An analysis of numerical methods for solving the particle trajectory equation: A modified replication study

Robert Kimelman

JHU, EN 625.61.81.FA23: Computational Methods

December 8, 2023

1 Problem Statement:

The purpose of this research project is to replicate a previous study which conducted numerical experiments for solving a differential equation known as the particle trajectory equation. The previous work compared their numerical results to experimental results in which a spherical object moves vertically in water and obtains a maximum depth of penetration. This scenario can be modeled by Newton's laws of motion, which results in a system of ordinary differential equations to be mathematically detailed later in this paper. The previous study implemented computer code to solve this system in the form of the following algorithms: forward Euler's method, backward Euler's method, the trapezoidal rule, a fourth-order Runge-Kutta method, a fourth-order Adams-Bashforth and Adams-Moulton method (which the researchers state as a predictor-corrector method), and a fourth-order iterative (Adams-Moulton) method [Primary1].

2 Mathematical Foundation:

2.1 Particle Trajectory Equation:

The system of differential equations that we are trying to solve is broken down as follows:

$$\frac{4}{3}\pi R_p^3(\rho_p + C_A\rho)\frac{du}{dt} = \frac{4}{3}\pi R_p^3 g(\rho_p - \rho) - C_D\frac{\pi R_p^2\rho}{2}u|u| \tag{1}$$

with

$$u = \frac{dz}{dt} \tag{2}$$

where g is a specific gravity of 0.99 with the initial condition that at the points $t = 0$ and $z = 0$, and $\rho_p = 600kg/m^3, \rho = 10^3kg/m^3, u = u_{entry}$. In this case, we start with the initial condition that $u_{entry} = 0$.

Moving forward towards obtaining a numerical solution, we can rearrange equation (1) as follows, isolating $\frac{du}{dt}$, by first dividing by the term being multiplied by $\frac{du}{dt}$:

$$\frac{du}{dt} = \frac{\rho_p - \rho}{\rho_p + C_A\rho}g - \frac{3C_D\rho}{8R_p}\frac{u|u|}{\rho_p + C_A\rho} \tag{3}$$

Integrating both sides in accordance with the Eulerian integration procedure, which is the integration strategy employed and cited by the original paper (for more detailed information, visit [Secondary4]), we obtain:

$$u_{t+\Delta t} = \frac{\rho_p - \rho}{\rho_p + C_A\rho}g\Delta t - \frac{3C_{D,t}\rho}{8R_p}\frac{u_t|u_t|}{\rho_p + C_A\rho}\Delta t + u_t \tag{4}$$

with $\Delta t = 10^{-3}s$. We also have $C_A = 0.5, C_D \approx 0.385$, and $R_p = 10^{-2}m$. All values are as given in the original paper or in [Primary2]. In addition, [Secondary4] mentions that our error in the Eulerian integration procedure will be proportional to the step size.

2.1.1 Forward Euler's Method:

For the forward Euler method, we compute the solution iteratively in the following manner:

$$u_{i+1} = u_i + h * \frac{du}{dt}(t_i, u_i) \tag{5}$$

2.1.2 Backward Euler's Method:

For the Backward Euler method, we compute the solution iteratively in the following manner:

$$u_{i+1} = u_i + h * \frac{du}{dt}(t_{i+1}, u_{i+1}) \tag{6}$$

2.1.3 Trapezoidal Rule:

For the Trapezoidal Rule, we compute the solution iteratively in the following manner:

$$k_1 = \frac{du}{dt}(t_i, u_i); \tag{7}$$

$$k_2 = \frac{du}{dt}(t_{i+1}, u_i + h * k_1); \tag{8}$$

$$u_{i+1} = u_i + (h/2) * (k_1 + k_2); \tag{9}$$

2.1.4 Fourth-Order Runge-Kutta Method:

For the Fourth-Order Runge-Kutta method, we compute the solution iteratively in the following manner:

$$k_1 = \frac{du}{dt}(t, u_i); \tag{10}$$

$$k_2 = \frac{du}{dt}(t + h/2, u_i + (h/2) * k_1); \tag{11}$$

$$k_3 = \frac{du}{dt}(t + h/2, u_i + (h/2) * k_2); \tag{12}$$

$$k_4 = \frac{du}{dt}(t + h, u_i + h * k_3); \tag{13}$$

$$u_{i+1} = u_i + (h/6) * (k_1 + 2 * k_2 + 2 * k_3 + k_4) \tag{14}$$

2.1.5 Adams-Bashforth and Adams-Moulton Method:

For the combined Adams-Bashforth and Adams-Moulton predictor-corrector method, the formula for computing the solution iteratively is as follows:

