Orbital Mechanics Library for C++ Documentation

Generated by Doxygen 1.9.5

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`
2. Make sure your compiler "sees" the libraries neccessary (which are located in the external folder):

- 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. (examplary ∗.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 seperate parts:

1.4.1 control

- **starttime** - *array* of three double type elemetents, 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 elemetents, 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 elemetents, 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.
- **n** - integer type *number*, the ammount of steps in the 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 orbitting or strationary [true/false] For orbitting 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:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

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

timestart	is Vector3D object at what time the engine will be turned on
timeend	is Vector3D object at what time the engine will be turned off
engforce	is Vector3D object the direction and magnitude of thrust force vector

See also

Vector3D for information about three-dimenstional 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 (bool save, double time)
  Move Planet around orbit.

Public Attributes

- std::vector< Vector3D > orb_data
  vector used for storing position data of orbitting 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 orbitting 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 (
            bool save,
            double time )  [inline]
```

Move Planet around orbit.

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

Parameters

time	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 orbitting 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 orbitting 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

- Vector3D dvdt (Vector3D f, double m)

  Derivative of Velocity.
- Vector3D dxdt (Vector3D v)

  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 extenstion)

  Function for Loading file from a directory.
- void Load_data (std::string &filename)

  Function Setting parameters from the file.
- bool Check_Collision (Planet &Planet)

  Collistion 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 Reset_Param ()

  Function for Reseting Parameters in RKIV method.
- void Recalculate_Forces (double time, double &mass, Vector3D position, Vector3D &force)

  Recalculating force in RKIV method.
- void Euler (Vector3D &velocity, Vector3D &position, Vector3D force, double mass)

  Euler method.
- void Runge_Kutta (Vector3D &velocity, Vector3D &position, Vector3D &force, double &mass)

  Runge-Kutta IV method.
- void Midpoint (Vector3D &velocity, Vector3D &position, Vector3D force, double mass)

  Midpoint method.
- void Adams_Bashforth (Vector3D &velocity, Vector3D &position, Vector3D &force, double &mass)

  Adams-Bashforth's method.
- void Solve ()

  Main solving function.
- void Push_Back ()

  Put all parameters in vectors.
- void Move_Orbit (bool save)

  Method for changing position of orbitting planets.
- void Save_Planets ()

  Function for saving planets.
- void Save_data ()

  Save simulation data.
- void Print_Pauses ()

  Pauses between simulation elements printing.

Public Attributes

- const double G = 6.67259e-11

    Universal Gravitational constant.
- int index = 0

    Index of current interval for RKIV.
- double temp_mass

    variable to store mass for RKIV
- Vector3D temp_force

    variable to store force for RKIV
- int n_steps

    Number of steps for simulation.
- Vector3D a

    variable to store current acceleration for AB
- Vector3D v

    variable to store current velocity for AB
- Vector3D a_1

    variable to store previous step acceleration for AB
- Vector3D a_2

    variable to store acceleration from two steps before for AB
- Vector3D v_1

    variable to store previous step velocity for AB
- Vector3D v_2

    variable to store velocity from two steps before for AB
- 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

    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_Bashforth()

```
void Solver::Adams_Bashforth (
            Vector3D & velocity,
```

```
Vector3D & position,
Vector3D & force,
double & mass )
```

Adams-Bashforth's method.

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

Parameters

velocity	is velocity at current time
position	is position at current time
force	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

Planet	is a Planet 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()

```
Vector3D Solver::dvdt (
            Vector3D f,
            double m )  [inline]
```

Derivative of Velocity.

Function returning the derivative of velocity.

Parameters

f	is force at time t
m	is mass of the object

Returns

Vector3D discreibing velocity change (acceleration) in the last interval

4.3.3.6 dxdt()

```
Vector3D Solver::dxdt (
            Vector3D v )  [inline]
```

Derivative of Position.

Function returning the derivative of position.

Parameters

v	is velocity at time t

Returns

Vector3D discreibing position (velocity) change in the last interval

4.3.3.7 Euler()

