# **Numerical Integration.**

## Question 1:

### Look at the simple dynamics exercise from week 1 and identify the source code where integration is evaluated. What integration method is used?

For rigidbodies, integration is performed within the step() method.

public void Step(float dt, World world)

{

...

...

//Euler Integration

Position = Position + LinearVelocity \* dt;

}

The method used is Euler integration. Seeing as both velocity and position are stored as vectors, there is no need to calculate all 3 dimensions separately. All 3 can be calculated within the same data structure.

## Question 2:

### Change the delta time used for the integration to a larger value. What effects does it have on the simulation?

The motion becomes more and more erratic and physically incorrect as the value of delta time increases. This is due to the incorrect evaluation of integrals when a high delta time value is used. Oscillation between correct values occurs at the highest values.

## Question 3:

### Implement and compare with other integration methods. E.g. 4th order Runge Kutta, mid-point, from the lecture.

...

...

//Euler Integration

//EulerIntegration(dt);

//Midpoint Integration.

//MidpointIntegration(world, acceleration, dt);

//RungeKutta integration.

RungeKuttaIntegration(acceleration, world, dt);

}

private void EulerIntegration(float dt)

{

Position = Position + LinearVelocity \* dt;

}

private void MidpointIntegration(World world, Vector2D acceleration, float dt)

{

//Calculate the mid point of the velocity.

Vector2D midVelocity = LinearVelocity + (acceleration + world.Gravity) \* 0.5f \* dt;

//Calculate the new velocity.

LinearVelocity = midVelocity \* dt;

//Update the position.

Position = Position + LinearVelocity \* dt;

}

private void RungeKuttaIntegration(Vector2D acceleration, World world, float dt)

{

Vector2D k1 = LinearVelocity + (acceleration + world.Gravity) \* dt;

Vector2D k2 = LinearVelocity + (k1 \* 0.5f \* dt);

Vector2D k3 = LinearVelocity + (k2 \* 0.5f \* dt);

Vector2D k4 = LinearVelocity + k3 \* dt;

Position = Position + (k1 + k2 \* 2.0f + k3 \* 2.0f + k4) \* (1.0f/6.0f) \* dt;

}

Midpoint provides the least accurate integration in my experiences. Euler’s method improves on it slightly but I find that the smoothest motion comes from the Runge Kutta integration. This makes sense given the additional terms and comparative irrelevance of dt in the Runge Kutta method.

## Question 4:

### It is often useful to decouple physics updates from rendering. How will you design a physics loop that is independent of the frame rate.

Unity shows how to achieve this very well, with an Update (every frame) and a fixedUpdate (every physics frame). I imagine that this method would calculate and update the stored positions and anything physics related on its fixedUpdate and then draw the frames in the update().

The most basic implementation of this would be a loop that includes a boolean on whether to update the rendering loop or not. Switching it every time the loop is ran would provide a 2:1 ratio for 120Hz physics processing to 60Hz rendering. This still does tie the frame rate to the physics implementation somewhat however, and may not be the best designed solution.