

Games Architecture and Mechanics Assignment 1 Report

by Sam Mifsud

Submission date: 30-Nov-2024 03:57PM (UTC+0000)

Submission ID: 244199114

File name:

50540_Sam_Mifsud_Games_Architecture_and_Mechanics_Assignment_1_Report_628791_1909551813.docx
(471.5K)

Word count: 760

Character count: 3645

2024

Assignment 1 Report

GAMES ARCHITECTURE AND MECHANICS ACCB5017
SAM MIFSUD (2305369)

Table of Contents

Part 1 – Numerical Integration Methods	2
Method 1 – Euler's Method	2
Method 2 – The Runge Kutta Method (RK4)	2
Part 2 – Simulation Programming Task	3
Simulation 1 – Euler's Method.....	3
Simulation 2 – The Runge-Kutta Method (RK4)	4
Simulation Results.....	4
Euler's.....	5
RK4.....	6
Testing.....	6
References.....	10

Part 1 – Numerical Integration Methods

Method 1 – Euler's Method



Method 2 – The Runge Kutta Method (RK4)

Part 2 – Simulation Programming Task

For the spring powered car simulation I ran two loops, one for each Numerical Integration method. The methods I used are the same two I discussed in Part 1, Euler's and The Runge-Kutta (Rk4) Methods. I had them write their results to separate files, and reused the variables between them by resetting them between loops.

For the code that was reused in multiple places and required testing I used a separate header file, and an additional header file was created for the testing code.

Simulation 1 – Euler's Method

```
while (!stopped) //main simulation loop
{
    //Simulation 1 - Euler's Method
    // We need to use Euler's Method twice, once for the velocity, to update it based on its acceleration, then its position, based on its current velocity
    //
    //Calculating the tractive force generated by the spring:
    tractForce = calculateTractForce(distance);

    //Calculating the drag on the vehicle
    dragForce = calculateDragForce(velocity, rollResist, vehMass, gravity, dragCoeff, airDensity, frontArea);

    acceleration = (tractForce - dragForce) / vehMass; //Calculating the change in acceleration of the vehicle
    velocity += acceleration * deltaTime; //Using Euler's Method to calculate change in velocity

    //Checking if drag force has reduced velocity to 0 or less after vehicle has started moving
    if (velocity <= 0 and distance > 0)
    {
        stopped = true; //Telling the loop to stop looping
        //Preventing the vehicle from going backwards when it should stop
        velocity = 0;
        acceleration = 0;
        dragForce = 0;
    }

    distance += velocity * deltaTime; //Using Euler's Method to calculate change in distance

    time += deltaTime; //Incrementing the total simulated time
    step++; //Incrementing the current simulation step
    fw.write(velocity, distance, step, time, acceleration); //Saving the current simulation step to the csv file
}
```

```
float integrateEulers(float changeInX, float deltaTime, float x)
{
    return x + changeInX * deltaTime;
}
```

Simulation 2 – The Runge-Kutta Method (RK4)

```
while (!stopped)
{
    //Simulation 2 - RK4 / The Runge Kutta Method
    // RK4 also needs to be used twice, for velocity and acceleration
    //
    //Calculating the tractive force generated by the spring:
    tractForce = calculateTractForce(distance);

    //Calculating the drag on the vehicle
    dragForce = calculateDragForce(velocity, rollResist, vehMass, gravity, dragCoeff, airDensity, frontArea);

    acceleration = (tractForce - dragForce) / vehMass; //Calculating the change in acceleration of the vehicle

    //Using RK4 to calculate velocity
    velocity = integrateRK4(acceleration, deltaTime, velocity);

    //Checking if drag force has reduced velocity to 0 or less after vehicle has started moving
    if (velocity <= 0 and distance > 0)
    {
        stopped = true; //Telling the loop to stop looping
        //Preventing the vehicle from going backwards when it should stop
        velocity = 0;
        acceleration = 0;
        dragForce = 0;
    }

    //Using RK4 to calculate position (distance) based on the velocity calculation
    distance = integrateRK4(velocity, deltaTime, distance);

    time += deltaTime; //Incrementing the total simulated time
    step++; //Increment current simulation step
    fw2.write(velocity, distance, step, time, acceleration); //Saving the current simulation step to the csv file
}

//RK4 method of numerical integration
float integrateRK4(float changeInX, float deltaTime, float x) //changeInX = y, deltaTime = h, x= x(n)
{
    float k1 = changeInX; //k1, 1st order, current change in x at start of step
    float k2 = changeInX + (deltaTime / 2) * k1; //k2, 2nd order, estimate of midpoint of the time step based on k1
    float k3 = changeInX + (deltaTime / 2) * k2; //k3, 3rd order, estimate of midpoint of the time step based on k2
    float k4 = changeInX + deltaTime * k3; //k4, 4th order, estimate of end point of the time step based on k3

    return x + (deltaTime / 6) * (k1 + (2 * k2) + (2 * k3) + k4); //Calculating new value using the k values calculated above
}
```