Predictor step:

$$u_{i+1} = u_i + (3/2) * h * \frac{du}{dt}(t_i, u_i) - (1/2) * h * \frac{du}{dt}(t_{i-1}), u_{i-1}) \tag{15}$$

Corrector step:

$$u_{i+1} = u_i + (1/2) * h * (\frac{du}{dt}(t_{i+1}, u_{i+1}) + \frac{du}{dt}(t_i, u_i)); \tag{16}$$

For more details on the derivation, please refer to the following source as found in [Secondary2].

2.1.6 Fourth-Order Iterative (Adams-Moulton) Method:

For the Fourth-Order Adams-Moulton method, we compute the solution to the given differential equation iteratively as follows:

$$u_{i+1} = u_i + (h/24) * (9 * \frac{du}{dt}(t_{i+1}, u_{i+1}) + 19 * \frac{du}{dt}(t_i, u_i) - 5 * \frac{du}{dt}(t_{i-1}, u_{i-1}) + \frac{du}{dt}(t_{i-2}, u_{i-2}))) \quad (17)$$

For more details on the derivation, please refer to the following source as found in [Secondary1] [Secondary3].

3 Algorithm Description:

The following algorithms will be implemented in MATLAB code, to be detailed later in this paper.

3.1 Forward Euler's Method:

From our lecture notes in Lecture 10B, we know that the basic idea behind Forward Euler's method is to compute each step successively without solving a particular equation. Forward Euler's method is a one-step method for computing approximations of the solution to the given differential equation.

Pseudo-code:

Algorithm 1 Forward Euler

function FORWARDEULER
    $\rho_p \leftarrow 600$
    $\rho \leftarrow 10^3$
    $\gamma \leftarrow 0.6$
    $u_{entry} \leftarrow 0$
    $delta_t \leftarrow 10^{-3}$
    $C_A \leftarrow 0.5$
    $C_D \leftarrow 0.385$
    $R_P \leftarrow 10^{-2}$
    $g \leftarrow 0.99$
    $n \leftarrow$ *1000*
    $h \leftarrow$ *1/n*
    diffEq $\leftarrow$ particle trajectory equation
    $solutionVector \leftarrow zeros(size = n + 1)$
    $solutionVector_1 \leftarrow u_{entry}$
    loop:
    $t \leftarrow$ *(i-1)*h*
    $solutionVector_{i+1} \leftarrow solutionVector_i + diffEq(t, solutionVector_i)$
    go to *top*
    exactSolution $\leftarrow$ *computeExact*
    maxError $\leftarrow$ *max(abs(solutionVector-exactSolution))*

3.2 Backward Euler's Method:

From our lecture notes in Lecture 10B, we recall that the Backward Euler method is similar to the Forward Euler method, except that when stepping from (t_i, y_i) to (t_{i+1}, u_{i+1}) we will compute the slope at the second point first. This is the basic difference between the two algorithms.

Pseudo-code:

Algorithm 2 Backward Euler

function BACKWARDEULER
    $\rho_p \leftarrow 600$
    $\rho \leftarrow 10^3$
    $\gamma \leftarrow 0.6$
    $u_{entry} \leftarrow 0$
    $delta_t \leftarrow 10^{-3}$
    $C_A \leftarrow 0.5$
    $C_D \leftarrow 0.385$
    $R_P \leftarrow 10^{-2}$
    $g \leftarrow 0.99$
    $n \leftarrow$ *1000*
    $h \leftarrow$ *1/n*
    $solutionVector \leftarrow zeros(size = n + 1)$
    $solutionVector_1 \leftarrow u_{entry}$
    diffEq $\leftarrow$ particle trajectory equation
    loop:
    $t \leftarrow$ *i*h*
    $solutionVector_{i+1} \leftarrow solutionVector_i + h * diffEq(t, solutionVector_{i+1})$
    go to *top*
    exactSolution $\leftarrow$ *computeExact*
    maxError $\leftarrow$ *max(abs(solutionVector-exactSolution))*

