

# Orbital Mechanics Library for C++ Documentation

Generated by Doxygen 1.9.5



<b>1 General Information</b>	<b>1</b>
1.0.0.1 C++ library for numerical calculations related to basic spacecraft motion issues, taking into account influence of gravity and vehicle propulsion.	1
1.1 Instalation	1
1.2 Operation	1
1.3 Units	2
1.4 json file	2
1.4.1 control	2
1.4.2 data	2
1.4.3 planets	3
1.4.4 ship	3
1.5 Example of Json file (Ship on the Earth orbit with a constant velocity):	3
<b>2 Class Index</b>	<b>5</b>
2.1 Class List	5
<b>3 File Index</b>	<b>7</b>
3.1 File List	7
<b>4 Class Documentation</b>	<b>9</b>
4.1 Control Class Reference	9
4.1.1 Detailed Description	9
4.1.2 Constructor & Destructor Documentation	10
4.1.2.1 Control()	10
4.1.3 Member Function Documentation	10
4.1.3.1 Check_input()	10
4.1.3.2 Print_Interval()	10
4.1.4 Member Data Documentation	10
4.1.4.1 engforce	10
4.1.4.2 timeend	11
4.1.4.3 timestart	11
4.2 Planet Class Reference	11
4.2.1 Detailed Description	12
4.2.2 Constructor & Destructor Documentation	12
4.2.2.1 Planet()	12
4.2.3 Member Function Documentation	12
4.2.3.1 Move_Planet()	12
4.2.3.2 Print_info()	12
4.2.4 Member Data Documentation	12
4.2.4.1 ang_velocity	13
4.2.4.2 isOrb	13
4.2.4.3 mass	13
4.2.4.4 name	13

4.2.4.5 orb_data	13
4.2.4.6 orb_pos	13
4.2.4.7 orb_radius	14
4.2.4.8 position	14
4.2.4.9 radius	14
4.2.4.10 start_ang	14
4.3 Solver Class Reference	14
4.3.1 Detailed Description	16
4.3.2 Member Enumeration Documentation	17
4.3.2.1 ode	17
4.3.3 Member Function Documentation	17
4.3.3.1 Adams_Bashford()	17
4.3.3.2 Calculate_Grav()	18
4.3.3.3 Calculate_Net()	18
4.3.3.4 Check_Collision()	18
4.3.3.5 dvdt()	18
4.3.3.6 dxdt()	19
4.3.3.7 Euler()	19
4.3.3.8 is_empty()	20
4.3.3.9 Load_data()	20
4.3.3.10 Load_file()	21
4.3.3.11 Midpoint()	21
4.3.3.12 Move_Orbit()	21
4.3.3.13 Populate()	22
4.3.3.14 Print_Pauses()	22
4.3.3.15 Push_Back()	22
4.3.3.16 Runge_Kutta()	22
4.3.3.17 Save_data()	23
4.3.3.18 Save_json()	23
4.3.3.19 Save_Planets()	23
4.3.3.20 Setup()	23
4.3.3.21 Solve()	23
4.3.3.22 UseEngine()	24
4.3.3.23 Validate_Json()	24
4.3.4 Member Data Documentation	24
4.3.4.1 distance	24
4.3.4.2 engine_data	24
4.3.4.3 engine_used	25
4.3.4.4 force_data	25
4.3.4.5 fuel_data	25
4.3.4.6 fuel_used	25
4.3.4.7 G	25

4.3.4.8 grav_forces . . . . .	25
4.3.4.9 kinetic_data . . . . .	26
4.3.4.10 mass_data . . . . .	26
4.3.4.11 method . . . . .	26
4.3.4.12 Planets . . . . .	26
4.3.4.13 position_data . . . . .	26
4.3.4.14 potential_data . . . . .	26
4.3.4.15 Ship . . . . .	27
4.3.4.16 step . . . . .	27
4.3.4.17 T . . . . .	27
4.3.4.18 time . . . . .	27
4.3.4.19 time_data . . . . .	27
4.3.4.20 TimeVect . . . . .	27
4.3.4.21 velocity_data . . . . .	28
4.4 Vector3D Class Reference . . . . .	28
4.4.1 Detailed Description . . . . .	29
4.4.2 Constructor & Destructor Documentation . . . . .	29
4.4.2.1 Vector3D() [1/2] . . . . .	29
4.4.2.2 Vector3D() [2/2] . . . . .	29
4.4.3 Member Function Documentation . . . . .	29
4.4.3.1 Add() . . . . .	29
4.4.3.2 Divide() . . . . .	30
4.4.3.3 Multiply() . . . . .	30
4.4.3.4 operator*() . . . . .	31
4.4.3.5 operator*=( ) . . . . .	31
4.4.3.6 operator+=( ) . . . . .	31
4.4.3.7 operator-=( ) . . . . .	32
4.4.3.8 operator/=( ) . . . . .	32
4.4.3.9 Subtract() . . . . .	32
4.4.3.10 VectorsEqual() . . . . .	33
4.4.3.11 Zero() . . . . .	33
4.4.4 Friends And Related Function Documentation . . . . .	33
4.4.4.1 operator* . . . . .	33
4.4.4.2 operator+ . . . . .	34
4.4.4.3 operator- . . . . .	34
4.4.4.4 operator/ . . . . .	35
4.4.4.5 operator<< . . . . .	35
4.4.5 Member Data Documentation . . . . .	35
4.4.5.1 x . . . . .	35
4.4.5.2 y . . . . .	36
4.4.5.3 z . . . . .	36
4.5 Vehicle Class Reference . . . . .	36

4.5.1 Detailed Description . . . . .	37
4.5.2 Constructor & Destructor Documentation . . . . .	37
4.5.2.1 Vehicle() [1/2] . . . . .	37
4.5.2.2 Vehicle() [2/2] . . . . .	38
4.5.3 Member Function Documentation . . . . .	38
4.5.3.1 Print_info() . . . . .	38
4.5.3.2 User_set() . . . . .	38
4.5.4 Member Data Documentation . . . . .	39
4.5.4.1 CalculatedEnergy . . . . .	39
4.5.4.2 displacement . . . . .	39
4.5.4.3 engine . . . . .	39
4.5.4.4 force . . . . .	39
4.5.4.5 fuel . . . . .	39
4.5.4.6 fuel_usage . . . . .	40
4.5.4.7 KineticEnergy . . . . .	40
4.5.4.8 mass . . . . .	40
4.5.4.9 name . . . . .	40
4.5.4.10 position . . . . .	40
4.5.4.11 PotentialEnergy . . . . .	40
4.5.4.12 velocity . . . . .	40
<b>5 File Documentation</b> . . . . .	<b>41</b>
5.1 Engineer_Thesis/Engineer_Thesis/Engineer_Thesis.cpp File Reference . . . . .	41
5.1.1 Function Documentation . . . . .	41
5.1.1.1 main() . . . . .	41
5.2 Engineer_Thesis/Engineer_Thesis/Instructions.md File Reference . . . . .	41
5.3 Engineer_Thesis/Engineer_Thesis/Planet.h File Reference . . . . .	41
5.4 Planet.h . . . . .	42
5.5 Engineer_Thesis/Engineer_Thesis/Solver.cpp File Reference . . . . .	42
5.6 Engineer_Thesis/Engineer_Thesis/Solver.h File Reference . . . . .	42
5.7 Solver.h . . . . .	43
5.8 Engineer_Thesis/Engineer_Thesis/Vector3D.cpp File Reference . . . . .	44
5.8.1 Function Documentation . . . . .	44
5.8.1.1 operator*() . . . . .	44
5.8.1.2 operator+() . . . . .	45
5.8.1.3 operator-() . . . . .	45
5.8.1.4 operator/() . . . . .	46
5.8.1.5 operator<<() . . . . .	46
5.9 Engineer_Thesis/Engineer_Thesis/Vector3D.h File Reference . . . . .	46
5.10 Vector3D.h . . . . .	46
5.11 Engineer_Thesis/Engineer_Thesis/Vehicle.cpp File Reference . . . . .	47
5.12 Engineer_Thesis/Engineer_Thesis/Vehicle.h File Reference . . . . .	47

---

5.13 Vehicle.h . . . . .	47
<b>Index</b>	<b>49</b>





# Chapter 1

## General Information

**1.0.0.1 C++ library for numerical calculations related to basic spacecraft motion issues, taking into account influence of gravity and vehicle propulsion.**

### 1.1 Instalation

To use the library you need to:

