simplified ODE.

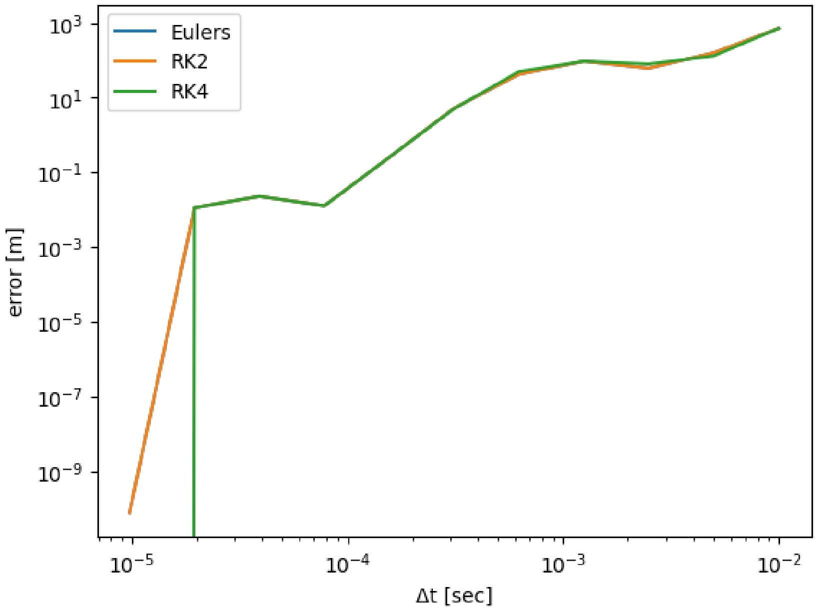


Figure 6: Rayleigh-Plesset System - Error of Solving Methods

Furthermore, to quantify the accuracy of each solution method when solving the R.P. equation, the grid convergence of each method's results is marked in Figure 7. For each method, the R value was taken at $t = 14.0$ using each of the time steps from $\Delta t = 0.01$ to $\Delta t = 9.765625 \times 10^{-6}$. Figure 7 shows that as Δt decreases, the value of R converges to 100 metres. As concluded earlier, the solution of RK4 becomes more accurate as the time step used becomes smaller. Since we do not have the analytical solution to the full R.P. equation, we can use the grid convergence to determine if the real solution will converge. Therefore, because all solution methods converge to $R = 100$m at $t = 14.0$ seconds, the actual result of the R.P. equation will approach the same solution.

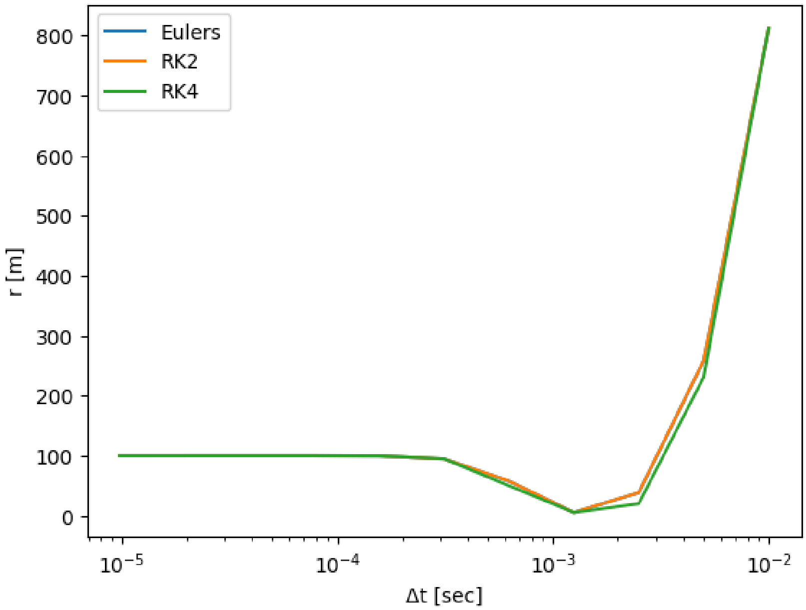
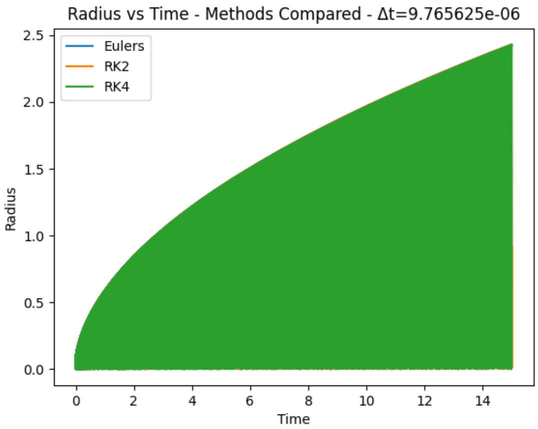