```
void Solver::Euler (
            Vector3D & velocity,
            Vector3D & position,
            Vector3D force,
            double mass )
```

Euler method.

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

Parameters

velocity	is velocity at current time
position	is position at current time
force	is force acting on the spaceship
mass	is mass of the spaceship

4.3.3.8 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

4.3.3.9 Load_file()

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

Function for Loading file from a directory.

Display files in directory and open file with a given filepath

Parameters

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

Returns

loaded file as ifstream

4.3.3.10 Midpoint()

```
void Solver::Midpoint (
            Vector3D & velocity,
            Vector3D & position,
            Vector3D force,
            double mass )
```

Midpoint method.

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

Parameters

velocity	is velocity at current time
position	is position at current time
force	is force acting on the spaceship
mass	is mass of the spaceship

4.3.3.11 Move_Orbit()

```
void Solver::Move_Orbit (
            bool save )
```

Method for changing position of orbitting planets.

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

4.3.3.12 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

4.3.3.13 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.

4.3.3.14 Push_Back()

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

Put all parameters in vectors.

Save all neccessary values at time t into corresponding vectors

4.3.3.15 Recalculate_Forces()

```
void Solver::Recalculate_Forces (
            double time,
            double & mass,
            Vector3D position,
            Vector3D & force )
```

Recalculating force in RKIV method.

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

Parameters

time	is time at which the force and mass should be recalculated
position	is position at the time given
force	is the force that will be recalculated
mass	is the mass that will be recalculated

4.3.3.16 Reset_Param()

```
void Solver::Reset_Param ( )
```

Function for Reseting Parameters in RKIV method.

Resets the paramaters changed for K2, K3, K4 coefficents of RKIV.

See also

Reset_Param() for the function changing paramaters

4.3.3.17 Runge_Kutta()

```
void Solver::Runge_Kutta (
            Vector3D & velocity,
            Vector3D & position,
            Vector3D & 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

velocity	is velocity at current time
position	is position at current time
force	is force acting on the spaceship
mass	is mass of the spaceship

4.3.3.18 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

4.3.3.19 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.

4.3.3.20 Save_Planets()

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

Function for saving planets.

Save all planets' paramaters to a seprate file

4.3.3.21 Setup()

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

Create all simulation elements.

Setup Particle and planets in simulation, fill all engine intervals

4.3.3.22 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_Bashforth(), Midpoint(), Euler(), Runge_Kutta() for more information about solving ODE's

4.3.3.23 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

4.3.3.24 Validate_Json()

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

Json Validation function.

Check if vector file validates against the schema

Parameters

filename	is a filepath for the json file that will be validated

4.3.4 Member Data Documentation

4.3.4.1 a

```
Vector3D Solver::a
```

variable to store current acceleration for AB

4.3.4.2 a_1

```
Vector3D Solver::a_1
```

variable to store previous step acceleration for AB

4.3.4.3 a_2

```
Vector3D Solver::a_2
```

variable to store acceleration from two steps before for AB

4.3.4.4 distance

```
Vector3D Solver::distance
```

4.3.4.5 engine_data

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

vector used for storing current engine data

4.3.4.6 engine_used

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

Boolean used for removing fuel used.

4.3.4.7 force_data

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

vector used for storing current force data

4.3.4.8 fuel_data

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

vectors for storing current fuel value

4.3.4.9 fuel_used

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

ammount of fuel_used at the iteration

4.3.4.10 G

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

Universal Gravitational constant.

4.3.4.11 grav_forces

```
Vector3D Solver::grav_forces
```

Gravitational force and distance from Planet at the time T.

4.3.4.12 index

```
int Solver::index = 0
```

Index of current interval for RKIV.

4.3.4.13 kinetic_data

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

vectors for storing current kinetic energy value

4.3.4.14 mass_data

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

vectors for storing current mass

4.3.4.15 method

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

Method stored as an int used for enum.

4.3.4.16 n_steps

```
int Solver::n_steps
```

Number of steps for simulation.

4.3.4.17 Planets

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

vector storing planets in the simulation

4.3.4.18 position_data

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

vector used for storing current position data

4.3.4.19 potential_data

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

vectors for storing current potential energy value

4.3.4.20 Ship

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

Ship member used in simulation.