1. Clone repository using: git clone [https://github.com/Spectyte5/Engineer\\_Thesis](https://github.com/Spectyte5/Engineer_Thesis)
2. Download and make sure your compiler sees the libraries neccessary:
  - json for modern C++: <https://github.com/tristanpenman/valijson>
  - valijson: <https://github.com/nlohmann/json>

### 1.2 Operation

There are two options of using the library:

1. Provide a command line argument of type string, which is a path to your \*.json\* file for simulation. (exemplary \*.json\* file can be found lower in this document) *example*:  
`.\Engineer_Thesis.cpp "./JSON_files/Sim1.json"`
2. Run the code with your compiler of choice and then choose if you want to create a new simulation or load a \*.json\* from this directory:  
`<Library directory>/JSON_files/`

## 1.3 Units

Units used are SI units:

- mass in kilograms [kg]
- position in meters [m]
- velocity in meters per second [m/s]
- force in Newtons [N]
- energy in Joule [J]
- angle in radians [rad]
- angular velocity in radians per second [rad/s]

## 1.4 json file

\*.json\* file has one object with 4 separate parts:

### 1.4.1 control

- **starttime** - *array* of three double type elements, time when engine will be turned on for x, y, z axis, allowing user to set each one independently.
- **endtime** - *array* of three double type elements, time when engine will be turned off for x, y, z axis, allowing user to set each one independently.
- **force** - *array* of three double type elements, magnitude and the direction of engine thrust for x, y, z axis, allowing user to set each one independently.

### 1.4.2 data

- **ode** - integer type *number* meaning which ODE solving method should be used: 0. Adams-Bashford

1. Euler
2. Midpoint
3. Runge-Kutta IV

- **step** - double type *number* equal to timesteps used for simulation.
- **time** - double type *number*, the final time of simulation.

### 1.4.3 planets

- **name** - *string* type, name of the planet
- **mass** - double type *number*, mass of the planet
- **radius** - double type *number*, radius of the planet
- **orbit** - *boolean* type, is orbiting or strationary [true/false] For orbiting Planets (orbit = true):
- **start\_angle** - double type *number*, phase used for orbital motion (start angle)
- **orbit\_radius** - double type *number*, radius of the orbit
- **ang\_velocity** - double type *number*, constant angular velocity of the planet
- **orbit\_pos** - *array* of three double type elemetents, position of the center of the orbit (x,y,z) **NOTE:** only x,z are taken into account y value is always 0 For stationary Planets (orbit = false):
- **position** - *array* of three double type elemetents, magnitude and the direction of position vector (x,y,z)

### 1.4.4 ship

- **fuel** - double type *number*, fuel mass of the ship
- **fuel\_usage** - double type *number*, constant ammount of fuel used when engines are turned on
- **mass** - double type *number*, total mass of the ship
- **name** - *string* type, name of the ship
- **position** - *array* of three double type elemetents, magnitude and the direction of position vector (x,y,z)
- **velocity** - *array* of three double type elemetents, magnitude and the direction of velocity vector (x,y,z)

## 1.5 Example of Json file (Ship on the Earth orbit with a constant velocity):

```
{
  "control": [
    {
      "endtime": [
        0.0,
        0.0,
        0.0
      ],
      "force": [
        0.0,
        0.0,
        0.0
      ],
      "starttime": [
        0.0,
        0.0,
        0.0
      ]
    }
  ],
  "data": {
    "ode": 3,
    "step": 1,
    "time": 42500
  },
  "planets": [
    {
      "name": "Earth",
      "mass": 5.972e24,
      "radius": 6.378e6,
      "orbit": false,

```

```
        "position": [
            0.0,
            0.0,
            0.0
        ]
    },
    ],
    "ship": {
        "fuel": 0.0,
        "fuel_usage": 0.0,
        "mass": 3000.0,
        "name": "Sim1",
        "position": [
            12742000.0,
            0.0,
            0.0
        ],
        "velocity": [
            0.0,
            5592.27,
            0.0
        ]
    }
}
```

## Chapter 2

# Class Index

### 2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">Control</a>	Class for engine intervals used in simulation . . . . .	<a href="#">9</a>
<a href="#">Planet</a>	Class for different Planets . . . . .	<a href="#">11</a>
<a href="#">Solver</a>	Main class, used for solving . . . . .	<a href="#">14</a>
<a href="#">Vector3D</a>	Class for Three-Dimensional Vectors . . . . .	<a href="#">28</a>
<a href="#">Vehicle</a>	Class for different spaceship objects . . . . .	<a href="#">36</a>



## Chapter 3

# File Index

### 3.1 File List

Here is a list of all files with brief descriptions:

Engineer_Thesis/Engineer_Thesis/ <a href="#">Engineer_Thesis.cpp</a> . . . . .	41
Engineer_Thesis/Engineer_Thesis/ <a href="#">Planet.h</a> . . . . .	41
Engineer_Thesis/Engineer_Thesis/ <a href="#">Solver.cpp</a> . . . . .	42
Engineer_Thesis/Engineer_Thesis/ <a href="#">Solver.h</a> . . . . .	42
Engineer_Thesis/Engineer_Thesis/ <a href="#">Vector3D.cpp</a> . . . . .	44
Engineer_Thesis/Engineer_Thesis/ <a href="#">Vector3D.h</a> . . . . .	46
Engineer_Thesis/Engineer_Thesis/ <a href="#">Vehicle.cpp</a> . . . . .	47
Engineer_Thesis/Engineer_Thesis/ <a href="#">Vehicle.h</a> . . . . .	47





## Chapter 4

# Class Documentation

### 4.1 Control Class Reference

Class for engine intervals used in simulation.

```
#include <Engineer_Thesis/Engineer_Thesis/Solver.h>
```

#### Public Member Functions

- [Control](#) ()
- void [Print\\_Interval](#) ()
- bool [Check\\_input](#) ([Solver](#) &method)

*Function for checking if intervals given by user are possible to be implemented.*

#### Public Attributes

- [Vector3D](#) [timestart](#) = { 0,0,0 }
- [Vector3D](#) [timeend](#) = { 0,0,0 }
- [Vector3D](#) [engforce](#) = { 0,0,0 }

#### 4.1.1 Detailed Description

Class for engine intervals used in simulation.

This class is used to create engine intervals, print and check input given by user for them.

##### Parameters

<i>timestart</i>	is <a href="#">Vector3D</a> object at what time the engine will be turned on
<i>timeend</i>	is <a href="#">Vector3D</a> object at what time the engine will be turned off
<i>engforce</i>	is <a href="#">Vector3D</a> object the direction and magnitude of thrust force vector

See also

[Vector3D](#) for information about three-dimensional vectors class

## 4.1.2 Constructor & Destructor Documentation

### 4.1.2.1 Control()

```
Control::Control ( ) [inline]
```

## 4.1.3 Member Function Documentation

### 4.1.3.1 Check\_input()

```
bool Control::Check_input (
    Solver & method ) [inline]
```

Function for checking if intervals given by user are possible to be implemented.

Iterates through all intervals given by the user, checks if the engine start, end and thrust values are correct and don't intersect each other.

Returns

true if all conditions are satisfied

### 4.1.3.2 Print\_Interval()

```
void Control::Print_Interval ( ) [inline]
```

## 4.1.4 Member Data Documentation

### 4.1.4.1 engforce

```
Vector3D Control::engforce = { 0,0,0 }
```

#### 4.1.4.2 timeend

```
Vector3D Control::timeend = { 0,0,0 }
```

#### 4.1.4.3 timestart

```
Vector3D Control::timestart = { 0,0,0 }
```

## 4.2 Planet Class Reference

Class for different Planets.

```
#include <Engineer_Thesis/Engineer_Thesis/Planet.h>
```

### Public Member Functions

- [Planet](#) ()  
*default constructor*
- void [Print\\_info](#) ()  
*Function printing information about planet.*
- void [Move\\_Planet](#) (double time)  
*Move [Planet](#) around orbit.*

### Public Attributes

- std::vector< [Vector3D](#) > [orb\\_data](#)  
*vector used for storing position data of orbiting planets*
- double [mass](#) = 0  
*mass of the planet [kg]*
- double [radius](#) = 0  
*radius of the planet [m]*
- double [orb\\_radius](#) = 0  
*radius of the orbit [m]*
- double [ang\\_velocity](#) = 0  
*angular velocity [rad/s]*
- double [start\\_ang](#) = 0  
*phase [rad]*
- [Vector3D](#) [position](#) = { 0,0,0 }  
*position of the ship*
- [Vector3D](#) [orb\\_pos](#) = { 0,0,0 }  
*position of the orbit in space*
- std::string [name](#) = ""  
*name of the planet*
- bool [isOrb](#) = false  
*Variable connected to the orbiting of planet if true it means that the [Planet](#) orbits around a given point.*

### 4.2.1 Detailed Description

Class for different Planets.

