PBPhys

P. Baillehache

November 4, 2018

Contents

Introduction

PBPhys is a C library providing structures and functions to simulate system of moving particles in any dimension.

Each particle is represented by a Shapoid, with a speed and acceleration vector, a mass and a drag coefficient. Particles can be fixed. The PBPhys is defined as GSet of particles. The system can emulate or ignore attraction between particles, downward gravity (applied to the second component of vectors, to simulate earth attraction for example), drag force on particle, elastic

collision between particles (considered as perfect spheres). The user can control the system's particle by position, speed or acceleration. The user can step the system by increment of time or until the next collision. The whole system and individual particles can be printed on a stream, saved/loaded in a file.

It uses the `PBErr`, `PBMath`, `Shapoid` and `GSet` libraries.

1 Definitions

1.1 Equation of movement

The movement of each particle is calculated as follow. Given $drag$ the drag coefficient of the particle ($drag \in \mathbb{R}^+$, 0.0 means no drag), $\overrightarrow{P}(t)$ the position of the particle at instant t, $\overrightarrow{S}(t)$ the speed of the particle at instant t and $\overrightarrow{A}(t)$ the acceleration of the particle at instant t:

$$\overrightarrow{P}(t+\delta t) = \overrightarrow{P}(t) + \overrightarrow{S}(t)\delta t + 0.5 * (\overrightarrow{A}(t) - drag * \overrightarrow{S}(t))\delta t^2 \quad (1)$$

$$\overrightarrow{S}(t+\delta t) = \overrightarrow{S}(t) + (\overrightarrow{A}(t) - drag * \overrightarrow{S}(t))\delta t \quad (2)$$

If a "downward gravity" is applied, its value is substracted to the second component of the acceleration. If "gravity" is applied each particle P_i 's acceleration is added an attraction force toward others particles equals to:

$$\overrightarrow{F}_i(t) = \sum_j \left(G.\frac{m_i.m_j}{||\overrightarrow{P}_j(t) - \overrightarrow{P}_i(t)||^2}.\frac{\overrightarrow{P_j} - \overrightarrow{P_i}}{||\overrightarrow{P}_j(t) - \overrightarrow{P}_i(t)||} \right) \quad (3)$$

where G is the gravity and m_i is the mass of the particle P_i.

1.2 Detection of collision

The detection of collision between particles is done while approximating particles to spheres.

The distance between two particles moving linearly is calculated as follow. Given $\overrightarrow{P}_0(t)$ and $\overrightarrow{S}_0(t)$ the position and speed of the first particle at time t, and $\overrightarrow{P}_1(t)$ and $\overrightarrow{S}_1(t)$ the position and speed of the second particle

at time t. The distance $D(t)$ betwen the two particles is given by:

$$D(t_0 + dt) = \sqrt{||\overrightarrow{V}(t_0)||^2 + 2(\overrightarrow{V}(t_0).\overrightarrow{W}(t_0))dt + ||\overrightarrow{W}(t_0)||^2 dt^2} \quad (4)$$

where $\overrightarrow{V}(t) = \overrightarrow{P}_1(t) - \overrightarrow{P}_0(t)$ and $\overrightarrow{W}(t) = \overrightarrow{S}_1(t) - \overrightarrow{S}_0(t)$.

One can notice that the square of the distance between particles is a polynomial function of order 2. The time to smallest distance between particles dt_n can then simply be calculated by searching the solution of $D'(t) = 0$, which gives:

$$dt_n = \frac{-b}{2a} \quad (5)$$

where a, b, c represents the coefficients of the polynomial: $D(t) = at^2 + bt + c$.

Finally, given R_0 and R_1 the radius of the spheres approximating the particle, the time dt_h to hit can be calculated as follow:

$$dt_h = \frac{-b - \sqrt{b^2 - 4a(c - (R_0 + R_1)^2)}}{2a} \quad (6)$$

1.3 Elastic collision

The speed after collision of two colliding particles is calculated as follow. Given m_0 and m_1 the masses of the particles, $\overrightarrow{P}_0$ and $\overrightarrow{P}_1$ the position of the particles, $\overrightarrow{S}_0$ and $\overrightarrow{S}_1$ the speed of the particles before collision and $\overrightarrow{S'}_0$ and $\overrightarrow{S'}_1$ the speed of the particles after collision:

$$\overrightarrow{S'}_0 = \overrightarrow{S}_0 - \frac{2m_1}{m_0 + m_1}.\frac{(\overrightarrow{S}_0 - \overrightarrow{S}_1).(\overrightarrow{P}_0 - \overrightarrow{P}_1)}{||\overrightarrow{P}_0 - \overrightarrow{P}_1||^2}(\overrightarrow{P}_0 - \overrightarrow{P}_1) \quad (7)$$

$$\overrightarrow{S'}_1 = \overrightarrow{S}_1 - \frac{2m_0}{m_0 + m_1}.\frac{(\overrightarrow{S}_1 - \overrightarrow{S}_0).(\overrightarrow{P}_1 - \overrightarrow{P}_0)}{||\overrightarrow{P}_1 - \overrightarrow{P}_0||^2}(\overrightarrow{P}_1 - \overrightarrow{P}_0) \quad (8)$$

2 Interface

```
// ============ PBPHYS.H ================

#ifndef PBPHYS_H
#define PBPHYS_H

// ================= Include =================
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "shapoid.h"

// ------------ PBPhysParticle

// ================= Define ==================

// ================= Data structure ===================

typedef struct PBPhysParticle {
  // Shapoid
  Shapoid* _shape;
  // Speed
  VecFloat* _speed;
  // User acceleration
  VecFloat* _accel;
  // System acceleration (used internally for calculation)
  VecFloat* _sysAccel;
  // Mass
  float _mass;
  // Drag per time unit
  float _drag;
  // Flag for fixed particle
  bool _fixed;
  // User data
  void* _data;
} PBPhysParticle;

// ================ Functions declaration ====================

// Create a new PBPhysParticle with dimension 'dim' and a 'shapeType'
// shapoid as shape
// Default values: _mass = 0.0, _drag = 0.0, _fixed = false
PBPhysParticle* PBPhysParticleCreate(const int dim,
  const ShapoidType shapeType);

// Free the memory used by the particle 'that'
void PBPhysParticleFree(PBPhysParticle** that);

// Return a clone of the particle 'that'
PBPhysParticle* PBPhysParticleClone(const PBPhysParticle* const that);

// Print the particle 'that' on the stream 'stream'
void PBPhysParticlePrintln(const PBPhysParticle* const that,
  FILE* const stream);

// Function which return the JSON encoding of 'that'
JSONNode* PBPhysParticleEncodeAsJSON(const PBPhysParticle* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool PBPhysParticleDecodeAsJSON(PBPhysParticle** that,
  const JSONNode* const json);

// Save the particle 'that' on the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if we could save the particle
```

```
// Return false else
// If user data is attached to the particle it must be saved by the user
bool PBPhysParticleSave(const PBPhysParticle* const that,
  FILE* const stream, const bool compact);

// Load the particle 'that' from the stream 'stream'
// Return true if we could load the particle
// Return false else
// If user data is attached to the particle it must be loaded by the user
bool PBPhysParticleLoad(PBPhysParticle** that, FILE* const stream);

// Return the dimension of the particle 'that'
#if BUILDMODE != 0
inline
#endif
int PBPhysParticleGetDim(const PBPhysParticle* const that);

// Return the shape type of the particle 'that'
#if BUILDMODE != 0
inline
#endif
ShapoidType PBPhysParticleGetShapeType(const PBPhysParticle* const that);

// Return the shape of the particle 'that'
#if BUILDMODE != 0
inline
#endif
const Shapoid* PBPhysParticleShape(const PBPhysParticle* const that);

// Return the 'iAxis'-th axis of the shape of the particle 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* PBPhysParticleAxis(const PBPhysParticle* const that,
  const int iAxis);

// Return the speed of the particle 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* PBPhysParticleSpeed(const PBPhysParticle* const that);

// Return the accel of the particle 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* PBPhysParticleAccel(const PBPhysParticle* const that);

// Return the sysAccel of the particle 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* PBPhysParticleSysAccel(const PBPhysParticle* const that);

// Return the position of the center of the particle 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* PBPhysParticleGetPos(const PBPhysParticle* const that);

// Set the speed of the particle 'that' to 'speed'
// If the particle is fixed do nothing
```

```
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetSpeed(const PBPhysParticle* const that,
  const VecFloat* const speed);

// Add to the speed of the particle 'that' the vector 'v' multiplied
// by 'c'
// If the particle is fixed do nothing
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleAddSpeed(const PBPhysParticle* const that,
  const VecFloat* const v, const float c);

// Add to the system accel of the particle 'that' the vector 'v'
// multiplied by 'c'
// If the particle is fixed do nothing
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleAddSysAccel(const PBPhysParticle* const that,
  const VecFloat* const v, const float c);

// Set the acceleration of the particle 'that' to 'accel'
// If the particle is fixed do nothing
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetAccel(PBPhysParticle* const that,
  const VecFloat* const accel);

// Reset the system acceleration of the particle 'that'
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleResetSysAccel(PBPhysParticle* const that);

// Add the gravity force 'gravity' to the system acceleration of the
// particle 'that' (substract 'gravity' to the axis y)
// If the particle is fixed do nothing
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleApplyGravity(PBPhysParticle* const that,
  const float gravity);

// Set the position of the center of the particle 'that' to 'pos'
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetPos(PBPhysParticle* const that,
  const VecFloat* const pos);

// Add to the position of the center of the particle 'that' the vector
// 'v' multiplied by 'c'
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleAddPos(PBPhysParticle* const that,
  const VecFloat* const v, const float c);
```

```
// Return true if the particle 'that' is the same is the particle 'tho'
// Return false else
// User data is not compared
#if BUILDMODE != 0
inline
#endif
bool PBPhysParticleIsSame(const PBPhysParticle* const that,
  const PBPhysParticle* const tho);

// Set the shape size of the particle 'that' to 'size'
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetSizeVec(PBPhysParticle* const that,
  const VecFloat* const size);
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetSizeScalar(PBPhysParticle* const that,
  const float size);

// Return the mass of the particle 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysParticleGetMass(const PBPhysParticle* const that);

// Set the mass of the particle 'that' to 'mass'
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleSetMass(PBPhysParticle* const that, const float mass);

// Return the drag of the particle 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysParticleGetDrag(const PBPhysParticle* const that);

// Set the drag of the particle 'that' to 'drag'
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleSetDrag(PBPhysParticle* const that, const float drag);

// Move the particle 'that' over a period of time 'dt'
// x(t+dt) = x(t) + v(t)*dt + 0.5*(a(t)-drag*v(t))*dt^2
// v(t+dt) = v(t) + (a(t)-drag*v(t))*dt
void PBPhysParticleMove(PBPhysParticle* const that, const float dt);

// Return true if the particle 'that' is fixed
// Return false else
#if BUILDMODE != 0
inline
#endif
bool PBPhysParticleIsFixed(const PBPhysParticle* const that);

// Set the fixed flag of the particle 'that' to 'fixed'
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleSetFixed(PBPhysParticle* const that,
```

```
  const bool fixed);

// Set the user data of the particle 'that' to 'data'
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleSetData(PBPhysParticle* const that, void* const data);

// Get the user data of the particle 'that'
#if BUILDMODE != 0
inline
#endif
void* PBPhysParticleData(const PBPhysParticle* const that);

// Correct the current speed of the two colliding particles 'that' and
// 'tho' under the hypothesis of elastic collision
// If at least one of the particle as a null mass, do nothing
void PBPhysParticleApplyElasticCollision(PBPhysParticle* const that,
  PBPhysParticle* const tho);

// Return the coefficients of the polynom describing the square of the
// distance between particles 'that' and 'tho'
// Return a vector such as dist^2(t)=v[0]+v[1]t+v[2]t^2
VecFloat3D PBPhysParticleGetDistPoly(const PBPhysParticle* const that,
  const PBPhysParticle* const tho);

// ------------ PBPhys

// ================= Define ==================

// units : meter, second, kilogram
#define PBPHYS_Gn 9.80665
#define PBPHYS_G 6.6740831e-11
#define PBPHYS_DELTAT 0.01

// ================= Data structure ==================

typedef struct PBPhys {
  // Dimension of space
  const int _dim;
  // Set of particles
  GSetPBPhysParticle _particles;
  // Delta time used in Step()
  float _deltaT;
  // Downward gravity
  float _downGravity;
  // Gravity between particles
  float _gravity;
  // Current time
  float _curTime;
} PBPhys;

// ================ Functions declaration ====================

// Create a new PBPhys for space dimension 'dim'
// Default values: _deltaT = PBPHYS_DELTAT, _downGravity = 0.0,
// _gravity = 0.0, _curTime = 0.0
PBPhys* PBPhysCreate(const int dim);

// Free memory used by the PBPhys 'that'
void PBPhysFree(PBPhys** that);
```

```
// Return a clone of the PBPhys 'that'
PBPhys* PBPhysClone(const PBPhys* const that);

// Print the PBPhys 'that' on the stream 'stream'
void PBPhysPrintln(const PBPhys* const that, FILE* const stream);

// Function which return the JSON encoding of 'that'
JSONNode* PBPhysEncodeAsJSON(const PBPhys* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool PBPhysDecodeAsJSON(PBPhys** that, const JSONNode* const json);

// Save the PBPhys 'that' on the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if we could save the PBPhys
// Return false else
bool PBPhysSave(const PBPhys* const that, FILE* const stream,
  const bool compact);

// Load the PBPhys 'that' from the stream 'stream'
// Return true if we could load the PBPhys
// Return false else
bool PBPhysLoad(PBPhys** that, FILE* const stream);

// Return the space dimension of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
int PBPhysGetDim(const PBPhys* const that);

// Return the set of particles of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
GSetPBPhysParticle* PBPhysParticles(const PBPhys* const that);

// Return the delta t of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysGetDeltaT(const PBPhys* const that);

// Return the current time of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysGetCurTime(const PBPhys* const that);

// Return the downward gravity of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysGetDownGravity(const PBPhys* const that);

// Return the gravity coefficient between particles of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysGetGravity(const PBPhys* const that);

// Return true if the PBPhys 'that' is the same as PBPhys 'tho'
```

```
bool PBPhysIsSame(const PBPhys* const that, const PBPhys* const tho);

// Set the delta t of the PBPhys 'that' to 'deltaT'
#if BUILDMODE != 0
inline
#endif
void PBPhysSetDeltaT(PBPhys* const that, const float deltaT);

// Set the current time of the PBPhys 'that' to 't'
#if BUILDMODE != 0
inline
#endif
void PBPhysSetCurTime(PBPhys* const that, const float t);

// Set the downward gravity of the PBPhys 'that' to 'gravity'
#if BUILDMODE != 0
inline
#endif
void PBPhysSetDownGravity(PBPhys* const that, float gravity);

// Set the gravity coefficient between particles of the PBPhys 'that'
// to 'gravity'
#if BUILDMODE != 0
inline
#endif
void PBPhysSetGravity(PBPhys* const that, float gravity);

// Step the PBPhys 'that' by that->_deltaT ignoring collision
void PBPhysNext(PBPhys* const that);

// Step the PBPhys 'that' by that->_deltaT managing collision(s)
void PBPhysStep(PBPhys* const that);

// Step the PBPhys 'that' by that->_deltaT or until a collision occured
// If no collision occured return NULL
// If a collision occured one can check the collision time with the
// current time that->_curTime, and the returned GSet contains the
// particles wich have collided
GSetPBPhysParticle* PBPhysStepToCollision(PBPhys* const that);

// Return the 'iParticle'-th particle of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
PBPhysParticle* PBPhysPart(const PBPhys* const that,
  const int iParticle);

// Get the number of particles of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
int PBPhysGetNbParticle(const PBPhys* const that);

// Add 'nb' particles of shape 'shape' into the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
void PBPhysAddParticles(PBPhys* const that, const int nb,
  const ShapoidType shape);

// ================= Polymorphism ==================
```

```
#define PBPhysParticleSetAccel(Particle, Accel) _Generic(Accel, \
  VecFloat*: _PBPhysParticleSetAccel, \
  VecFloat2D*: _PBPhysParticleSetAccel, \
  VecFloat3D*: _PBPhysParticleSetAccel, \
  const VecFloat*: _PBPhysParticleSetAccel, \
  const VecFloat2D*: _PBPhysParticleSetAccel, \
  const VecFloat3D*: _PBPhysParticleSetAccel, \
  default: PBErrInvalidPolymorphism)(Particle, \
    (const VecFloat* const)(Accel))

#define PBPhysParticleSetSpeed(Particle, Speed) _Generic(Speed, \
  VecFloat*: _PBPhysParticleSetSpeed, \
  VecFloat2D*: _PBPhysParticleSetSpeed, \
  VecFloat3D*: _PBPhysParticleSetSpeed, \
  const VecFloat*: _PBPhysParticleSetSpeed, \
  const VecFloat2D*: _PBPhysParticleSetSpeed, \
  const VecFloat3D*: _PBPhysParticleSetSpeed, \
  default: PBErrInvalidPolymorphism)(Particle, \
    (const VecFloat* const)(Speed))

#define PBPhysParticleAddSpeed(Particle, Vec, Coeff) _Generic(Vec, \
  VecFloat*: _PBPhysParticleAddSpeed, \
  VecFloat2D*: _PBPhysParticleAddSpeed, \
  VecFloat3D*: _PBPhysParticleAddSpeed, \
  const VecFloat*: _PBPhysParticleAddSpeed, \
  const VecFloat2D*: _PBPhysParticleAddSpeed, \
  const VecFloat3D*: _PBPhysParticleAddSpeed, \
  default: PBErrInvalidPolymorphism)(Particle, \
    (const VecFloat* const)(Vec), Coeff)

#define PBPhysParticleAddSysAccel(Particle, Vec, Coeff) _Generic(Vec, \
  VecFloat*: _PBPhysParticleAddSysAccel, \
  VecFloat2D*: _PBPhysParticleAddSysAccel, \
  VecFloat3D*: _PBPhysParticleAddSysAccel, \
  const VecFloat*: _PBPhysParticleAddSysAccel, \
  const VecFloat2D*: _PBPhysParticleAddSysAccel, \
  const VecFloat3D*: _PBPhysParticleAddSysAccel, \
  default: PBErrInvalidPolymorphism)(Particle, \
    (const VecFloat* const)(Vec), Coeff)

#define PBPhysParticleSetPos(Particle, Pos) _Generic(Pos, \
  VecFloat*: _PBPhysParticleSetPos, \
  VecFloat2D*: _PBPhysParticleSetPos, \
  VecFloat3D*: _PBPhysParticleSetPos, \
  const VecFloat*: _PBPhysParticleSetPos, \
  const VecFloat2D*: _PBPhysParticleSetPos, \
  const VecFloat3D*: _PBPhysParticleSetPos, \
  default: PBErrInvalidPolymorphism)(Particle, \
    (const VecFloat* const)(Pos))

#define PBPhysParticleAddPos(Particle, Vec, Coeff) _Generic(Vec, \
  VecFloat*: _PBPhysParticleAddPos, \
  VecFloat2D*: _PBPhysParticlevPos, \
  VecFloat3D*: _PBPhysParticleAddPos, \
  const VecFloat*: _PBPhysParticleAddPos, \
  const VecFloat2D*: _PBPhysParticlevPos, \
  const VecFloat3D*: _PBPhysParticleAddPos, \
  default: PBErrInvalidPolymorphism)(Particle, \
    (const VecFloat* const)(Vec), Coeff)

#define PBPhysParticleSetSize(Particle, Size) _Generic(Size, \
  VecFloat*: _PBPhysParticleSetSizeVec, \
```

```
  const VecFloat*: _PBPhysParticleSetSizeVec, \
  float: _PBPhysParticleSetSizeScalar, \
  default: PBErrInvalidPolymorphism)(Particle, Size)

// ================ Inliner ====================

#if BUILDMODE != 0
#include "pbphys-inline.c"
#endif


#endif
```

