# NOTES

## 1. Mechanics

### 1.1. Derivation and integration.

### 1.2. Position, velocity and acceleration.

Let $p(t)$, $v(t)$ and $a(t)$ be the position, velocity and acceleration of an object in the plane or in three dimensional space. The relationship between these three are as follows

$$\frac{dp}{dt} = v(t)$$

$$\frac{dv}{dt} = a(t)$$

If we know the acceleration of an object we can integrate to find the velocity and position.

For example, assume an object is dropped $20m$ above groundlevel. We will calculate the time it takes for the object to hit the ground as well as the speed at time of impact. The only acceleration on the object is due to gravity, which for simplicity is assumed to be equal to

$$a(t) = -10$$

The velocity is

$$v(t) = -10t$$

and the position is

$$p(t) = -5t^2 + 20$$

Here we have added 20 to the position, since the position is 20 at time $t = 0$. We solve for $p(t) = 0$ to find the time of impact, which gives $t = 2$. The velocity at time $t = 2$ is

$$v(2) = -10 \cdot 2 = -20.$$

Often it is too difficult or inconvenent to integrate analytically, in which case we need to use numerical methods. The simplest is the (unstable) **explicit Euler integrator**

```
void GameObject::update(float dt)
{
    pos += vel * dt;
    vel += acc * dt;
}
```

The explicit Euler integrator adds energy to the system, so for instance a bouncing ball (without drag or loss of kinetic energy at collision) will increase its height over time. This integrator should only be used with with damping or by adding friction.

We can do slightly better with the **implicit Euler integrator**

```
void GameObject::update(float dt)
{
    vel += acc * dt;
    pos += vel * dt;
}
```

The implicit Euler integrator removes energy from the system, so for instance a bouncing ball will eventually stop moving. In other words, this integrator has damping built into it.

Numerical integration is a large and complicated field of study. There are a number of other integrators that can be used, some better than others. I refeer to the Fronter page for links to some of them.

## Problems:

**Exercise. 1.** *A ball is thrown from ground level directly upwards with a velocity* 10*. Calculate*

a) *The time it takes for the ball to reach its highest point.*
b) *The height above ground of its highest point.*
c) *The time it takes for the ball to hit the ground.*

**Exercise. 2.** *A cannonball is fired from the origin with speed* 100 *and a given angle* $\theta$*.*

a) *Calculate how far along the* $x$*-axis the cannonball travels if the firing angle is* $30°$*.*
b) *Calculate how far along the* $x$*-axis the cannonball travels if the firing angle is* $45°$*.*

**Exercise. 3.** *A grenade is thrown from the origin. The speed and angle can be adjusted, but the grenade explodes after exactly* 5 *seconds.*

*The target is located at distance* 10 *from the origin on the positive* $x$*-axis, and moves along the* $x$*-axis with a constant velocity* 4*. Calculate the angle and speed which makes the grenade hit the target exactly as it explodes.*

1.3. **Forces.** Forces are needed to create motion. The relationship is given by Newtons Law, which for us is equal to,

$$a = F/m$$

where $a$ is the acceleration, $F$ is the force and $m$ is the mass of the object.

```
class PhysicsObject
{
public:
      Vector pos, vel, acc, force;
      // good idea to store invMass = 1/mass:
      // One less division in update, and moreover
      // invMass == 0 means unmovable.
      float invMass, mass;

      void applyForce(vector nForce);
      void update(float dt);

};


void PhysicsObject::applyForce(vector nForce)
{
      force += nForce;
}


void PhysicsObject::update(float dt)
{
      acc = force * invMass;

      // integrate with the simplest possible integrator
      vel += acc * dt;
      pos += vel * dt;

      // reset
```

```
      force = Vector(0,0);

     // if system is unstable, you can try damping
     vel *= (1-dt)  // comes to complete stop in 1 second
     // or physically correct:  you can use applyForce with friction

    // you might also try putting vel = 0 if it is small enough
    // and do
    // vel *= maxVelocity / vel.length();
    // if the velocity is too high.
}

PhysicsObject po;

// What should happen when "a" is pressed
po.applyForce(Vector(-20,0));

// Gravity, if we are looking at objects from the side
po.applyForce(Vector(0,10) * op.mass);

// Friction, if we are looking at objects from above
po.applyForce(-op.vel * 10 * op.mass / op.vel.length());

// drag at high velocities
// dragConstant depends on the medium we are travelling through
// and the properties of the object
po.applyForce(-op.vel * op.vel.length() * dragConstant);

// drag at low velocities
// dragConstant depends on the medium we are travelling through
// and the properties of the object
po.applyForce(-op.vel * dragConstant);

po.update(timePassed);
```

