NBody-Marzagora-Tortorella-Vignolo

Final version

Generated by Doxygen 1.10.0

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

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 documented files with brief descriptions:

Chapter 4

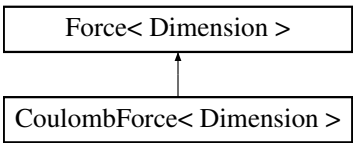
Class Documentation

4.1 CoulombForce< Dimension > Class Template Reference

CoulombForce class models the Coulomb force and inherits from Force.

`#include <force.hpp>`

Inheritance diagram for CoulombForce< Dimension >:



Public Member Functions

- std::array< double, Dimension > calculateForce (const Particle< Dimension > &k, const Particle< Dimension > &q) const override

  Override of function for calculating the force between two particles in order to adapt it to the Coulomb force.

Public Member Functions inherited from Force< Dimension >

- **Force** ()

  Default constructor for the Force class.
- virtual ~**Force** ()

  Default destructor for the Force class.

4.1.1 Detailed Description

template<size_t Dimension>
class CoulombForce< Dimension >

CoulombForce class models the Coulomb force and inherits from Force.

Template Parameters

Dimension	Number of dimensions of the simulation

4.1.2 Member Function Documentation

4.1.2.1 calculateForce()

```
template<size_t Dimension>
std::array< double, Dimension > CoulombForce< Dimension >::calculateForce (
            const Particle< Dimension > & k,
            const Particle< Dimension > & q ) const  [inline], [override], [virtual]
```

Override of function for calculating the force between two particles in order to adapt it to the Coulomb force.

Parameters

k	First particle involved in the force calculation.
q	Second particle involved in the force calculation.

Returns

Array representing the force vector between particles.

Implements Force< Dimension >.

The documentation for this class was generated from the following file:

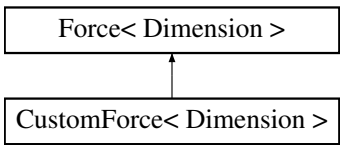
- C:/Users/PROPRIETARIO/NBody-Marzagora-Tortorella-Vignolo/src/utils/force.hpp

4.2 CustomForce< Dimension > Class Template Reference

CustomForce class models a custom force similar to the gravitational force and inherits from Force. For test purposes only.

```
#include <force.hpp>
```

Inheritance diagram for CustomForce< Dimension >:



Public Member Functions

- **CustomForce** (double G)
- std::array< double, Dimension > calculateForce (const Particle< Dimension > &k, const Particle< Dimension > &q) const override

  Pure virtual function for calculating the force between two particles.

Public Member Functions inherited from Force< Dimension >

- **Force** ()

  Default constructor for the Force class.
- virtual ~**Force** ()

  Default destructor for the Force class.

4.2.1 Detailed Description

template<size_t Dimension>
class CustomForce< Dimension >

CustomForce class models a custom force similar to the gravitational force and inherits from Force. For test purposes only.

Template Parameters

Dimension	Number of dimensions of the simulation

4.2.2 Member Function Documentation

4.2.2.1 calculateForce()

```
template<size_t Dimension>
std::array< double, Dimension > CustomForce< Dimension >::calculateForce (
            const Particle< Dimension > & p1,
            const Particle< Dimension > & p2 ) const  [inline], [override], [virtual]
```

Pure virtual function for calculating the force between two particles.

Parameters

p1	First particle involved in the force calculation.
p2	Second particle involved in the force calculation.

Returns

Array representing the force vector between particles.

Implements Force< Dimension >.

The documentation for this class was generated from the following file:

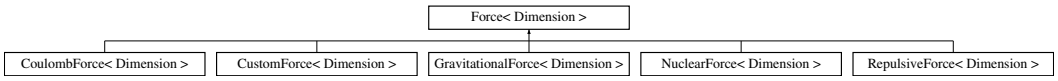
- C:/Users/PROPRIETARIO/NBody-Marzagora-Tortorella-Vignolo/src/utils/force.hpp

4.3 Force< Dimension > Class Template Reference

Force class for particle interactions.

`#include <force.hpp>`

Inheritance diagram for Force< Dimension >:



Public Member Functions

- **Force** ()

  Default constructor for the Force class.
- virtual std::array< double, Dimension > calculateForce (const Particle< Dimension > &p1, const Particle< Dimension > &p2) const =0

  Pure virtual function for calculating the force between two particles.
- virtual ~**Force** ()

  Default destructor for the Force class.

4.3.1 Detailed Description

template<size_t Dimension>
class Force< Dimension >

Force class for particle interactions.

Template Parameters

Dimension	Number of dimensions of the simulation

4.3.2 Member Function Documentation

4.3.2.1 calculateForce()

```
template<size_t Dimension>
virtual std::array< double, Dimension > Force< Dimension >::calculateForce (
            const Particle< Dimension > & p1,
            const Particle< Dimension > & p2 ) const  [pure virtual]
```

Pure virtual function for calculating the force between two particles.

Parameters

p1	First particle involved in the force calculation.
p2	Second particle involved in the force calculation.

Returns

Array representing the force vector between particles.

Implemented in GravitationalForce< Dimension >, CoulombForce< Dimension >, NuclearForce< Dimension >, RepulsiveForce< Dimension >, and CustomForce< Dimension >.

The documentation for this class was generated from the following file:

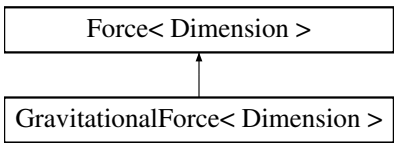
- C:/Users/PROPRIETARIO/NBody-Marzagora-Tortorella-Vignolo/src/utils/force.hpp

4.4 GravitationalForce< Dimension > Class Template Reference

GravitationalForce class models the gravitational force and inherits from Force.

`#include <force.hpp>`

Inheritance diagram for GravitationalForce< Dimension >:



Public Member Functions

- std::array< double, Dimension > calculateForce (const Particle< Dimension > &k, const Particle< Dimension > &q) const override

  Override of function for calculating the force between two particles in order to adapt it to the gravitational force.

Public Member Functions inherited from Force< Dimension >

- **Force** ()

  Default constructor for the Force class.
- virtual ~**Force** ()

  Default destructor for the Force class.

4.4.1 Detailed Description

template<size_t Dimension>
class GravitationalForce< Dimension >

GravitationalForce class models the gravitational force and inherits from Force.

Template Parameters

Dimension	Number of dimensions of the simulation

4.4.2 Member Function Documentation

4.4.2.1 calculateForce()

```
template<size_t Dimension>
std::array< double, Dimension > GravitationalForce< Dimension >::calculateForce (
            const Particle< Dimension > & k,
            const Particle< Dimension > & q ) const  [inline], [override], [virtual]
```

Override of function for calculating the force between two particles in order to adapt it to the gravitational force.

Parameters

p1	First particle involved in the force calculation.
p2	Second particle involved in the force calculation.

Returns

Array representing the force vector between particles.

Implements Force< Dimension >.