Simulation Results

Both the Euler's and RK4 method simulations produced similar results, with both their final distances being within .3m of one another and Euler's only taking one additional step. I did not have time to create a graph demonstrating both of their results

Euler's

649	0.4931	42.0697	648	6.48	-2.0606
650	0.4725	42.0744	649	6.49	-2.0606
651	0.4519	42.0789	650	6.5	-2.0606
652	0.4313	42.0832	651	6.51	-2.0605
653	0.4107	42.0873	652	6.52	-2.0605
654	0.3901	42.0912	653	6.53	-2.0604
655	0.3695	42.0949	654	6.54	-2.0604
656	0.3489	42.0984	655	6.55	-2.0604
657	0.3283	42.1017	656	6.56	-2.0603
658	0.3077	42.1048	657	6.57	-2.0603
659	0.2871	42.1077	658	6.58	-2.0603
660	0.2665	42.1103	659	6.59	-2.0603
661	0.2459	42.1128	660	6.6	-2.0602
662	0.2253	42.115	661	6.61	-2.0602
663	0.2047	42.1171	662	6.62	-2.0602
664	0.1841	42.1189	663	6.63	-2.0602
665	0.1635	42.1206	664	6.64	-2.0602
666	0.1429	42.122	665	6.65	-2.0602
667	0.1222	42.1232	666	6.66	-2.0601
668	0.1016	42.1242	667	6.67	-2.0601
669	0.081	42.125	668	6.68	-2.0601
670	0.0604	42.1256	669	6.69	-2.0601
671	0.0398	42.126	670	6.7	-2.0601
672	0.0192	42.1262	671	6.71	-2.0601
673	0	42.1262	672	6.72	0
674					

RK4

648	0.4919	42.3477	647	6.47	-2.0606
649	0.4712	42.3524	648	6.48	-2.0606
650	0.4505	42.357	649	6.49	-2.0606
651	0.4298	42.3613	650	6.5	-2.0605
652	0.4091	42.3654	651	6.51	-2.0605
653	0.3884	42.3693	652	6.52	-2.0604
654	0.3677	42.373	653	6.53	-2.0604
655	0.347	42.3765	654	6.54	-2.0604
656	0.3263	42.3798	655	6.55	-2.0603
657	0.3056	42.3828	656	6.56	-2.0603
658	0.2849	42.3857	657	6.57	-2.0603
659	0.2642	42.3884	658	6.58	-2.0603
660	0.2435	42.3908	659	6.59	-2.0602
661	0.2227	42.393	660	6.6	-2.0602
662	0.202	42.3951	661	6.61	-2.0602
663	0.1813	42.3969	662	6.62	-2.0602
664	0.1606	42.3985	663	6.63	-2.0602
665	0.1399	42.3999	664	6.64	-2.0602
666	0.1192	42.4011	665	6.65	-2.0601
667	0.0985	42.4021	666	6.66	-2.0601
668	0.0778	42.4029	667	6.67	-2.0601
669	0.0571	42.4035	668	6.68	-2.0601
670	0.0364	42.4038	669	6.69	-2.0601
671	0.0157	42.404	670	6.7	-2.0601
672	0	42.404	671	6.71	0
673					

Testing

To ensure my program was working correctly I created a series of tests for each function, to ensure their outputs were correct. I had to decide on test data for each function and come up with an input and calculate what the output should be if the code is working correctly.

I used a calculator to generate the data to test against. I would have liked to use premade programs specifically for numerical integration using Euler's and RK4 methods, as these would ensure that my functions are implemented correctly, as when manually calculating I could make the same mistakes in my formulas that I could have made in the code. However, I could not find a suitable calculator for

either method, as most required significantly more inputs and produced outputs in different formats to mine.

I created a testing harness via a function in a separate header file. It calls each function, sends it test inputs, checks its outputs against the pre-calculated results and prints a string to the console depending on whether it is correct or not. For each test that passed I would increment a value that stored the number of passed tests, so at the end I could show how many of the tests passed.