3.3 Trapezoidal Rule:

From our lecture notes in Lecture 10B, we remember that the Trapezoidal Rule is similar to the Backward Euler method, in that we compute the slope as the average at both end points in the interval $[t_1, t_2]$, with the main similarity being that we compute the right end point backward. We also implement a different formula for ϕ, which will be detailed in the mathematical foundation section of this paper.

Pseudo-code:

Algorithm 3 Trapezoidal Rule

function TRAPEZOIDALRULE
    diffEq ← particle trajectory equation
    $\rho_p \leftarrow 600$
    $\rho \leftarrow 10^3$
    $\gamma \leftarrow 0.6$
    $u_{entry} \leftarrow 0$
    $delta_t \leftarrow 10^{-3}$
    $C_A \leftarrow 0.5$
    $C_D \leftarrow 0.385$
    $R_P \leftarrow 10^{-2}$
    $g \leftarrow 0.99$
    $n \leftarrow$ *1000*
    $h \leftarrow$ *1/n*
    $solutionVector \leftarrow zeros(size = n + 1)$
    $solutionVector_1 \leftarrow u_{entry}$
    diffEq ← particle trajectory equation
    loop:
    $k_1 \leftarrow diffEq(t_i, solutionVector_i)$
    $k_2 \leftarrow diffEq(t_{i+1}, solutionVector_i + h * k1)$
    $solutionVector_{i+1} \leftarrow solutionVector_i + (h/2) * (k1 + k2)$
    go to *top*
    exactSolution ← *computeExact*
    maxError ← *max(abs(solutionVector-exactSolution))*

3.4 Fourth-Order Runge-Kutta Method:

The Fourth-Order Runge-Kutta method is similar to the Trapezoidal Method but of order four.

Pseudo-code:

Algorithm 4 Runge-Kutta

function RUNGEKUTTA
    $\rho_p \leftarrow 600$
    $\rho \leftarrow 10^3$
    $\gamma \leftarrow 0.6$
    $u_{entry} \leftarrow 0$
    $delta_t \leftarrow 10^{-3}$
    $C_A \leftarrow 0.5$
    $C_D \leftarrow 0.385$
    $R_P \leftarrow 10^{-2}$
    $g \leftarrow 0.99$
    $n \leftarrow$ *1000*
    $h \leftarrow$ *1/n*
    $solutionVector \leftarrow zeros(size = n + 1)$
    $solutionVector_1 \leftarrow u_{entry}$
    diffEq $\leftarrow$ particle trajectory equation
    loop:
    $t \leftarrow (i - 1) * h$
    $k_1 \leftarrow diffEq(t, solutionVector_i)$
    $k_2 \leftarrow diffEq(t + h/2, solutionVector_i + (h/2) * k_1)$
    $k_3 \leftarrow diffEq(t + h/2, solutionVector_i + (h/2) * k_2)$
    $k_4 \leftarrow diffEq(t + h, solutionVector_i + h * k_3)$
    $solutionVector_{i+1} \leftarrow solutionVector_i + (h/6) * (k_1 + 2 * k_2 + 2 * k_3 + k_4)$
    go to *top*
    exactSolution $\leftarrow$ *computeExact*
    maxError $\leftarrow$ *max(abs(solutionVector-exactSolution))*

3.5 Adams-Bashforth and Adams-Moulton Method:

The Adams-Bansforth method is a two-step method, for which the mathematical details will be described later in this paper.