The documentation for this class was generated from the following file:

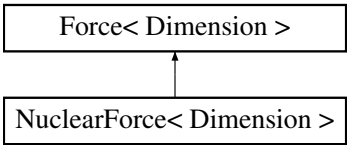
- C:/Users/PROPRIETARIO/NBody-Marzagora-Tortorella-Vignolo/src/utils/force.hpp

4.5 NuclearForce< Dimension > Class Template Reference

NuclearForce class models the nuclear force and inherits from Force.

```
#include <force.hpp>
```

Inheritance diagram for NuclearForce< Dimension >:



Public Member Functions

- std::array< double, Dimension > calculateForce (const Particle< Dimension > &k, const Particle< Dimension > &q) const override

  Override of function for calculating the force between two particles in order to adapt it to the nuclear force.

Public Member Functions inherited from Force< Dimension >

- **Force** ()

  Default constructor for the Force class.
- virtual ~**Force** ()

  Default destructor for the Force class.

4.5.1 Detailed Description

template<size_t Dimension>
class NuclearForce< Dimension >

NuclearForce class models the nuclear force and inherits from Force.

Template Parameters

Dimension	Number of dimensions of the simulation

4.5.2 Member Function Documentation

4.5.2.1 calculateForce()

```
template<size_t Dimension>
std::array< double, Dimension > NuclearForce< Dimension >::calculateForce (
            const Particle< Dimension > & k,
            const Particle< Dimension > & q ) const  [inline], [override], [virtual]
```

Override of function for calculating the force between two particles in order to adapt it to the nuclear force.

Parameters

k	First particle involved in the force calculation.
q	Second particle involved in the force calculation.

Returns

Array representing the force vector between particles.

Implements Force< Dimension >.

The documentation for this class was generated from the following file:

- C:/Users/PROPRIETARIO/NBody-Marzagora-Tortorella-Vignolo/src/utils/force.hpp

4.6 Particle< Dimension > Class Template Reference

Particle class that models particles.

```
#include <particle.hpp>
```

Public Member Functions

- Particle (int id, double p, std::array< double, Dimension > pos, std::array< double, Dimension > v, double radius, bool type)

  Constructor that initializes the mass, position, and velocity of the particle.
- void setVel (const std::array< double, Dimension > &v)

  Setter method for velocity of the particle.
- void setProperty (double p)

  Setter method for property of the particle.
- double getProperty () const

  Getter method for property of the particle.
- std::array< double, Dimension > **getAccel** () const
- std::array< double, Dimension > getPos () const

  Getter method for positions of the particle.
- std::array< double, Dimension > getVel () const

  Getter method for velocity of the particle.
- std::array< double, Dimension > getForce () const

  Getter method for force of the particle.
- int getId () const

  Getter method for the ID of the particle.
- double getRadius () const

  Getter method for the radius of the particle.
- bool getType () const

  Getter method for the ID of the particle.
- void **resetForce** ()
- double **squareDistance** (const Particle< Dimension > &p) const
- void addForce (const std::array< double, Dimension > &force_qk)

  Method that adds force.
- void update (const double delta_t)

  Method that updates positions and velocities using Euler integration.
- void **velocityVerletUpdate** (const double delta_t)
- void updateAndReset (const double delta_t)

  Method that updates positions and velocities using Euler integration and resets force.
- void **printStates** () const

  Method that prints info about the particles.
- void manageCollision (Particle< Dimension > &p, double dim)

  Method that manages collisions between two particles and between a particle and the borders of the simulation (visual purposes only)
- bool hitsBoundary (double dim)

  Method that checks if the boundary has been touched.
- ~**Particle** ()

  Default destructor of Particle class.

4.6.1 Detailed Description

template<size_t Dimension>
class Particle< Dimension >

Particle class that models particles.

Template Parameters

Dimension	Number of dimensions of the simulation

4.6.2 Constructor & Destructor Documentation

4.6.2.1 Particle()

```
template<size_t Dimension>
Particle< Dimension >::Particle (
            int id,
            double p,
            std::array< double, Dimension > pos,
            std::array< double, Dimension > v,
            double radius,
            bool type )  [inline]
```

Constructor that initializes the mass, position, and velocity of the particle.

Parameters

id	Id of the particle
p	Property of particle: mass for gravitational force, charge for Coulomb force
pos	Array of positions of the particle
v	Array of velocities of the particle
radius	Radius of the particle
type	Type of the particle: 0 for gravitational, 1 for Coulomb

4.6.3 Member Function Documentation

4.6.3.1 addForce()

```
template<size_t Dimension>
void Particle< Dimension >::addForce (
            const std::array< double, Dimension > & force_qk )  [inline]
```

Method that adds force.

Parameters

force_qk	Array of components of the force between particles q and k

4.6.3.2 getForce()

```
template<size_t Dimension>
std::array< double, Dimension > Particle< Dimension >::getForce ( ) const  [inline]
```

Getter method for force of the particle.

Returns

force Array of force of the particle

4.6.3.3 getId()

```
template<size_t Dimension>
int Particle< Dimension >::getId ( ) const  [inline]
```

Getter method for the ID of the particle.

Returns

id ID of the particle

4.6.3.4 getPos()

```
template<size_t Dimension>
std::array< double, Dimension > Particle< Dimension >::getPos ( ) const  [inline]
```

Getter method for positions of the particle.

Returns

pos Array of positions of the particle

4.6.3.5 getProperty()

```
template<size_t Dimension>
double Particle< Dimension >::getProperty ( ) const  [inline]
```

Getter method for property of the particle.

Returns

property Property of particle: mass for gravitational force, charge for Coulomb force

4.6.3.6 getRadius()

```
template<size_t Dimension>
double Particle< Dimension >::getRadius ( ) const  [inline]
```

Getter method for the radius of the particle.

Returns

radius Radius of the particle

4.6.3.7 getType()

```
template<size_t Dimension>
bool Particle< Dimension >::getType ( ) const  [inline]
```

Getter method for the ID of the particle.

Returns

type Type of the particle: 0 for gravitational, 1 for Coulomb

4.6.3.8 getVel()

```
template<size_t Dimension>
std::array< double, Dimension > Particle< Dimension >::getVel ( ) const  [inline]
```

Getter method for velocity of the particle.

Returns

vel Array of velocity of the particle

4.6.3.9 hitsBoundary()

```
template<size_t Dimension>
bool Particle< Dimension >::hitsBoundary (
            double dim )  [inline]
```

Method that checks if the boundary has been touched.

Parameters

dim	Number of dimensions of the simulation

Returns

bound_touched true if the border has been touched, false otherwise

4.6.3.10 manageCollision()

```
template<size_t Dimension>
void Particle< Dimension >::manageCollision (
            Particle< Dimension > & p,
            double dim )  [inline]
```

Method that manages collisions between two particles and between a particle and the borders of the simulation (visual purposes only)

Parameters

p	Particle which has collided with this particle
dim	Number of dimensions of the simulation

4.6.3.11 setProperty()

```
template<size_t Dimension>
void Particle< Dimension >::setProperty (
            double p )  [inline]
```

Setter method for property of the particle.