For the FileWriter function I created a new testing file, wrote some data to it and read the data back. If the read function output the column headings the file writer automatically generated, then the function is working correctly. The file can be manually read by the user through an external program to ensure the values passed were written correctly also.

```
int passedTests = 0;
//Testing the File Writer
FileWriter testFW("test.csv");
testFW.write(1, 2, 3, 4, 5);
testFW.close();
std::ifstream testFile;
testFile.open("test.csv");
std::string testString;
std::getline(testFile >> std::ws, testString);
std::cout << "Test string = " << testString << std::endl;
testFile.close();
if (testString == "Velocity(m/s), Position(m), Time Step(s), Time Elapsed(s), Acceleration (m/s/s)")
{
    std::cout << "FileWriter Test Passed" << std::endl;
    passedTests++;
}
else
{
    std::cout << "FileWriter Test Failed" << std::endl;
}
```

For the tractive force calculation function I ran the function twice with 2 different values for distance. One below 6 that should output a normal tractive force, and one above that should output 0. I checked that both outputs were correct, however I ran into a problem during testing where the function would return 14.4 and 0, the expected values the testing harness was looking for, but it would claim the test failed regardless. I believe this was caused by floating point inaccuracy [1] and was fixed by checking if the value for the first output was in a very small range around 14.4 instead.

```

//Testing the tractive force function
float testFloat = calculateTractForce(4);
std::cout << testFloat << std::endl;
float testFloat2 = calculateTractForce(10);
std::cout << testFloat2 << std::endl;
if (testFloat > 14.399999 and testFloat < 14.400001 and testFloat2 == 0)//Cl
{
    std::cout << "calculateTractForce Test Passed" << std::endl;
    passedTests++;
}
else
{
    std::cout << "calculateTractForce Test Failed" << std::endl;
}

```

For the drag force calculation I ran the function with a velocity of 10 and set all the other values to what they would be in the program normally. I once again checked that the value was within a small range to account for floating point error.

```

//Testing the drag force calculation
testFloat = calculateDragForce(10, 0.21, 1, 9.81, 0.34, 1.2, 0.01)
std::cout << testFloat << std::endl;
if (testFloat > 2.264 and testFloat < 2.2642)
{
    std::cout << "calculateDragForce Test Passed" << std::endl;
    passedTests++;
}
else
{
    std::cout << "calculateDragForce Test Failed" << std::endl;
}

```

For the Euler's Method calculation function I simply checked that if x started as 1 and increased by 2 per second for 1 second it would calculate the result as 3.

```

//Testing the Euler's integration method
testFloat = integrateEulers(2, 1, 1);
std::cout << testFloat << std::endl;
if (testFloat == 3)
{
    std::cout << "integrateEulers Test Passed" << std::endl;
    passedTests++;
}
else
{
    std::cout << "integrateEulers Test Failed" << std::endl;
}
}

```

Due to a lack of time, I was unable to test the RK4 calculation function, as its calculations are more complex and would require a more complex testing method than I could have implemented.

```
Velocity: 0, Position: 42.404, Time Step: 0.1, Time Elapsed: 0.71000, Acceleration: 0
Press any key to run function tests...

Velocity: 1, Position: 2, Time Step: 3, Time Elapsed: 4, Acceleration: 5
Test string = Velocity(m/s), Position(m), Time Step(s), Time Elapsed(s), Acceleration (m/s/s)
FileWriter Test Passed
14.4
0
calculateTractForce Test Passed
2.2641
calculateDragForce Test Passed
3
integrateEulers Test Passed
Total tests passed: 4/4
```

All of the tests I ran produced expected values and therefore all tests were passed. While the RK4 method did not get tested, in practice it appeared close to Euler's method, suggesting it is likely working correctly.

References

- [1] D. GOLDBERG, "What every computer scientist should know about floating-point arithmetic," *ACM computing surveys*, vol. 23, no. 1, p. 5–48, 1991.

Games Architecture and Mechanics Assignment 1 Report

GRADEMARK REPORT

FINAL GRADE

45/100

GENERAL COMMENTS

Engagement with Literature: 1/15

Knowledge and Understanding: 2/15

Cognitive and Intellectual Skills: 10/20

Practical Skills: 40/45

Transferable Skills: 2/5

Overall 45/100

Comment:

Assuming there were time constraints based on the lack of content in the report?

Great, well-commented code, but let down by a severe lack of content in the report? This could have been a great assignment!

See inline comments also

PAGE 1

PAGE 2

PAGE 3



Comment 1

Did you upload the wrong file?

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11