3 Code

3.1 pbphys.c

```
// ============ PBPHYS.C ================

// ================= Include =================

#include "pbphys.h"
#if BUILDMODE == 0
#include "pbphys-inline.c"
#endif

// ------------ PBPhysParticle

// ================ Functions declaration ====================

// Return the displacement of the particle from current position to
// the position after dt
VecFloat* PBPhysParticleGetNextDisplacement(
  const PBPhysParticle* const that, const float dt);

// Return the coefficients of the polynom describing the square of the
// distance between two lines passing through 'posA' and 'posB' and
// colinear to 'dirA' and 'dirB' respectively
// Return a vector such as dist^2(t)=v[0]+v[1]t+v[2]t^2
VecFloat3D PBPhysGetDistPoly(const VecFloat* const posA,
  const VecFloat* const dirA, const VecFloat* const posB,
  const VecFloat* const dirB);

// ================ Functions implementation ====================

// Create a new PBPhysParticle with dimension 'dim' and a 'shapeType'
// shapoid as shape
// Default values: _mass = 0.0, _drag = 0.0, _fixed = false
PBPhysParticle* PBPhysParticleCreate(const int dim,
  const ShapoidType shapeType) {
#if BUILDMODE == 0
  if (dim <= 0) {
    PBPhysErr->_type = PBErrTypeInvalidArg;
    sprintf(PBPhysErr->_msg, "'dim' is invalid (0<%d)", dim);
    PBErrCatch(PBPhysErr);
  }
#endif
  // Allocate memory
```

```
  PBPhysParticle *that = PBErrMalloc(PBPhysErr, sizeof(PBPhysParticle));
  // Set properties
  that->_shape = ShapoidCreate(dim, shapeType);
  that->_speed = VecFloatCreate(dim);
  that->_accel = VecFloatCreate(dim);
  that->_sysAccel = VecFloatCreate(dim);
  that->_mass = 0.0;
  that->_drag = 0.0;
  that->_fixed = false;
  that->_data = NULL;
  // Return the new PBPhysParticle
  return that;
}

// Free the memory used by the particle 'that'
void PBPhysParticleFree(PBPhysParticle** that) {
  // Check arguments
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  ShapoidFree(&((*that)->_shape));
  VecFree(&((*that)->_speed));
  VecFree(&((*that)->_accel));
  VecFree(&((*that)->_sysAccel));
  free(*that);
  *that = NULL;
}

// Return a clone of the particle 'that'
PBPhysParticle* PBPhysParticleClone(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Declare the clone
  PBPhysParticle* clone = PBPhysParticleCreate(
    PBPhysParticleGetDim(that), PBPhysParticleGetShapeType(that));
  // Copy properties
  PBPhysParticleSetSpeed(clone, PBPhysParticleSpeed(that));
  PBPhysParticleSetAccel(clone, PBPhysParticleAccel(that));
  PBPhysParticleSetMass(clone, PBPhysParticleGetMass(that));
  PBPhysParticleSetFixed(clone, PBPhysParticleIsFixed(that));
  PBPhysParticleSetDrag(clone,
    PBPhysParticleGetDrag(that));
  VecFloat* center = PBPhysParticleGetPos(that);
  PBPhysParticleSetPos(clone, center);
  VecFree(&center);
  // Return the clone
  return clone;
}

// Print the particle 'that' on the stream 'stream'
void PBPhysParticlePrintln(const PBPhysParticle* const that,
  FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
```

```
    PBErrCatch(PBPhysErr);
  }
  if (stream == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'stream' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  ShapoidPrintln(PBPhysParticleShape(that), stream);
  fprintf(stream, "speed: ");
  VecPrint(PBPhysParticleSpeed(that), stream);
  fprintf(stream, "\n");
  fprintf(stream, "accel: ");
  VecPrint(PBPhysParticleAccel(that), stream);
  fprintf(stream, "\n");
  fprintf(stream, "mass: %.3f\n", PBPhysParticleGetMass(that));
  fprintf(stream, "drag: %.3f\n",
    PBPhysParticleGetDrag(that));
  if (PBPhysParticleIsFixed(that))
    fprintf(stream, "fixed\n");
  else
    fprintf(stream, "unfixed\n");
}

// Function which return the JSON encoding of 'that'
JSONNode* PBPhysParticleEncodeAsJSON(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the dim
  sprintf(val, "%d", PBPhysParticleGetDim(that));
  JSONAddProp(json, "_dim", val);
  // Encode the type
  sprintf(val, "%d", PBPhysParticleGetShapeType(that));
  JSONAddProp(json, "_type", val);
  // Encode the shape
  JSONAddProp(json, "_shape",
    ShapoidEncodeAsJSON(PBPhysParticleShape(that)));
  // Encode the speed
  JSONAddProp(json, "_speed",
    VecEncodeAsJSON(PBPhysParticleSpeed(that)));
  // Encode the acceleration
  JSONAddProp(json, "_accel",
    VecEncodeAsJSON(PBPhysParticleAccel(that)));
  // Encode the mass
  sprintf(val, "%f", that->_mass);
  JSONAddProp(json, "_mass", val);
  // Encode the drag
  sprintf(val, "%f", that->_drag);
  JSONAddProp(json, "_drag", val);
  // Encode the fixed
  sprintf(val, "%d", that->_fixed);
  JSONAddProp(json, "_fixed", val);
  // Return the created JSON
```

```
  return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool PBPhysParticleDecodeAsJSON(PBPhysParticle** that,
  const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (json == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'json' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // If 'that' is already allocated
  if (*that != NULL)
    // Free memory
    PBPhysParticleFree(that);
  // Get the dim from the JSON
  JSONNode* prop = JSONProperty(json, "_dim");
  if (prop == NULL) {
    return false;
  }
  int dim = atoi(JSONLabel(JSONValue(prop, 0)));
  // Get the type from the JSON
  prop = JSONProperty(json, "_type");
  if (prop == NULL) {
    return false;
  }
  int type = atoi(JSONLabel(JSONValue(prop, 0)));
  // If the data is invalid
  if (dim <= 0)
    return false;
  // Allocate memory
  *that = PBPhysParticleCreate(dim, type);
  // Decode the shape
  prop = JSONProperty(json, "_shape");
  if (prop == NULL) {
    return false;
  }
  if (!ShapoidDecodeAsJSON(&((*that)->_shape), prop)) {
    return false;
  }
  // Decode the speed
  prop = JSONProperty(json, "_speed");
  if (prop == NULL) {
    return false;
  }
  if (!VecDecodeAsJSON(&((*that)->_speed), prop)) {
    return false;
  }
  // Decode the accel
  prop = JSONProperty(json, "_accel");
  if (prop == NULL) {
    return false;
  }
  if (!VecDecodeAsJSON(&((*that)->_accel), prop)) {
    return false;
```

```
  }
  // Get the mass from the JSON
  prop = JSONProperty(json, "_mass");
  if (prop == NULL) {
    return false;
  }
  (*that)->_mass = atof(JSONLabel(JSONValue(prop, 0)));
  // Get the drag from the JSON
  prop = JSONProperty(json, "_drag");
  if (prop == NULL) {
    return false;
  }
  (*that)->_drag = atof(JSONLabel(JSONValue(prop, 0)));
  // Get the fixed from the JSON
  prop = JSONProperty(json, "_fixed");
  if (prop == NULL) {
    return false;
  }
  (*that)->_fixed = atoi(JSONLabel(JSONValue(prop, 0)));
  // Return the success code
  return true;
}

// Save the particle 'that' on the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if we could save the particle
// Return false else
// If user data is attached to the particle it must be saved by the user
bool PBPhysParticleSave(const PBPhysParticle* const that,
  FILE* const stream, const bool compact) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (stream == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'stream' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Get the JSON encoding
  JSONNode* json = PBPhysParticleEncodeAsJSON(that);
  // Save the JSON
  if (!JSONSave(json, stream, compact)) {
    return false;
  }
  // Free memory
  JSONFree(&json);
  // Return success code
  return true;
}

// Load the particle 'that' from the stream 'stream'
// Return true if we could load the particle
// Return false else
// If user data is attached to the particle it must be loaded by the
// user
bool PBPhysParticleLoad(PBPhysParticle** that, FILE* const stream) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (stream == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'stream' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (!JSONLoad(json, stream)) {
    return false;
  }
  // Decode the data from the JSON
  if (!PBPhysParticleDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return the success code
  return true;
}

// Move the particle 'that' over a period of time 'dt'
// x(t+dt) = x(t) + v(t)*dt + 0.5*(a(t)-drag*v(t))*dt^2
// v(t+dt) = v(t) + (a(t)-drag*v(t))*dt
void PBPhysParticleMove(PBPhysParticle* const that, const float dt) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  if (!PBPhysParticleIsFixed(that)) {
    // Get the displacement
    VecFloat* disp = PBPhysParticleGetNextDisplacement(that, dt);
    // Update the position
    VecFloat* v = VecGetOp(
      ShapoidPos(PBPhysParticleShape(that)), 1.0, disp, 1.0);
    PBPhysParticleSetPos(that, v);
    VecFree(&v);
    VecFree(&disp);
    // Update the speed
    PBPhysParticleAddSpeed(that,
      PBPhysParticleSpeed(that), -dt * PBPhysParticleGetDrag(that));
    PBPhysParticleAddSpeed(that, PBPhysParticleAccel(that), dt);
    PBPhysParticleAddSpeed(that, PBPhysParticleSysAccel(that), dt);
  }
}

// Return the displacement of the particle from current position to
// the position after dt
VecFloat* PBPhysParticleGetNextDisplacement(
  const PBPhysParticle* const that, const float dt) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  VecFloat* v = VecGetOp(PBPhysParticleAccel(that), 1.0,
    PBPhysParticleSpeed(that), -1.0 * PBPhysParticleGetDrag(that));
  VecOp(v, 1.0, PBPhysParticleSysAccel(that), 1.0);
  VecOp(v, 0.5 * fsquare(dt), PBPhysParticleSpeed(that), dt);
  return v;
}

// Correct the current speed of the two colliding particles 'that' and
// 'tho' under the hypothesis of elastic collision
// Particles' mass must not be null
void PBPhysParticleApplyElasticCollision(PBPhysParticle* const that,
  PBPhysParticle* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (tho == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'tho' is null");
    PBErrCatch(PBPhysErr);
  }
  if (fabs(PBPhysParticleGetMass(that)) < PBMATH_EPSILON) {
    PBPhysErr->_type = PBErrTypeInvalidArg;
    sprintf(PBPhysErr->_msg, "mass of 'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (fabs(PBPhysParticleGetMass(tho)) < PBMATH_EPSILON) {
    PBPhysErr->_type = PBErrTypeInvalidArg;
    sprintf(PBPhysErr->_msg, "mass of 'tho' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Get the center of particles
  VecFloat* posA = PBPhysParticleGetPos(that);
  VecFloat* posB = PBPhysParticleGetPos(tho);
  // Get the difference in pos
  VecFloat* v = VecGetOp(posA, 1.0, posB, -1.0);
  // Get the difference in speed
  VecFloat* w = VecGetOp(PBPhysParticleSpeed(that), 1.0,
    PBPhysParticleSpeed(tho), -1.0);
  // Get the prod of differences
  float prod = VecDotProd(v, w);
  // Get the norm of difference in pos
  float norm = VecNorm(v);
  // Calculate a temporary value for following calculation
  float c = 2.0 * prod /
    ((PBPhysParticleGetMass(that) + PBPhysParticleGetMass(tho)) *
    fsquare(norm));
  // Update the speed of 'that' if it's not fixed
  if (!PBPhysParticleIsFixed(that))
    PBPhysParticleAddSpeed(that, v,
      -1.0 * c * PBPhysParticleGetMass(tho));
  // Update the speed of 'tho' if it's not fixed
  if (!PBPhysParticleIsFixed(tho))
    PBPhysParticleAddSpeed(tho, v,
      c * PBPhysParticleGetMass(that));
```

```
  // Free memory
  VecFree(&posA);
  VecFree(&posB);
  VecFree(&v);
  VecFree(&w);
}

// Return the coefficients of the polynom describing the square of the
// distance between particles 'that' and 'tho'
// Return a vector such as dist^2(t)=v[0]+v[1]t+v[2]t^2
VecFloat3D PBPhysParticleGetDistPoly(const PBPhysParticle* const that,
  const PBPhysParticle* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (tho == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'tho' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  VecFloat* posA = PBPhysParticleGetPos(that);
  VecFloat* posB = PBPhysParticleGetPos(tho);
  VecFloat3D ret = PBPhysGetDistPoly(posA, PBPhysParticleSpeed(that),
    posB, PBPhysParticleSpeed(tho));
  VecFree(&posA);
  VecFree(&posB);
  return ret;
}

// ------------ PBPhys

// ================ Functions declaration ====================

// Calculate the system acceleration of the particle 'part' in the
// PBPhys 'that'
void PBPhysUpdateSysAccel(const PBPhys* const that,
  PBPhysParticle* const part);

// Return the time to collision between two particles of radius 'rA'
// and 'rB' and the polynom of the square distance between particles
// over time 'distPoly'
float PBPhysGetTimeToHit(float rA, float rB, VecFloat3D* distPoly);

// ================ Functions implementation ====================

// Create a new PBPhys for space dimension 'dim'
// Default values: _deltaT = 0.01, _downGravity = 0.0, _gravity = 0.0,
// _curTime = 0.0
PBPhys* PBPhysCreate(const int dim) {
#if BUILDMODE == 0
  if (dim <= 0) {
    PBPhysErr->_type = PBErrTypeInvalidArg;
    sprintf(PBPhysErr->_msg, "'dim' is invalid (0<%d)", dim);
    PBErrCatch(PBPhysErr);
  }
#endif
  // Allocate memory
  PBPhys* that = PBErrMalloc(PBPhysErr, sizeof(PBPhys));
```

```
  // Set properties
  *(int*)&(that->_dim) = dim;
  that->_particles = GSetPBPhysParticleCreateStatic();
  that->_deltaT = PBPHYS_DELTAT;
  that->_downGravity = 0.0;
  that->_gravity = false;
  that->_curTime = 0.0;
  // Return the new PBPhys
  return that;
}

// Free memory used by the PBPhys 'that'
void PBPhysFree(PBPhys** that) {
  // Check argument
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  while (PBPhysGetNbParticle(*that) > 0) {
    PBPhysParticle* particle = GSetPop(PBPhysParticles(*that));
    PBPhysParticleFree(&particle);
  }
  free(*that);
  *that = NULL;
}

// Return a clone of the PBPhys 'that'
PBPhys* PBPhysClone(const PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Declare the clone
  PBPhys* clone = PBPhysCreate(PBPhysGetDim(that));
  // Copy the properties
  PBPhysSetGravity(clone, PBPhysGetGravity(that));
  PBPhysSetCurTime(clone, PBPhysGetCurTime(that));
  PBPhysSetDeltaT(clone, PBPhysGetDeltaT(that));
  PBPhysSetDownGravity(clone, PBPhysGetDownGravity(that));
  // Copy the particles
  if (PBPhysGetNbParticle(that) > 0) {
    GSetIterForward iter =
      GSetIterForwardCreateStatic(PBPhysParticles(that));
    do {
      PBPhysParticle* part = GSetIterGet(&iter);
      PBPhysParticle* clonePart = PBPhysParticleClone(part);
      GSetAppend(PBPhysParticles(clone), clonePart);
    } while (GSetIterStep(&iter));
  }
  // Return the clone
  return clone;
}

// Print the PBPhys 'that' on the stream 'stream'
void PBPhysPrintln(const PBPhys* const that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
```

```
    PBErrCatch(PBPhysErr);
  }
  if (stream == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'stream' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  fprintf(stream, "dimension: %d\n", PBPhysGetDim(that));
  fprintf(stream, "t: %f\n", PBPhysGetCurTime(that));
  fprintf(stream, "dt: %f\n", PBPhysGetDeltaT(that));
  fprintf(stream, "down gravity: %f\n", PBPhysGetDownGravity(that));
  fprintf(stream, "gravity: %f\n", PBPhysGetGravity(that));
  fprintf(stream, "nb particles: %d\n", PBPhysGetNbParticle(that));
  if (PBPhysGetNbParticle(that) > 0) {
    GSetIterForward iter =
      GSetIterForwardCreateStatic(PBPhysParticles(that));
    int iPart = 0;
    do {
      fprintf(stream, "particle #%d:\n", iPart);
      PBPhysParticle* part = GSetIterGet(&iter);
