PBMath

P. Baillehache

October 30, 2017

Contents

Introduction

PBMath is C library providing mathematical structures and functions.

The `VecFloat` structure and its functions can be used to manipulate vectors of float values.

The `VecShort` structure and its functions can be used to manipulate vectors of short values.

The `MatFloat` structure and its functions can be used to manipulate matrices of float values.

The `Shapoid` structure and its functions can be used to manipulate Shapoid objects (see next section for details).

The `Gauss` structure and its functions can be used to get values of the Gauss function and random values distributed accordingly with a Gauss distribution.

The `Smoother` functions can be used to get values of the SmoothStep and SmootherStep functions.

1 Definitions

1.1 Angle between two vectors

The problem is as follow: given two vectors $\vec{V}$ and $\vec{W}$ not null, how to calculate the angle θ from $\vec{V}$ to $\vec{W}$.

Let's call M the rotation matrix: $M\vec{V} = \vec{W}$, and the components of M as follow:

$$M = \begin{bmatrix} Ma & Mb \\ Mc & Md \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{1}$$

Then, $M\vec{V} = \vec{W}$ can be written has

$$\begin{cases} W_x = M_a V_x + M_b V_y \\ W_y = M_c V_x + M_d V_y \end{cases} \tag{2}$$

Equivalent to

$$\begin{cases} W_x = M_a V_x + M_b V_y \\ W_y = -M_b V_x + M_a V_y \end{cases} \tag{3}$$

where $M_a = cos(\theta)$ and $M_b = -sin(\theta)$.
If $Vx \neq 0.0$, we can write

$$\begin{cases} M_b = \frac{M_a V_y - W_y}{V_x} \\ M_a = \frac{W_x + W_y V_y / V_x}{V_x + V_y^2 / V_x} \end{cases} \tag{4}$$

Or, if $Vx = 0.0$, we can write

$$\begin{cases} Ma = \frac{W_y + M_b V_x}{V_y} \\ Mb = \frac{W_x - W_y V_x / V_y}{V_y + V_x^2 / V_y} \end{cases} \tag{5}$$

Then we have $\theta = \pm cos^{-1}(M_a)$ where the sign can be determined by verifying that the sign of $sin(\theta)$ matches the sign of $-M_b$: if $sin(cos^{-1}(M_a)) * M_b > 0.0$ then multiply $\theta = -cos^{-1}(M_a)$ else $\theta = cos^{-1}(M_a)$.

1.2 Shapoid

1.2.1 Definition

A Shapoid is a geometry defined by its dimension $D \in \mathbb{N}_+^*$ equals to the number of dimensions of the space it exists in, its position $\overrightarrow{P}$, and its axis $(\overrightarrow{A_0}, \overrightarrow{A_1}, ..., \overrightarrow{A_{D-1}})$. A_i and P are vectors of dimension D. In what follows I'll use I as notation for the interval $[0, D-1]$ for simplification.

Shapoids are classified in three groups: Facoid, Pyramidoid and Spheroid. The volume of a Shapoid is defined by, for a Facoid:

$$\left\{ \sum_{i \in I} v_i \overrightarrow{A_i} + \overrightarrow{P} \right\}, v_i \in [0.0, 1.0] \tag{6}$$

for a Pyramidoid:

$$\left\{ \sum_{i \in I} v_i \overrightarrow{A_i} + \overrightarrow{P} \right\}, v_i \in [0.0, 1.0], \sum_{i \in I} v_i \leq 1.0 \tag{7}$$

and for a Spheroid:

$$\begin{array}{r} \left\{ \sum_{i \in I} (v_i - 0.5) \overrightarrow{A_i} + \overrightarrow{P} \right\}, \\ v_i \in [0.0, 1.0], \quad \sum_{i \in I} (v_i - 0.5)^2 \leq 0.25 \end{array} \tag{8}$$

1.2.2 Transformation

A translation of a Shapoid by $\overrightarrow{T}$ is obtained as follow:

$$\left(\overrightarrow{P}, \left\{ \overrightarrow{A_i} \right\}_{i \in I} \right) \mapsto \left(\overrightarrow{P} + \overrightarrow{T}, \left\{ \overrightarrow{A_i} \right\}_{i \in I} \right) \tag{9}$$