Pseudo-code:

Algorithm 5 Adams-Bashforth and Adams-Moulton

function AdamsBashforth
    $\rho_p \leftarrow 600$
    $\rho \leftarrow 10^3$
    $\gamma \leftarrow 0.6$
    $u_{entry} \leftarrow 0$
    $delta_t \leftarrow 10^{-3}$
    $C_A \leftarrow 0.5$
    $C_D \leftarrow 0.385$
    $R_P \leftarrow 10^{-2}$
    $g \leftarrow 0.99$
    $n \leftarrow$ *1000*
    $h \leftarrow$ *1/n*
    $solutionVector \leftarrow zeros(size = n + 1)$
    $solutionVector_1 \leftarrow u_{entry}$
    diffEq $\leftarrow$ particle trajectory equation
    loop:
    $solutionVector_{i+1} = solutionVector_i + (3/2) * h * diffEq(t_i, solutionVector_i) - (1/2) * h * diffEq(t_{i-1}, solutionVector_{i-1})$
    $solutionVector_{i+1} = solutionVector_i + (1/2) * h * (diffEq(t_{i+1}, solutionVector_{i+1}) + diffEq(t_i, solutionVector_i))$
    go to *top*
    exactSolution $\leftarrow$ *computeExact*
    maxError $\leftarrow$ *max(abs(solutionVector-exactSolution))*

3.6 Fourth-Order Iterative (Adams-Moulton) method:

The Adams-Moulton method is similar to the Fourth-Order Runge-Kutta method, but the solution can be determined in one line of code, as shown below.

Pseudo-code:

Algorithm 6 Adams-Moulton

1: **function** AdamsBashforth
2: $\rho_p \leftarrow 600$
3: $\rho \leftarrow 10^3$
4: $\gamma \leftarrow 0.6$
5: $u_{entry} \leftarrow 0$
6: $delta_t \leftarrow 10^{-3}$
7: $C_A \leftarrow 0.5$
8: $C_D \leftarrow 0.385$
9: $R_P \leftarrow 10^{-2}$
10: $g \leftarrow 0.99$
11: $n \leftarrow$ *1000*
12: $h \leftarrow$ *1/n*
13: $solutionVector \leftarrow zeros(size = n + 1)$
14: $solutionVector_1 \leftarrow u_{entry}$
15: *diffEq* $\leftarrow$ particle trajectory equation
16: *loop*:
17: $solutionVector_{i+1} = solutionVector_i + (h/24)(9 * diffEq(t_{i+1}, solutionVector_{i+1}) + 19 * diffEq(t_i, solutionVector_i) - 5 * diffEq(t_{i-1}, solutionVector_{i-1}) + diffEq(t_{i-2}, solutionVector_{i-2})))$
18: **go to** *top*
19: *exactSolution* $\leftarrow$ *computeExact*
20: *maxError* $\leftarrow$ *max(abs(solutionVector-exactSolution))*

4 Well-Posedness (including stability):

4.1 Particle Trajectory Equation:

As given in our Lecture Notes in Lecture 1B, the definition for a well-posed problem is as follows:

1. A solution exists
2. The solution is unique
3. The solution's behavior changes continuously with the initial conditions. (Stability)

---

1. For our particular purposes, an analytical solution does exist, so long as we abide by particular assumptions about physical constraints and values, namely, we assume that C_D is constant [Primary1].
2. The solution is unique in this case, as detailed in the study that we are replicating here (again, since we are assuming that C_D is constant) [Primary1].
3. In our case, we can test for stability using code by implementing minor errors in our input values for our algorithms, seeing if they lead to very large changes in the resulting solution to the differential equation.

The basic code for (3) is as follows. We are just making a change of $\epsilon = 0.01$ to the measurement data:

```
epsilon = 0.01;
rho_p = 600+epsilon;
rho = 10^3+epsilon;
gamma = 0.6+epsilon;
u_entry = 0+epsilon;
delta_t = 10^(-3)+epsilon;
```

```
C_A = 0.5+epsilon;

C_D = 0.385+epsilon;

R_P=10^(-2)+epsilon;

g=0.99+epsilon;
```

Here is the output for each respective method (n=1000):