      PBPhysParticlePrintln(part, stream);
      ++iPart;
    } while (GSetIterStep(&iter));
  }
}

// Function which return the JSON encoding of 'that'
JSONNode* PBPhysEncodeAsJSON(const PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the dimension
  sprintf(val, "%d", that->_dim);
  JSONAddProp(json, "_dim", val);
  // Encode the curTime
  sprintf(val, "%f", that->_curTime);
  JSONAddProp(json, "_curTime", val);
  // Encode the deltat
  sprintf(val, "%f", that->_deltaT);
  JSONAddProp(json, "_deltaT", val);
  // Encode the downGravity
  sprintf(val, "%f", that->_downGravity);
  JSONAddProp(json, "_downGravity", val);
  // Encode the gravity
  sprintf(val, "%f", that->_gravity);
  JSONAddProp(json, "_gravity", val);
  // Encode the nbParticle
  sprintf(val, "%d", PBPhysGetNbParticle(that));
  JSONAddProp(json, "_nbParticle", val);
  // Encode the particles
  // Declare an array of structures converted to string
  JSONArrayStruct setPart = JSONArrayStructCreateStatic();
  if (PBPhysGetNbParticle(that) > 0) {
```

```
    GSetIterForward iter =
      GSetIterForwardCreateStatic(PBPhysParticles(that));
    do {
      PBPhysParticle* part = GSetIterGet(&iter);
      JSONArrayStructAdd(&setPart,
        PBPhysParticleEncodeAsJSON(part));

    } while (GSetIterStep(&iter));
  }
  // Add a key with the array of structures
  JSONAddProp(json, "_particles", &setPart);
  // Free memory
  JSONArrayStructFlush(&setPart);
  // Return the created JSON
  return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool PBPhysDecodeAsJSON(PBPhys** that, const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (json == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'json' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // If 'that' is already allocated
  if (*that != NULL)
    // Free memory
    PBPhysFree(that);
  // Decode the dimension
  JSONNode* prop = JSONProperty(json, "_dim");
  if (prop == NULL) {
    return false;
  }
  int dim = atoi(JSONLabel(JSONValue(prop, 0)));
  // If data is invalid
  if (dim <= 0)
    return false;
  // Allocate memory
  *that = PBPhysCreate(dim);
  // Decode the curTime
  prop = JSONProperty(json, "_curTime");
  if (prop == NULL) {
    return false;
  }
  (*that)->_curTime = atof(JSONLabel(JSONValue(prop, 0)));
  // Decode the deltaT
  prop = JSONProperty(json, "_deltaT");
  if (prop == NULL) {
    return false;
  }
  (*that)->_deltaT = atof(JSONLabel(JSONValue(prop, 0)));
  // Decode the downGravity
  prop = JSONProperty(json, "_downGravity");
  if (prop == NULL) {
    return false;
```

```
  }
  (*that)->_downGravity = atof(JSONLabel(JSONValue(prop, 0)));
  // Decode the gravity
  prop = JSONProperty(json, "_gravity");
  if (prop == NULL) {
    return false;
  }
  (*that)->_gravity = atof(JSONLabel(JSONValue(prop, 0)));
  // Decode the nbParticle
  prop = JSONProperty(json, "_nbParticle");
  if (prop == NULL) {
    return false;
  }
  int nbParticle = atoi(JSONLabel(JSONValue(prop, 0)));
  // Decode the particle
  prop = JSONProperty(json, "_particles");
  if (prop == NULL) {
    return false;
  }
  if (JSONGetNbValue(prop) != nbParticle) {
    return false;
  }
  for (int iPart = 0; iPart < nbParticle; ++iPart) {
    JSONNode* part = JSONValue(prop, iPart);
    PBPhysParticle* p = NULL;
    if (!PBPhysParticleDecodeAsJSON(&p, part))
      return false;
    GSetAppend(PBPhysParticles(*that), p);
  }
  // Return the success code
  return true;
}

// Save the PBPhys 'that' on the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if we could save the PBPhys
// Return false else
bool PBPhysSave(const PBPhys* const that, FILE* const stream,
  const bool compact) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (stream == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'stream' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Get the JSON encoding
  JSONNode* json = PBPhysEncodeAsJSON(that);
  // Save the JSON
  if (!JSONSave(json, stream, compact)) {
    return false;
  }
  // Free memory
  JSONFree(&json);
  // Return success code
  return true;
```

```
}

// Load the PBPhys 'that' from the stream 'stream'
// Return true if we could load the PBPhys
// Return false else
bool PBPhysLoad(PBPhys** that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (stream == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'stream' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (!JSONLoad(json, stream)) {
    return false;
  }
  // Decode the data from the JSON
  if (!PBPhysDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return the success code
  return true;
}

// Step the PBPhys 'that' by that->_deltaT ignoring collision
void PBPhysNext(PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // If there is particle
  if (PBPhysGetNbParticle(that) > 0) {
    // Loop on particles
    GSetIterForward iter =
      GSetIterForwardCreateStatic(PBPhysParticles(that));
    do {
      PBPhysParticle* part = GSetIterGet(&iter);
      // Calculate the system acceleration of the particle
      PBPhysUpdateSysAccel(that, part);
      // Move the particle
      PBPhysParticleMove(part, PBPhysGetDeltaT(that));
    } while (GSetIterStep(&iter));
  }
  // Update current time
  PBPhysSetCurTime(that,
    PBPhysGetCurTime(that) + PBPhysGetDeltaT(that));
}

// Calculate the system acceleration of the particle 'part' in the
```

```
// PBPhys 'that'
void PBPhysUpdateSysAccel(const PBPhys* const that,
  PBPhysParticle* particle) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (particle == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'particle' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Reset the system acceleration
  PBPhysParticleResetSysAccel(particle);
  // If the particle is fixed there is nothing to do
  if (PBPhysParticleIsFixed(particle))
    return;
  // If the down gravity is active
  if (fabs(PBPhysGetDownGravity(that)) > PBMATH_EPSILON) {
    // Substract the down gravity to the y axis of the system
    // acceleration
    PBPhysParticleApplyGravity(particle, PBPhysGetDownGravity(that));
  }
  // If the gravity is active
  if (fabs(PBPhysGetGravity(that)) > PBMATH_EPSILON) {
    // Get the center pos of the particle
    VecFloat* centerParticle = PBPhysParticleGetPos(particle);
    // Loop on particles
    GSetIterForward iter =
      GSetIterForwardCreateStatic(PBPhysParticles(that));
    do {
      PBPhysParticle* part = GSetIterGet(&iter);
      // If the current particle is not the particle in argument
      if (particle != part) {
        // Get the distance between the two particles
        VecFloat* centerPart = PBPhysParticleGetPos(part);
        float dist = VecDist(centerParticle, centerPart);
        if (fabs(dist) > PBMATH_EPSILON) {
          // Get the magnitude of the attraction
          float mag = PBPhysGetGravity(that) *
            PBPhysParticleGetMass(particle) *
            PBPhysParticleGetMass(part) / fsquare(dist);
          // Apply the attraction toward the other particle
          VecOp(centerPart, 1.0, centerParticle, -1.0);
          VecNormalise(centerPart);
          PBPhysParticleAddSysAccel(particle, centerPart, mag);
        }
        // Free memory
        VecFree(&centerPart);
      }
    } while (GSetIterStep(&iter));
    // Free memory
    VecFree(&centerParticle);
  }
}

// Step the PBPhys 'that' by that->_deltaT managing collision(s)
void PBPhysStep(PBPhys* const that) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Declare a variable to memorize the goal time
  float goalT = PBPhysGetCurTime(that) + PBPhysGetDeltaT(that);
  // Declare a variable to memorize the initial deltat
  float origDeltaT = PBPhysGetDeltaT(that);
  // Loop until we reach the goal time
  while (PBPhysGetCurTime(that) < goalT) {
    // Step until next collision
    GSetPBPhysParticle* set = PBPhysStepToCollision(that);
    // If there has been collision
    if (set != NULL) {
      // Manage the collision
      PBPhysParticleApplyElasticCollision(
        GSetGet(set, 0), GSetGet(set, 1));
      // Correct the deltat to reach the initial goal time
      PBPhysSetDeltaT(that, goalT - PBPhysGetCurTime(that));
      // Free the set
      GSetFree(&set);
    }
  }
  // Reset the initial deltat
  PBPhysSetDeltaT(that, origDeltaT);
}

// Step the PBPhys 'that' for that->_deltaT or until a collision occured
// If no collision occured return NULL
// If a collision occured one can check the collision time with the
// current time that->_curTime, and the returned GSet contains the
// particles wich have collided
GSetPBPhysParticle* PBPhysStepToCollision(PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Declare a variable to memorize the deltat until next collision
  float deltat = PBPhysGetDeltaT(that);
  // Declare a set to memorize the colliding particle
  GSetPBPhysParticle* setCollision = GSetPBPhysParticleCreate();
  // If there is particle
  if (PBPhysGetNbParticle(that) > 0) {
    // Loop on particles
    GSetIterForward iter =
      GSetIterForwardCreateStatic(PBPhysParticles(that));
    do {
      PBPhysParticle* part = GSetIterGet(&iter);
      // Calculate the system acceleration of the particle
      PBPhysUpdateSysAccel(that, part);
    } while (GSetIterStep(&iter));
    // If there is at least two particles
    if (PBPhysGetNbParticle(that) > 1) {
      // Declare a variabe to memorize the inverse of deltat
      float invDeltaT = 1.0 / PBPhysGetDeltaT(that);
      // Loop on particles once again from the beginning
      GSetIterReset(&iter);
```

```
do {
  PBPhysParticle* part = GSetIterGet(&iter);
  // Get the pos of the center of the particle
  VecFloat* posPart = PBPhysParticleGetPos(part);
  // Get the displacement vector for the current particle
  VecFloat* vPart = PBPhysParticleGetNextDisplacement(part,
    PBPhysGetDeltaT(that));
  // Scale to have the displacement per time unit
  VecScale(vPart, invDeltaT);
  // Get the bounding radius of the particle
  float radPart =
    ShapoidGetBoundingRadius(PBPhysParticleShape(part));
  // Create a new iterator to loop through following particles
  GSetIterForward iterPair = iter;
  GSetIterStep(&iterPair);
  do {
    PBPhysParticle* pair = GSetIterGet(&iterPair);
    // Get the pos of the center of the particle
    VecFloat* posPair = PBPhysParticleGetPos(pair);
    // Get the displacement vector for the current pair
    VecFloat* vPair = PBPhysParticleGetNextDisplacement(pair,
      PBPhysGetDeltaT(that));
    // Scale to have the displacement per time unit
    VecScale(vPair, invDeltaT);
    // Get the bounding radius of the pair
    float radPair =
      ShapoidGetBoundingRadius(PBPhysParticleShape(pair));
    // Check the pair trajectory to determine at what time they
    // are at the closest and what is this closest distance
    VecFloat3D distPoly = PBPhysGetDistPoly(posPart, vPart,
      posPair, vPair);
    float tNearest = deltat;
    if (fabs(VecGet(&distPoly, 2)) > PBMATH_EPSILON)
      tNearest = -0.5 * VecGet(&distPoly, 1) /
        VecGet(&distPoly, 2);
    float distNearest = sqrt(VecGet(&distPoly, 0) +
      tNearest * VecGet(&distPoly, 1) +
      fsquare(tNearest) * VecGet(&distPoly, 2));
    // If there is an impact in future
    if (tNearest > 0.0 && distNearest < radPart + radPair) {
      // Get the exact time at which particles hit
      float tHit =
        PBPhysGetTimeToHit(radPart, radPair, &distPoly);
      // If the time at hit is sooner than current delta
      if (tHit < deltat) {
        // Remove the eventual previous colliding particles
        GSetFlush(setCollision);
        // Add the colliding particles
        GSetAppend(setCollision, part);
        GSetAppend(setCollision, pair);
        // Update the time at hit
        deltat = tHit;
      }
    }
    // Free memory
    VecFree(&vPair);
    VecFree(&posPair);
  } while (GSetIterStep(&iterPair));
  // Free memory
  VecFree(&posPart);
  VecFree(&vPart);
} while (GSetIterStep(&iter) && !GSetIterIsLast(&iter));
```

```
    }
    // Move the particles
    GSetIterReset(&iter);
    do {
      PBPhysParticle* part = GSetIterGet(&iter);
      // Move the particle
      PBPhysParticleMove(part, deltat);
    } while (GSetIterStep(&iter));
  }
  // Update current time
  PBPhysSetCurTime(that, PBPhysGetCurTime(that) + deltat);
  // If the set of collision is empty free it
  if (GSetNbElem(setCollision) == 0)
    GSetFree(&setCollision);
  // Return the set of colliding particles
  return setCollision;
}

// Return the time to collision between two particles of radius 'rA'
// and 'rB' and the polynom of the square distance between particles
// over time 'distPoly'
float PBPhysGetTimeToHit(float rA, float rB, VecFloat3D* distPoly) {
  float dist = fsquare(rA + rB);
  float tHit = (-1.0 * VecGet(distPoly, 1) -
    sqrt(fsquare(VecGet(distPoly, 1)) - 4.0 * VecGet(distPoly, 2) *
    (VecGet(distPoly, 0) - dist))) / (2.0 * VecGet(distPoly, 2));
  return tHit;
}

// Return the coefficients of the polynom describing the square of the
// distance between two lines passing through 'posA' and 'posB' and
// colinear to 'dirA' and 'dirB' respectively
// Return a vector such as dist^2(t)=v[0]+v[1]t+v[2]t^2
VecFloat3D PBPhysGetDistPoly(const VecFloat* const posA,
  const VecFloat* const dirA, const VecFloat* const posB,
  const VecFloat* const dirB) {
#if BUILDMODE == 0
  if (posA == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'posA' is null");
    PBErrCatch(PBPhysErr);
  }
  if (posB == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'posB' is null");
    PBErrCatch(PBPhysErr);
  }
  if (dirA == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'dirA' is null");
    PBErrCatch(PBPhysErr);
  }
  if (dirB == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'dirB' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  // Declare the vector result
  VecFloat3D res = VecFloatCreateStatic3D();
  // Loop on dimensions
  for (long iDim = VecGetDim(posA); iDim--;) {
```

```
    VecSetAdd(&res, 0,
      fsquare(VecGet(posA, iDim) - VecGet(posB, iDim)));
    VecSetAdd(&res, 1,
      (VecGet(posA, iDim) - VecGet(posB, iDim)) *
      (VecGet(dirA, iDim) - VecGet(dirB, iDim)));
    VecSetAdd(&res, 2,
      fsquare(VecGet(dirA, iDim) - VecGet(dirB, iDim)));
  }
  VecSet(&res, 1, VecGet(&res, 1) * 2.0);
  // Return the result
  return res;
}

// Return true if the PBPhys 'that' is the same as PBPhys 'tho'
bool PBPhysIsSame(const PBPhys* const that, const PBPhys* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  if (that->_dim != tho->_dim ||
    !ISEQUALF(that->_deltaT, tho->_deltaT) ||
    !ISEQUALF(that->_curTime, tho->_curTime) ||
    !ISEQUALF(that->_downGravity, tho->_downGravity) ||
    that->_gravity != tho->_gravity ||
    PBPhysGetNbParticle(that) != PBPhysGetNbParticle(tho))
    return false;
  if (PBPhysGetNbParticle(that) > 0) {
    GSetIterForward iterA =
      GSetIterForwardCreateStatic(PBPhysParticles(that));
    GSetIterForward iterB =
      GSetIterForwardCreateStatic(PBPhysParticles(tho));
    do {
      PBPhysParticle* partA = GSetIterGet(&iterA);
      PBPhysParticle* partB = GSetIterGet(&iterB);
      if (!PBPhysParticleIsSame(partA, partB))
        return false;
    } while (GSetIterStep(&iterA) && GSetIterStep(&iterB));
  }
  return true;
}
```