Parameters

p	Property of particle: mass for gravitational force, charge for Coulomb force

4.6.3.12 setVel()

```
template<size_t Dimension>
void Particle< Dimension >::setVel (
            const std::array< double, Dimension > & v )  [inline]
```

Setter method for velocity of the particle.

Parameters

v	Array of velocities of the particle

4.6.3.13 update()

```
template<size_t Dimension>
void Particle< Dimension >::update (
            const double delta_t )  [inline]
```

Method that updates positions and velocities using Euler integration.

Parameters

delta_t	Time step after which the state of the particle is being updated

4.6.3.14 updateAndReset()

```
template<size_t Dimension>
```

```
void Particle< Dimension >::updateAndReset (
            const double delta_t )  [inline]
```

Method that updates positions and velocities using Euler integration and resets force.

Parameters

delta_t	Time step after which the state of the particle is being updated

The documentation for this class was generated from the following files:

- C:/Users/PROPRIETARIO/NBody-Marzagora-Tortorella-Vignolo/src/utils/force.hpp
- C:/Users/PROPRIETARIO/NBody-Marzagora-Tortorella-Vignolo/src/utils/particle.hpp

4.7 QuadtreeNode< Dimension > Class Template Reference

QuadtreeNode class that models a node of the quadtree needed for the algorithm of Barnes-Hut.

```
#include <quadtreeNode.hpp>
```

Public Member Functions

- QuadtreeNode (double x, double y, double w, int maxDepthValue=10)

  Constructor that initializes the center of the node and its width.
- ~**QuadtreeNode** ()

  Default destructor for the quadtreeNode class.
- const std::array< double, Dimension > & getCenter () const

  Getter function that returns the coordinates of the center of the node.
- double getWidth () const

  Getter function that returns the width of the region.
- bool isLeaf () const

  Boolean function that returns true if a node is leaf, false otherwise.
- const std::shared_ptr< Particle< Dimension > > & getParticle () const

  Getter function that returns the particle inside the node.
- const std::array< double, Dimension > & getTotalCenter () const

  Getter function that returns the coordinates of the center of mass of the node.
- double getTotalMass () const

  Getter function that returns the sum of the masses of the paticles inside the node.
- int getCount () const

  Getter function that returns the number of particles inside a node and its children.
- const std::array< std::unique_ptr< QuadtreeNode< Dimension > >, 4 > & getChildren () const

  Getter function that returns the children of a node.
- void insertNode (std::unique_ptr< QuadtreeNode< Dimension > > subTreeRoot, int depth=0)

  Function that inserts a subtree as a child of a father node: if the current node is a leaf, it becomes an internal node and the subtree is inserted as a child of the father node. Otherwise, it finds the quadrant in which the subtree has to be inserted and calls: if the quadrant node exists and the child node does not exist, it is initialized with the subtree, otherwise the subtree is inserted in the child node.
- void **split** ()

Function that creates the children of a node: it creates the four children of a node and inserts them in the children array.

- int getQuadrantIndex (const std::array< double, Dimension > &pos)

  Function that returns the index of the quadrant in which the particle has to be inserted.

- void insert (std::shared_ptr< Particle< Dimension > > p, int depth=0)

  Function that updates the attributes of the node and then inserts a particle in the quadtree: if the node is a leaf and it is empty, the particle is inserted in the node. Otherwise, if a paricle is already inside the node, the particle is repositioned inside one of the children, based on its coordinates, and the method is recursively called for the new particle, in order to handle even the case where two particles should go in the same child.

- void updateAttributes (std::shared_ptr< Particle< Dimension > > newParticle, int numParticles=1)

  Function that updates the attributes of the node: it updates the total mass of the node and its children and the coordinates of the of the center of mass, in case it is needed to create the approximated particle for the evaluation of the force.

- void **printTree** (int depth=0) const

  Function that prints the current node's details: the level where the node is inside the tree, the coordinates of the center, the mass, the number of particles inside the node and if the node is a leaf or an internal node. If the node is a leaf, it displays the particle and exits, otherwise it recursively displays the child nodes.

- std::unique_ptr< Particle< Dimension > > createApproximateParticle ()

  Function that creates the approximated particle which represent a group of particles identified by the center of mass of the particles inside the quadtree. It is used to evaluate the force between a particle a group of particles that are sufficiently far from each other, so that the approximation is accurate enough. The new particle has id -1 in order not to confuse it for a real particle, the mass is the sum of the masses of the particles inside the node and as center the center of mass of the particles (now initialized as 0.0).

4.7.1 Detailed Description

template<size_t Dimension>
class QuadtreeNode< Dimension >

QuadtreeNode class that models a node of the quadtree needed for the algorithm of Barnes-Hut.

Template Parameters

Dimension	Number of dimensions of the simulation

4.7.2 Constructor & Destructor Documentation

4.7.2.1 QuadtreeNode()

```
template<size_t Dimension>
QuadtreeNode< Dimension >::QuadtreeNode (
            double x,
            double y,
            double w,
            int maxDepthValue = 10 )  [inline]
```

Constructor that initializes the center of the node and its width.

Parameters

width	Width of the region covered by the node

Parameters

leaf	Boolean value that indicates if the node is a leaf or not
totalCenter	Coordinates of the center of mass of the particles of the node
center	Coordinates of the center of the node
totalMass	Total mass of the particles' of the node and its children
count	Number of particles inside the node

4.7.3 Member Function Documentation

4.7.3.1 createApproximateParticle()

```
template<size_t Dimension>
std::unique_ptr< Particle< Dimension > > QuadtreeNode< Dimension >::createApproximate↩
Particle ( )  [inline]
```

Function that creates the approximated particle which represent a group of particles identified by the center of mass of the particles inside the quadtree. It is used to evaluate the force between a particle a group of particles that are sufficiently far from each other, so that the approximation is accurate enough. The new particle has id -1 in order not to confuse it for a real particle, the mass is the sum of the masses of the particles inside the node and as center the center of mass of the particles (now initialized as 0.0).

Returns

Unique pointer to the approximated particle

4.7.3.2 getCenter()

```
template<size_t Dimension>
const std::array< double, Dimension > & QuadtreeNode< Dimension >::getCenter ( ) const  [inline]
```

Getter function that returns the coordinates of the center of the node.

Returns

Coordinates of the center

4.7.3.3 getChildren()

```
template<size_t Dimension>
const std::array< std::unique_ptr< QuadtreeNode< Dimension > >, 4 > & QuadtreeNode< Dimension
>::getChildren ( ) const  [inline]
```

Getter function that returns the children of a node.

Returns

Array that contains the unique pointers to the children of a node

4.7.3.4 getCount()

```
template<size_t Dimension>
int QuadtreeNode< Dimension >::getCount ( ) const  [inline]
```

Getter function that returns the number of particles inside a node and its children.

Returns

Number of particles

4.7.3.5 getParticle()

```
template<size_t Dimension>
const std::shared_ptr< Particle< Dimension > > & QuadtreeNode< Dimension >::getParticle ( )
const  [inline]
```

Getter function that returns the particle inside the node.