This class handles creating, moving and printing information about a planet

### 4.2.2 Constructor & Destructor Documentation

#### 4.2.2.1 Planet()

```
Planet::Planet ( ) [inline]
```

default constructor

### 4.2.3 Member Function Documentation

#### 4.2.3.1 Move\_Planet()

```
void Planet::Move_Planet (
    double time ) [inline]
```

Move [Planet](#) around orbit.

Function moving planet around orbit with given angular velocity starting from given angle

Parameters

<i>time</i>	is current time of simulation
-------------	-------------------------------

#### 4.2.3.2 Print\_info()

```
void Planet::Print_info ( ) [inline]
```

Function printing information about planet.

### 4.2.4 Member Data Documentation

#### 4.2.4.1 ang\_velocity

```
double Planet::ang_velocity = 0
```

angular velocity [rad/s]

#### 4.2.4.2 isOrb

```
bool Planet::isOrb = false
```

Variable connected to the orbiting of planet if true it means that the [Planet](#) orbits around a given point.

#### 4.2.4.3 mass

```
double Planet::mass = 0
```

mass of the planet [kg]

#### 4.2.4.4 name

```
std::string Planet::name = ""
```

name of the planet

#### 4.2.4.5 orb\_data

```
std::vector<Vector3D> Planet::orb_data
```

vector used for storing position data of orbiting planets

#### 4.2.4.6 orb\_pos

```
Vector3D Planet::orb_pos = { 0,0,0 }
```

position of the orbit in space

#### Note

orbit is actually only x,z so the y component will always be 0

#### 4.2.4.7 orb\_radius

```
double Planet::orb_radius = 0
```

radius of the orbit [m]

#### 4.2.4.8 position

```
Vector3D Planet::position = { 0,0,0 }
```

position of the ship

#### 4.2.4.9 radius

```
double Planet::radius = 0
```

radius of the planet [m]

#### 4.2.4.10 start\_ang

```
double Planet::start_ang = 0
```

phase [rad]

## 4.3 Solver Class Reference

Main class, used for solving.

```
#include <Engineer_Thesis/Engineer_Thesis/Solver.h>
```

### Public Types

- enum `ode` { `adams` , `euler` , `midpoint` , `runge` }

*Enum of different solving ODEs methods.*

## Public Member Functions

- double [dvdt](#) (double t, double v, double f, double m)  
*Derivative of Velocity.*
- double [dxdt](#) (double t, double v, double x)  
*Derivative of Position.*
- void [Populate](#) ()  
*Define planets in simulation.*
- void [Setup](#) ()  
*Create all simulation elements.*
- bool [Validate\\_Json](#) (std::string &filename)  
*Json Validation function.*
- void [Save\\_json](#) ()  
*Save simulation as a Json file.*
- std::ifstream [Load\\_file](#) (std::string sys\_path, std::string filepath, std::string extension)  
*Function for Loading file from a directory.*
- void [Load\\_data](#) (std::string &filename)  
*Function Setting parameters from the file.*
- bool [Check\\_Collision](#) ([Planet](#) &[Planet](#))  
*Collision checking function.*
- bool [UseEngine](#) ()  
*Applying the thrust force from the engines.*
- void [Calculate\\_Grav](#) ()  
*Function for calculating gravity.*
- void [Calculate\\_Net](#) ()  
*Function for Calculating Net force.*
- void [Euler](#) (double &[time](#), double &velocity, double &position, double &dt, double &force, double &mass)  
*Euler method.*
- void [Runge\\_Kutta](#) (double &[time](#), double &velocity, double &position, double &dt, double &force, double &mass)  
*Runge-Kutta IV method.*
- void [Midpoint](#) (double &[time](#), double &velocity, double &position, double &dt, double &force, double &mass)  
*Midpoint method.*
- void [Adams\\_Bashford](#) (double &[time](#), double &velocity, double &position, double &dt, double &force, double &mass)  
*Adams-Bashforth's method.*
- void [Solve](#) ()  
*Main solving function.*
- void [Push\\_Back](#) ()  
*Put all parameters in vectors.*
- void [Move\\_Orbit](#) ()  
*Method for changing position of orbiting planets.*
- void [Save\\_Planets](#) ()  
*Function for saving planets.*
- void [Save\\_data](#) ()  
*Save simulation data.*
- bool [is\\_empty](#) (std::ifstream &pFile)  
*Funtion checking if a given file is empty.*
- void [Print\\_Pauses](#) ()  
*Pauses between simulation elements printing.*

## Public Attributes

- const double `G` = 6.67259e-11  
*Gravitational constant.*
- std::vector< double > `time_data`  
*vectors for storing current time value*
- std::vector< double > `mass_data`  
*vectors for storing current mass*
- std::vector< double > `fuel_data`  
*vectors for storing current fuel value*
- std::vector< double > `kinetic_data`  
*vectors for storing current kinetic energy value*
- std::vector< double > `potential_data`  
*vectors for storing current potential energy value*
- std::vector< Vector3D > `position_data`  
*vector used for storing current position data*
- std::vector< Vector3D > `velocity_data`  
*vector used for storing current velocity data*
- std::vector< Vector3D > `engine_data`  
*vector used for storing current engine data*
- std::vector< Vector3D > `force_data`  
*vector used for storing current force data*
- std::vector< Planet > `Planets`  
*vector storing planets in the simulation*
- std::vector< Control > `TimeVect`  
*vector storing force intervals of type Control*
- `Vehicle Ship` = `Vehicle`("", 0, 0, 0, 0, 0, 0, 0, 0, 0)  
*Ship member used in simulation.*
- `Vector3D grav_forces`  
*Gravitational force and distance from Planet at the time T.*
- `Vector3D distance`
- bool `engine_used` = false  
*Boolean used for removing fuel used.*
- int `method` = 0  
*Method stored as an int used for enum.*
- double `T` = 0  
*time of simulation*
- double `step` = 0  
*time step between increments*
- double `time` = 0  
*current simulation time*
- double `fuel_used` = 0  
*ammount of fuel\_used at the iteration*

### 4.3.1 Detailed Description

Main class, used for solving.

Class taking care of loading, validating, saving files and calculating results using different solvers



## 4.3.2 Member Enumeration Documentation

### 4.3.2.1 ode

```
enum Solver::ode
```

Enum of different solving ODEs methods.

Enumerator

adams	
euler	
midpoint	
runge	

## 4.3.3 Member Function Documentation

### 4.3.3.1 Adams\_Bashford()

```
void Solver::Adams_Bashford (
    double & time,
    double & velocity,
    double & position,
    double & dt,
    double & force,
    double & mass )
```

Adams-Bashforth's method.

Function solving and ODE using the Adams-Bashforth's predictor and corrector method for calculating and setting parameters of the ship

Parameters

<i>time</i>	is current time of simulation
<i>velocity</i>	is velocity at current time
<i>position</i>	is position at current time
<i>dt</i>	is step between iterations
<i>force</i>	is force acting on the spaceship @mass is mass of the spaceship

#### 4.3.3.2 Calculate\_Grav()

```
void Solver::Calculate_Grav ( )
```

Function for calculating gravity.

Iterate through planets and calculate Gravitation forces acting on the ship and it's potential energy

#### 4.3.3.3 Calculate\_Net()

```
void Solver::Calculate_Net ( )
```

Function for Calculating Net force.

Sets the value of net force taking in to account gravitational and thrust forces

See also

[Calculate\\_Grav\(\)](#) and [UseEngine\(\)](#) for more information about forces calculation

#### 4.3.3.4 Check\_Collision()

```
bool Solver::Check_Collision (
    Planet & Planet )
```

Collistion checking function.

Function calculating distance between the Ship and [Planet](#).

Parameters

<a href="#">Planet</a>	is a <a href="#">Planet</a> Class object which we are checking ships collistion with
------------------------	--

Returns

true if we have a collistion and false if there not

Attention

If Ship is exactly on the [Planet](#)'s surface it does not count as a collistion

#### 4.3.3.5 dvdt()

```
double Solver::dvdt (
    double t,
```

```
double v,  
double f,  
double m ) [inline]
```

Derivative of Velocity.

Function returning the derivative of velocity.