3.2 pbphys-inline.c

```
// ============ PBPHYS-INLINE.C ================

// ------------ PBPhysParticle

// ================ Functions implementation ====================

// Return the dimension of the particle 'that'
#if BUILDMODE != 0
inline
#endif
int PBPhysParticleGetDim(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return ShapoidGetDim(that->_shape);
}

// Return the shape type of the particle 'that'
#if BUILDMODE != 0
inline
#endif
ShapoidType PBPhysParticleGetShapeType(
  const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return ShapoidGetType(that->_shape);
}

// Return the shape of the particle 'that'
#if BUILDMODE != 0
inline
#endif
const Shapoid* PBPhysParticleShape(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_shape;
}

// Return the 'iAxis'-th axis of the shape of the particle 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* PBPhysParticleAxis(const PBPhysParticle* const that,
  const int iAxis) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (iAxis < 0 || iAxis >= PBPhysParticleGetDim(that)) {
    PBPhysErr->_type = PBErrTypeInvalidArg;
    sprintf(PBPhysErr->_msg, "'iAxis' is invalid (0<=%d<%d)",
      iAxis, PBPhysParticleGetDim(that));
    PBErrCatch(PBPhysErr);
  }
#endif
  return ShapoidAxis(PBPhysParticleShape(that), iAxis);
}

// Return the speed of the particle 'that'
```

```
#if BUILDMODE != 0
inline
#endif
const VecFloat* PBPhysParticleSpeed(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_speed;
}

// Return the acceleration of the particle 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* PBPhysParticleAccel(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_accel;
}

// Return the sysAccel of the particle 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* PBPhysParticleSysAccel(
  const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_sysAccel;
}

// Return the position of the center of the particle 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* PBPhysParticleGetPos(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return ShapoidGetCenter(that->_shape);
}

// Set the speed of the particle 'that' to 'speed'
// If the particle is fixed do nothing
```

```
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetSpeed(const PBPhysParticle* const that,
  const VecFloat* const speed) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (speed == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'speed' is null");
    PBErrCatch(PBPhysErr);
  }
  if (VecGetDim(speed) != PBPhysParticleGetDim(that)) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'speed' 's dimension is invalid (%ld=%d)",
      VecGetDim(speed), PBPhysParticleGetDim(that));
    PBErrCatch(PBPhysErr);
  }
#endif
  if (!PBPhysParticleIsFixed(that))
    VecCopy(that->_speed, speed);
}

// Add to the speed of the particle 'that' the vector 'v' multiplied
// by 'c'
// If the particle is fixed do nothing
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleAddSpeed(const PBPhysParticle* const that,
  const VecFloat* const v, const float c) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (v == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'v' is null");
    PBErrCatch(PBPhysErr);
  }
  if (VecGetDim(v) != PBPhysParticleGetDim(that)) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'v' 's dimension is invalid (%ld=%d)",
      VecGetDim(v), PBPhysParticleGetDim(that));
    PBErrCatch(PBPhysErr);
  }
#endif
  if (!PBPhysParticleIsFixed(that))
    VecOp(that->_speed, 1.0, v, c);
}

// Add to the system accel of the particle 'that' the vector 'v'
// multiplied by 'c'
// If the particle is fixed do nothing
#if BUILDMODE != 0
inline
```

```
#endif
void _PBPhysParticleAddSysAccel(const PBPhysParticle* const that,
  const VecFloat* const v, const float c) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (v == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'v' is null");
    PBErrCatch(PBPhysErr);
  }
  if (VecGetDim(v) != PBPhysParticleGetDim(that)) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'v' 's dimension is invalid (%ld=%d)",
      VecGetDim(v), PBPhysParticleGetDim(that));
    PBErrCatch(PBPhysErr);
  }
#endif
  if (!PBPhysParticleIsFixed(that))
    VecOp(that->_sysAccel, 1.0, v, c);
}



// Set the acceleration of the particle 'that' to 'accel'
// If the particle is fixed do nothing
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetAccel(PBPhysParticle* const that,
  const VecFloat* const accel) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (accel == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'accel' is null");
    PBErrCatch(PBPhysErr);
  }
  if (VecGetDim(accel) != PBPhysParticleGetDim(that)) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'accel' 's dimension is invalid (%ld=%d)",
      VecGetDim(accel), PBPhysParticleGetDim(that));
    PBErrCatch(PBPhysErr);
  }
#endif
  if (!PBPhysParticleIsFixed(that))
    VecCopy(that->_accel, accel);
}

// Reset the system acceleration of the particle 'that'
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleResetSysAccel(PBPhysParticle* const that) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  VecSetNull(that->_accel);
}

// Add the gravity force 'gravity' to the system acceleration of the
// particle 'that' (substract 'gravity' to the axis y)
// If the particle is fixed do nothing
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleApplyGravity(PBPhysParticle* const that,
  const float gravity) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  if (!PBPhysParticleIsFixed(that))
    VecSetAdd(that->_accel, 1, -1.0 * gravity);
}

// Set the position of the center of the particle 'that' to 'pos'
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetPos(PBPhysParticle* const that,
  const VecFloat* const pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (pos == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'pos' is null");
    PBErrCatch(PBPhysErr);
  }
  if (VecGetDim(pos) != PBPhysParticleGetDim(that)) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'pos' 's dimension is invalid (%ld=%d)",
      VecGetDim(pos), PBPhysParticleGetDim(that));
    PBErrCatch(PBPhysErr);
  }
#endif
  ShapoidSetCenterPos(that->_shape, pos);
}

// Add to the position of the center of the particle 'that' the
// vector 'v' multiplied by 'c'
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleAddPos(PBPhysParticle* const that,
  const VecFloat* const v, const float c) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (v == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'v' is null");
    PBErrCatch(PBPhysErr);
  }
  if (VecGetDim(v) != PBPhysParticleGetDim(that)) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'v' 's dimension is invalid (%ld=%d)",
      VecGetDim(v), PBPhysParticleGetDim(that));
    PBErrCatch(PBPhysErr);
  }
#endif
  VecFloat* pos = ShapoidGetCenter(PBPhysParticleShape(that));
  VecOp(pos, 1.0, v, c);
  ShapoidSetCenterPos(that->_shape, pos);
  VecFree(&pos);
}

// Return true if the particle 'that' is the same is the particle 'tho'
// Return false else
// User data is not compared
#if BUILDMODE != 0
inline
#endif
bool PBPhysParticleIsSame(const PBPhysParticle* const that,
  const PBPhysParticle* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (tho == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'tho' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  if (!ShapoidIsEqual(PBPhysParticleShape(that),
    PBPhysParticleShape(tho)) ||
    !VecIsEqual(PBPhysParticleSpeed(that), PBPhysParticleSpeed(tho)) ||
    !VecIsEqual(PBPhysParticleAccel(that), PBPhysParticleAccel(tho)) ||
    !ISEQUALF(PBPhysParticleGetMass(that), PBPhysParticleGetMass(tho)) ||
    PBPhysParticleIsFixed(that) != PBPhysParticleIsFixed(tho))
    return false;
  return true;
}

// Set the shape size of the particle 'that' to 'size'
#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetSizeVec(PBPhysParticle* const that,
  const VecFloat* const size) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (size == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'size' is null");
    PBErrCatch(PBPhysErr);
  }
  if (VecGetDim(size) != PBPhysParticleGetDim(that)) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'size' 's dimension is invalid (%ld=%d)",
      VecGetDim(size), PBPhysParticleGetDim(that));
    PBErrCatch(PBPhysErr);
  }
#endif
  for (int iAxis = PBPhysParticleGetDim(that); iAxis--;) {
    float scale = VecGet(size, iAxis) /
      VecNorm(ShapoidAxis(PBPhysParticleShape(that), iAxis));
    ShapoidAxisScale(that->_shape, iAxis, scale);
  }
}

#if BUILDMODE != 0
inline
#endif
void _PBPhysParticleSetSizeScalar(PBPhysParticle* const that,
  const float size) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  for (int iAxis = PBPhysParticleGetDim(that); iAxis--;) {
    float scale = size /
      VecNorm(ShapoidAxis(PBPhysParticleShape(that), iAxis));
    ShapoidAxisScale(that->_shape, iAxis, scale);
  }
}

// Return the mass of the particle 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysParticleGetMass(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_mass;
}

// Set the mass of the particle 'that' to 'mass'
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleSetMass(PBPhysParticle* const that,
```

```
  const float mass) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  that->_mass = mass;
}

// Return the drag of the particle 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysParticleGetDrag(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_drag;
}

// Set the drag of the particle 'that' to 'drag'
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleSetDrag(PBPhysParticle* const that,
  const float drag) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  that->_drag = drag;
}

// Return true if the particle 'that' is fixed
// Return false else
#if BUILDMODE != 0
inline
#endif
bool PBPhysParticleIsFixed(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_fixed;
}

// Set the fixed flag of the particle 'that' to 'fixed'
#if BUILDMODE != 0
inline
#endif
```

```
void PBPhysParticleSetFixed(PBPhysParticle* const that,
  const bool fixed) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  that->_fixed = fixed;
  if (fixed) {
    VecSetNull(that->_speed);
    VecSetNull(that->_accel);
  }
}

// Set the user data of the particle 'that' to 'data'
#if BUILDMODE != 0
inline
#endif
void PBPhysParticleSetData(PBPhysParticle* const that,
  void* const data) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  that->_data = data;
}

// Get the user data of the particle 'that'
#if BUILDMODE != 0
inline
#endif
void* PBPhysParticleData(const PBPhysParticle* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_data;
}

// ------------ PBPhys

// ================ Functions implementation ====================

// Return the space dimension of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
int PBPhysGetDim(const PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
```

```
#endif
  return that->_dim;
}

// Return the set of particles of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
GSetPBPhysParticle* PBPhysParticles(const PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return (GSetPBPhysParticle*)&(that->_particles);
}

// Return the delta t of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysGetDeltaT(const PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_deltaT;
}

// Return the current time of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysGetCurTime(const PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_curTime;
}

// Return the downward gravity of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysGetDownGravity(const PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_downGravity;
```

```
}

// Return the gravity coefficient between particles of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
float PBPhysGetGravity(const PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return that->_gravity;
}

// Set the delta t of the PBPhys 'that' to 'deltaT'
#if BUILDMODE != 0
inline
#endif
void PBPhysSetDeltaT(PBPhys* const that, const float deltaT) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  that->_deltaT = deltaT;
}

// Set the current time of the PBPhys 'that' to 't'
#if BUILDMODE != 0
inline
#endif
void PBPhysSetCurTime(PBPhys* const that, const float t) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  that->_curTime = t;
}

// Set the downward gravity of the PBPhys 'that' to 'gravity'
#if BUILDMODE != 0
inline
#endif
void PBPhysSetDownGravity(PBPhys* const that, float gravity) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  that->_downGravity = gravity;
}
```

```
// Set the gravity coefficient between particles of the PBPhys 'that'
// to 'gravity'
#if BUILDMODE != 0
inline
#endif
void PBPhysSetGravity(PBPhys* const that, float gravity) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  that->_gravity = gravity;
}

// Return the 'iParticle'-th particle of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
PBPhysParticle* PBPhysPart(const PBPhys* const that,
  const int iParticle) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (iParticle < 0 || iParticle >= PBPhysGetNbParticle(that)) {
    PBPhysErr->_type = PBErrTypeInvalidArg;
    sprintf(PBPhysErr->_msg, "'iParticle' is invalid (0<=%d<%d)",
      iParticle, PBPhysGetNbParticle(that));
    PBErrCatch(PBPhysErr);
  }
#endif
  return GSetGet(&(that->_particles), iParticle);
}

// Get the number of particles of the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
int PBPhysGetNbParticle(const PBPhys* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
#endif
  return GSetNbElem(&(that->_particles));
}

// Add 'nb' particles of shape 'shape' into the PBPhys 'that'
#if BUILDMODE != 0
inline
#endif
void PBPhysAddParticles(PBPhys* const that, const int nb,
  const ShapoidType shape) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBPhysErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(PBPhysErr->_msg, "'that' is null");
    PBErrCatch(PBPhysErr);
  }
  if (nb <= 0) {
    PBPhysErr->_type = PBErrTypeInvalidArg;
    sprintf(PBPhysErr->_msg, "'nb' is invalid (0<%d)", nb);
    PBErrCatch(PBPhysErr);
  }
#endif
  for (int iParticle = nb; iParticle--;) {
    PBPhysParticle* particle =
      PBPhysParticleCreate(PBPhysGetDim(that), shape);
    GSetAppend(&(that->_particles), particle);
  }
}
```