Returns

Particle contained inside the node

4.7.3.6 getQuadrantIndex()

```
template<size_t Dimension>
int QuadtreeNode< Dimension >::getQuadrantIndex (
            const std::array< double, Dimension > & pos )  [inline]
```

Function that returns the index of the quadrant in which the particle has to be inserted.

Parameters

pos	Coordinates of the particle to be inserted

Returns

Index of the quadrant in which the particle has to be inserted

4.7.3.7 getTotalCenter()

```
template<size_t Dimension>
const std::array< double, Dimension > & QuadtreeNode< Dimension >::getTotalCenter ( ) const
[inline]
```

Getter function that returns the coordinates of the center of mass of the node.

Returns

Coordinates of the center of mass of the node

4.7.3.8 getTotalMass()

```
template<size_t Dimension>
double QuadtreeNode< Dimension >::getTotalMass ( ) const  [inline]
```

Getter function that returns the sum of the masses of the paticles inside the node.

Returns

sum of the masses of the paticles inside the node

4.7.3.9 getWidth()

```
template<size_t Dimension>
double QuadtreeNode< Dimension >::getWidth ( ) const  [inline]
```

Getter function that returns the width of the region.

Returns

Width of the region covered by the center

4.7.3.10 insert()

```
template<size_t Dimension>
void QuadtreeNode< Dimension >::insert (
            std::shared_ptr< Particle< Dimension > > p,
            int depth = 0 )  [inline]
```

Function that updates the attributes of the node and then inserts a particle in the quadtree: if the node is a leaf and it is empty, the particle is inserted in the node. Otherwise, if a paricle is already inside the node, the particle is repositioned inside one of the children, based on its coordinates, and the method is recursively called for the new particle, in order to handle even the case where two particles should go in the same child.

Parameters

p	Particle to be inserted inside the node

4.7.3.11 insertNode()

```
template<size_t Dimension>
void QuadtreeNode< Dimension >::insertNode (
            std::unique_ptr< QuadtreeNode< Dimension > > subTreeRoot,
            int depth = 0 )  [inline]
```

Function that inserts a subtree as a child of a father node: if the current node is a leaf, it becomes an internal node and the subtree is inserted as a child of the father node. Otherwise, it finds the quadrant in which the subtree has to be inserted and calls: if the quadrant node exists and the child node does not exist, it is initialized with the subtree, otherwise the subtree is inserted in the child node.

Parameters

subTreeRoot	Unique pointer to the subtree to be inserted

4.7.3.12 isLeaf()

```
template<size_t Dimension>
bool QuadtreeNode< Dimension >::isLeaf ( ) const  [inline]
```

Boolean function that returns true if a node is leaf, false otherwise.

Returns

True if a node is leaf, false otherwise

4.7.3.13 updateAttributes()

```
template<size_t Dimension>
void QuadtreeNode< Dimension >::updateAttributes (
            std::shared_ptr< Particle< Dimension > > newParticle,
            int numParticles = 1 )  [inline]
```

Function that updates the attributes of the node: it updates the total mass of the node and its children and the coordinates of the of the center of mass, in case it is needed to create the approximated particle for the evaluation of the force.

Parameters

newParticle	Particle to be inserted inside the node
numParticles	Number of particles to be inserted the node (default is 1, needed to update the count of the particles inside the node)

The documentation for this class was generated from the following file:

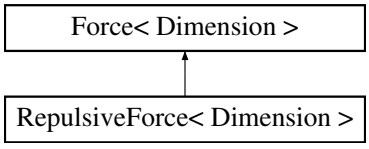
- C:/Users/PROPRIETARIO/NBody-Marzagora-Tortorella-Vignolo/src/utils/quadtreeNode.hpp

4.8 RepulsiveForce< Dimension > Class Template Reference

RepulsiveForce class models the repulsive force and inherits from Force.

```
#include <force.hpp>
```

Inheritance diagram for RepulsiveForce< Dimension >:



Public Member Functions

- std::array< double, Dimension > calculateForce (const Particle< Dimension > &k, const Particle< Dimension > &q) const override

  Override of function for calculating the force between two particles in order to adapt it to the repulsive force.

Public Member Functions inherited from Force< Dimension >

- **Force** ()

  Default constructor for the Force class.
- virtual ~**Force** ()

  Default destructor for the Force class.

4.8.1 Detailed Description

template<size_t Dimension>
class RepulsiveForce< Dimension >

RepulsiveForce class models the repulsive force and inherits from Force.

Template Parameters

Dimension	Number of dimensions of the simulation

4.8.2 Member Function Documentation

4.8.2.1 calculateForce()

```
template<size_t Dimension>
std::array< double, Dimension > RepulsiveForce< Dimension >::calculateForce (
            const Particle< Dimension > & k,
            const Particle< Dimension > & q ) const  [inline], [override], [virtual]
```

Override of function for calculating the force between two particles in order to adapt it to the repulsive force.

Parameters

k	First particle involved in the force calculation.
q	Second particle involved in the force calculation.

Returns

Array representing the force vector between particles.

Implements Force< Dimension >.

The documentation for this class was generated from the following file:

- C:/Users/PROPRIETARIO/NBody-Marzagora-Tortorella-Vignolo/src/utils/force.hpp

Chapter 5

File Documentation

5.1 fileUtilities.hpp

```
00001 #include <vector>
00002 #include <fstream>
00003 #include <iostream>
00004 #include <random>
00005 #include "particle.hpp"
00006
00023 template <size_t Dimension>
00024 void printInitialStateOnFile(std::vector<Particle<Dimension>> *particles, int dim, std::string
      fileName, int it, int speedUp, size_t numFilesAndThreads)
00025 {
00026     std::ofstream file(fileName);
00027     if (file.is_open())
00028     {
00029         file « (*particles).size() « std::endl;
00030         file « dim « std::endl;
00031         file « Dimension « std::endl;
00032         file « it / speedUp « std::endl;
00033         file « numFilesAndThreads « std::endl;
00034
00035         for (const Particle<Dimension> &p : (*particles))
00036             file « p.getRadius() « std::endl;
00037         for (const Particle<Dimension> &p : (*particles))
00038         {
00039             file « p.getId() « ",";
00040             const auto &pos = p.getPos();
00041             for (size_t i = 0; i < Dimension; ++i)
00042             {
00043                 file « pos[i];
00044                 if (i < Dimension - 1)
00045                     file « ",";
00046             }
00047             file « std::endl;
00048         }
00049     }
00050     else
00051     {
00052         std::cout « "Unable to open file";
00053     }
00054     file.close();
00055 }
00056
00065 template <size_t Dimension>
00066 void writeParticlePositionsToFile(const std::vector<Particle<Dimension>> &particles, FILE* file)
00067 {
00068     for (const auto &q : particles)
00069     {
00070         fprintf(file, "%d,", q.getId());
00071         const auto& pos = q.getPos();
00072         for (size_t i = 0; i < Dimension; ++i) {
00073             fprintf(file, "%f", pos[i]);
00074             if (i < Dimension - 1) {
00075                 fprintf(file, ",");
00076             }
00077         }
00078         fprintf(file, "\n");
00079     }
00080 }
```