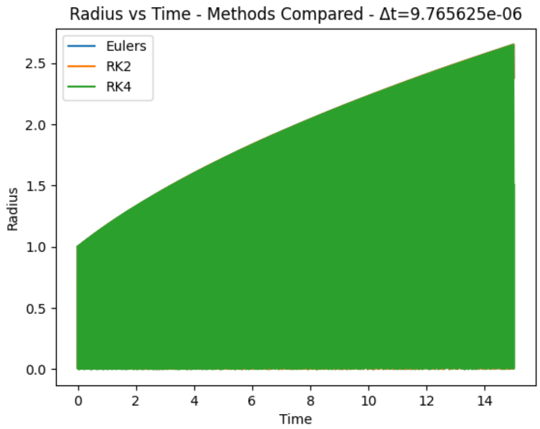
Figure 7: Rayleigh-Plesset System - R values at $t = 14.0$

3.4 Radius Variation

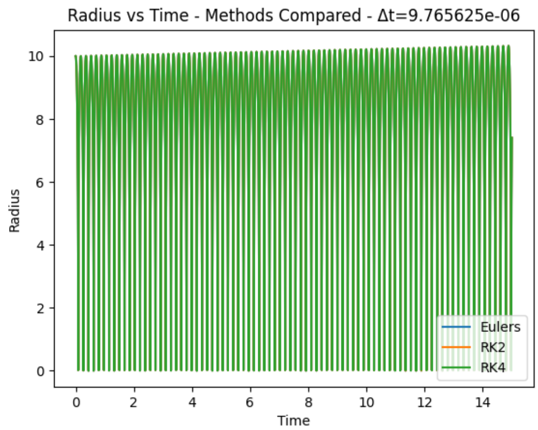
In order to observe the behaviour of smaller vacuum bubbles, the system was solved for a range of different radiuses: 0.1, 1, 10, and 100 meters. The results of which are shown in the figure 8.



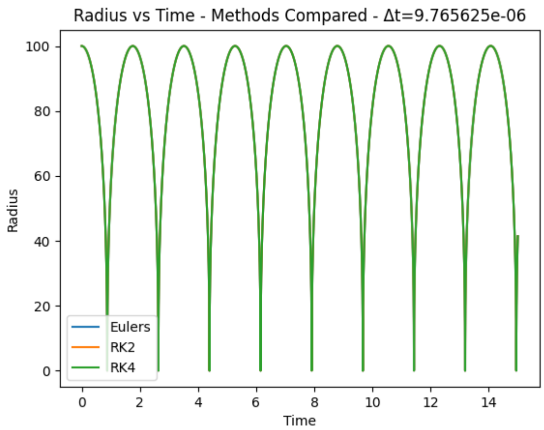
(a) Radius vs Time for $R_0 = 0.1\,[m]$



(b) Radius vs Time for $R_0 = 1\,[m]$



(c) Radius vs Time for $R_0 = 10\,[m]$



(d) Radius vs Time for $R_0 = 100\,[m]$

Figure 8: Rayleigh-Plesset System - Solutions with different R_0 values

The most obvious thing about the graphs is that, as R_0 increases, so does the period of oscillation. Table 1 shows that the time of the first collapse varies almost linearly with the initial radius. Due to the precision of this numerical solver, it is assumed that the variation is error, and that the values are linearly related.

Table 1: First Collapse Time vs Initial Radius

$R_0\,[m]$	$T_1^*\,[sec]$
0.1	0.0009765625
1	0.008984375000000001
10	0.08826171875
100	0.8790625

Additionally, in figure 8, it is easy to see that the smaller R_0 values create errors with the current solver implementation. This is likely due to the accumulation in the very small error at the rebound point when the sign inversion correction from section 3.2.3 is enacted. Table 2 shows the R and $\dot{R}$ immediately before and after the sign inversion kicks in.