4 Makefile

```
# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=0

all: pbmake_wget main

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Rules to make the executable
repo=pbphys
$($(repo)_EXENAME): \
$($(repo)_EXENAME).o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) `echo "$($(repo)_EXE_DEP) $($(repo)_EXENAME).o" | tr ' ' '\n' | sort -u` $(LINK_ARG) $($(repo)_LINK_ARG)

$($(repo)_EXENAME).o: \
$($(repo)_DIR)/$($(repo)_EXENAME).c \
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/$
```

5 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
```

```
#include <sys/time.h>
#include "pberr.h"
#include "pbphys.h"

#define RANDOMSEED 0

void UnitTestPBPhysParticleCreateFreePrint() {
  PBPhysParticle* particle = PBPhysParticleCreate(2,
    ShapoidTypeSpheroid);
  if (particle == NULL ||
    particle->_shape == NULL ||
    particle->_speed == NULL ||
    particle->_accel == NULL ||
    particle->_fixed == true ||
    ISEQUALF(particle->_mass, 0.0) == false ||
    ISEQUALF(particle->_drag, 0.0) == false ||
    VecGetDim(particle->_speed) != 2 ||
    VecGetDim(particle->_accel) != 2 ||
    ShapoidGetDim(particle->_shape) != 2 ||
    ShapoidGetType(particle->_shape) != ShapoidTypeSpheroid) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleCreate failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysParticlePrintln(particle, stdout);
  PBPhysParticleFree(&particle);
  if (particle != NULL) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleFree failed");
    PBErrCatch(PBPhysErr);
  }
  printf("UnitTestPBPhysParticleCreateFreePrint OK\n");
}

void UnitTestPBPhysParticleGetSet() {
  int dim = 2;
  ShapoidType type = ShapoidTypeSpheroid;
  PBPhysParticle* particle = PBPhysParticleCreate(dim, type);
  if (PBPhysParticleGetDim(particle) != dim) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleGetDim failed");
    PBErrCatch(PBPhysErr);
  }
  if (PBPhysParticleGetShapeType(particle) != type) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleGetShapeType failed");
    PBErrCatch(PBPhysErr);
  }
  if (PBPhysParticleShape(particle) != particle->_shape) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleShape failed");
    PBErrCatch(PBPhysErr);
  }
  if (PBPhysParticleSpeed(particle) != particle->_speed) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleSpeed failed");
    PBErrCatch(PBPhysErr);
  }
  if (PBPhysParticleAccel(particle) != particle->_accel) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleAccel failed");
    PBErrCatch(PBPhysErr);
```