5.2 force.hpp

```
00001 #ifndef FORCE_H
00002 #define FORCE_H
00003
00004 #include <array>
00005 #include <cmath>
00006
00007 template<size_t Dimension>
00008 class Particle;
00009
00014 template<size_t Dimension>
00015 class Force{
00016     public:
00020         Force(){}
00021
00028         virtual std::array<double,Dimension> calculateForce(const Particle<Dimension> &p1, const
      Particle<Dimension> &p2) const = 0;
00029
00033         virtual ~Force() {}
00034 };
00035
00040 template<size_t Dimension>
00041 class GravitationalForce : public Force<Dimension>{
00042     public:
00049         std::array<double,Dimension> calculateForce(const Particle<Dimension> &k, const
      Particle<Dimension> &q) const override{
00050             const auto& k_pos = k.getPos();
00051             const auto& q_pos = q.getPos();
00052
00053             std::array<double, Dimension> pos_diff;
00054             for(size_t i = 0; i < Dimension; ++i) pos_diff[i] = q_pos[i] - k_pos[i];
00055
00056             double dist_squared = 0.0;
00057             for(size_t i = 0; i < Dimension; ++i) dist_squared = pos_diff[i] * pos_diff[i] +
      dist_squared;
00058             double dist = std::sqrt(dist_squared);
00059             double dist_cubed = dist * dist * dist;
00060
00061             double gravF = G * q.getProperty() * k.getProperty() / dist_cubed;
00062
00063             std::array<double, Dimension> force_qk;
00064             for(size_t i = 0; i < Dimension; ++i) force_qk[i] = gravF * pos_diff[i];
00065
00066             return force_qk;
00067         }
00068     private:
00069         double const G = 100;
00070 };
00071
00076 template<size_t Dimension>
00077 class CoulombForce : public Force<Dimension>{
00078     public:
00085         std::array<double,Dimension> calculateForce(const Particle<Dimension> &k, const
      Particle<Dimension> &q) const override{
00086             const auto& k_pos = k.getPos();
00087             const auto& q_pos = q.getPos();
00088
00089             std::array<double, Dimension> pos_diff;
00090             for(size_t i = 0; i < Dimension; ++i) pos_diff[i] = q_pos[i] - k_pos[i];
00091
00092             double dist_squared = 0.0;
00093             for(size_t i = 0; i < Dimension; ++i) dist_squared = pos_diff[i] * pos_diff[i] +
      dist_squared;
00094             double dist = std::sqrt(dist_squared);
00095             double dist_cubed = dist * dist * dist;
00096
00097             double coulF = K * q.getProperty() * k.getProperty() / dist_cubed;
00098
00099             std::array<double, Dimension> force_qk;
00100             for(size_t i = 0; i < Dimension; ++i) force_qk[i] = -coulF * pos_diff[i];
00101
00102             return force_qk;
00103         }
00104     private:
00105         double const K = 8.987e-09;
00106 };
00107
00112 template<size_t Dimension>
00113 class NuclearForce : public Force<Dimension>{
00114     public:
00121         std::array<double,Dimension> calculateForce(const Particle<Dimension> &k, const
      Particle<Dimension> &q) const override{
00122             const auto& k_pos = k.getPos();
00123             const auto& q_pos = q.getPos();
00124
00125             std::array<double, Dimension> pos_diff;
```

```
00126             for(size_t i = 0; i < Dimension; ++i) pos_diff[i] = k_pos[i] - q_pos[i];
00127
00128             double dist = 0.0;
00129             for(size_t i = 0; i < Dimension; ++i) dist = pos_diff[i] * pos_diff[i] + dist;
00130
00131             double nukeF = -std::exp(-dist/r0)/dist;
00132
00133             std::array<double, Dimension> force_qk;
00134
00135             for(size_t i = 0; i < Dimension; ++i) force_qk[i] = nukeF * pos_diff[i];
00136
00137             return force_qk;
00138         }
00139     private:
00140         double const r0 = 1.0;
00141 };
00142
00147 template<size_t Dimension>
00148 class RepulsiveForce : public Force<Dimension>{
00149     public:
00156         std::array<double, Dimension> calculateForce(const Particle<Dimension> &k, const
      Particle<Dimension> &q) const override{
00157             const auto& k_pos = k.getPos();
00158             const auto& q_pos = q.getPos();
00159
00160             std::array<double, Dimension> pos_diff;
00161             for(size_t i = 0; i < Dimension; ++i) pos_diff[i] = k_pos[i] - q_pos[i];
00162
00163             double dist = 0.0;
00164             for(size_t i = 0; i < Dimension; ++i) dist = pos_diff[i] * pos_diff[i] + dist;
00165
00166             double repF = kp / dist;
00167
00168             std::array<double, Dimension> force_qk;
00169
00170             for(size_t i = 0; i < Dimension; ++i) force_qk[i] = repF * pos_diff[i] / dist;
00171
00172             return force_qk;
00173         }
00174     private:
00175         double const kp = 1.0;
00176 };
00177
00183 template<size_t Dimension>
00184 class CustomForce : public Force<Dimension>{
00185     public:
00186         CustomForce(double G) : G(G) {}
00187         std::array<double,Dimension> calculateForce(const Particle<Dimension> &k, const
      Particle<Dimension> &q) const override{
00188             const auto& k_pos = k.getPos();
00189             const auto& q_pos = q.getPos();
00190
00191             std::array<double, Dimension> pos_diff;
00192             for(size_t i = 0; i < Dimension; ++i) pos_diff[i] = q_pos[i] - k_pos[i];
00193
00194             double dist_squared = 0.0;
00195             for(size_t i = 0; i < Dimension; ++i) dist_squared = pos_diff[i] * pos_diff[i] +
      dist_squared;
00196             double dist = std::sqrt(dist_squared);
00197             double dist_cubed = dist * dist * dist;
00198
00199             double customF = G * q.getProperty() * k.getProperty() / dist_cubed;
00200
00201             std::array<double, Dimension> force_qk;
00202             for(size_t i = 0; i < Dimension; ++i) force_qk[i] = customF*pos_diff[i];
00203
00204             return force_qk;
00205         }
00206     private:
00207         double const G;
00208 };
00209
00210 #endif
```