#### Parameters

$t$	is current time of simulation
$v$	is velocity at time $t$
$f$	is force at time $t$
$m$	is mass of the object

#### Returns

double discribing velocity change in the last interval

#### 4.3.3.6 dxdt()

```
double Solver::dxdt (  
    double t,  
    double v,  
    double x ) [inline]
```

Derivative of Position.

Function returning the derivative of position.

#### Parameters

$t$	is current time of simulation
$v$	is velocity at time $t$
$x$	is position at time $t$

#### Returns

double discribing position change in the last interval

#### 4.3.3.7 Euler()

```
void Solver::Euler (  
    double & time,  
    double & velocity,
```

```
double & position,
double & dt,
double & force,
double & mass )
```

Euler method.

Function solving and ODE using the Euler's method and setting parameters of the ship

#### Parameters

<i>time</i>	is current time of simulation
<i>velocity</i>	is velocity at current time
<i>position</i>	is position at current time
<i>dt</i>	is step between iterations
<i>force</i>	is force acting on the spaceship @mass is mass of the spaceship

#### 4.3.3.8 is\_empty()

```
bool Solver::is_empty (
    std::ifstream & pFile ) [inline]
```

Funtion checking if a given file is empty.

#### Parameters

<i>pFile</i>	is path and name of the file
--------------	------------------------------

#### Returns

true if empty and false if not

#### 4.3.3.9 Load\_data()

```
void Solver::Load_data (
    std::string & filename )
```

Function Setting parameters from the file.

Loaded file is used to set paramaters

#### See also

[Load\\_file\(\)](#) for information about loading file

**4.3.3.10 Load\_file()**

```
std::ifstream Solver::Load_file (
    std::string sys_path,
    std::string filepath,
    std::string extension )
```

Function for Loading file from a directory.

Display files in directory and open file with a given filepath

**Parameters**

<i>sys_path</i>	is directory in which we are looking for files
<i>filepath</i>	is path to the file
<i>extension</i>	is extension of the file ex. ".txt"

**Returns**

loaded file as ifstream

**4.3.3.11 Midpoint()**

```
void Solver::Midpoint (
    double & time,
    double & velocity,
    double & position,
    double & dt,
    double & force,
    double & mass )
```

Midpoint method.

Function solving and ODE using the modified Euler's method (Midpoint method) and setting parameters of the ship

**Parameters**

<i>time</i>	is current time of simulation
<i>velocity</i>	is velocity at current time
<i>position</i>	is position at current time
<i>dt</i>	is step between iterations
<i>force</i>	is force acting on the spaceship @mass is mass of the spaceship

**4.3.3.12 Move\_Orbit()**

```
void Solver::Move_Orbit ( )
```

Method for changing position of orbiting planets.

Checks if planets is orbiting around a point and if yes changes its position and saves it to vector

#### 4.3.3.13 Populate()

```
void Solver::Populate ( )
```

Define planets in simulation.

Gets ammount of planets in simulation, sets parameters for planet and puts it in the planets vector

#### 4.3.3.14 Print\_Pauses()

```
void Solver::Print_Pauses ( ) [inline]
```

Pauses between simulation elements printing.

Function printing '=' signs to allow better seperation between simulation elements and improve comfort of reading the text displayed.

#### 4.3.3.15 Push\_Back()

```
void Solver::Push_Back ( )
```

Put all parameters in vectors.

Save all neccessary values at time t into corresponding vectors

#### 4.3.3.16 Runge\_Kutta()

```
void Solver::Runge_Kutta (
    double & time,
    double & velocity,
    double & position,
    double & dt,
    double & force,
    double & mass )
```

Runge-Kutta IV method.

Function solving and ODE using the Runge-Kutta IV-order method and setting parameters of the ship

##### Parameters

<i>time</i>	is current time of simulation
<i>velocity</i>	is velocity at current time
<i>position</i>	is position at current time
<i>dt</i>	is step between iterations
<i>force</i>	is force acting on the spaceship @mass is mass of the spaceship

#### 4.3.3.17 Save\_data()

```
void Solver::Save_data ( )
```

Save simulation data.

Saves all parameters and calls the function for saving planets' data.

See also

[Save\\_Planets\(\)](#) for more information about saving planets

#### 4.3.3.18 Save\_json()

```
void Solver::Save_json ( )
```

Save simulation as a Json file.

Function used in create a simulation mode to save all parameters of the ship and planets in a json file which then can be reloaded in load mode.

#### 4.3.3.19 Save\_Planets()

```
void Solver::Save_Planets ( )
```

Function for saving planets.

Save all planets' paramaters to a seprate file

#### 4.3.3.20 Setup()

```
void Solver::Setup ( )
```

Create all simulation elements.

Setup Particle and planets in simulation, fill all engine intervals

#### 4.3.3.21 Solve()

```
void Solver::Solve ( )
```

Main solving function.

This function loops through time interval calling all functions used for calculation and prints result on screen.

See also

[Adams\\_Bashford\(\)](#), [Midpoint\(\)](#), [Euler\(\)](#), [Runge\\_Kutta\(\)](#) for more information about solving ODE's

#### 4.3.3.22 UseEngine()

```
bool Solver::UseEngine ( )
```

Applying the thrust force from the engines.

Checks if we are in any of intervals defined by user and if yes and fuel is available it applies engine force

##### Returns

true if engine was used and no if not

#### 4.3.3.23 Validate\_Json()

```
bool Solver::Validate_Json (
    std::string & filename )
```

Json Validation function.

Check if vector file validates against the schema

##### Parameters

<i>filename</i>	is a filepath for the json file that will be validated
-----------------	--

### 4.3.4 Member Data Documentation

#### 4.3.4.1 distance

```
Vector3D Solver::distance
```

#### 4.3.4.2 engine\_data

```
std::vector<Vector3D> Solver::engine_data
```

vector used for storing current engine data



#### 4.3.4.3 engine\_used

```
bool Solver::engine_used = false
```

Boolean used for removing fuel used.

#### 4.3.4.4 force\_data

```
std::vector<Vector3D> Solver::force_data
```

vector used for storing current force data

#### 4.3.4.5 fuel\_data

```
std::vector<double> Solver::fuel_data
```

vectors for storing current fuel value

#### 4.3.4.6 fuel\_used

```
double Solver::fuel_used = 0
```

ammount of fuel\_used at the iteration

#### 4.3.4.7 G

```
const double Solver::G = 6.67259e-11
```

Gravitational constant.

#### 4.3.4.8 grav\_forces

```
Vector3D Solver::grav_forces
```

Gravitational force and distance from [Planet](#) at the time T.

#### 4.3.4.9 kinetic\_data

```
std::vector<double> Solver::kinetic_data
```

vectors for storing current kinetic energy value

#### 4.3.4.10 mass\_data

```
std::vector<double> Solver::mass_data
```

vectors for storing current mass

#### 4.3.4.11 method

```
int Solver::method =0
```

Method stored as an int used for enum.

#### 4.3.4.12 Planets

```
std::vector<Planet> Solver::Planets
```

vector storing planets in the simulation

#### 4.3.4.13 position\_data

```
std::vector<Vector3D> Solver::position_data
```

vector used for storing current position data

#### 4.3.4.14 potential\_data

```
std::vector<double> Solver::potential_data
```

vectors for storing current potential energy value

#### 4.3.4.15 Ship

```
Vehicle Solver::Ship = Vehicle("", 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Ship member used in simulation.

#### 4.3.4.16 step

```
double Solver::step = 0
```

time step between increments

#### 4.3.4.17 T

```
double Solver::T = 0
```

time of simulation

#### 4.3.4.18 time

```
double Solver::time = 0
```

current simulation time

#### 4.3.4.19 time\_data

```
std::vector<double> Solver::time_data
```

vectors for storing current time value

#### 4.3.4.20 TimeVect

```
std::vector<Control> Solver::TimeVect
```

vector storing force intervals of type [Control](#)

#### 4.3.4.21 velocity\_data

```
std::vector<Vector3D> Solver::velocity_data
```

vector used for storing current velocity data

## 4.4 Vector3D Class Reference

Class for Three-Dimensional Vectors.