Method	Max. Error	ϵ
Forward Euler	2.2204e-16	0
Backward Euler	0.00038047	0
Runge-Kutta	3.012e-07	0
Trapezoidal	3.0158e-07	0
Bashforth and Moulton	0.00036	0
Adams-Moulton	0.00036	0
Forward Euler	0.20073	0.01
Backward Euler	0.0039925	0.01
Runge-Kutta	0.0036303	0.01
Trapezoidal	0.0036303	0.01
Bashforth and Moulton	0.0060288	0.01
Adams-Moulton	0.0060288	0.01
Forward Euler	0.056505	0.001
Backward Euler	0.00072224	0.001
Runge-Kutta	0.00036005	0.001
Trapezoidal	0.00036153	0.001
Bashforth and Moulton	0.00036004	0.001
Adams-Moulton	0.00072007	0.001
Forward Euler	0.0076787	0.0001
Backward Euler	0.00039674	0.0001
Runge-Kutta	3.6005e-05	0.0001
Trapezoidal	3.618e-05	0.0001
Bashforth and Moulton	0.000296	0.0001
Adams-Moulton	0.000396	0.0001

Therefore, we can see that all methods are stable as the maximum errors are still relatively low. The similarity in error between Adams-Bashforth and Adams-Moulton, and Adams-

Moulton is likely due to the fact that the former is an adaptation of the latter.

5 Rate of convergence (instead of conditioning as mentioned in office hours):

I modified my code for each of the following algorithms in the following way (n=1000):

```
rate_of_convergence = (u(1000)-u(999))/(u(999)-u(998));


disp(['Rate of convergence: ' num2str(rate_of_convergence)]);
```

[Primary3]

I got the following rates of convergence for each algorithm (n=1000):

Method	Rate of Convergence
Forward Euler	0.99578
Backward Euler	1.001
Runge-Kutta	0.99577
Trapezoidal	0.99579
Bashforth and Moulton	0.99577
Adams-Moulton	0.99678

Therefore, we can see that each algorithm has a rate of convergence of ≈ 1, as desired.

6 Theoretical error analysis:

According to our Lecture Notes in Lecture 11B as well as the computations in the other two algorithms, we have the following orders of accuracy:

- Forward Euler: k = 1
- Backward Euler: k =1
- Trapezoidal: k=2
- Fourth-Order Runge-Kutta: k = 4
- Adams-Bashforth and Adams-Moulton: k = 1
- Adams-Moulton: k = 1

Also, as given in our textbook Lyche et. al, we have the definition for consistency error as:

$$\epsilon(u) := \Sigma_{i=1}^{n} \delta_i^h(u) \tag{18}$$

and the scheme is consistent if

$$lim_{H\to 0} max_i \epsilon(u) = 0 \tag{19}$$

for any solution u. Equivalently, we have

$$lim_{H\to 0} max_i [\frac{\delta_i^h(u)}{h_i}] = 0 \tag{20}$$

where

$$\delta_i^h(u) := ||u(t_{i+1}) - u(t_i) - h_i \phi(t_i, u(t_i), h_i)|| \tag{21}$$

for i=1,...,n.

6.1 Forward Euler:

We have, as above:

$$\delta_i^h(u) := ||u(t_{i+1}) - u(t_i) - h_i\phi(t_i, u(t_i), h_i)|| \tag{22}$$

for i=1,...,n.

Then given Forward Euler's method, we have

$$u_{i+1} - u_i = h\frac{du}{dt}(t_i, u_i) \tag{23}$$

Hence

$$u_{i+1} - u_i - h\frac{du}{dt}(t_i, u_i) = 0 \tag{24}$$

Therefore

$$lim_{H\to 0} max_i[\frac{\delta_i^h(u)}{h_i}] = 0 \tag{25}$$

So the Forward Euler method is a consistent scheme.

6.2 Backward Euler:

We have, as above:

$$\delta_i^h(u) := ||u(t_{i+1}) - u(t_i) - h_i\phi(t_i, u(t_i), h_i)|| \tag{26}$$

for i=1,...,n.

Then given Backward Euler's method, we have

$$u_{i+1} - u_i = h\frac{du}{dt}(t_{i+1}, u_{i+1}) \tag{27}$$

Hence

$$u_{i+1} - u_i - h\frac{du}{dt}(t_{i+1}, u_{i+1}) = 0 \tag{28}$$

So we can conclude that the Backward Euler method is a consistent scheme.