4.3.4.21 step

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

time step between increments

4.3.4.22 T

```
double Solver::T
```

time of simulation

4.3.4.23 temp_force

```
Vector3D Solver::temp_force
```

variable to store force for RKIV

4.3.4.24 temp_mass

```
double Solver::temp_mass
```

variable to store mass for RKIV

4.3.4.25 time

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

current simulation time

4.3.4.26 time_data

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

vectors for storing current time value

4.3.4.27 TimeVect

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

vector storing force intervals of type Control

4.3.4.28 v

```
Vector3D Solver::v
```

variable to store current velocity for AB

4.3.4.29 v_1

```
Vector3D Solver::v_1
```

variable to store previous step velocity for AB

4.3.4.30 v_2

```
Vector3D Solver::v_2
```

variable to store velocity from two steps before for AB

4.3.4.31 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)
  Substact 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)
  Substract 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 double &d)
  Multiply vector by scale.
- Vector3D operator/ (const double &d)
  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+ (const Vector3D &v1, const Vector3D &v2)

    Add two vectors with + operator.
- Vector3D operator- (const Vector3D &v1, const Vector3D &v2)

    Substract two vectors with - operator.
- Vector3D operator∗ (const Vector3D &v1, const Vector3D &v2)

    Multiply two vectors with ∗ operator.
- Vector3D operator/ (const Vector3D &v1, const Vector3D &v2)

    Divide two vectors with / operator.
- std::ostream & operator<< (std::ostream &output, const Vector3D &vect)

    overload of << opearator 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

vect	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

vect	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

vect	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 double & d )
```

Multiply vector by scale.

Parameters

d	is double value by which we want to multiply our vector

Returns

Vector3D with values multipliyed by d

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

v1	is vector multiplyied
v2	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

v1	is vector which we are adding into
v2	is vector being added

Returns

v1 increased by v2 value

4.4.3.7 operator-=()

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

Substract two vectors with -= operator.

4.4.3.7.1 Example `v1 -= v2`

Parameters

v1	is vector which we are substracting from
v2	is vector being substracted

Returns

v1 decreased by v2 value

4.4.3.8 operator/()

```
Vector3D Vector3D::operator/ (
        const double & d )
```

Multiply vector by scale.

Parameters

d	is double value by which we want to divide our vector

Returns

Vector3D with values divided by d

4.4.3.9 operator/=()

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

Divide two vectors with /= operator.

4.4.3.9.1 Example `v1 /= v2`

Parameters

v1	is vector divided
v2	is vector we are dividing by

Returns

v1 divided by v2 value

4.4.3.10 Subtract()

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

Substact two vectors.

4.4.3.10.1 Example

```
v1.Substract(v2) // which equals to v1 - v2
```

Parameters

vect	is vector being substracted from the vector calling this method

Returns

Vector3D that is a vector on which method was called with vect value substracted from it

4.4.3.11 VectorsEqual()

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

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

Parameters

vect	is vector compared to the vector calling this method

Returns

true if two vectors are the same, false if not

4.4.3.12 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* (
            const Vector3D & v1,
            const Vector3D & v2 )  [friend]
```

Multiply two vectors with * operator.

4.4.4.1.1 Example `result = v1 * v2`

Parameters

v1	is first vector being multiplyied
v2	is second vector we are multiplying by

Returns

vector equal to v1 * v2

4.4.4.2 operator+

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

Add two vectors with + operator.

4.4.4.2.1 Example `result = v1 + v2`

Parameters

v1	is first vector being addded
v2	is second vector being added

Returns

vector equal to v1 + v2

4.4.4.3 operator-

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

Substract two vectors with - operator.

4.4.4.3.1 Example `result = v1 - v2`

Parameters

v1	is first vector being substracted
v2	is second vector being substracted

Returns

vector equal to v1 - v2

4.4.4.4 operator/

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

Divide two vectors with / operator.

4.4.4.4.1 Example `result = v1 / v2`

Parameters

v1	is first vector being devided
v2	is second vector we are deviding by

Returns

vector equal to v1 / v2

4.4.4.5 operator<<

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

overload of << opearator for printing vectors

Parameters

output	is ofstream where we will print data
vect	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-dimenstional 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 assiging given paramaters.