Table 2: Singularity Rebound Amplification

i	t	$\dot{R}$	R
98	0.00095703125	-366.564575925969	0.00642150348534173
99	0.000966796875	-376.330200925969	0.00284177129856469
100	0.0009765625	376.330200925969	0.00284177129856469
101	0.000986328125	366.564575925969	0.00651687091698236

Between $i = 99$ and $i = 100$, the new state is calculated to have $R < 0$. This causes the slope to be inverted for the next state point. The issue arises due to how the slope is handled. The R value is calculated for $i = 98 \rightarrow i = 99$ using the R and $\dot{R}$ of $i = 98$, but when it rebounds and takes the first increasing step of $i = 100 \rightarrow i = 101$ using the R and $\dot{R}$ of $i = 100$. Looking at these values, it is evident that $\dot{R}_{i=98} < -\dot{R}_{100}$. This subtle amplification is not significant when there are small number of rebound events, such as when a large R_0 is used. But, as the frequency of rebounds increases, the effect becomes far more pronounced. For future implementations, this issue could be resolved by storing the $\dot{R}$ value for 1 previous state, allowing the sign inversion correction to utilize the previous slope value instead of the current. This should remove the amplification and provide more consistent data. Unfortunately, this could not be implemented in time to have new data calculated.

4 References

[1] The wizards of orbits, Aug 2001. *European Space Agency (ESA).*

[2] Donald R. Askeland, Pradeep P. Fulay, and Wendelin J. Wright. *The Science and Engineering of Material*. Global Engineering, 6th edition, 2006. pp. 880.

[3] Michael P. Brenner, Sascha Hilgenfeldt, and Detlef Lohse. Single bubble sonoluminescense. *Reviews of modern physics*, 74(2), 2002. pp. 425-484.

[4] Brian Dunbar. The iss to date (03/09/2011), Mar 2011.

[5] Lord Rayleigh O.M. F.R.S. Viii. on the pressure developed in a liquid during the collapse of a spherical cavity. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 34(200):94–98, 1917.

[6] Sascha Hilgenfeldt, Michael P. Brenner, Siegfried Grossmann, and Detlef Lohse. Analysis of rayleigh plesset dynamics for sonoluminescing bubbles. *Journal of fluid mechanics*, 1998(365), 1998. pp. 171-204.

[7] David A. Jacqmin and Quang-Viet Nguyen. A study of cavitation-ignition bubble combustion. Technical memorandum (tm), NASA, August 2005.

[8] John Malik. The yields of hiroshima and nagasaki nuclear explosions. Technical report, Los Alamos National Labratory, September 1985.

5 Appendix

5.1 Equations

Primary ODE simplification

$$\ddot{R} + \lambda^2 (R - R_0) = \frac{-3}{2}\frac{P_0}{\rho_l R_0}, \quad \text{where } \lambda^2 = \frac{3P_0}{\rho_l R_0^2} \tag{14a}$$

$$\downarrow$$

$$\ddot{R} + \lambda^2 R - \lambda^2 R_0 = \frac{-3}{2}\frac{P_0}{\rho_l} R_0$$

$$\ddot{R} + \underbrace{\lambda^2 P}_{\text{const. } j} = \underbrace{\frac{-3}{2}\frac{P_0}{\rho_0 R_0} - \lambda^2 R_0}_{\text{const. } k}$$

$$\begin{array}{l|l} & k = -\frac{3}{2}\frac{P_0}{\rho_0 R_0} - \lambda^2 R_0 \\ j = \lambda^2 & \\ j = \frac{3P_0}{\rho_l R_0^2} & k = -\frac{3}{2}\frac{P_0}{\rho_l R_0} - \frac{3P_0}{\rho_l R_0} \\ & k = -\frac{9}{2}\frac{P_0}{\rho_l R_0} \end{array}$$

$$\therefore \quad \ddot{R} + jR = k\,, \quad \text{where } j = \lambda^2 = \frac{3P_0}{\rho_l R_0^2} \quad \& \quad k = -\frac{9}{2}\frac{P_0}{\rho_l R_0} \tag{14b}$$

Homogeneous Solution

$$a\ddot{R} + b\dot{R} + cR = 0$$

$$a = 1\,, \quad b = 0\,, \quad c = j$$

$$R_h(t) = e^{\lambda_h t}\,, \quad \text{where } \lambda_h = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$\lambda_h = \frac{0 \pm \sqrt{0 - 4(1)(j)}}{2(1)}$$

$$\text{Since non-real solution :} \quad R_h(t) = c_a e^{\alpha t}\cos(\beta t) + c_b e^{\beta t}\sin(\beta t)$$

$$\text{where,} \quad \lambda_{h_p} = \alpha + i\beta\,, \quad \lambda_{h_p} = \alpha + i\beta$$

$$\text{and,} \quad \alpha = 0\,, \quad \beta = \sqrt{j}$$

$$\therefore \quad R_h(t) = c_a \cos\left(t\sqrt{j}\right) + c_b \sin\left(t\sqrt{j}\right) \tag{14c}$$