6.3 Trapezoidal Rule:

We have, as above:

$$\delta_i^h(u) := ||u(t_{i+1}) - u(t_i) - h_i\phi(t_i, u(t_i), h_i)|| \tag{29}$$

for i=1,...,n.

Then given the Trapezoidal rule, we have

$$u_{i+1} - u_i = (h/2) * (\frac{du}{dt}(t_i, u_i) + \frac{du}{dt}(t_{i+1}, u_i + h * \frac{du}{dt}(t_i, u_i)) \tag{30}$$

Hence

$$u_{i+1} - u_i - (h/2) * (\frac{du}{dt}(t_i, u_i) + \frac{du}{dt}(t_{i+1}, u_i + h * \frac{du}{dt}(t_i, u_i)) = 0 \tag{31}$$

Therefore

$$lim_{H\to 0} max_i[\frac{\delta_i^h(u)}{h_i}] = 0 \tag{32}$$

So the Trapezoidal rule is a consistent scheme.

6.4 Fourth-Order Runge-Kutta:

We have, as above:

$$\delta_i^h(u) := ||u(t_{i+1}) - u(t_i) - h_i\phi(t_i, u(t_i), h_i)|| \tag{33}$$

for i=1,...,n.

Then given the Fourth-Order Runge-Kutta Method, we have

$$u_{i+1} - u_i = (h/2) * (\frac{du}{dt}(t_i, u_i) + \frac{du}{dt}(t_{i+1}, u_i + h * \frac{du}{dt}(t_i, u_i)) \tag{34}$$

Hence due to the same logic as the Trapezoidal rule, we have

$$lim_{H \to 0} max_i [\frac{\delta_i^h(u)}{h_i}] = 0 \tag{35}$$

So the Fourth-Order Runge-Kutta Method is a consistent scheme.

6.5 Adams-Bashforth:

We have, as above:

$$\delta_i^h(u) := ||u(t_{i+1}) - u(t_i) - h_i \phi(t_i, u(t_i), h_i)|| \tag{36}$$

for i=1,...,n.

Then given the Adams-Bashforth and Adams-Moulton method, we have in the Corrector Step:

$$u_{i+1} - u_i - (h/2) * (\frac{du}{dt}(t_i, u_i) + \frac{du}{dt}(t_{i+1}, u_{i+1}) + h * \frac{du}{dt}(t_i, u_i)) = 0 \tag{37}$$

Hence due to the same logic as the Fourth-Order Runge-Kutta method, we have

$$lim_{H \to 0} max_i [\frac{\delta_i^h(u)}{h_i}] = 0 \tag{38}$$

So the Adams-Bashforth and Adams-Moulton method is a consistent scheme.

6.6 Adams-Moulton Method:

We have, as above:

$$\delta_i^h(u) := ||u(t_{i+1}) - u(t_i) - h_i \phi(t_i, u(t_i), h_i)|| \tag{39}$$

for i=1,...,n.

Given the formula for the Adams-Moulton method:

$$u_{i+1} = u_i + (h/24) * (9 * \frac{du}{dt}(t_{i+1}, u_{i+1}) + 19 * \frac{du}{dt}(t_i, u_i) - 5 * \frac{du}{dt}(t_{i-1}, u_{i-1}) + \frac{du}{dt}(t_{i-2}, u_{i-2}))) \quad (40)$$

This formula yields a consistent scheme.

7 Numerical error analysis:

I tested several values of n to see what the error output is for each method and got the following table on the next page:

Method	Max. Error	N
Forward Euler	0.13652	100
Backward Euler	0.0036632	100
Runge-Kutta	0.0032476	100
Trapezoidal	0.0032479	100
Bashforth and Moulton	0.0032391	100
Adams-Moulton	0.00324	100
Forward Euler	0.12209	150
Backward Euler	0.0024406	150
Runge-Kutta	0.0020455	150
Trapezoidal	0.0020456	150
Bashforth and Moulton	0.0020397	150
Adams-Moulton	0.00204	150
Forward Euler	0.097923	250
Backward Euler	0.0014683	250
Runge-Kutta	0.0010832	250
Trapezoidal	0.0010832	250
Bashforth and Moulton	0.0010799	250
Adams-Moulton	0.00108	250
Forward Euler	2.2204e-16	1000
Backward Euler	0.00038047	1000
Runge-Kutta	3.012e-07	1000
Trapezoidal	3.0158e-07	1000
Bashforth and Moulton	0.00036	1000
Adams-Moulton	0.00036	1000