```
#include <Engineer_Thesis/Engineer_Thesis/Vector3D.h>
```

### Public Member Functions

- [Vector3D](#) ()  
*default constructor*
- [Vector3D](#) (double [x](#), double [y](#), double [z](#))  
*constructor with x, y and z values*
- [Vector3D](#) & [Add](#) (const [Vector3D](#) &vect)  
*Add two vectors.*
- [Vector3D](#) & [Subtract](#) (const [Vector3D](#) &vect)  
*Subtract two vectors.*
- [Vector3D](#) & [Multiply](#) (const [Vector3D](#) &vect)  
*Multiply two vectors.*
- [Vector3D](#) & [Divide](#) (const [Vector3D](#) &vect)  
*Divide two vectors.*
- [Vector3D](#) & [operator+=](#) (const [Vector3D](#) &vect)  
*Add two vectors with += operator.*
- [Vector3D](#) & [operator-=](#) (const [Vector3D](#) &vect)  
*Subtract two vectors with -= operator.*
- [Vector3D](#) & [operator\\*=](#) (const [Vector3D](#) &vect)  
*Multiply two vectors with \*= operator.*
- [Vector3D](#) & [operator/=](#) (const [Vector3D](#) &vect)  
*Divide two vectors with /= operator.*
- [Vector3D](#) & [operator\\*](#) (const int &i)  
*Multiply vector by scale.*
- [Vector3D](#) & [Zero](#) ()  
*Sets values of the x,y,z to 0.*
- bool [VectorsEqual](#) (const [Vector3D](#) &vect)  
*Function checking if two vectors have the same x,y,z components.*

### Public Attributes

- double [x](#)  
*x component of the vector*
- double [y](#)  
*y component of the vector*
- double [z](#)  
*z component of the vector*

## Friends

- `Vector3D & operator+ (Vector3D &v1, const Vector3D &v2)`  
*Add two vectors with + operator.*
- `Vector3D & operator- (Vector3D &v1, const Vector3D &v2)`  
*Subtract two vectors with - operator.*
- `Vector3D & operator* (Vector3D &v1, const Vector3D &v2)`  
*Multiply two vectors with \* operator.*
- `Vector3D & operator/ (Vector3D &v1, const Vector3D &v2)`  
*Divide two vectors with / operator.*
- `std::ostream & operator<< (std::ostream &output, const Vector3D &vect)`  
*overload of << operator for printing vectors*

### 4.4.1 Detailed Description

Class for Three-Dimensional Vectors.

This class is used for operations and storing parameters of the three-dimensional vectors

### 4.4.2 Constructor & Destructor Documentation

#### 4.4.2.1 Vector3D() [1/2]

```
Vector3D::Vector3D ( )
```

default constructor

#### 4.4.2.2 Vector3D() [2/2]

```
Vector3D::Vector3D (
    double x,
    double y,
    double z )
```

constructor with x, y and z values

### 4.4.3 Member Function Documentation

#### 4.4.3.1 Add()

```
Vector3D & Vector3D::Add (
    const Vector3D & vect )
```

Add two vectors.

**4.4.3.1.1 Example** `v1.Add(v2) // which equals to v1 + v2`

#### Parameters

<i>vect</i>	is vector being added to the vector calling this method
-------------	---

#### Returns

[Vector3D](#) that is a vector on which method was called with *vect* value added to it

### 4.4.3.2 Divide()

```
Vector3D & Vector3D::Divide (  
    const Vector3D & vect )
```

Divide two vectors.

**4.4.3.2.1 Example** `v1.Divide(v2) // which equals to v1 / v2`

#### Parameters

<i>vect</i>	is vector which the vector calling this method is divided by
-------------	--

#### Returns

[Vector3D](#) that is a vector on which method was called divided by *vect* value

### 4.4.3.3 Multiply()

```
Vector3D & Vector3D::Multiply (  
    const Vector3D & vect )
```

Multiply two vectors.

**4.4.3.3.1 Example** `v1.Multiply(v2) // which equals to v1 * v2`

#### Parameters

<i>vect</i>	is vector which the vector calling this method is multiplyied by
-------------	--

#### Returns

[Vector3D](#) that is a vector on which method was called multiplyied by *vect* value

#### 4.4.3.4 operator\*()

```
Vector3D & Vector3D::operator* (
    const int & i )
```

Multiply vector by scale.

##### Parameters

<i>i</i>	is integer value by which we want to multiply our vector
----------	--

#### 4.4.3.5 operator\*=( )

```
Vector3D & Vector3D::operator*= (
    const Vector3D & vect )
```

Multiply two vectors with \*= operator.

##### 4.4.3.5.1 Example `v1 *= v2`

##### Parameters

<i>v1</i>	is vector multiplied
<i>v2</i>	is vector we are multiplying by

##### Returns

`v1` multiplied by `v2` value

#### 4.4.3.6 operator+=( )

```
Vector3D & Vector3D::operator+= (
    const Vector3D & vect )
```

Add two vectors with += operator.

##### 4.4.3.6.1 Example `v1 += v2`

##### Parameters

<i>v1</i>	is vector which we are adding into
<i>v2</i>	is vector being added

**Returns**

v1 increased by v2 value

**4.4.3.7 operator+=( )**

```
Vector3D & Vector3D::operator+= (
    const Vector3D & vect )
```

Subtract two vectors with -= operator.

**4.4.3.7.1 Example** `v1 -= v2`**Parameters**

<code>v1</code>	is vector which we are subtracting from
<code>v2</code>	is vector being subtracted

**Returns**

v1 decreased by v2 value

**4.4.3.8 operator/=( )**

```
Vector3D & Vector3D::operator/= (
    const Vector3D & vect )
```

Divide two vectors with /= operator.

**4.4.3.8.1 Example** `v1 /= v2`**Parameters**

<code>v1</code>	is vector divided
<code>v2</code>	is vector we are dividing by

**Returns**

v1 divided by v2 value

**4.4.3.9 Subtract()**

```
Vector3D & Vector3D::Subtract (
    const Vector3D & vect )
```

Subtract two vectors.



**4.4.3.9.1 Example** `v1.Subtract(v2) // which equals to v1 - v2`**Parameters**

<code>vect</code>	is vector being subtracted from the vector calling this method
-------------------	--

**Returns**

[Vector3D](#) that is a vector on which method was called with vect value subtracted from it

**4.4.3.10 VectorsEqual()**

```
bool Vector3D::VectorsEqual (
    const Vector3D & vect )
```

Function checking if two vectors have the same x,y,z components.

**Parameters**

<code>vect</code>	is vector compared to the vector calling this method
-------------------	--

**Returns**

true if two vectors are the same, false if not

**4.4.3.11 Zero()**

```
Vector3D & Vector3D::Zero ( )
```

Sets values of the x,y,z to 0.

**4.4.4 Friends And Related Function Documentation****4.4.4.1 operator\***

```
Vector3D & operator* (
    Vector3D & v1,
    const Vector3D & v2 ) [friend]
```

Multiply two vectors with \* operator.

**4.4.4.1.1 Example** `v1 * v2`

**Parameters**

<i>v1</i>	is vector multiplyied
<i>v2</i>	is vector we are multiplying by

**Returns**

*v1* multiplied by *v2* value

**4.4.4.2 operator+**

```
Vector3D & operator+ (  
    Vector3D & v1,  
    const Vector3D & v2 ) [friend]
```

Add two vectors with + operator.

**4.4.4.2.1 Example**  $v1 + v2$ **Parameters**

<i>v1</i>	is vector which we are adding into
<i>v2</i>	is vector being added

**Returns**

*v1* increased by *v2* value

**4.4.4.3 operator-**

```
Vector3D & operator- (  
    Vector3D & v1,  
    const Vector3D & v2 ) [friend]
```

Subtract two vectors with - operator.

**4.4.4.3.1 Example**  $v1 - v2$ **Parameters**

<i>v1</i>	is vector which we are subtracting from
<i>v2</i>	is vector being subtracted

**Returns**

v1 decreased by v2 value

**4.4.4.4 operator/**

```
Vector3D & operator/ (
    Vector3D & v1,
    const Vector3D & v2 ) [friend]
```

Divide two vectors with / operator.

**4.4.4.4.1 Example** `v1 / v2`**Parameters**

<i>v1</i>	is vector divided
<i>v2</i>	is vector we are dividing by

**Returns**

v1 divided by v2 value

**4.4.4.5 operator<<**

```
std::ostream & operator<< (
    std::ostream & output,
    const Vector3D & vect ) [friend]
```

overload of << operator for printing vectors

**Parameters**

<i>output</i>	is ofstream where we will print data
<i>vect</i>	is a vector being printed

**4.4.5 Member Data Documentation****4.4.5.1 x**

```
double Vector3D::x
```

x component of the vector

#### 4.4.5.2 y

```
double Vector3D::y
```

y component of the vector

#### 4.4.5.3 z

```
double Vector3D::z
```

z component of the vector

## 4.5 Vehicle Class Reference

Class for different spaceship objects.