Particular Solution

Since RHS is const., no derivatives needed:

$$R_p(t) = c_0 \tag{14d}$$

Substitute to find c_0 :

$$\ddot{R}_p + jR_p = k$$

$$(0) + j(c_0) = k$$

$$c_0 = \frac{k}{j} = \frac{-\frac{9}{2}\frac{P_0}{\rho_l R_0}}{\frac{3P_0}{\rho_l R_0^2}} = \frac{1}{2}R_0 \tag{14e}$$

General Solution

$$R(t) = R_h(t) + R_p(t)$$

$$R(t) = c_a \cos\left(t\sqrt{j}\right) + c_b \sin\left(t\sqrt{j}\right) + \frac{1}{2}R_0 \tag{14f}$$

$$R(t) = c_a \cos\left(t\sqrt{\frac{3P_0}{\rho_l R_0^2}}\right) + c_b \sin\left(t\sqrt{\frac{3P_0}{\rho_l R_0^2}}\right) + \frac{1}{2}R_0 \tag{14g}$$

Initial Value Solution

$$\begin{aligned}
R(t) &= c_a \cos\left(t\sqrt{j}\right) + c_b \sin\left(t\sqrt{j}\right) + \frac{1}{2}R_0 \\
R(0) &= c_a \cos\left((0)\sqrt{j}\right) + c_b \sin\left((0)\sqrt{j}\right) + \frac{1}{2}R_0 \\
0.002 &= c_a(1) + c_b(0) + \frac{1}{2}(0.002) \\
c_a &= 0.002 - \frac{1}{2}0.002 = 0.001
\end{aligned}$$

$$\begin{aligned}
\dot{R}(t) &= -c_a\sqrt{j}\ \sin\left(t\sqrt{j}\right) + c_b \cos\left(t\sqrt{j}\right) \\
\dot{R}(0) &= -c_a\sqrt{j}\ \sin\left((0)\sqrt{j}\right) + c_b \cos\left((0)\sqrt{j}\right) \\
0 &= -c_a\sqrt{j}\,(0) + c_b(1) \\
c_b &= 0
\end{aligned}$$

$$j = \frac{3P_0}{\rho_l R_0^2} = \frac{3 \times 100981}{1000 \times 0.002^2} = 7.573575 \times 10^7$$

$$\sqrt{j} = \sqrt{7.573575 \times 10^7} = 8702.629$$

$$R(t) = 0.001\ \cos(8702.629t) + 0.001 \tag{14h}$$

5.2 Full Code Listings

Listing 10: ODE Solver Object — Solvy_boi.py

```python
__author__ = "Austin W. Milne"
__credits__ = ["Austin W. Milne", "Japmeet Brar", "Joshua Selvanayagam", "Kevin Chu"]
__email__ = "awbmilne@uwaterloo.ca"
__version__ = "1.0"
__date__ = "March 8, 2022"

"""
This code was written for Project #1 of ME303 "Advanced Engineering Mathmatics" in the
    Winter 2022 term.
The goal is to numerically solve an ODE relating to the occilation of a collapsing vacuum
    bubble in
water. There are a number of solution methods implemented and compared. In some cases, these
     solutions
are also compared to the analytical solution.
"""

import copy
import math
import numpy as np
from sympy import E, Matrix
from sympy.matrices import Matrix

class Solvy_boi:  #
    """
    ODE numerical solution class.
    This Class takes a system of first order ODEs and provides a number of
    methods available for solving the system numerically.
    """
    def __init__(self, symbol, symbols, functions, s_lim, analytical):
        # Store the ODE system
        self.symbol = symbol
        self.symbols = symbols
        self.functions = functions
        self.s_lim = s_lim
        self.analytical = analytical #

    # Exclusive Euler solution methodology
    def e_eul(self, state, dt, t):
        # Increment the positions using Explicit Euler method
        v_subs_dict = list(zip(self.symbols, state)) # Dictionary for value substitution
        slopes = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.
            functions]) # Solve for slopes
        new_state = state + dt * slopes # Apply slopes to state
        # Check for sign inversion
        return self.sign_inversion_correction(state, new_state, t) # Return the corrected
            state

    # Exclusive Runge-Kutta (RK2) solution methodology
    def e_rk2(self, state, dt, t):