5.3 particle.hpp

```
00001 #ifndef PARTICLE_H
00002 #define PARTICLE_H
00003
00004 #include <vector>
00005 #include <iostream>
00006 #include <array>
00007
```

```
00008
00009 template<size_t Dimension>
00010 class Force;
00011
00016 template<size_t Dimension>
00017 class Particle {
00018     public:
00028         Particle(int id, double p, std::array<double, Dimension> pos, std::array<double, Dimension> v,
      double radius, bool type)
00029             : id(id), property(p), pos{pos}, vel{v}, force{}, type(type), radius(radius) {
00030                 for(size_t j = 0; j < Dimension; ++j) accel[j] = 0.0;
00031             }
00032
00038         void setVel(const std::array<double, Dimension> &v){
00039             for(size_t i=0; i < Dimension; ++i) vel[i] = v[i];
00040         }
00041
00046         void setProperty(double p){
00047             property = p;
00048         }
00049
00054         double getProperty() const {
00055             return property;
00056         }
00057
00058         std::array<double, Dimension> getAccel() const {
00059             return accel;
00060         }
00061
00066         std::array<double, Dimension> getPos() const{
00067             return pos;
00068         }
00069
00074         std::array<double, Dimension> getVel() const{
00075             return vel;
00076         }
00077
00082         std::array<double, Dimension> getForce() const{
00083         return force;
00084         }
00085
00086
00091         int getId() const{
00092             return id;
00093         }
00094
00099         double getRadius() const{
00100             return radius;
00101         }
00102
00107         bool getType() const{
00108             return type;
00109         }
00110
00111         //method that resets total force for next implementation
00112         void resetForce() {
00113             for (size_t i = 0; i < Dimension; ++i ) {
00114                 force[i] = 0.0;
00115             }
00116         }
00117
00118         //method that calculates the square of the distance
00119         double squareDistance(const Particle<Dimension> &p) const{
00120             const auto& k_pos = getPos();
00121             const auto& p_pos = p.getPos();
00122
00123             std::array<double, Dimension> diff;
00124             for(size_t i=0; i < Dimension; ++i) diff[i] = k_pos[i] - p_pos[i];
00125
00126             double square_dist = 0.0;
00127             for(const auto& d : diff) square_dist += d * d;
00128
00129             return square_dist;
00130         }
00131
00137         void addForce(const std::array<double, Dimension> &force_qk) {
00138
00139             for(size_t i = 0; i < Dimension; ++i){
00140                 force[i] += force_qk[i];
00141             }
00142
00143         }
00144
00150         void update(const double delta_t) {
00151             for(size_t i = 0; i < Dimension; ++i)
00152             {
00153                 pos[i] += vel[i] * delta_t;
```

```
00154                 vel[i] += (force[i] / ((property<0)? -property:property)) * delta_t;
00155             }
00156         }
00157 
00158         void velocityVerletUpdate(const double delta_t) {
00159             std::array<double, Dimension> prevAccel;
00160             for(size_t i=0; i<Dimension; ++i)
00161             {
00162                 prevAccel[i] = accel[i];
00163                 accel[i] = (force[i] / ((property<0)? -property:property));
00164                 pos[i] += vel[i] * delta_t + 0.5 * prevAccel[i] * delta_t * delta_t;
00165                 vel[i] += 0.5 * (prevAccel[i] + accel[i]) * delta_t;
00166             }
00167         }
00168 
00173         void updateAndReset(const double delta_t) {
00174 
00175             for(size_t i = 0; i < Dimension; ++i) pos[i] += vel[i] * delta_t;
00176             for(size_t i = 0; i < Dimension; ++i) vel[i] += (force[i] / property) * delta_t;
00177 
00178             resetForce();
00179         }
00180 
00184         void printStates() const{
00185             std::cout « "Id: " « id « std::endl;
00186 
00187             std::cout « "Position: ";
00188             for (size_t i = 0; i < Dimension; ++i) {
00189                 std::cout « pos[i];
00190                 if (i < Dimension - 1) {
00191                     std::cout « " ";
00192                 }
00193             }
00194             std::cout « std::endl;
00195 
00196             (!type? std::cout « "Mass: " : std::cout « "Charge: ");
00197             std::cout « property « std::endl;
00198 
00199             std::cout « "Force: ";
00200             for (size_t i = 0; i < Dimension; ++i) {
00201                 std::cout « force[i];
00202                 if (i < Dimension - 1) {
00203                     std::cout « " ";
00204                 }
00205             }
00206             std::cout « std::endl;
00207 
00208             std::cout « "Velocity: ";
00209             for (size_t i = 0; i < Dimension; ++i) {
00210                 std::cout « vel[i];
00211                 if (i < Dimension - 1) {
00212                     std::cout « " ";
00213                 }
00214             }
00215             std::cout « std::endl;
00216         }
00217 
00223         void manageCollision(Particle<Dimension> &p, double dim){
00224             if(dim){
00225                 for(size_t i = 0; i < Dimension; ++i){
00226                     if (pos[i] + radius > dim || pos[i] - radius < -dim){
00227                         vel[i] = -vel[i];
00228                     }
00229                 }
00230             }
00231             else{
00232                 std::array<double, Dimension> prev_vel, new_vel;
00233 
00234                 for(size_t i = 0; i < Dimension; ++i) prev_vel[i] = vel[i];
00235                 for(size_t i = 0; i < Dimension; ++i) vel[i] = ((property -
    p.getProperty())*vel[i]+2*p.getProperty()*p.getVel()[i]) / (property + p.getProperty());
00236                 for(size_t i = 0; i < Dimension; ++i) new_vel[i] = ((p.getProperty() -
    property)*p.getVel()[i]+2*property*prev_vel[i]) / (property + p.getProperty());
00237                 p.setVel(new_vel);
00238             }
00239         }
00240 
00246         bool hitsBoundary(double dim){
00247             bool bound_touched = false;
00248             for(size_t i = 0; i < Dimension; ++i){
00249                 if(getPos()[i] + getRadius() > dim || getPos()[i] - getRadius() < -dim ){
00250                     bound_touched = true;
00251                     return bound_touched;
00252                 }
00253             }
00254 
00255             return bound_touched;
```

```
00256         }
00257
00261         ~Particle(){}
00262     private:
00263         int id;
00264         double property;
00265         std::array<double, Dimension> pos;
00266         std::array<double, Dimension> vel;
00267         std::array<double, Dimension> force;
00268         std::array<double, Dimension> accel;
00269         bool type;
00270         double radius;
00271 };
00272
00273 #endif
```