8 Comparison of theoretical and numerical errors:

8.1 Forward Euler's Method:

$$h_{theoretical} = (1/1000)^k = (1/1000)^1 = 0.001, e_{numerical} = 2.2204e - 16 \tag{41}$$

8.2 Backward Euler's Method:

$$h_{theoretical} = (1/1000)^k = (1/1000)^1 = 0.001, e_{numerical} = 0.00038047 \tag{42}$$

8.3 Trapezoidal Rule:

$$h_{theoretical} = (1/1000)^k = (1/1000)^2 = 0.000001, e_{numerical} = 3.0158e - 07 \tag{43}$$

8.4 Fourth-Order Runge-Kutta Method:

$$h_{theoretical} = (1/1000)^k = (1/1000)^4, e_{numerical} = 3.012e - 07 \tag{44}$$

8.5 Adams-Bashforth and Adams-Moulton Method:

$$h_{theoretical} = (1/1000)^k = (1/1000)^1 = 0.001, e_{numerical} = 0.00036 \tag{45}$$

8.6 Fourth-Order Iterative (Adams-Moulton) Method:

$$h_{theoretical} = (1/1000)^k = (1/1000)^1 = 0.001, e_{numerical} = 0.00036 \tag{46}$$

Additionally, we have the following table:

Method	Consistency
Forward Euler	Consistent
Backward Euler	Consistent
Runge-Kutta	Consistent
Trapezoidal	Consistent
Bashforth and Moulton	Consistent
Adams-Moulton	Consistent

According to the theoretical and numerical results above, we see that only the Runge-Kutta Method Code is not in agreement with theoretical errors, but the error is still extremely small for n = 1000. Therefore, for the particle trajectory equation, it would be best not to utilize this method if possible.

9 Numerical Results:

```
>> exact_solution(n=1000)


ans =


   -0.1614
```

9.1 Forward Euler's Method:

```
>> u(n=1000)


ans =


    -0.1609
```

9.2 Backward Euler's Method:

```
>> u(n=1000)


ans =


    -0.1705
```

9.3 Trapezoidal Rule:

```
>> u(n=1000)
```

```
ans =

   -0.1608
```

9.4 Fourth-Order Runge-Kutta Method:

```
>> u(n=1000)

ans =

   -0.1614
```

9.5 Adams-Bashforth and Adams-Moulton Method:

```
>> u(n=1000)

ans =

   -0.1614
```

9.6 Fourth-Order Iterative (Adams-Moulton) Method:

```
>> u(n=1000)

ans =

    -0.1963
```

10 Appendix (Code):

Through a great deal of experimentation, I optimized n to find the n at which the error is minimized for each method: first through trial-and-error, and then by running codes.

10.1 Forward Euler's Method:

```
% Define values for each variable in the differential equation
rho_p = 600;
rho = 10^3;
gamma = 0.6;
u_entry = 0;
delta_t = 10^(-3);
C_A = 0.5;
C_D = 0.385;
R_P=10^(-2);
g=0.99;
% Define the differential equation function
dudt = @(t,u) (rho_p-rho)/(rho_p+C_A*rho)*g-(3*C_D*rho*u*abs(u)
   )/(8*R_P*(rho_p+C_A*rho));

% Define the number of steps
n = 1000;

% Compute the step size
h = 1/n;

% Initialize the solution vectors
```

```
u = zeros(n+1, 1);
exact_solution = zeros(n+1, 1);
exact_solution(1) = u_entry;
u(1) = u_entry;

% Perform the forward Euler method
for i = 1:n
    t = (i-1)*h;
    u(i+1) = u(i) + h*dudt(t, u(i));
end

% Exact solution as given in the paper
% Obtain an analytical solution using u_{t+\Delta t}
t_exact = linspace(0, 1, n+1);
for i = 1:n
    t_exact(i+1)=t_exact(i)+delta_t;
    exact_solution(i+1) = u(i) + (rho_p-rho)/(rho_p+C_A*rho)*g*
       delta_t-(3*C_D*rho*u(i)*abs(u(i)))/(8*R_P*(rho_p+C_A*rho
       ))*delta_t;
end


% Compute the maximum error
max_error = max(abs(u - exact_solution));

% Display the maximum error
disp(['Maximum error: ' num2str(max_error)]);
```