        # Increment the state using Runge-Kutta (RK2) method
        state_0 = copy.copy(state)
        # Solve for k1
        v_subs_dict = list(zip(self.symbols, state))
        k1 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.
            functions])
        # Solve for k2 using k1
        new_state = state_0 + 0.5 * dt * k1
        v_subs_dict = list(zip(self.symbols, state))
        k2 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.
            functions])
        # Apply the k2 slope to the function
        new_state = state_0 + dt * k2
        # Check for sign inversion
```

```
        return self.sign_inversion_correction(state, new_state, t) # Return the corrected state

    # Exclusive Runge-Kutta (RK4) solution methodology
    def e_rk4(self, state, dt, t):
        # Increment the state using Runge-Kutta (RK4) method
        state_0 = copy.copy(state)
        # Solve for k1
        v_subs_dict = list(zip(self.symbols, state))
        k1 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions]) #
        # Solve for k2 using k1
        new_state = state_0 + 0.5 * dt * k1
        v_subs_dict = list(zip(self.symbols, state))
        k2 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions])
        # Solve for k3 using k2
        new_state = state_0 + 0.5 * dt * k2
        v_subs_dict = list(zip(self.symbols, state))
        k3 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions])
        # Solve for k4 using k3
        new_state = state_0 + dt * k3
        v_subs_dict = list(zip(self.symbols, state))
        k4 = Matrix([np.clip(f.subs(v_subs_dict),-self.s_lim,self.s_lim) for f in self.functions])
        # Apply the k slopes to the function
        new_state = state_0 + dt * (1.0/6.0) * (k1 + 2*k2 + 2*k3 + k4)
        # Check for sign
        return self.sign_inversion_correction(state, new_state, t) # Return the corrected state

    # Sign inversion correction for vacuum bubble collapse
    def sign_inversion_correction(self, state, new_state, t):
        i = self.symbols.index(self.symbol)
        if math.copysign(1, state[i]) != math.copysign(1, new_state[i]):
            if (state[i] != 0) and (new_state[i] != 0):
                # print(f"Collapse @ t={t}") # Debug output
                state[i] = -state[i] # Invert the value of consequence
                return -state # Invert the entire state (reverts value of consquence)
        return new_state # Return the new state if no sign change

    # Analytical solution incrementor
    def anl(self, state, dt, t):
        # Solve given lambda for the specified t
        return [lmb(t) for lmb in self.analytical] # Return the solution list to the supplied lambda list

    # Solution incrementor function
    def run_solution(self, method, state_0, d_t ,t):
        state = state_0 # Not necessary, just cleanliness
        data_set = [[0] + [v for v in state_0]] # Store the 0 initial data point
        for i in range(int(t / d_t)): # For every incremental step
            time = d_t*(i+1)
            state = method(self, state, d_t, time) # Update the state using the specified method
            data_set.append([d_t*(i+1)] + [v for v in state]) # Append the data point to the array
        return data_set # Return the list of data
```