Parameters

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

```
#pragma once
#include "Vector3D.h"

class Planet {

public:

    std::vector <Vector3D> orb_data;
    double mass = 0;
    double radius = 0;
    double orb_radius = 0;
    double ang_velocity = 0;
    double start_ang = 0;

    Vector3D position = { 0,0,0 };
    Vector3D orb_pos = { 0,0,0 };
    std::string name = "";
    bool isOrb = false;

    Planet() {}

    void Print_info() {

        std::cout « "\nName:  " « name « "\nMass:  " « mass « " kg" « "\nRadius:  " « radius « " m" « "\nPosition:  " « position « " m";

        if (isOrb) {
            std::cout « "\nOrbit Radius:  " « orb_radius « " m" « "\nOrbit Velocity:  " « ang_velocity « " rad/s" « "\nOrbit Center:  " « orb_pos « " m";
        }

        std::cout « "\n";
    }
    void Move_Planet(bool save, double time) {

        position.x = orb_pos.x + orb_radius * cos(start_ang + ang_velocity * time);
        position.z = orb_pos.z + orb_radius * sin(start_ang + ang_velocity * time);

        if (save) {
            orb_data.push_back(position);
        }
    }
};
```

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.

```
#pragma once
#include <fstream>
#include <iomanip>
#include "Vehicle.h"
#include "Planet.h"


//forward declare class
class Control;

class Solver {

public:

    const double G = 6.67259e-11;
    int index = 0;
    double temp_mass;
    Vector3D temp_force;
    int n_steps;
    Vector3D a;
    Vector3D v;
    Vector3D a_1;
    Vector3D a_2;
    Vector3D v_1;
    Vector3D v_2;
    std::vector <double> time_data;
    std::vector <double> mass_data;
    std::vector <double> fuel_data;
    std::vector <double> kinetic_data;
    std::vector <double> potential_data;
    std::vector <Vector3D> position_data;
    std::vector <Vector3D>velocity_data;
    std::vector <Vector3D> engine_data;
    std::vector <Vector3D> force_data;
    std::vector <Planet> Planets;
    std::vector <Control> TimeVect;
    Vehicle Ship = Vehicle("", 0, 0, 0, 0, 0, 0, 0, 0, 0);
    Vector3D grav_forces, distance;
    bool engine_used = false;
    enum ode { adams, euler, midpoint, runge};
    int method=0;
    double T;
    double step = 0;
    double time = 0;
    double fuel_used = 0;
    inline Vector3D dvdt(Vector3D f, double m) { return f/m; }
    inline Vector3D dxdt(Vector3D v) { return v; }
    void Populate();
    void Setup();
    bool Validate_Json(std::string& filename);
    void Save_json();
    std::ifstream Load_file(std::string sys_path, std::string filepath, std::string extenstion);
    void Load_data(std::string& filename);
    bool Check_Collision(Planet& Planet);
    bool UseEngine();
    void Calculate_Grav();
    void Calculate_Net();
    void Reset_Param();

    void Recalculate_Forces(double time, double& mass, Vector3D position, Vector3D& force);
    void Euler(Vector3D& velocity, Vector3D& position, Vector3D force, double mass);
    void Runge_Kutta(Vector3D& velocity, Vector3D& position, Vector3D& force, double& mass);
    void Midpoint(Vector3D& velocity, Vector3D& position, Vector3D force, double mass);
    void Adams_Bashforth(Vector3D& velocity, Vector3D& position, Vector3D& force, double& mass);
    void Solve();
    void Push_Back();
    void Move_Orbit(bool save);
    void Save_Planets();
```

```
    void Save_data();

    void Print_Pauses() {
        std::cout  « std::setfill('=') « std::setw(120) « "\n";
    }
};

class Control{

public:
    Vector3D timestart= { 0,0,0 }, timeend = { 0,0,0 }, engforce = { 0,0,0 };

    Control() {};

    void Print_Interval() {
        std::cout « "\nStart times(x,y,z):  " « timestart « " s"
            « "\nEnd times(x,y,z):  " « timeend « " s"
            « "\nEngine force(x,y,z):  " « engforce « " N" « std::endl;
    }
    bool Check_input(Solver& method) {
            //initial check:
        if (method.TimeVect.empty()) {
            //timestart
            if (timestart.x < 0 || timestart.y < 0 || timestart.z < 0) return false;
            //timeend
            if (timeend.x > method.T || timeend.y > method.T || timeend.z > method.T) return false;
            if (timeend.x < timestart.x || timeend.y < timestart.y || timeend.z < timestart.z) return
   false;
        }
        else {
            if (timestart.x < 0 || timestart.y < 0 || timestart.z < 0) return false;
            if (timeend.x > method.T || timeend.y > method.T || timeend.z > method.T) return false;
            if (timeend.x < timestart.x || timeend.y < timestart.y || timeend.z < timestart.z) return
   false;
            //check if interval does not intersect previous interval
            if (timestart.x < method.TimeVect.back().timeend.x || timestart.y <
   method.TimeVect.back().timeend.y || timestart.z < method.TimeVect.back().timeend.z) return false;
        }
        return true;
    }
};
```