10.2 Backward Euler's Method:

For the Backward Euler's method, I modified the above code as follows:

```
% Perform the backward Euler method
for i = 1:n
    t = (i+1)*h;
    u(i+1) = u(i) + h*dudt(t, u(i+1));
end
```

10.3 Trapezoidal Rule:

For the Trapezoidal Rule, I modified the Forward Euler method code as follows:

```
% Perform the trapezoidal method
for i = 1:n
    k1 = dudt(t(i), u(i));
    k2 = dudt(t(i+1), u(i) + h*k1);
    u(i+1) = u(i) + (h/2)*(k1 + k2);
end
```

10.4 Fourth-Order Runge-Kutta Method:

For the Fourth-Order Runge-Kutta Method, I modified the Forward Euler method code as follows:

```
% Perform the fourth-order Runge-Kutta method
for i = 1:n
    t = (i-1)*h;
    k1 = dudt(t, u(i));
    k2 = dudt(t + h/2, u(i) + (h/2)*k1);
```

```
    k3 = dudt(t + h/2, u(i) + (h/2)*k2);
    k4 = dudt(t + h, u(i) + h*k3);
    u(i+1) = u(i) + (h/6)*(k1 + 2*k2 + 2*k3 + k4);
end
```

10.5 Adams-Bashforth and Adams-Moulton Method:

For the Adams-Bansforth and Adams-Moulton Method, I modified the Forward Euler method code as follows:

```
% Perform the Adams-Bansforth and Adams-Moulton method
for i = 2:n
    % Predictor Step
    u(i+1) = u(i) + (3/2)*h*dudt(t(i), u(i))-(1/2)*h*dudt(t(i
       -1),u(i-1));
    % Corrector Step
    u(i+1) = u(i) + (1/2)*h*(dudt(t(i+1), u(i+1))+dudt(t(i),u(i
       )));
end
```

10.6 Fourth-Order Iterative (Adams-Moulton) Method:

For the Adams-Moulton Method, I modified the Forward Euler method code as follows:

```
% Perform the Adams-Moulton method
for i = 3:n
    u(i+1) = u(i) + (h/24)*(9*dudt(t(i+1), u(i+1)+19*dudt(t(i),
        u(i))-5*dudt(t(i-1), u(i-1))+dudt(t(i-2), u(i-2))));
end
```

References

Robert Kimelman

PRIMARY SOURCES

[Primary1] D. Mazumdar and R. I. L. Guthrie, "An analysis of numerical methods for solving the particle trajectory equation," *Appl. Math. Modelling*, 1988.

[Primary2] D. Mazumdar, "Fluid flow, particle motion and mixing in ladle metallurgy operations," Ph.D. dissertation, McGill University, Canada, 1985.

[Primary3] A. H. Van Tuyl, "Acceleration of convergence of a family of logarithmically convergent sequences," *Mathematics of Computation*, 1994.

[Primary1] [Primary2] [Primary3]

SECONDARY SOURCES

[Secondary1] "Adams methods." Available at https://www.cs.unc.edu/~smp/COMP205/LECTURES/DIFF/lec19/node2.html.

[Secondary2] "Adams-bashforth and adams-moulton methods," Available at https://en.wikiversity.org/wiki/Adams-Bashforth_and_Adams-Moulton_methods.

[Secondary3] "Write pseudo-code in latex," Available at https://tex.stackexchange.com/questions/163768/write-pseudo-code-in-latex.

[Secondary4] "Euler integration." Available at https://kevgildea.github.io/blog/Kinematic-differential-equations/.

[Secondary1] [Secondary2] [Secondary3] [Secondary4]