```
}
if (PBPhysParticleIsFixed(particle) != particle->_fixed) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysParticleIsFixed failed");
  PBErrCatch(PBPhysErr);
}
VecFloat2D v = VecFloatCreateStatic2D();
VecSet(&v, 0, 2.0); VecSet(&v, 1, 3.0);
PBPhysParticleSetSpeed(particle, &v);
if (VecIsEqual(PBPhysParticleSpeed(particle), &v) == false) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysParticleSetSpeed failed");
  PBErrCatch(PBPhysErr);
}
VecSet(&v, 0, 4.0); VecSet(&v, 1, 5.0);
PBPhysParticleSetPos(particle, &v);
VecFloat* pos = PBPhysParticleGetPos(particle);
if (VecIsEqual(pos, &v) == false) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysParticleSetPos failed");
  PBErrCatch(PBPhysErr);
}
VecFree(&pos);
VecSet(&v, 0, 6.0); VecSet(&v, 1, 7.0);
PBPhysParticleSetAccel(particle, &v);
if (VecIsEqual(PBPhysParticleAccel(particle), &v) == false) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysParticleSetAccel failed");
  PBErrCatch(PBPhysErr);
}
for (int iAxis = dim; iAxis--;)
  if (PBPhysParticleAxis(particle, iAxis) !=
    ShapoidAxis(PBPhysParticleShape(particle), iAxis)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleAxis failed");
    PBErrCatch(PBPhysErr);
  }
PBPhysParticleSetSize(particle, (VecFloat*)&v);
for (int iAxis = dim; iAxis--;)
  if (!ISEQUALF(VecNorm(PBPhysParticleAxis(particle, iAxis)),
    VecGet(&v, iAxis))) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleSetSize failed");
    PBErrCatch(PBPhysErr);
  }
float size = 0.5;
PBPhysParticleSetSize(particle, size);
for (int iAxis = dim; iAxis--;)
  if (!ISEQUALF(VecNorm(PBPhysParticleAxis(particle, iAxis)), size)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleSetSize failed");
    PBErrCatch(PBPhysErr);
  }
float mass = 0.1;
PBPhysParticleSetMass(particle, mass);
if (!ISEQUALF(particle->_mass, mass)) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysParticleSetMass failed");
  PBErrCatch(PBPhysErr);
}
if (!ISEQUALF(PBPhysParticleGetMass(particle), mass)) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
```

```
    sprintf(PBPhysErr->_msg, "PBPhysParticleGetMass failed");
    PBErrCatch(PBPhysErr);
  }
  float drag = 0.2;
  PBPhysParticleSetDrag(particle, drag);
  if (!ISEQUALF(particle->_drag, drag)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleSetDrag failed");
    PBErrCatch(PBPhysErr);
  }
  if (!ISEQUALF(PBPhysParticleGetDrag(particle), drag)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleGetDrag failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysParticleSetFixed(particle, true);
  if (PBPhysParticleIsFixed(particle) != true) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleSetFixed failed");
    PBErrCatch(PBPhysErr);
  }
  char data[2];
  particle->_data = data;
  if (PBPhysParticleData(particle) != data) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleData failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysParticleSetData(particle, data + 1);
  if (PBPhysParticleData(particle) != data + 1) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleSetData failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysParticleFree(&particle);
  printf("UnitTestPBPhysParticleGetSet OK\n");
}

void UnitTestPBPhysParticleCloneIsSame() {
  PBPhysParticle* particle = PBPhysParticleCreate(2,
    ShapoidTypeSpheroid);
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 2.0); VecSet(&v, 1, 3.0);
  PBPhysParticleSetPos(particle, &v);
  VecFloat2D w = VecFloatCreateStatic2D();
  VecSet(&w, 0, 4.0); VecSet(&w, 1, 5.0);
  PBPhysParticleSetSpeed(particle, &w);
  VecSet(&w, 0, 6.0); VecSet(&w, 1, 7.0);
  PBPhysParticleSetAccel(particle, &w);
  PBPhysParticle* clone = PBPhysParticleClone(particle);
  if (PBPhysParticleIsSame(clone, particle) == false) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleClone failed");
    PBErrCatch(PBPhysErr);
  }
  VecSet(&w, 0, 1.0); VecSet(&w, 1, 5.0);
  PBPhysParticleSetSpeed(particle, &w);
  if (PBPhysParticleIsSame(clone, particle) == true) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleIsSame failed");
    PBErrCatch(PBPhysErr);
  }
```

```
  PBPhysParticleFree(&particle);
  PBPhysParticleFree(&clone);
  printf("UnitTestPBPhysParticleCloneIsSame OK\n");
}

void UnitTestPBPhysParticleLoadSave() {
  PBPhysParticle* particle = PBPhysParticleCreate(2,
    ShapoidTypeSpheroid);
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 2.0); VecSet(&v, 1, 3.0);
  PBPhysParticleSetPos(particle, &v);
  VecFloat2D w = VecFloatCreateStatic2D();
  VecSet(&w, 0, 4.0); VecSet(&w, 1, 5.0);
  PBPhysParticleSetSpeed(particle, &w);
  VecSet(&w, 0, 6.0); VecSet(&w, 1, 7.0);
  PBPhysParticleSetAccel(particle, &w);
  PBPhysParticleSetMass(particle, 8.0);
  FILE* fd = fopen("./particle.txt", "w");
  if (PBPhysParticleSave(particle, fd, false) == false) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleSave failed");
    PBErrCatch(PBPhysErr);
  }
  fclose(fd);
  fd = fopen("./particle.txt", "r");
  PBPhysParticle* loaded = PBPhysParticleCreate(2,
    ShapoidTypeSpheroid);
  if (PBPhysParticleLoad(&loaded, fd) == false ||
    PBPhysParticleIsSame(loaded, particle) == false) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysParticleLoad failed");
    PBErrCatch(PBPhysErr);
  }
  fclose(fd);
  PBPhysParticleFree(&loaded);
  PBPhysParticleFree(&particle);
  printf("UnitTestPBPhysParticleLoadSave OK\n");
}

void UnitTestPBPhysParticleAccelMove() {
  PBPhysParticle* particle = PBPhysParticleCreate(2,
    ShapoidTypeSpheroid);
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 1.0); VecSet(&v, 1, -0.5);
  PBPhysParticleSetAccel(particle, &v);
  float dt = 0.1;
  float checkA[20] = {
    0.005000,-0.002500,0.020000,-0.010000,0.045000,-0.022500,0.080000,
    -0.040000,0.125000,-0.062500,0.180000,-0.090000,0.245000,-0.122500,
    0.320000,-0.160000,0.405000,-0.202500,0.500000,-0.250000};
  const VecFloat* pos = ShapoidPos(PBPhysParticleShape(particle));
  for (int i = 0; i < 10; ++i) {
    PBPhysParticleMove(particle, dt);
    if (ISEQUALF(VecGet(PBPhysParticleSpeed(particle), 0),
      (float)(i + 1) * dt) == false ||
      ISEQUALF(VecGet(PBPhysParticleSpeed(particle), 1),
      (float)(i + 1) * dt * -0.5) == false) {
      PBPhysErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBPhysErr->_msg, "PBPhysParticleMove failed");
      PBErrCatch(PBPhysErr);
    }
    if (ISEQUALF(VecGet(pos, 0), checkA[2 * i]) == false ||
```

```
      ISEQUALF(VecGet(pos, 1), checkA[2 * i + 1]) == false) {
      PBPhysErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBPhysErr->_msg, "PBPhysParticleMove failed");
      PBErrCatch(PBPhysErr);
    }
  }
  float drag = 0.1;
  PBPhysParticleSetDrag(particle, drag);
  VecFloat2D vecNull = VecFloatCreateStatic2D();
  ShapoidSetPos(particle->_shape, &vecNull);
  PBPhysParticleSetSpeed(particle, &vecNull);
  float checkC[20] = {
    0.100000,-0.050000,0.199000,-0.099500,0.297010,-0.148505,0.394040,
    -0.197020,0.490099,-0.245050,0.585199,-0.292599,0.679347,-0.339673,
    0.772553,-0.386277,0.864828,-0.432414,0.956179,-0.478090};
  float checkD[20] = {
    0.005000,-0.002500,0.019950,-0.009975,0.044751,-0.022375,0.079303,
    -0.039651,0.123510,-0.061755,0.177275,-0.088637,0.240502,-0.120251,
    0.313097,-0.156549,0.394966,-0.197483,0.486017,-0.243008};
  for (int i = 0; i < 10; ++i) {
    PBPhysParticleMove(particle, dt);
    if (ISEQUALF(VecGet(PBPhysParticleSpeed(particle), 0),
      checkC[2 * i]) == false ||
      ISEQUALF(VecGet(PBPhysParticleSpeed(particle), 1),
      checkC[2 * i + 1]) == false) {
      PBPhysErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBPhysErr->_msg, "PBPhysParticleMove failed");
      PBErrCatch(PBPhysErr);
    }
    if (ISEQUALF(VecGet(pos, 0), checkD[2 * i]) == false ||
      ISEQUALF(VecGet(pos, 1), checkD[2 * i + 1]) == false) {
      PBPhysErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBPhysErr->_msg, "PBPhysParticleMove failed");
      PBErrCatch(PBPhysErr);
    }
  }
  PBPhysParticleFree(&particle);
  printf("UnitTestPBPhysParticleAccelMove OK\n");
}

void UnitTestPBPhysParticleTestTrajectory() {
  PBPhysParticle* am = PBPhysParticleCreate(2,
    ShapoidTypeSpheroid);
  PBPhysParticle* amd = PBPhysParticleCreate(2,
    ShapoidTypeSpheroid);
  VecFloat2D accel = VecFloatCreateStatic2D();
  VecSet(&accel, 0, 4.0); VecSet(&accel, 1, 4.0);
  PBPhysParticleSetSpeed(am, &accel);
  PBPhysParticleSetSpeed(amd, &accel);
  VecSet(&accel, 0, 0.0); VecSet(&accel, 1, -1.0 * PBPHYS_Gn);
  PBPhysParticleSetAccel(am, &accel);
  PBPhysParticleSetAccel(amd, &accel);
  float drag = 0.2;
  PBPhysParticleSetDrag(amd, drag);
  float dt = 0.01;
  float t = 0.0;
  FILE* fd = fopen("./traj.txt", "w");
  const VecFloat* posam = ShapoidPos(PBPhysParticleShape(am));
  const VecFloat* posamd = ShapoidPos(PBPhysParticleShape(amd));
  for (int i = 0; i < 100; ++i) {
    PBPhysParticleMove(am, dt);
    PBPhysParticleMove(amd, dt);
```

```
    t += dt;
    fprintf(fd, "%f %f %f %f %f\n", t,
      VecGet(posam, 0), VecGet(posam, 1),
      VecGet(posamd, 0), VecGet(posamd, 1));
  }
  fclose(fd);
  PBPhysParticleFree(&am);
  PBPhysParticleFree(&amd);
  printf("UnitTestPBPhysParticleTestTrajectory OK\n");
}

void UnitTestPBPhysParticle() {
  UnitTestPBPhysParticleCreateFreePrint();
  UnitTestPBPhysParticleGetSet();
  UnitTestPBPhysParticleCloneIsSame();
  UnitTestPBPhysParticleLoadSave();
  UnitTestPBPhysParticleAccelMove();
  UnitTestPBPhysParticleTestTrajectory();
  printf("UnitTestPBPhysParticle OK\n");
}

void UnitTestPBPhysCreateFreePrint() {
  int dim = 2;
  PBPhys* phys = PBPhysCreate(dim);
  if (phys == NULL ||
    phys->_dim != 2 ||
    !ISEQUALF(phys->_deltaT, PBPHYS_DELTAT) ||
    !ISEQUALF(phys->_downGravity, 0.0) ||
    !ISEQUALF(phys->_curTime, 0.0) ||
    phys->_gravity != 0 ||
    GSetNbElem(PBPhysParticles(phys)) != 0) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysCreate failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysAddParticles(phys, 2, ShapoidTypeSpheroid);
  PBPhysParticleSetMass(PBPhysPart(phys, 1), 1.0);
  PBPhysPrintln(phys, stdout);
  PBPhysFree(&phys);
  if (phys != NULL) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysFree failed");
    PBErrCatch(PBPhysErr);
  }
  printf("UnitTestPBPhysCreateFreePrint OK\n");
}

void UnitTestPBPhysGetSetAdd() {
  int dim = 2;
  PBPhys* phys = PBPhysCreate(dim);
  if (PBPhysGetDim(phys) != dim) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysGetDim failed");
    PBErrCatch(PBPhysErr);
  }
  phys->_curTime = 0.1;
  if (!ISEQUALF(PBPhysGetCurTime(phys), phys->_curTime)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysGetCurTime failed");
    PBErrCatch(PBPhysErr);
  }
  if (!ISEQUALF(PBPhysGetDeltaT(phys), PBPHYS_DELTAT)) {
```