```
#include <Engineer_Thesis/Engineer_Thesis/Vehicle.h>
```

### Public Member Functions

- [Vehicle](#) ()  
*Default Constructor.*
- [Vehicle](#) (std::string n, double rx, double ry, double rz, double vx, double vy, double vz, double m, double [fuel](#), double [fuel\\_usage](#))  
*Constructor assiging given paramaters.*
- void [Print\\_info](#) ()  
*Print information about the Ship.*
- void [User\\_set](#) ()  
*User set ships parameters.*

## Public Attributes

- `std::string name`  
*name of the ship*
- `Vector3D position`  
*Vector3D position on x,y,z axis [m].*
- `Vector3D velocity`  
*Vector3D velocity on x,y,z axis [m/s].*
- `Vector3D engine = { 0,0,0 }`  
*engine is a Vector3D thrust force on x,y,z axis [N]*
- `Vector3D force = { 0,0,0 }`  
*Vector3D net force acting on spaceship on x,y,z axis [N].*
- `Vector3D displacement = { 0,0,0 }`  
*displacement of the ship from initial position[m]*
- `double mass = 0`  
*total mass of the ship with fuel [kg]*
- `double fuel = 0`  
*mass of fuel carried by the ship[kg]*
- `double fuel_usage = 0`  
*ammount of fuel used by engines [kg/s]*
- `double PotentialEnergy = 0`  
*total potential energy from all planets acting on the spaceship [J]*
- `double KineticEnergy = 0`  
*energy from velocity whith which spaceship is moving [J]*
- `bool CalculatedEnergy = 0`  
*true or false depending on whether the planets where already initialized*

### 4.5.1 Detailed Description

Class for different spaceship objects.

Ship is a body having no size, no rotation (Point-mass)

#### Note

In few places the mass actually is a mass without fuel, inputed by user that then has the fuel mass added to it.

#### See also

[Vector3D](#) for more information about three-dimentional vector objects

### 4.5.2 Constructor & Destructor Documentation

#### 4.5.2.1 Vehicle() [1/2]

```
Vehicle::Vehicle ( ) [inline]
```

Default Constructor.

#### 4.5.2.2 Vehicle() [2/2]

```
Vehicle::Vehicle (
    std::string n,
    double rx,
    double ry,
    double rz,
    double vx,
    double vy,
    double vz,
    double m,
    double fuel,
    double fuel_usage )
```

Constructor assigning given paramaters.

##### Parameters

<i>n</i>	is name of the ship
<i>rx</i>	is position on x axis [m]
<i>ry</i>	is position on y axis [m]
<i>rz</i>	is position on z axis [m]
<i>vx</i>	is velocity on x axis [m/s]
<i>vy</i>	is velocity on y axis [m/s]
<i>vz</i>	is velocity on z axis [m/s]
<i>m</i>	is mass of the ship [kg]
<i>fuel</i>	is mass of fuel carried by the ship [kg]
<i>fuel_usage</i>	is ammount of fuel used in [kg/s]

### 4.5.3 Member Function Documentation

#### 4.5.3.1 Print\_info()

```
void Vehicle::Print_info ( )
```

Print information about the Ship.

Function for printing each paramter of the ship on screen

#### 4.5.3.2 User\_set()

```
void Vehicle::User_set ( )
```

User set ships parameters.

Function allowing user to set values of the Ship, used in create a simulation mode.

## 4.5.4 Member Data Documentation

### 4.5.4.1 CalculatedEnergy

```
bool Vehicle::CalculatedEnergy = 0
```

true or false depending on whether the planets where already initialized

See also

[Planet](#) more info about planets

### 4.5.4.2 displacement

```
Vector3D Vehicle::displacement = { 0,0,0 }
```

displacement of the ship from initial position[m]

### 4.5.4.3 engine

```
Vector3D Vehicle::engine = { 0,0,0 }
```

engine is a [Vector3D](#) thrust force on x,y,z axis [N]

### 4.5.4.4 force

```
Vector3D Vehicle::force = { 0,0,0 }
```

[Vector3D](#) net force acting on spaceship on x,y,z axis [N].

### 4.5.4.5 fuel

```
double Vehicle::fuel = 0
```

mass of fuel carried by the ship[kg]

#### 4.5.4.6 fuel\_usage

```
double Vehicle::fuel_usage = 0
```

ammount of fuel used by engines [kg/s]

#### 4.5.4.7 KineticEnergy

```
double Vehicle::KineticEnergy = 0
```

energy from velocity whith which spaceship is moving [J]

#### 4.5.4.8 mass

```
double Vehicle::mass = 0
```

total mass of the ship with fuel [kg]

#### 4.5.4.9 name

```
std::string Vehicle::name
```

name of the ship

#### 4.5.4.10 position

```
Vector3D Vehicle::position
```

Vector3D position on x,y,z axis [m].

#### 4.5.4.11 PotentialEnergy

```
double Vehicle::PotentialEnergy = 0
```

total potential energy from all planets acting on the spaceship [J]

#### 4.5.4.12 velocity

```
Vector3D Vehicle::velocity
```

Vector3D velocity on x,y,z axis [m/s].



## Chapter 5

# File Documentation

### 5.1 Engineer\_Thesis/Engineer\_Thesis/Engineer\_Thesis.cpp File Reference

```
#include <iostream>
#include "Solver.h"
```

#### Functions

- int [main](#) (int argc, char \*argv[])

#### 5.1.1 Function Documentation

##### 5.1.1.1 main()

```
int main (
    int argc,
    char * argv[] )
```

### 5.2 Engineer\_Thesis/Engineer\_Thesis/Instructions.md File Reference

### 5.3 Engineer\_Thesis/Engineer\_Thesis/Planet.h File Reference

```
#include "Vector3D.h"
```

## Classes

- class [Planet](#)

*Class for different Planets.*

## 5.4 Planet.h

[Go to the documentation of this file.](#)

```

1 #pragma once
2 #include "Vector3D.h"
3
4
5
6
7 class Planet {
8
9 public:
10
11
12     std::vector <Vector3D> orb_data;
13
14     double mass = 0;
15
16     double radius = 0;
17
18     double orb_radius = 0;
19
20     double ang_velocity = 0;
21
22     double start_ang = 0;
23
24
25     Vector3D position = { 0,0,0 };
26     Vector3D orb_pos = { 0,0,0 };
27
28     std::string name = "";
29
30     bool isOrb = false;
31
32
33
34
35     Planet() {}
36
37
38     void Print_info() {
39
40         std::cout << "\nName: " << name << "\nMass: " << mass << " kg" << "\nRadius: " << radius << " m" <<
41         "\nPosition: " << position << " m";
42
43         if (isOrb) {
44             std::cout << "\nOrbit Radius: " << orb_radius << " m" << "\nOrbit Velocity: " << ang_velocity <<
45             " rad/s" << "\nOrbit Center: " << orb_pos << " m";
46         }
47
48         std::cout << "\n";
49     }
50
51     void Move_Planet(double time) {
52
53         position.x = orb_pos.x + orb_radius * cos(start_ang + ang_velocity * time);
54         position.z = orb_pos.z + orb_radius * sin(start_ang + ang_velocity * time);
55         orb_data.push_back(position);
56     }
57
58 };

```

## 5.5 Engineer\_Thesis/Engineer\_Thesis/Solver.cpp File Reference

```

#include "Solver.h"
#include <filesystem>
#include <json.hpp>
#include <valijson_nlohmann_bundled.hpp>

```

## 5.6 Engineer\_Thesis/Engineer\_Thesis/Solver.h File Reference

```

#include <fstream>
#include <iomanip>
#include "Vehicle.h"
#include "Planet.h"

```

## Classes

- class [Solver](#)  
*Main class, used for solving.*
- class [Control](#)  
*Class for engine intervals used in simulation.*

## 5.7 Solver.h

[Go to the documentation of this file.](#)