5.4 quadtreeNode.hpp

```
00001 #ifndef QUADTREE_NODE_HPP
00002 #define QUADTREE_NODE_HPP
00003
00004 #include <array>
00005 #include <memory>
00006 #include <vector>
00007
00008 #include "particle.hpp"
00009
00010 using namespace std;
00011
00012 template <size_t Dimension>
00013 class Particle;
00014
00019 template <size_t Dimension>
00020 class QuadtreeNode {
00021    public:
00031     QuadtreeNode(double x, double y, double w, int maxDepthValue = 10)
00032         : width(w),
00033           leaf(true),
00034           totalCenter(std::array<double, Dimension>()),
00035           center({x, y}),
00036           totalMass(0.0),
00037           count(0),
00038           maxDepth(maxDepthValue) {
00039     }
00043     ~QuadtreeNode() {}
00044
00050     const std::array<double, Dimension>& getCenter() const { return center; }
00051
00057     double getWidth() const { return width; }
00058
00064     bool isLeaf() const { return leaf; }
00065
00071     const std::shared_ptr<Particle<Dimension»& getParticle() const { return particle; }
00072
00078     const std::array<double, Dimension>& getTotalCenter() const { return totalCenter; }
00079
00085     double getTotalMass() const { return totalMass; }
00086
00092     int getCount() const { return count; }
00093
00099     const std::array<std::unique_ptr<QuadtreeNode<Dimension», 4>& getChildren() const { return
      children; }
00100
00109     void insertNode(std::unique_ptr<QuadtreeNode<Dimension>> subTreeRoot, int depth = 0) {
00110         updateAttributes(subTreeRoot->createApproximateParticle(), subTreeRoot->getCount());
00111
00112         if (!subTreeRoot) {
00113             cerr « "Error: the node  being inserted is null" « endl;
00114             return;
00115         }
00116
00117         if (leaf) {
00118
00119             split();
00120             leaf = false;
00121         }
00122
00123
00124         int index = getQuadrantIndex(subTreeRoot->getCenter());
00125         if (index == -1) {
00126             cerr « "Error: invalid index for the node to be inserted" « endl;
00127             return;
00128         }
```

```
00129
00130         if (!children[index]->getParticle()) {
00131             children[index] = std::move(subTreeRoot);
00132         } else {
00133             children[index]->insertNode(std::move(subTreeRoot), depth + 1);
00134         }
00135     }
00136
00140     void split() {
00141         leaf = false;
00142
00143         double halfWidth = width / 2.0;
00144         double quarterWidth = width / 4.0;
00145
00146         double x = center[0];
00147         double y = center[1];
00148
00149         if (width == 0) return;
00150
00151         children[0] = std::make_unique<QuadtreeNode<Dimension»(x - quarterWidth, y - quarterWidth,
      halfWidth);
00152         children[1] = std::make_unique<QuadtreeNode<Dimension»(x + quarterWidth, y - quarterWidth,
      halfWidth);
00153         children[2] = std::make_unique<QuadtreeNode<Dimension»(x - quarterWidth, y + quarterWidth,
      halfWidth);
00154         children[3] = std::make_unique<QuadtreeNode<Dimension»(x + quarterWidth, y + quarterWidth,
      halfWidth);
00155     }
00156
00163     int getQuadrantIndex(const std::array<double, Dimension>& pos) {
00164         bool isLeft = pos[0] < center[0];
00165         bool isTop = pos[1] < center[1];
00166
00167         if (isLeft && isTop) return 0;    // Top-left
00168         if (!isLeft && isTop) return 1;   // Top-right
00169         if (isLeft && !isTop) return 2;   // Bottom-left
00170         if (!isLeft && !isTop) return 3;  // Bottom-right
00171
00172         return -1;
00173     }
00174
00183     void insert(std::shared_ptr<Particle<Dimension>> p, int depth = 0) {
00184         updateAttributes(p);
00185
00186         if (leaf && particle == nullptr) {
00187             particle = p;
00188             return;
00189         }
00190
00191         if (leaf && particle != nullptr) {
00192             split();
00193             int index = getQuadrantIndex(particle->getPos());
00194             if (index != -1)
00195                 children[index]->insert(particle);
00196             else {
00197                 cerr « "Error: quadrant index invalid while repositioning" « endl;
00198             }
00199             particle = nullptr;
00200         }
00201
00202         int index = getQuadrantIndex(p->getPos());
00203         if (index != -1) {
00204             if (children[index] == nullptr) {
00205                 cerr « "Error: child not initialised" « endl;
00206                 return;
00207             }
00208             children[index]->insert(p, depth + 1);
00209         } else {
00210             cerr « "Error: invalid index for the new particle" « endl;
00211         }
00212     }
00213
00221     void updateAttributes(std::shared_ptr<Particle<Dimension>> newParticle, int numParticles = 1) {
00222         double oldTotalMass = totalMass;
00223         totalMass += newParticle->getProperty();  // Property of particle: mass for gravitational
      force, charge for Coulomb force.
00224         count += numParticles;
00225         for (size_t i = 0; i < Dimension; ++i) {
00226             double sumWeights = totalCenter[i] * oldTotalMass;
00227             sumWeights += newParticle->getPos()[i] * newParticle->getProperty();
00228             totalCenter[i] = sumWeights / totalMass;
00229         }
00230     }
00231
00237     void printTree(int depth = 0) const {
00238         // Print the current node's details
00239         std::cout « std::string(depth * 4, ' ') « "Node at depth " « depth « ": [" « center[0] « ", "
```

```
    « center[1]
00240                   « ", " « width « "], "
00241                   « "Mass: " « totalMass « ", Particles: " « count « ",";
00242         // Check if the node is a leaf or an internal node
00243         std::cout « (leaf ? " Leaf node\n" : " Internal node\n");
00244 
00245         if (leaf) {
00246 
00247             if (particle != nullptr) {
00248                 std::cout « std::string((depth + 1) * 4, ' ') « "Particle: " « particle->getId() « ")
"
00249                           « particle->getPos()[0] « ", " « particle->getPos()[1]
00250                           « ". mass: " « particle->getProperty()
00251                           « "\n";
00252             }
00253         } else {
00254             for (const auto& child : children) {
00255                 if (child != nullptr) {
00256                     child->printTree(depth + 1);
00257                 }
00258             }
00259         }
00260     }
00261 
00270     std::unique_ptr<Particle<Dimension» createApproximateParticle() {
00271         std::array<double, Dimension> pos;
00272         std::array<double, Dimension> vel;
00273         for (size_t i = 0; i < Dimension; ++i) {
00274             pos[i] = totalCenter[i];
00275             vel[i] = 0.0;
00276         }
00277         return std::make_unique<Particle<Dimension»(-1, totalMass, pos, vel, 1.0, false);
00278     }
00279 
00280    private:
00281     double width;
00282     bool leaf;
00283     std::shared_ptr<Particle<Dimension» particle;
00284     std::array<std::unique_ptr<QuadtreeNode<Dimension», 4> children;
00285     std::array<double, Dimension> totalCenter;
00286     std::array<double, Dimension> center;
00287     double totalMass;
00288     int count;
00289     int maxDepth;
00290 };
00291 
00292 #endif  // QUADTREE_NODE_H
```