Listing 11: Question 1 Runner — Question_1.py

```python
__author__ = "Austin W. Milne"
__credits__ = ["Austin W. Milne", "Japmeet Brar", "Joshua Selvanayagam", "Kevin Chu"]
__email__ = "awbmilne@uwaterloo.ca"
__version__ = "1.0"
__date__ = "March 8, 2022"

"""
This code was written for Project #1 of ME303 "Advanced Engineering Mathmatics" in the Winter 2022 term.
The goal is to numerically solve an ODE relating to the occilation of a collapsing vacuum bubble in
water. There are a number of solution methods implemented and compared. In some cases, these solutions
are also compared to the analytical solution.
"""

import os
import csv
import numpy
import timeit
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path
from sympy import Matrix, symbols, cos
from sympy.matrices import Matrix

from Solvy_boi import Solvy_boi

def run_q1(delta_t, T_star, slope_lim, root, r_0=0.002, show_output=False):
    # CONFIGURATION ------------------------------------------------------------------- #
    r_0 = 0.002
    rho = 1000
    p_0 = 100981
    lmda_sqr = (3 * p_0) / (rho * r_0**2)
    out_p = root / f'dt_{delta_t}'

    # Ensure output directory exists
    if not os.path.exists(out_p):
        os.makedirs(out_p)

    # Create the necessary symbols
    P, R, t = symbols("P R t") # sympy.symbol objects for symbolic formula

    # Create the ODE object
    system = Solvy_boi(
        R, # Variable of consequence. Used to determine error.
        [P, R], # List of variables, The slopes of which are the LHS of the below equations
        [-(3/2) * (p_0/(rho*r_0)) + lmda_sqr*r_0 - lmda_sqr*R, P], # List of functions, RHS of system
        slope_lim, # Limiting value for slope (absolute value)
        [lambda t: None, lambda t : 0.001 * cos(8702.629 * t) + 0.001] # Lambda functions for analytical solution
    )
    state_0 = Matrix([0,0.002]) # Initial state of system

    # CALCULATIONS -------------------------------------------------------------------- #
    # Run the computation using each method and collect data
    data = {} # Empty data dictionary
    time = {} # Empty time dictionary
    methods = [["Eulers",     Solvy_boi.e_eul],
               ["RK2",        Solvy_boi.e_rk2],
               ["RK4",        Solvy_boi.e_rk4],
               ["Analytical", Solvy_boi.anl]]
    for method in methods:
        start = timeit.default_timer() # Start a timer
        data[method[0]] = system.run_solution(method[1], state_0, delta_t, T_star)
```

```
        time[method[0]] = timeit.default_timer() - start # Store the time required to solve
    data.update((label, pd.DataFrame(set)) for label, set in data.items()) # Convert data sets to dataframes

    # Add column names for data in each data frame (prettify)
    plot_symbols = ['t'] + [repr(sym) for sym in system.symbols] # List of symbols (prepend 't')
    for _, set in data.items():
        set.rename(columns=dict(enumerate(plot_symbols, start=0)), inplace=True) # Name colums of the data sets

    # DATA OUTPUT ------------------------------------------------------------------- #
    # Print the Data Sets for posterity
    for label, set in data.items():
        if(show_output): print(f"\n -- Data for system solved using {label} method --")
        if(show_output): print(numpy.shape(set), type(set), set, sep='\n') # Print to stdout
        print(f"Solution time ({label}): {time[label]}")
        set.to_csv(out_p / f"{label}_data.csv") # Save CSV file to 'out' folder

    with open(out_p / f"solve_times", 'w') as file:
        labels = [label for label,_ in time.items()]
        writer = csv.DictWriter(file, labels)
        # writer.writerow(labels)
        writer.writerow(time)

    # Create a plot for each data set
    for label, set in data.items():
        plt.plot(set[repr(t)], set[repr(R)]) # Plot each line with its symbol
        # plt.title(f"Radius vs Time - {label} Method -  t ={delta_t}")
        plt.xlabel("Time")
        plt.ylabel("Radius")
        plt.savefig(out_p / f"{label}_graph.png", bbox_inches = 'tight')
        plt.clf()

    # Create combined plot for all solutions
    for label, set in data.items():
        plt.plot(set[repr(t)], set[repr(R)], label=label)
    # plt.title(f"Radius vs Time - Methods Compared -  t ={delta_t}")
    plt.xlabel("Time [sec]")
    plt.ylabel("Radius [m]")
    plt.legend()
    plt.savefig(out_p / f"combined_graph.png", bbox_inches = 'tight')
    if show_output: plt.show()
    plt.clf()

if __name__ == "__main__":
    delta_t = 0.0000001
    T_star = 0.0003609935174
    slope_lim = 10e5
    root = Path(f'./out/Question_1')
    run_q1(delta_t, T_star, slope_lim, root, show_output=True)
```