```
1 #pragma once
2 #include <fstream>
3 #include <iomanip>
4 #include "Vehicle.h"
5 #include "Planet.h"
6
7
8 //forward declare class
9 class Control;
10
11 class Solver {
12 public:
13     const double G = 6.67259e-11;
14     std::vector<double> time_data;
15     std::vector<double> mass_data;
16     std::vector<double> fuel_data;
17     std::vector<double> kinetic_data;
18     std::vector<double> potential_data;
19
20     std::vector<Vector3D> position_data;
21     std::vector<Vector3D> velocity_data;
22     std::vector<Vector3D> engine_data;
23     std::vector<Vector3D> force_data;
24     std::vector<Planet> Planets;
25     std::vector<Control> TimeVect;
26     Vehicle Ship = Vehicle("", 0, 0, 0, 0, 0, 0, 0, 0, 0);
27     Vector3D grav_forces, distance;
28     bool engine_used = false;
29     enum ode { adams, euler, midpoint, runge};
30     int method=0;
31
32     double T = 0;
33     double step = 0;
34     double time = 0;
35     double fuel_used = 0;
36
37     double dvdt(double t, double v, double f, double m) { return f / m; }
38     double dxdt(double t, double v, double x) { return v; }
39     void Populate();
40     void Setup();
41     bool Validate_Json(std::string& filename);
42     void Save_json();
43     std::ifstream Load_file(std::string sys_path, std::string filepath, std::string extenstion);
44     void Load_data(std::string& filename);
45     bool Check_Collision(Planet& Planet);
46     bool UseEngine();
47     void Calculate_Grav();
48     void Calculate_Net();
49     void Euler(double& time, double& velocity, double& position, double& dt, double& force, double&
50 mass);
51     void Runge_Kutta(double& time, double& velocity, double& position, double& dt, double& force,
52 double& mass);
53     void Midpoint(double& time, double& velocity, double& position, double& dt, double& force, double&
54 mass);
55     void Adams_Bashford(double& time, double& velocity, double& position, double& dt, double& force,
56 double& mass);
57     void Solve();
58     void Push_Back();
59     void Move_Orbit();
60     void Save_Planets();
61     void Save_data();
62     bool is_empty(std::ifstream& pFile)
63     {
64         return pFile.peek() == std::ifstream::traits_type::eof();
65     }
66     void Print_Pauses() {
67         std::cout << std::setfill('=') << std::setw(120) << "\n";
68     }
69 }
```

```

205     }
206 };
207
215 class Control{
216
217 public:
218     Vector3D timestart= { 0,0,0 }, timeend = { 0,0,0 }, engforce = { 0,0,0 };
219
220     Control() {};;
221
222     void Print_Interval() {
223         std::cout << "\nStart times(x,y,z): " << timestart << " s"
224             << "\nEnd times(x,y,z): " << timeend << " s"
225             << "\nEngine force(x,y,z): " << engforce << " N" << std::endl;
226     }
227
231     bool Check_input(Solver& method) {
232         //initial check:
233         if (method.TimeVect.empty()) {
234             //timestart
235             if (timestart.x < 0 || timestart.y < 0 || timestart.z < 0) return false;
236             //timeend
237             if (timeend.x > method.T || timeend.y > method.T || timeend.z > method.T) return false;
238             if (timeend.x < timestart.x || timeend.y < timestart.y || timeend.z < timestart.z) return
false;
239         }
240         else {
241             if (timestart.x < 0 || timestart.y < 0 || timestart.z < 0) return false;
242             if (timeend.x > method.T || timeend.y > method.T || timeend.z > method.T) return false;
243             if (timeend.x < timestart.x || timeend.y < timestart.y || timeend.z < timestart.z) return
false;
244             //check if interval does not intersect previous interval
245             if (timestart.x < method.TimeVect.back().timeend.x || timestart.y <
method.TimeVect.back().timeend.y || timestart.z < method.TimeVect.back().timeend.z) return false;
246         }
247         return true;
248     }
249 };

```

## 5.8 Engineer\_Thesis/Engineer\_Thesis/Vector3D.cpp File Reference

```
#include "Vector3D.h"
```

### Functions

- [Vector3D & operator+ \(Vector3D &v1, const Vector3D &v2\)](#)
- [Vector3D & operator- \(Vector3D &v1, const Vector3D &v2\)](#)
- [Vector3D & operator\\* \(Vector3D &v1, const Vector3D &v2\)](#)
- [Vector3D & operator/ \(Vector3D &v1, const Vector3D &v2\)](#)
- [std::ostream & operator<< \(std::ostream &output, const Vector3D &vect\)](#)

### 5.8.1 Function Documentation

#### 5.8.1.1 operator\*()

```

Vector3D & operator* (
    Vector3D & v1,
    const Vector3D & v2 )

```

##### 5.8.1.1.1 Example v1 \* v2

**Parameters**

<i>v1</i>	is vector multiplyied
<i>v2</i>	is vector we are multiplying by

**Returns**

*v1* multiplied by *v2* value

**5.8.1.2 operator+()**

```
Vector3D & operator+ (
    Vector3D & v1,
    const Vector3D & v2 )
```

**5.8.1.2.1 Example**  $v1 + v2$ **Parameters**

<i>v1</i>	is vector which we are adding into
<i>v2</i>	is vector being added

**Returns**

*v1* increased by *v2* value

**5.8.1.3 operator-()**

```
Vector3D & operator- (
    Vector3D & v1,
    const Vector3D & v2 )
```

**5.8.1.3.1 Example**  $v1 - v2$ **Parameters**

<i>v1</i>	is vector which we are substracting from
<i>v2</i>	is vector being substracted

**Returns**

*v1* decreased by *v2* value

#### 5.8.1.4 operator/()

```
Vector3D & operator/ (
    Vector3D & v1,
    const Vector3D & v2 )
```

##### 5.8.1.4.1 Example `v1 / v2`

###### Parameters

<code>v1</code>	is vector divided
<code>v2</code>	is vector we are dividing by

###### Returns

`v1` divided by `v2` value

#### 5.8.1.5 operator<<()

```
std::ostream & operator<< (
    std::ostream & output,
    const Vector3D & vect )
```

###### Parameters

<code>output</code>	is ofstream where we will print data
<code>vect</code>	is a vector being printed

## 5.9 Engineer\_Thesis/Engineer\_Thesis/Vector3D.h File Reference

```
#include <iostream>
```

### Classes

- class `Vector3D`  
*Class for Three-Dimensional Vectors.*

## 5.10 Vector3D.h

[Go to the documentation of this file.](#)

```
1 #pragma once
2 #include <iostream>
3
```

```

7  class Vector3D {
8
9  public:
11     double x;
13     double y;
15     double z;
16
18     Vector3D();
20     Vector3D(double x, double y, double z);
21
30     Vector3D& Add(const Vector3D& vect);
39     Vector3D& Subtract(const Vector3D& vect);
48     Vector3D& Multiply(const Vector3D& vect);
57     Vector3D& Divide(const Vector3D& vect);
58
68     friend Vector3D& operator+ (Vector3D& v1, const Vector3D& v2);
78     friend Vector3D& operator- (Vector3D& v1, const Vector3D& v2);
88     friend Vector3D& operator* (Vector3D& v1, const Vector3D& v2);
98     friend Vector3D& operator/ (Vector3D& v1, const Vector3D& v2);
99
109    Vector3D& operator+=(const Vector3D& vect);
119    Vector3D& operator-=(const Vector3D& vect);
129    Vector3D& operator*=(const Vector3D& vect);
139    Vector3D& operator/=(const Vector3D& vect);
140
144    Vector3D& operator*(const int& i);
145
147    Vector3D& Zero();
148
153    friend std::ostream& operator < (std::ostream& output, const Vector3D& vect);
154
159    bool VectorsEqual(const Vector3D& vect);
160 };

```

## 5.11 Engineer\_Thesis/Engineer\_Thesis/Vehicle.cpp File Reference

```
#include "Vehicle.h"
```

## 5.12 Engineer\_Thesis/Engineer\_Thesis/Vehicle.h File Reference

```
#include "Vector3D.h"
#include <vector>
```

### Classes

- class [Vehicle](#)  
*Class for different spaceship objects.*

## 5.13 Vehicle.h

[Go to the documentation of this file.](#)

```

1  #pragma once
2  #include "Vector3D.h"
3  #include <vector>
4
10 class Vehicle {
11
12 public:
14     std::string name;
16     Vector3D position;
18     Vector3D velocity;

```