5.8 Engineer_Thesis/Engineer_Thesis/Vector3D.cpp File Reference

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

Functions

- Vector3D operator+ (const Vector3D &v1, const Vector3D &v2)
- Vector3D operator- (const Vector3D &v1, const Vector3D &v2)
- Vector3D operator* (const Vector3D &v1, const Vector3D &v2)
- Vector3D operator/ (const 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* (
            const Vector3D & v1,
            const Vector3D & v2 )
```

5.8.1.1.1 Example `result = v1 * v2`

Parameters

v1	is first vector being multiplyied
v2	is second vector we are multiplying by

Returns

vector equal to v1 $*$ v2

5.8.1.2 operator+()

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

5.8.1.2.1 Example `result = v1 + v2`

Parameters

v1	is first vector being addded
v2	is second vector being added

Returns

vector equal to v1 + v2

5.8.1.3 operator-()

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

5.8.1.3.1 Example `result = v1 - v2`

Parameters

v1	is first vector being substracted
v2	is second vector being substracted

Returns

vector equal to v1 - v2

5.8.1.4 operator/()

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

5.8.1.4.1 Example `result = v1 / v2`

Parameters

v1	is first vector being devided
v2	is second vector we are deviding by

Returns

vector equal to v1 / v2

5.8.1.5 operator<<()

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

Parameters

output	is ofstream where we will print data
vect	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.

```
#pragma once
#include <iostream>

```

```
class Vector3D {

public:
    double x;
    double y;
    double z;

    Vector3D();
    Vector3D(double x, double y, double z);

    Vector3D& Add(const Vector3D& vect);
    Vector3D& Subtract(const Vector3D& vect);
    Vector3D& Multiply(const Vector3D& vect);
    Vector3D& Divide(const Vector3D& vect);
    friend Vector3D operator+ (const Vector3D& v1, const Vector3D& v2);
    friend Vector3D operator- (const Vector3D& v1, const Vector3D& v2);
    friend Vector3D operator* (const Vector3D& v1, const Vector3D& v2);
    friend Vector3D operator/ (const Vector3D& v1, const Vector3D& v2);
    Vector3D& operator+=(const Vector3D& vect);
    Vector3D& operator-=(const Vector3D& vect);
    Vector3D& operator*=(const Vector3D& vect);
    Vector3D& operator/=(const Vector3D& vect);

    Vector3D operator*(const double& d);

    Vector3D operator/(const double& d);

    Vector3D& Zero();

    friend std::ostream& operator « (std::ostream& output, const Vector3D& vect);

    bool VectorsEqual(const Vector3D& vect);
};
```

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.

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

class Vehicle {

public:
    std::string name;
    Vector3D position;
    Vector3D velocity;
```

```
    Vector3D engine = { 0,0,0 };
    Vector3D force = { 0,0,0 };
    Vector3D displacement = { 0,0,0 };
    double mass = 0;
    double fuel = 0;
    double fuel_usage = 0;
    double PotentialEnergy = 0;
    double KineticEnergy = 0;
    bool CalculatedEnergy = 0;

    Vehicle() {};

    Vehicle(std::string n, double rx, double ry, double rz, double vx, double vy, double vz, double m,
   double fuel, double fuel_usage);
    void Print_info();
    void User_set();
};
```

Index