Listing 12: Question 2 Runner — Question_2.py

```python
__author__ = "Austin W. Milne"
__credits__ = ["Austin W. Milne", "Japmeet Brar", "Joshua Selvanayagam", "Kevin Chu"]
__email__ = "awbmilne@uwaterloo.ca"
__version__ = "1.0"
__date__ = "March 8, 2022"

"""
This code was written for Project #1 of ME303 "Advanced Engineering Mathmatics" in the Winter 2022 term.
The goal is to numerically solve an ODE relating to the occilation of a collapsing vacuum bubble in
water. There are a number of solution methods implemented and compared. In some cases, these solutions
are also compared to the analytical solution.
"""

import os
import csv
import numpy
import timeit
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path
from sympy import Matrix, symbols
from sympy.matrices import Matrix

from Solvy_boi import Solvy_boi

def run_q2(delta_t, T_star, slope_lim, root, r_0=100, show_output=False):
    # CONFIGURATION -------------------------------------------------------------------- #
    rho = 996
    mu = 0.798e-3
    sigma = 0.072
    p_0 = 10e5 + 1000*9.81*100
    out_p = root / f'dt_{delta_t}'

    # Ensure output directory exists
    if not os.path.exists(out_p):
        os.makedirs(out_p)

    # Create the necessary symbols
    P, R, t = symbols("P R t")

    # Create the ODE object
    system = Solvy_boi(
        R, # The actual output variable
        [P, R], # List of variables, The slopes of which are the LHS of the below equations
        [(-p_0 - 4*mu*(P/R)-2*(sigma/R)-(3/2)*(rho*P**2))/(rho * R), P], # List of functions, RHS of system
        slope_lim,
        [lambda t: None, lambda t : None]
    )
    state_0 = Matrix([0,r_0]) # Initial state of system

    # CALCULATIONS --------------------------------------------------------------------- #
    # Run the computation using each method and collect data
    data = {}
    time = {}
    methods = [["Eulers",     Solvy_boi.e_eul],
               ["RK2",        Solvy_boi.e_rk2],
               ["RK4",        Solvy_boi.e_rk4],]
    for method in methods:
        start = timeit.default_timer()
        data[method[0]] = system.run_solution(method[1], state_0, delta_t, T_star)
        time[method[0]] = timeit.default_timer() - start
    data.update((label, pd.DataFrame(set)) for label, set in data.items()) # Convert data
```

```
        sets to dataframes

    # Add column names for data in each data frame (prettify)
    plot_symbols = ['t'] + [repr(sym) for sym in system.symbols] # List of symbols (prepend 't')
    for _, set in data.items():
        set.rename(columns=dict(enumerate(plot_symbols, start=0)), inplace=True) # Name colums of the data sets

    # DATA OUTPUT ------------------------------------------------------------------- #
    # Print the Data Sets for posterity
    for label, set in data.items():
        if show_output: print(f"\n -- Data for system solved using {label} method --")
        if show_output: print(numpy.shape(set), type(set), set, sep='\n') # Print to stdout
        print(f"Solution time ({label}): {time[label]}")
        set.to_csv(out_p / f"{label}_data.csv") # Save CSV file to 'out' folder

    with open(out_p / f"solve_times", 'w') as file:
        labels = [label for label,_ in time.items()]
        writer = csv.DictWriter(file, labels)
        # writer.writerow(labels)
        writer.writerow(time)

    # Create a plot for each data set
    for label, set in data.items():
        plt.plot(set[repr(t)], set[repr(R)]) # Plot each line with its symbol
        # plt.title(f"Radius vs Time - {label} Method -  t ={delta_t}")
        plt.xlabel("Time")
        plt.ylabel("Radius")
        plt.savefig(out_p / f"{label}_graph.png", bbox_inches = 'tight')
        plt.clf()

    # Create combined plot for all solutions
    for label, set in data.items():
        plt.plot(set[repr(t)], set[repr(R)], label=label)
    # plt.title(f"Radius vs Time - Methods Compared -  t ={delta_t}")
    plt.xlabel("Time")
    plt.ylabel("Radius")
    plt.legend()
    plt.savefig(out_p / f"combined_graph.png", bbox_inches = 'tight')
    if show_output: plt.show()
    plt.clf()


if __name__ == '__main__':
    delta_t = 0.0001
    T_star = 15
    slope_limit = 10e6
    root = Path(f'./out/Question_2')
    run_q2(delta_t, T_star, slope_limit, root, show_output=True)
```