```
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysGetDeltaT failed");
  PBErrCatch(PBPhysErr);
}
phys->_downGravity = 0.2;
if (!ISEQUALF(PBPhysGetDownGravity(phys), phys->_downGravity)) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysGetDownGravity failed");
  PBErrCatch(PBPhysErr);
}
phys->_gravity = 0.3;
if (!ISEQUALF(PBPhysGetGravity(phys), phys->_gravity)) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysGetGravity failed");
  PBErrCatch(PBPhysErr);
}
PBPhysSetCurTime(phys, 0.2);
if (!ISEQUALF(PBPhysGetCurTime(phys), phys->_curTime)) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysSetCurTime failed");
  PBErrCatch(PBPhysErr);
}
PBPhysSetDeltaT(phys, 0.3);
if (!ISEQUALF(PBPhysGetDeltaT(phys), phys->_deltaT)) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysSetDeltaT failed");
  PBErrCatch(PBPhysErr);
}
PBPhysSetDownGravity(phys, 0.4);
if (!ISEQUALF(PBPhysGetDownGravity(phys), phys->_downGravity)) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysSetDownGravity failed");
  PBErrCatch(PBPhysErr);
}
PBPhysSetGravity(phys, 0.5);
if (!ISEQUALF(PBPhysGetGravity(phys), phys->_gravity)) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysSetGravity failed");
  PBErrCatch(PBPhysErr);
}
if (PBPhysParticles(phys) != &(phys->_particles)) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysParticles failed");
  PBErrCatch(PBPhysErr);
}
if (PBPhysGetNbParticle(phys) != 0) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysGetNbParticle failed");
  PBErrCatch(PBPhysErr);
}
PBPhysAddParticles(phys, 2, ShapoidTypeSpheroid);
if (GSetNbElem(PBPhysParticles(phys)) != 2) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysAddParticles failed");
  PBErrCatch(PBPhysErr);
}
if (PBPhysGetNbParticle(phys) != 2) {
  PBPhysErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBPhysErr->_msg, "PBPhysGetNbParticle failed");
  PBErrCatch(PBPhysErr);
}
if (GSetGet(PBPhysParticles(phys), 0) != PBPhysPart(phys, 0) ||
```

```
    GSetGet(PBPhysParticles(phys), 1) != PBPhysPart(phys, 1)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysPart failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysFree(&phys);
  printf("UnitTestPBPhysGetSetAdd OK\n");
}

void UnitTestPBPhysCloneIsSame() {
  int dim = 2;
  PBPhys* phys = PBPhysCreate(dim);
  PBPhysAddParticles(phys, 2, ShapoidTypeSpheroid);
  PBPhysParticleSetMass(PBPhysPart(phys, 1), 1.0);
  PBPhys* clone = PBPhysClone(phys);
  if (!PBPhysIsSame(clone, phys)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysClone failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysParticleSetMass(PBPhysPart(clone, 1), 2.0);
  if (PBPhysIsSame(clone, phys)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysIsSame failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysFree(&phys);
  PBPhysFree(&clone);
  printf("UnitTestPBPhysCloneIsSame OK\n");
}

void UnitTestPBPhysLoadSave() {
  int dim = 2;
  PBPhys* phys = PBPhysCreate(dim);
  PBPhysAddParticles(phys, 2, ShapoidTypeSpheroid);
  PBPhysParticleSetMass(PBPhysPart(phys, 1), 1.0);
  FILE* fd = fopen("./phys.txt", "w");
  if (!PBPhysSave(phys, fd, false)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysSave failed");
    PBErrCatch(PBPhysErr);
  }
  fclose(fd);
  PBPhys* loaded = PBPhysCreate(dim);
  fd = fopen("./phys.txt", "r");
  if (!PBPhysLoad(&loaded, fd) ||
    !PBPhysIsSame(phys, loaded)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysLoad failed");
    PBErrCatch(PBPhysErr);
  }
  fclose(fd);
  PBPhysFree(&phys);
  PBPhysFree(&loaded);
  printf("UnitTestPBPhysLoadSave OK\n");
}

void UnitTestPBPhysStepFree() {
  int dim = 2;
  PBPhys* phys = PBPhysCreate(dim);
  PBPhysAddParticles(phys, 1, ShapoidTypeSpheroid);
  VecFloat2D v = VecFloatCreateStatic2D();
```

```
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.5);
  PBPhysParticleSetSpeed(PBPhysPart(phys, 0), &v);
  float check[20] = {
    0.010000,0.005000,0.020000,0.010000,0.030000,0.015000,0.040000,
    0.020000,0.050000,0.025000,0.060000,0.030000,0.070000,0.035000,
    0.080000,0.040000,0.090000,0.045000,.100000,0.050000};
  for (int i = 0; i < 10; ++i) {
    PBPhysNext(phys);
    VecSet(&v, 0, check[2 * i]); VecSet(&v, 1, check[2 * i + 1]);
    if (!VecIsEqual(
      ShapoidPos(PBPhysParticleShape(PBPhysPart(phys, 0))),
      &v)) {
      PBPhysErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBPhysErr->_msg, "PBPhysStep failed");
      PBErrCatch(PBPhysErr);
    }
  }
  PBPhysFree(&phys);
  printf("UnitTestPBPhysStepFree OK\n");
}

void UnitTestPBPhysStepDownGravity() {
  int dim = 2;
  PBPhys* phys = PBPhysCreate(dim);
  PBPhysSetDownGravity(phys, PBPHYS_Gn);
  PBPhysAddParticles(phys, 1, ShapoidTypeSpheroid);
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.5);
  PBPhysParticleSetSpeed(PBPhysPart(phys, 0), &v);
  float check[20] = {
    0.010000,0.004510,0.020000,0.008039,0.030000,0.010587,0.040000,
    0.012155,0.050000,0.012742,0.060000,0.012348,0.070000,0.010974,
    0.080000,0.008619,0.090000,0.005283,0.100000,0.000967};
  for (int i = 0; i < 10; ++i) {
    PBPhysNext(phys);
    VecSet(&v, 0, check[2 * i]); VecSet(&v, 1, check[2 * i + 1]);
    if (!VecIsEqual(
      ShapoidPos(PBPhysParticleShape(PBPhysPart(phys, 0))),
      &v)) {
      PBPhysErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBPhysErr->_msg, "PBPhysStep failed");
      PBErrCatch(PBPhysErr);
    }
  }
  PBPhysFree(&phys);
  printf("UnitTestPBPhysStepDownGravity OK\n");
}

void UnitTestPBPhysStepGravity() {
  int dim = 2;
  PBPhys* phys = PBPhysCreate(dim);
  PBPhysSetGravity(phys, 1.0);
  PBPhysAddParticles(phys, 2, ShapoidTypeSpheroid);
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.5);
  PBPhysParticleSetSpeed(PBPhysPart(phys, 0), &v);
  PBPhysParticleSetMass(PBPhysPart(phys, 0), 1.0);
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 1.0);
  PBPhysParticleSetPos(PBPhysPart(phys, 1), &v);
  PBPhysParticleSetMass(PBPhysPart(phys, 1), 1.0);
  PBPhysParticleSetFixed(PBPhysPart(phys, 1), true);
  float check[20] = {
```

```
    0.010018,0.005018,0.020089,0.010089,0.030249,0.015249,0.040535,
    0.020537,0.050984,0.025988,0.061633,0.031642,0.072520,0.037538,
    0.083684,0.043716,0.095164,0.050216,0.107000,0.057080
    };
  for (int i = 0; i < 10; ++i) {
    PBPhysNext(phys);
    VecSet(&v, 0, check[2 * i]); VecSet(&v, 1, check[2 * i + 1]);
    if (!VecIsEqual(
      ShapoidPos(PBPhysParticleShape(PBPhysPart(phys, 0))),
      &v)) {
      PBPhysErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBPhysErr->_msg, "PBPhysStep failed");
      PBErrCatch(PBPhysErr);
    }
  }
  PBPhysFree(&phys);
  printf("UnitTestPBPhysStepGravity OK\n");
}

void UnitTestPBPhysStepToCollisionApplyElasticCollision() {
  int dim = 2;
  PBPhys* phys = PBPhysCreate(dim);
  PBPhysAddParticles(phys, 3, ShapoidTypeSpheroid);
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 1.0);
  PBPhysParticleSetSpeed(PBPhysPart(phys, 0), &v);
  PBPhysParticleSetMass(PBPhysPart(phys, 0), 1.0);
  VecSet(&v, 0, 0.0); VecSet(&v, 1, 2.0);
  PBPhysParticleSetPos(PBPhysPart(phys, 1), &v);
  VecSet(&v, 0, 0.5); VecSet(&v, 1, -0.5);
  PBPhysParticleSetSpeed(PBPhysPart(phys, 1), &v);
  PBPhysParticleSetMass(PBPhysPart(phys, 1), 1.0);
  VecSet(&v, 0, 2.0); VecSet(&v, 1, 2.0);
  PBPhysParticleSetPos(PBPhysPart(phys, 2), &v);
  VecSet(&v, 0, -1.0); VecSet(&v, 1, -1.0);
  PBPhysParticleSetSpeed(PBPhysPart(phys, 2), &v);
  PBPhysParticleSetMass(PBPhysPart(phys, 2), 2.0);
  PBPhysSetDeltaT(phys, 2.0);
  GSetPBPhysParticle* set = PBPhysStepToCollision(phys);
  if (!ISEQUALF(PBPhysGetCurTime(phys), 0.646447) ||
    !ISEQUALF(PBPhysPart(phys,0)->_shape->_pos->_val[0], 0.646447) ||
    !ISEQUALF(PBPhysPart(phys,0)->_shape->_pos->_val[1], 0.646447) ||
    !ISEQUALF(PBPhysPart(phys,1)->_shape->_pos->_val[0], 0.323223) ||
    !ISEQUALF(PBPhysPart(phys,1)->_shape->_pos->_val[1], 1.676777) ||
    !ISEQUALF(PBPhysPart(phys,2)->_shape->_pos->_val[0], 1.353553) ||
    !ISEQUALF(PBPhysPart(phys,2)->_shape->_pos->_val[1], 1.353553) ||
    GSetNbElem(set) != 2 ||
    GSetGet(set, 0) != PBPhysPart(phys,0) ||
    GSetGet(set, 1) != PBPhysPart(phys,2)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysStepToCollision failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysParticleApplyElasticCollision(GSetGet(set, 0), GSetGet(set, 1));
  if (!ISEQUALF(PBPhysPart(phys,0)->_speed->_val[0], -1.666667) ||
    !ISEQUALF(PBPhysPart(phys,0)->_speed->_val[1], -1.666667) ||
    !ISEQUALF(PBPhysPart(phys,2)->_speed->_val[0], 0.333334) ||
    !ISEQUALF(PBPhysPart(phys,2)->_speed->_val[1], 0.333334)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg,
      "PBPhysParticleApplyElasticCollision failed");
    PBErrCatch(PBPhysErr);
```

```
  }
  GSetFree(&set);
  set = PBPhysStepToCollision(phys);
  if (!ISEQUALF(PBPhysGetCurTime(phys), 0.878937) ||
    !ISEQUALF(PBPhysPart(phys,0)->_shape->_pos->_val[0], 0.258963) ||
    !ISEQUALF(PBPhysPart(phys,0)->_shape->_pos->_val[1], 0.258963) ||
    !ISEQUALF(PBPhysPart(phys,1)->_shape->_pos->_val[0], 0.439468) ||
    !ISEQUALF(PBPhysPart(phys,1)->_shape->_pos->_val[1], 1.560532) ||
    !ISEQUALF(PBPhysPart(phys,2)->_shape->_pos->_val[0], 1.431050) ||
    !ISEQUALF(PBPhysPart(phys,2)->_shape->_pos->_val[1], 1.431050) ||
    GSetNbElem(set) != 2 ||
    GSetGet(set, 0) != PBPhysPart(phys,1) ||
    GSetGet(set, 1) != PBPhysPart(phys,2)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg, "PBPhysStepToCollision failed");
    PBErrCatch(PBPhysErr);
  }
  PBPhysParticleApplyElasticCollision(GSetGet(set, 0), GSetGet(set, 1));
  if (!ISEQUALF(PBPhysPart(phys,1)->_speed->_val[0], 0.138846) ||
    !ISEQUALF(PBPhysPart(phys,1)->_speed->_val[1], -0.452840) ||
    !ISEQUALF(PBPhysPart(phys,2)->_speed->_val[0], 0.513910) ||
    !ISEQUALF(PBPhysPart(phys,2)->_speed->_val[1], 0.309754)) {
    PBPhysErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBPhysErr->_msg,
      "PBPhysParticleApplyElasticCollision failed");
    PBErrCatch(PBPhysErr);
  }
  GSetFree(&set);
  VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0);
  PBPhysParticleSetPos(PBPhysPart(phys, 0), &v);
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 1.0);
  PBPhysParticleSetSpeed(PBPhysPart(phys, 0), &v);
  VecSet(&v, 0, 0.0); VecSet(&v, 1, 2.0);
  PBPhysParticleSetPos(PBPhysPart(phys, 1), &v);
  VecSet(&v, 0, 0.5); VecSet(&v, 1, -0.5);
  PBPhysParticleSetSpeed(PBPhysPart(phys, 1), &v);
  VecSet(&v, 0, 2.0); VecSet(&v, 1, 2.0);
  PBPhysParticleSetPos(PBPhysPart(phys, 2), &v);
  VecSet(&v, 0, -1.0); VecSet(&v, 1, -1.0);
  PBPhysParticleSetSpeed(PBPhysPart(phys, 2), &v);
  ShapoidScale(PBPhysParticleShape(PBPhysPart(phys, 2)), (float)2.0);
  PBPhysSetDeltaT(phys, 0.05);
  PBPhysSetCurTime(phys, 0.0);
  FILE* fd = fopen("./collision.txt", "w");
  for (int i = 0; i < 20; ++i) {
    PBPhysStep(phys);
    fprintf(fd, "%f %f %f %f %f %f %f\n", PBPhysGetCurTime(phys),
      PBPhysPart(phys,0)->_shape->_pos->_val[0],
      PBPhysPart(phys,0)->_shape->_pos->_val[1],
      PBPhysPart(phys,1)->_shape->_pos->_val[0],
      PBPhysPart(phys,1)->_shape->_pos->_val[1],
      PBPhysPart(phys,2)->_shape->_pos->_val[0],
      PBPhysPart(phys,2)->_shape->_pos->_val[1]);
  }
  fclose(fd);
  PBPhysFree(&phys);
  printf("UnitTestPBPhysStepToCollisionApplyElasticCollision OK\n");
}

void UnitTestPBPhysNext() {
  UnitTestPBPhysStepFree();
  UnitTestPBPhysStepDownGravity();
```