5.5 simulationUtilities.hpp

```
00001 #include <vector>
00002 #include <fstream>
00003 #include <iostream>
00004 #include <random>
00005 #include "particle.hpp"
00006 #include "quadtreeNode.hpp"
00007 #include "force.hpp"
00008 
00041 template <size_t Dimension>
00042 std::vector<Particle<Dimension» generateRandomParticles(int N, int posBoundary = 100, int minProperty
      = 1,
00043                                                         int maxProperty = 99, int maxVel = 100, int
      minRadius = 0,
00044                                                         int maxRadius = 15, bool type = false) {
00045     int maxRetry = 15, counter = 0;
00046     bool overlapping = false;
00047     double x, y, r, property, squareDistance = 0.0;
00048     std::vector<Particle<Dimension» particles;
00049     std::array<double, Dimension> pos, vel;
00050 
00051     std::random_device rd;
00052     std::mt19937 gen(rd());
00053     std::uniform_real_distribution<double> disProperty(minProperty, maxProperty);
00054     std::uniform_real_distribution<double> disVel(-maxVel, maxVel);
00055     std::uniform_real_distribution<double> disRadius(minRadius, maxRadius);
00056     std::uniform_real_distribution<double> disPos(-posBoundary + maxRadius, posBoundary - maxRadius);
00057 
00058     while (particles.size() < N) {
00059         if (counter == maxRetry)
00060             throw std::runtime_error(
00061                 "ERROR: the dimension of the simulation area is too little, "
00062                 "please specify a bigger area or generate less particles.");
00063 
```

```
00064         overlapping = false;
00065
00066         r = disRadius(gen);
00067
00068         for (size_t i = 0; i < Dimension; ++i) pos[i] = disPos(gen);
00069
00070         for (const Particle<Dimension>& p : particles) {
00071             squareDistance = 0.0;
00072             for (size_t i = 0; i < Dimension; ++i)
00073                 squareDistance = squareDistance + ((p.getPos()[i] - pos[i]) * (p.getPos()[i] -
      pos[i]));
00074             if (sqrt(squareDistance) < p.getRadius() + r) {
00075                 counter++;
00076                 overlapping = true;
00077                 break;
00078             }
00079         }
00080
00081         if (!overlapping) {
00082             counter = 0;
00083
00084             property = disProperty(gen);
00085
00086             for (size_t i = 0; i < Dimension; ++i) vel[i] = disVel(gen);
00087
00088             Particle<Dimension> p(particles.size(), property, pos, vel, r, type);
00089
00090             particles.push_back(p);
00091         } else {
00092             counter++;
00093         }
00094     }
00095
00096     return particles;
00097 }
00098
00099
00100
00108 template <size_t Dimension>
00109 void printAllParticlesStateAndDistance(const std::vector<Particle<Dimension>>* particles) {
00110     std::cout << "Print All Particles State And Distance:\n";
00111     double squareDistance = 0.0;
00112     for (int i = 0; i < (*particles).size(); ++i) {
00113         Particle<Dimension>& p = (*particles)[i];
00114         std::cout << "--------------------------------------\n";
00115         p.printStates();
00116         squareDistance = 0.0;
00117         for (size_t i = 0; i < Dimension; ++i) squareDistance = squareDistance + (p.getPos()[i] *
      p.getPos()[i]);
00118         std::cout << "Distance from origin: " << sqrt(squareDistance) << "\n";
00119     }
00120 }
00121
00122
00132 template <size_t Dimension>
00133 std::vector<Particle<Dimension>> generateOrbitTestParticles(double size, double costantForce) {
00134     std::vector<Particle<Dimension>> particles;
00135     double orbitRadius = size / 2.0;
00136     double mass1 = 100;
00137     double mass2 = 1;
00138     double force = costantForce * (mass1 * mass2) / (orbitRadius * orbitRadius);
00139     double velocity = sqrt(force * orbitRadius / mass2);
00140     std::array<double, Dimension> pos1;
00141     std::array<double, Dimension> pos2;
00142     std::array<double, Dimension> vel1;
00143     std::array<double, Dimension> vel2;
00144
00145     for (int i = 0; i < Dimension; ++i) {
00146         pos1[i] = 0;
00147         pos2[i] = i == 0 ? orbitRadius : 0;
00148         vel1[i] = 0;
00149         vel2[i] = i == (Dimension - 1) ? velocity : 0;
00150     }
00151
00152     Particle<Dimension> p1(0, mass1, pos1, vel1, size / 100, false);
00153     Particle<Dimension> p2(1, mass2, pos2, vel2, size / 100, false);
00154     particles.push_back(p1);
00155     particles.push_back(p2);
00156
00157     return particles;
00158 }
```

5.6 treeUtilities.hpp

```
00001 #include <vector>
00002 #include <fstream>
00003 #include <iostream>
00004 #include <random>
00005 #include "particle.hpp"
00006 #include "quadtreeNode.hpp"
00007 #include "force.hpp"
00008
00018 template <size_t Dimension>
00019 std::unique_ptr<QuadtreeNode<Dimension» createQuadTree(std::vector<Particle<Dimension>>& particles,
00020                                                         double dimSimulationArea) {
00021     if (particles.empty()) {
00022         return nullptr;
00023     }
00024
00025     double minX = -1 * dimSimulationArea * 0.5;
00026     double maxX = dimSimulationArea * 0.5;
00027     double minY = minX;
00028     double maxY = maxX;
00029
00030
00031     double width = std::max(maxX - minX, maxY - minY);
00032     std::unique_ptr<QuadtreeNode<Dimension» root =
00033         std::make_unique<QuadtreeNode<Dimension»(minX + width / 2, minY + width / 2, width, 20);
00034
00035     for (auto& particle : particles) {
00036         std::shared_ptr<Particle<Dimension» p = std::make_shared<Particle<Dimension»(particle);
00037         root->insert(p);
00038     }
00039
00040     return root;
00041 }
00042
00053 template <size_t Dimension>
00054 void assignRegions(double regionWidth, double dim, double x, double y,
00055                    std::vector<std::array<double, Dimension»* centers) {
00056     if (dim == regionWidth) {
00057         centers->push_back({x - regionWidth / 2, y + regionWidth / 2});
00058         centers->push_back({x + regionWidth / 2, y + regionWidth / 2});
00059         centers->push_back({x - regionWidth / 2, y - regionWidth / 2});
00060         centers->push_back({x + regionWidth / 2, y - regionWidth / 2});
00061
00062     } else {
00063         assignRegions(regionWidth, dim / 2, x - dim / 2, y + dim / 2, centers);
00064         assignRegions(regionWidth, dim / 2, x + dim / 2, y + dim / 2, centers);
00065         assignRegions(regionWidth, dim / 2, x - dim / 2, y - dim / 2, centers);
00066         assignRegions(regionWidth, dim / 2, x + dim / 2, y - dim / 2, centers);
00067     }
00068 }
00069
00078 template <size_t Dimension>
00079 std::vector<std::array<double, Dimension» getSubRegionsCoordinates(int num_threads, double dim) {
00080     std::vector<std::array<double, Dimension» regions;
00081
00082     int rows = std::sqrt(num_threads);
00083     while (num_threads % rows != 0) {
00084         rows--;
00085     }
00086     int cols = num_threads / rows;
00087
00088     double regionWidth = dim / cols;
00089     double regionHeight = dim / rows;
00090
00091     assignRegions(regionWidth, dim / 2, 0.0, 0.0, &regions);
00092
00093     return regions;
00094 }
00095
00104 template <size_t Dimension>
00105 std::array<double, Dimension> getSubRegionsDimension(int num_threads, double dim) {
00106     std::array<double, Dimension> regionsDim;
00107
00108     int rows = std::sqrt(num_threads);
00109
00110     while (num_threads % rows != 0) {
00111         rows--;
00112     }
00113
00114     int cols = num_threads / rows;
00115
00116     double regionWidth = dim / cols;
00117     double regionHeight = dim / rows;
00118     regionsDim[0] = regionWidth;
00119     regionsDim[1] = regionHeight;
00120
```

```
00121     return regionsDim;
00122 }
00123
00124
00125
```

Index