Listing 13: Auto Runner — Auto_runner.py

```python
__author__ = "Austin W. Milne"
__credits__ = ["Austin W. Milne", "Japmeet Brar", "Joshua Selvanayagam", "Kevin Chu"]
__email__ = "awbmilne@uwaterloo.ca"
__version__ = "1.0"
__date__ = "March 8, 2022"

"""
This code was written for Project #1 of ME303 "Advanced Engineering Mathmatics" in the Winter 2022 term.
The goal is to numerically solve an ODE relating to the occilation of a collapsing vacuum bubble in
water. There are a number of solution methods implemented and compared. In some cases, these solutions
are also compared to the analytical solution.
"""

import os
import re
import csv
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path
from Question_1 import run_q1
from Question_2 import run_q2

# Q1 AUTORUN CONFIGURATION ---------------------------------------------------------- #
Q1_dt_variance_root = Path(f'./out/dt_variance/Question_1')
Q1_delta_t_max = 0.00001
Q1_delta_t_steps = 10
Q1_delta_t_factor = 2

Q1_T_star = 0.0003609935174
Q1_slope_lim = 10e5

Q1_error_ref_time = 0.0003

# Logarithmic  t  set
Q1_dt_set = [Q1_delta_t_max / (Q1_delta_t_factor**i) for i in range(Q1_delta_t_steps+1)]

# Q2 AUTORUN CONFIGURATION ---------------------------------------------------------- #
Q2_dt_variance_root = Path(f'./out/dt_variance/Question_2')
Q2_delta_t_max = 0.01
Q2_delta_t_steps = 10
Q2_delta_t_factor = 2

Q2_T_star = 15
Q2_slope_lim = 10e5

Q_2_error_time = 9

# Logarithmic  t  set
Q2_dt_set = [Q2_delta_t_max / (Q2_delta_t_factor**i) for i in range(Q2_delta_t_steps+1)]

Q2_size_variance_root = Path(f'./out/size_variance/Question_2')
Q2_size_set = [0.1, 1, 10, 100]

if __name__ == '__main__':
    # AUTORUN --------------------------------------------------------------------------- #
    # Debugging output of  t  sets
    #print(f"Q1 dt set:\n{Q1_dt_set}")
    #print(f"Q2 dt set:\n{Q2_dt_set}")

    # Run the Q1 set
    for i, dt in enumerate(Q1_dt_set, start=1):
        print(f"\nRunning ({i}/{len(Q1_dt_set)}) Q1 with dt = {dt}")
```

```
        run_q1(dt, Q1_T_star, Q1_slope_lim, Q1_dt_variance_root)

    # Run the Q2 set
    for i, dt in enumerate(Q2_dt_set, start=1):
        print(f"\nRunning ({i}/{len(Q2_dt_set)}) Q2 with dt = {dt}")
        run_q2(dt, Q2_T_star, Q2_slope_lim, Q2_dt_variance_root)


    # ERROR DETERMINATION AND GRAPHING --------------------------------------------------- #
    # Determine error for each question
    error_root = Path('./out/Error')
    questions = ['Question_1', 'Question_2']
    roots = [Q1_dt_variance_root, Q2_dt_variance_root]
    reference_times = [0.00034, 14.0]
    reference_methods = ['Analytical', 'RK4']
    methods = [["Eulers", "RK2", "RK4"],
               ["Eulers", "RK2", "RK4"]]
    for q, root, time, method, methods in zip(questions, roots, reference_times, reference_methods, methods):
        # Create a sorted list of the  t  values
        dts = []
        for dir in os.listdir(root):
            m = re.search(r'(?<=dt_).*', dir)
            dts.append(float(m.group(0)))
        dts.sort(reverse=True)

        # Set the reference point for error calculation
        ref_pnt = 0.0
        with open(root / f'dt_{dts[-1]}' / f'{method}_data.csv', newline='') as csvfile:
            reader = csv.reader(csvfile)
            for row in reader:
                if row[1] == str(time):
                    ref_pnt = float(row[3])
                    break

        # Collect list of error data
        error_list = []
        for dt in dts:
            error_frame = [dt]
            for method in methods:
                with open(root / f'dt_{dt}' / f'{method}_data.csv', newline='') as csvfile:
                    reader = csv.reader(csvfile)
                    for row in reader:
                        if row[1] == str(time):
                            error_frame.append(abs(float(ref_pnt) - float(row[3])))
            error_list.append(error_frame)

        # Create labeled dataframe for easier data manipulation
        df = pd.DataFrame(error_list)
        columns = ['dt'] + methods
        df.rename(columns=dict(enumerate(columns, start=0)), inplace=True)

        # Create combined plot of data values
        out_file = error_root / q / f"error.png"
        if not os.path.isdir(Path(out_file).parent):
            os.makedirs(Path(out_file).parent)
        for method in methods:
            plt.plot(df['dt'], df[method], label=method) # Plot the single method
        # plt.title(f"Method error vs  t  - {q.replace('_',' ')}")
        plt.xlabel(" t  [sec]")
        plt.ylabel("error [m]")
        plt.yscale('log')
        plt.xscale('log')
        plt.legend()
        plt.savefig(error_root / q / f"error.png",  bbox_inches = 'tight')
        plt.clf()

        # CREATE R@T VS  T  GRAPH ---------------------------------------------------- #
```

```
        # Collect list of R values
        r_list = []
        for dt in dts:
            r_frame = [dt]
            for method in methods:
                with open(root / f'dt_{dt}' / f'{method}_data.csv', newline='') as csvfile:
                    reader = csv.reader(csvfile)
                    for row in reader:
                        if row[1] == str(time):
                            r_frame.append(float(row[3]))
            r_list.append(r_frame)

        # Create labeled dataframe for easier data manipulation
        df = pd.DataFrame(r_list)
        columns = ['dt'] + methods
        df.rename(columns=dict(enumerate(columns, start=0)), inplace=True)

        # Create combined plot of data values
        out_file = error_root / q / f"r_at_t.png"
        if not os.path.isdir(Path(out_file).parent):
            os.makedirs(Path(out_file).parent)
        for method in methods:
            plt.plot(df['dt'], df[method], label=method) # Plot the single method
        # plt.title(f"Method error vs  t  - {q.replace('_',' ')}")
        plt.xlabel(" t  [sec]")
        plt.ylabel("r [m]")
        plt.xscale('log')
        plt.legend()
        plt.savefig(error_root / q / f"r_at_t.png",  bbox_inches = 'tight')
        plt.clf()

    # VARIED BUBBLE DIAMETER ---------------------------------------------------------- #

    # Run the Q2 set
    for i, size in enumerate(Q2_size_set, start=1):
        dt = Q2_dt_set[-2]
        print(f"\nRunning ({i}/{len(Q2_size_set)}) Q2 with size = {size}")
        run_q2(dt, Q2_T_star, Q2_slope_lim, Q2_size_variance_root / str(size), r_0=size)
```