```
  UnitTestPBPhysStepGravity();
  UnitTestPBPhysStepToCollisionApplyElasticCollision();
  printf("UnitTestPBPhysStep OK\n");
}

void UnitTestPBPhys() {
  UnitTestPBPhysCreateFreePrint();
  UnitTestPBPhysGetSetAdd();
  UnitTestPBPhysCloneIsSame();
  UnitTestPBPhysLoadSave();
  UnitTestPBPhysNext();

  printf("UnitTestPBPhys OK\n");
}

void UnitTestAll() {
  UnitTestPBPhysParticle();
  UnitTestPBPhys();
  printf("UnitTestAll OK\n");
}

int main() {
  UnitTestAll();
  // Return success code
  return 0;
}
```

6 Unit tests output

```
Type: Spheroid
Dim: 2
Pos: <0.000,0.000>
Axis(0): <1.000,0.000>
Axis(1): <0.000,1.000>
speed: <0.000,0.000>
accel: <0.000,0.000>
mass: 0.000
drag: 0.000
unfixed
UnitTestPBPhysParticleCreateFreePrint OK
UnitTestPBPhysParticleGetSet OK
UnitTestPBPhysParticleCloneIsSame OK
UnitTestPBPhysParticleLoadSave OK
UnitTestPBPhysParticleAccelMove OK
UnitTestPBPhysParticleTestTrajectory OK
UnitTestPBPhysParticle OK
dimension: 2
t: 0.000000
dt: 0.010000
down gravity: 0.000000
gravity: 0.000000
nb particles: 2
particle #0:
Type: Spheroid
Dim: 2
Pos: <0.000,0.000>
Axis(0): <1.000,0.000>
Axis(1): <0.000,1.000>
```

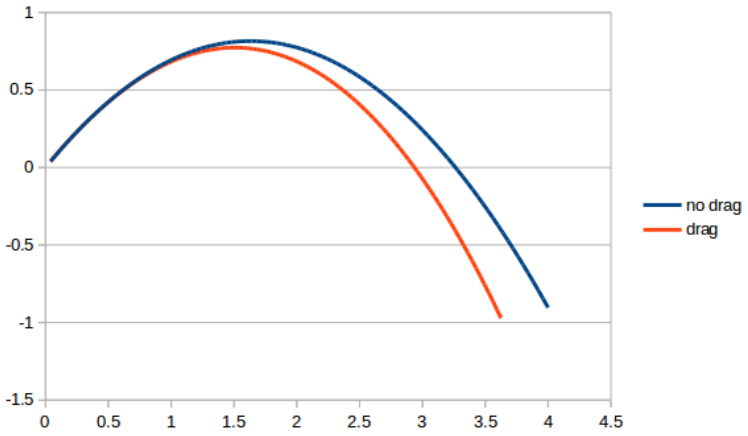
```
speed: <0.000,0.000>
accel: <0.000,0.000>
mass: 0.000
drag: 0.000
unfixed
particle #1:
Type: Spheroid
Dim: 2
Pos: <0.000,0.000>
Axis(0): <1.000,0.000>
Axis(1): <0.000,1.000>
speed: <0.000,0.000>
accel: <0.000,0.000>
mass: 1.000
drag: 0.000
unfixed
UnitTestPBPhysCreateFreePrint OK
UnitTestPBPhysGetSetAdd OK
UnitTestPBPhysCloneIsSame OK
UnitTestPBPhysLoadSave OK
UnitTestPBPhysStepFree OK
UnitTestPBPhysStepDownGravity OK
UnitTestPBPhysStepGravity OK
UnitTestPBPhysStepToCollisionApplyElasticCollision OK
UnitTestPBPhysStep OK
UnitTestPBPhys OK
UnitTestAll OK
```

traj.txt:

```
0.010000 0.040000 0.039510 0.039960 0.039470
0.020000 0.080000 0.078039 0.079840 0.077880
0.030000 0.120000 0.115587 0.119640 0.115232
0.040000 0.160000 0.152155 0.159361 0.151530
0.050000 0.200000 0.187742 0.199002 0.186773
0.060000 0.240000 0.222348 0.238564 0.220966
0.070000 0.280000 0.255974 0.278047 0.254110
0.080000 0.320000 0.288619 0.317451 0.286207
0.090000 0.360000 0.320283 0.356776 0.317259
0.100000 0.400000 0.350967 0.396023 0.347268
0.110000 0.440000 0.380670 0.435191 0.376236
0.120000 0.480000 0.409392 0.474280 0.404166
0.130000 0.520000 0.437134 0.513292 0.431060
0.140000 0.560000 0.463895 0.552225 0.456918
0.150000 0.600000 0.489675 0.591081 0.481745
0.160000 0.640000 0.514475 0.629858 0.505541
0.170000 0.680000 0.538294 0.668559 0.528309
0.180000 0.720000 0.561132 0.707182 0.550051
0.190000 0.760000 0.582990 0.745727 0.570768
0.200000 0.800000 0.603867 0.784196 0.590464
0.210000 0.840000 0.623763 0.822587 0.609139
0.220000 0.880000 0.642679 0.860902 0.626797
0.230000 0.920000 0.660614 0.899140 0.643438
0.240000 0.960000 0.677568 0.937302 0.659066
0.250000 1.000000 0.693542 0.975388 0.673681
0.260000 1.040000 0.708535 1.013397 0.687287
0.270000 1.080000 0.722547 1.051330 0.699885
0.280000 1.120000 0.735579 1.089187 0.711477
0.290000 1.160000 0.747630 1.126969 0.722065
0.300000 1.200000 0.758701 1.164675 0.731651
0.310000 1.240000 0.768790 1.202306 0.740237
```

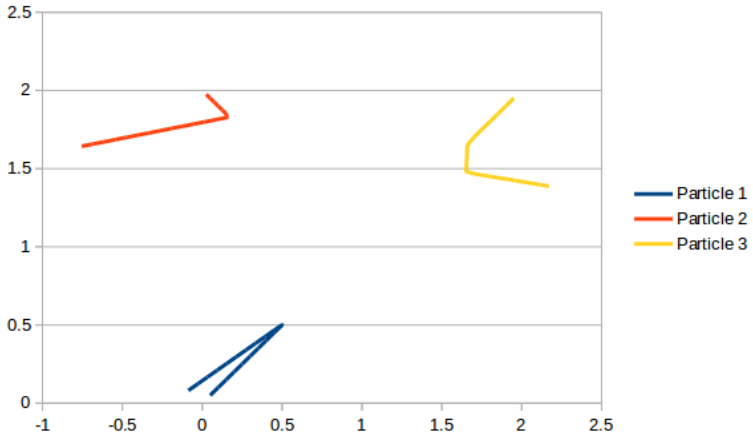
```
0.320000 1.280000 0.777899 1.239861 0.747826
0.330000 1.320000 0.786028 1.277341 0.754419
0.340000 1.360000 0.793175 1.314746 0.760018
0.350000 1.400000 0.799342 1.352077 0.764625
0.360000 1.440000 0.804529 1.389333 0.768242
0.370000 1.480000 0.808735 1.426514 0.770871
0.380000 1.520000 0.811960 1.463621 0.772514
0.390000 1.560000 0.814204 1.500654 0.773174
0.400000 1.600000 0.815468 1.537613 0.772851
0.410000 1.640000 0.815751 1.574497 0.771548
0.420000 1.679999 0.815053 1.611308 0.769268
0.430000 1.719999 0.813375 1.648046 0.766011
0.440000 1.759999 0.810716 1.684710 0.761780
0.450000 1.799999 0.807076 1.721300 0.756577
0.460000 1.839999 0.802456 1.757818 0.750403
0.470000 1.879999 0.796855 1.794262 0.743262
0.480000 1.919999 0.790274 1.830634 0.735153
0.490000 1.959999 0.782711 1.866932 0.726081
0.500000 1.999999 0.774168 1.903159 0.716046
0.510000 2.039999 0.764645 1.939312 0.705050
0.520000 2.079999 0.754140 1.975394 0.693096
0.530000 2.119999 0.742656 2.011403 0.680185
0.540000 2.159999 0.730190 2.047340 0.666319
0.550000 2.199999 0.716744 2.083205 0.651500
0.560000 2.239999 0.702317 2.118999 0.635730
0.570000 2.279999 0.686909 2.154721 0.619011
0.580000 2.319999 0.670521 2.190372 0.601345
0.590000 2.359999 0.653152 2.225951 0.582733
0.600000 2.399999 0.634803 2.261459 0.563178
0.610000 2.439999 0.615472 2.296896 0.542681
0.620000 2.479999 0.595161 2.332262 0.521245
0.630000 2.519999 0.573870 2.367558 0.498871
0.640000 2.559999 0.551598 2.402783 0.475561
0.650000 2.599999 0.528345 2.437937 0.451317
0.660000 2.639999 0.504111 2.473021 0.426141
0.670000 2.679999 0.478897 2.508035 0.400034
0.680000 2.719999 0.452702 2.542979 0.372999
0.690000 2.759999 0.425526 2.577853 0.345038
0.700000 2.799999 0.397370 2.612658 0.316152
0.710000 2.839998 0.368233 2.647392 0.286342
0.720000 2.879998 0.338116 2.682057 0.255612
0.730000 2.919998 0.307018 2.716653 0.223963
0.740000 2.959998 0.274939 2.751180 0.191396
0.750000 2.999998 0.241879 2.785638 0.157914
0.760000 3.039998 0.207839 2.820027 0.123517
0.770000 3.079998 0.172818 2.854347 0.088210
0.780000 3.119998 0.136816 2.888598 0.051992
0.790000 3.159998 0.099834 2.922781 0.014865
0.800000 3.199998 0.061871 2.956895 -0.023167
0.810000 3.239998 0.022928 2.990941 -0.062104
0.820000 3.279998 -0.016996 3.024919 -0.101944
0.830000 3.319998 -0.057901 3.058830 -0.142685
0.840000 3.359998 -0.099787 3.092672 -0.184326
0.849999 3.399998 -0.142653 3.126446 -0.226863
0.859999 3.439998 -0.186500 3.160154 -0.270296
0.869999 3.479998 -0.231327 3.193793 -0.314623
0.879999 3.519998 -0.277136 3.227366 -0.359842
0.889999 3.559998 -0.323925 3.260871 -0.405951
0.899999 3.599998 -0.371694 3.294309 -0.452949
0.909999 3.639998 -0.420444 3.327681 -0.500833
0.919999 3.679998 -0.470175 3.360985 -0.549602
0.929999 3.719998 -0.520887 3.394223 -0.599255
```

```
0.939999 3.759998 -0.572579 3.427395 -0.649788
0.949999 3.799998 -0.625252 3.460500 -0.701201
0.959999 3.839998 -0.678905 3.493539 -0.753493
0.969999 3.879997 -0.733539 3.526512 -0.806660
0.979999 3.919997 -0.789154 3.559419 -0.860701
0.989999 3.959997 -0.845750 3.592260 -0.915615
0.999999 3.999997 -0.903326 3.625036 -0.971400
```



collision.txt:

```
0.050000 0.050000 0.050000 0.025000 1.975000 1.950000 1.950000
0.100000 0.100000 0.100000 0.050000 1.950000 1.900000 1.900000
0.150000 0.150000 0.150000 0.075000 1.925000 1.850000 1.850000
0.200000 0.200000 0.200000 0.100000 1.900000 1.800000 1.800000
0.250000 0.250000 0.250000 0.125000 1.875000 1.750000 1.750000
0.300000 0.300000 0.300000 0.150000 1.850000 1.700000 1.700000
0.350000 0.350000 0.350000 0.155565 1.827215 1.659718 1.648893
0.400000 0.400000 0.400000 0.085599 1.813041 1.657201 1.593480
0.450000 0.450000 0.450000 0.015633 1.798866 1.654684 1.538068
0.500000 0.500000 0.500000 -0.054334 1.784691 1.652167 1.482655
0.550000 0.454424 0.469154 -0.124300 1.770516 1.697439 1.467666
0.600000 0.394305 0.426006 -0.194267 1.756341 1.749981 1.458827
0.650000 0.334186 0.382858 -0.264233 1.742166 1.802524 1.449989
0.700000 0.274067 0.339710 -0.334199 1.727992 1.855067 1.441150
0.750000 0.213948 0.296562 -0.404166 1.713817 1.907609 1.432312
0.800000 0.153829 0.253414 -0.474132 1.699642 1.960152 1.423473
0.850000 0.093709 0.210266 -0.544099 1.685467 2.012695 1.414634
0.900000 0.033590 0.167118 -0.614065 1.671292 2.065238 1.405796
0.950000 -0.026529 0.123970 -0.684032 1.657117 2.117780 1.396957
1.000000 -0.086648 0.080822 -0.753998 1.642943 2.170323 1.388119
```


2.5
2
1.5
1
0.5
0
-1
-0.5
0
0.5
1
1.5
2
2.5
Particle 1
Particle 2
Particle 3