## 1.4. **Elastic and inelastic collisions.**

1.4.1. *1d collisions.* We are given two objects moving along the $x$-axis, with masses $m_1$ and $m_2$, velocities before the collision $v_1$ and $v_2$, and velocities after the collision $u_1$ and $u_2$. We would like to calculate the velocities after collision, using the masses and the velocities before

collision. Collision also causes rotation, but this is beyond the scope of this course.

Using preservation of momentum

$$m_1v_1 + m_2v_2 = m_1u_1 + m_2u_2$$

and preservation of kinetic energy

$$\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 = \frac{1}{2}m_1u_1^2 + \frac{1}{2}m_2u_2^2$$

we get the formulas

$$v_1 = \frac{(m_1 - m_2)u_1 + 2m_2u_2}{m_1 + m_2}$$

$$v_2 = \frac{(m_2 - m_1)u_2 + 2m_1u_1}{m_1 + m_2}$$

If the collision is not elastic, the formulas are

$$v_1 = \frac{Cm_2(u_2 - u_1) + m_1u_1 + m_2u_2}{m_1 + m_2}$$

$$v_2 = \frac{Cm_1(u_1 - u_2) + m_1u_1 + m_2u_2}{m_1 + m_2}$$

where $C$ determines the elasticity. If $C = 1$, the collision is elastic, and we recover the formulas above. For a completely inelastic collision we set $C = 0$. An arbitrary collision would have coefficient $0 < C < 1$.

## Problems:

**Exercise. 4.** *Two objects are moving along the x-axis. Draw figures explaining the situation and calculate what happens when colliding, when*

a) $m_1 = 1 = m_2$, $u_1 = 1$ *and* $u_2 = -1$
b) $m_1 = 1 = m_2$, $u_1 = 2$ *and* $u_2 = -2$
c) $m_1 = 0$, $m_2 = 1$, $u_1 = 2$ *and* $u_1 = 1$
d) $m_1 = 0$, $m_2 = 1$, $u_1 = 2$ *and* $u_1 = -1$
e) $m_1 = 2$, $m_1 = 1$, $u_1 = 1$ *and* $u_2 = -1$

1.4.2. *2d collisions.* Collisions in 2d and 3d are very similar, and so we only consider the 2d case. Let the position of the two colliding objects $p_1$ and $p_2$, respectively, and let

$$n = \frac{p_2 - p_1}{|p_2 - p_1|},$$

and $t$ be a unit vector orthogonal to $n$. We decompose

$$u_1 = u_{1n}n + u_{1t}t \text{ and } u_2 = u_{2n}n + u_{2t}t$$

where $u_{in} = u_i \cdot n$ and $u_{it} = u_i \cdot t$. The velocities after the collision are given by

$$v_1 = v_{1n}n + v_{1t}t \text{ and } v_2 = v_{2n}n + v_{2t}t$$

where $v_{it} = u_{it}$ and $v_{in}$ is computed as a $1d$-collision. That is,

$$v_{1n} = \frac{(m_1 - m_2)u_{1n} + 2m_2u_{2n}}{m_1 + m_2}$$

$$v_{2n} = \frac{(m_2 - m_1)u_{2n} + 2m_1u_{1n}}{m_1 + m_2}$$

1.5. **Particle systems.**

```
class Particle
{
public:
    sf::Vector2f pos, vel;
    float life;
    Particle();
    void spawn(sf::Vector2f nPos, sf::Vector2f nVel, float nLife);
    void update(float dt);
    void draw(sf::RenderWindow& w);
};

void Particle::update(float dt)
{
    life -= dt;
    if(life > 0) {

// code for updating a single particle

    }
}


class Generator
{
public:
    Particle particles[500];
    float timeSinceLast;
    Generator();
```

```
    void update(float dt);
    void draw(sf::RenderWindow& w);
};


void Generator::update(float dt)
{
    timeSinceLast -= dt;
    for(int i = 0; i < 500; i++) {
        if(particles[i].life > 0) {
            particles[i].update(dt);
        } else if(timeSinceLast < 0) {
            // reset timeSinceLast
            // spawn particles[i] with constraints
        }
    }
}
```