```
20     Vector3D engine = { 0,0,0 };
22     Vector3D force = { 0,0,0 };
24     Vector3D displacement = { 0,0,0 };
26     double mass = 0;
28     double fuel = 0;
30     double fuel_usage = 0;
32     double PotentialEnergy = 0;
34     double KineticEnergy = 0;
38     bool CalculatedEnergy = 0;
39
41     Vehicle() {};
54
55     Vehicle(std::string n, double rx, double ry, double rz, double vx, double vy, double vz, double m,
double fuel, double fuel_usage);
59     void Print_info();
63     void User_set();
64 };
```



# Index

- adams
  - Solver, [17](#)
- Adams\_Bashford
  - Solver, [17](#)
- Add
  - Vector3D, [29](#)
- ang\_velocity
  - Planet, [12](#)
- Calculate\_Grav
  - Solver, [17](#)
- Calculate\_Net
  - Solver, [18](#)
- CalculatedEnergy
  - Vehicle, [39](#)
- Check\_Collision
  - Solver, [18](#)
- Check\_input
  - Control, [10](#)
- Control, [9](#)
  - Check\_input, [10](#)
  - Control, [10](#)
  - engforce, [10](#)
  - Print\_Interval, [10](#)
  - timeend, [10](#)
  - timestart, [11](#)
- displacement
  - Vehicle, [39](#)
- distance
  - Solver, [24](#)
- Divide
  - Vector3D, [30](#)
- dvdT
  - Solver, [18](#)
- dxdt
  - Solver, [19](#)
- engforce
  - Control, [10](#)
- engine
  - Vehicle, [39](#)
- engine\_data
  - Solver, [24](#)
- engine\_used
  - Solver, [24](#)
- Engineer\_Thesis.cpp
  - main, [41](#)
- Engineer\_Thesis/Engineer\_Thesis/Engineer\_Thesis.cpp,  
[41](#)
- Engineer\_Thesis/Engineer\_Thesis/Instructions.md, [41](#)
- Engineer\_Thesis/Engineer\_Thesis/Planet.h, [41](#), [42](#)
- Engineer\_Thesis/Engineer\_Thesis/Solver.cpp, [42](#)
- Engineer\_Thesis/Engineer\_Thesis/Solver.h, [42](#), [43](#)
- Engineer\_Thesis/Engineer\_Thesis/Vector3D.cpp, [44](#)
- Engineer\_Thesis/Engineer\_Thesis/Vector3D.h, [46](#)
- Engineer\_Thesis/Engineer\_Thesis/Vehicle.cpp, [47](#)
- Engineer\_Thesis/Engineer\_Thesis/Vehicle.h, [47](#)
- Euler
  - Solver, [19](#)
- euler
  - Solver, [17](#)
- force
  - Vehicle, [39](#)
- force\_data
  - Solver, [25](#)
- fuel
  - Vehicle, [39](#)
- fuel\_data
  - Solver, [25](#)
- fuel\_usage
  - Vehicle, [39](#)
- fuel\_used
  - Solver, [25](#)
- G
  - Solver, [25](#)
- grav\_forces
  - Solver, [25](#)
- is\_empty
  - Solver, [20](#)
- isOrb
  - Planet, [13](#)
- kinetic\_data
  - Solver, [25](#)
- KineticEnergy
  - Vehicle, [40](#)
- Load\_data
  - Solver, [20](#)
- Load\_file
  - Solver, [20](#)
- main
  - Engineer\_Thesis.cpp, [41](#)
- mass
  - Planet, [13](#)
  - Vehicle, [40](#)

- mass\_data
  - Solver, 26
- method
  - Solver, 26
- Midpoint
  - Solver, 21
- midpoint
  - Solver, 17
- Move\_Orbit
  - Solver, 21
- Move\_Planet
  - Planet, 12
- Multiply
  - Vector3D, 30
- name
  - Planet, 13
  - Vehicle, 40
- ode
  - Solver, 17
- operator<<
  - Vector3D, 35
  - Vector3D.cpp, 46
- operator\*
  - Vector3D, 30, 33
  - Vector3D.cpp, 44
- operator\*=
  - Vector3D, 31
- operator+
  - Vector3D, 34
  - Vector3D.cpp, 45
- operator+=
  - Vector3D, 31
- operator-
  - Vector3D, 34
  - Vector3D.cpp, 45
- operator-=
  - Vector3D, 32
- operator/
  - Vector3D, 35
  - Vector3D.cpp, 45
- operator/=
  - Vector3D, 32
- orb\_data
  - Planet, 13
- orb\_pos
  - Planet, 13
- orb\_radius
  - Planet, 13
- Planet, 11
  - ang\_velocity, 12
  - isOrb, 13
  - mass, 13
  - Move\_Planet, 12
  - name, 13
  - orb\_data, 13
  - orb\_pos, 13
  - orb\_radius, 13
  - Planet, 12
  - position, 14
  - Print\_info, 12
  - radius, 14
  - start\_ang, 14
- Planets
  - Solver, 26
- Populate
  - Solver, 22
- position
  - Planet, 14
  - Vehicle, 40
- position\_data
  - Solver, 26
- potential\_data
  - Solver, 26
- PotentialEnergy
  - Vehicle, 40
- Print\_info
  - Planet, 12
  - Vehicle, 38
- Print\_Interval
  - Control, 10
- Print\_Pauses
  - Solver, 22
- Push\_Back
  - Solver, 22
- radius
  - Planet, 14
- runge
  - Solver, 17
- Runge\_Kutta
  - Solver, 22
- Save\_data
  - Solver, 23
- Save\_json
  - Solver, 23
- Save\_Planets
  - Solver, 23
- Setup
  - Solver, 23
- Ship
  - Solver, 26
- Solve
  - Solver, 23
- Solver, 14
  - adams, 17
  - Adams\_Bashford, 17
  - Calculate\_Grav, 17
  - Calculate\_Net, 18
  - Check\_Collision, 18
  - distance, 24
  - dvd, 18
  - dxdt, 19
  - engine\_data, 24
  - engine\_used, 24

- Euler, [19](#)
- euler, [17](#)
- force\_data, [25](#)
- fuel\_data, [25](#)
- fuel\_used, [25](#)
- G, [25](#)
- grav\_forces, [25](#)
- is\_empty, [20](#)
- kinetic\_data, [25](#)
- Load\_data, [20](#)
- Load\_file, [20](#)
- mass\_data, [26](#)
- method, [26](#)
- Midpoint, [21](#)
- midpoint, [17](#)
- Move\_Orbit, [21](#)
- ode, [17](#)
- Planets, [26](#)
- Populate, [22](#)
- position\_data, [26](#)
- potential\_data, [26](#)
- Print\_Pauses, [22](#)
- Push\_Back, [22](#)
- runge, [17](#)
- Runge\_Kutta, [22](#)
- Save\_data, [23](#)
- Save\_json, [23](#)
- Save\_Planets, [23](#)
- Setup, [23](#)
- Ship, [26](#)
- Solve, [23](#)
- step, [27](#)
- T, [27](#)
- time, [27](#)
- time\_data, [27](#)
- TimeVect, [27](#)
- UseEngine, [23](#)
- Validate\_Json, [24](#)
- velocity\_data, [27](#)
- start\_ang
  - Planet, [14](#)
- step
  - Solver, [27](#)
- Subtract
  - Vector3D, [32](#)
- T
  - Solver, [27](#)
- time
  - Solver, [27](#)
- time\_data
  - Solver, [27](#)
- timeend
  - Control, [10](#)
- timestart
  - Control, [11](#)
- TimeVect
  - Solver, [27](#)
- UseEngine
  - Solver, [23](#)
- User\_set
  - Vehicle, [38](#)
- Validate\_Json
  - Solver, [24](#)
- Vector3D, [28](#)
  - Add, [29](#)
  - Divide, [30](#)
  - Multiply, [30](#)
  - operator<<, [35](#)
  - operator\*, [30](#), [33](#)
  - operator\*=:, [31](#)
  - operator+, [34](#)
  - operator+=, [31](#)
  - operator-, [34](#)
  - operator-=, [32](#)
  - operator/, [35](#)
  - operator/=, [32](#)
  - Subtract, [32](#)
  - Vector3D, [29](#)
  - VectorsEqual, [33](#)
  - x, [35](#)
  - y, [36](#)
  - z, [36](#)
  - Zero, [33](#)
- Vector3D.cpp
  - operator<<, [46](#)
  - operator\*, [44](#)
  - operator+, [45](#)
  - operator-, [45](#)
  - operator/, [45](#)
- VectorsEqual
  - Vector3D, [33](#)
- Vehicle, [36](#)
  - CalculatedEnergy, [39](#)
  - displacement, [39](#)
  - engine, [39](#)
  - force, [39](#)
  - fuel, [39](#)
  - fuel\_usage, [39](#)
  - KineticEnergy, [40](#)
  - mass, [40](#)
  - name, [40](#)
  - position, [40](#)
  - PotentialEnergy, [40](#)
  - Print\_info, [38](#)
  - User\_set, [38](#)
  - Vehicle, [37](#)
  - velocity, [40](#)
- velocity
  - Vehicle, [40](#)
- velocity\_data
  - Solver, [27](#)
- x
  - Vector3D, [35](#)

y

Vector3D, [36](#)

z

Vector3D, [36](#)

Zero

Vector3D, [33](#)