A scale of a Shapoid by $\overrightarrow{S}$ is obtained as follow:

$$\left(\overrightarrow{P}, \left\{ \overrightarrow{A_i} \right\}_{i \in I} \right) \mapsto \left(\overrightarrow{P}, \left\{ \overrightarrow{A_i'} \right\}_{i \in I} \right) \tag{10}$$

where

$$\overrightarrow{A'_i} = S_i\overrightarrow{A_i} \tag{11}$$

For Shapoid whose dimension D is equal to 2, a rotation by angle θ is obtained as follow:

$$\left(\overrightarrow{P}, \overrightarrow{A_0}, \overrightarrow{A_1}\right) \mapsto \left(\overrightarrow{P}, \overrightarrow{A'_0}, \overrightarrow{A'_1}\right) \tag{12}$$

where

$$\overrightarrow{A'_i} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \overrightarrow{A_i} \tag{13}$$

1.2.3 Shapoid's coordinate system

The Shapoid's coordinate system is the system having $\overrightarrow{P}$ as origin and $\overrightarrow{A_i}$ as axis. One can change from the Shapoid's coordinate system ($\overrightarrow{X^S}$) to the standard coordinate system ($\overrightarrow{X}$) as follow:

$$\overrightarrow{X} = \left[\left(\overrightarrow{A_0}\right)\left(\overrightarrow{A_1}\right)...\left(\overrightarrow{A_{D-1}}\right)\right]\overrightarrow{X^S} + \overrightarrow{P} \tag{14}$$

and reciprocally, from the standard coordinate system to the Shapoid's coordinate system:

$$\overrightarrow{X^S} = \left[\left(\overrightarrow{A_0}\right)\left(\overrightarrow{A_1}\right)...\left(\overrightarrow{A_{D-1}}\right)\right]^{-1}\left(\overrightarrow{X} - \overrightarrow{P}\right) \tag{15}$$

1.2.4 Insideness

$\overrightarrow{X}$ is inside the Shapoid S if, for a Facoid:

$$\forall i \in I, 0.0 \leq X_i^S \leq 1.0 \tag{16}$$

for a Pyramidoid:

$$\begin{cases} \forall i \in I, 0.0 \leq X_i^S \leq 1.0 \\ \sum_{i \in I} X_i^S \leq 1.0 \end{cases} \tag{17}$$

for a Spheroid:

$$\left|\left|\overrightarrow{X^S}\right|\right| \leq 0.5 \tag{18}$$

1.2.5 Bounding box

A bounding box of a Shapoid is a Facoid whose axis are colinear to axis of the standard coordinate system, and including the Shapoid in its volume. While the smallest possible bounding box can be easily obtained for Facoid and Pyramidoid, it's more complicate for Spheroid. Then we will consider for the Spheroid the bounding box of the equivalent Facoid $\left(\overrightarrow{P}-\sum_{i\in I}\left(0.5*\overrightarrow{A_i}\right),\left\{\overrightarrow{A_i}\right\}_{i\in I}\right)$ which gives the smallest bounding box when axis of the Spheroid are colinear to axis of the standard coordinate system and a bounding box slightly too large when not colinear.
The bounding box is defined as follow, for a Facoid:

$$\left(\overrightarrow{P'},\left\{\overrightarrow{A'_i}\right\}_{i\in I}\right) \tag{19}$$

where

$$\left\{\begin{array}{l} P'_i=P_i+\sum_{j\in I^-}A_{ji} \\ A'_{ij}=0.0, i\neq j \\ A'_{ij}=\sum_{k\in I^+}A_{kj}-\sum_{k\in I^-}A_{kj}, i=j \end{array}\right. \tag{20}$$

and, I^+ and I^- are the subsets of I such as $\forall j\in I^+, A_{ij}\geq 0.0$ and $\forall j\in I^-, A_{ij}<0.0$.

for a Pyramidoid:

$$\left(\overrightarrow{P'},\left\{\overrightarrow{A'_i}\right\}_{i\in I}\right) \tag{21}$$

where

$$\left\{\begin{array}{l} P'_i=P_i+Min\left(Min_{j\in I}(A_{ji}),0.0\right) \\ A'_{ij}=0.0, i\neq j \\ A'_{ij}=Max_{k\in I}(A_{kj})-Min\left(Min_{k\in I}(A_{kj}),0.0\right), i=j \end{array}\right. \tag{22}$$

1.2.6 Depth and Center

Depth $\mathbf{D}_S(\overrightarrow{X})$ of position $\overrightarrow{X}$ a Shapoid S is a value ranging from 0.0 if $\overrightarrow{X}$ is on the surface of the Shapoid, to 1.0 if $\overrightarrow{X}$ is at the farthest location from the surface inside the Shapoid. Depth is by definition equal to 0.0 if $\overrightarrow{X}$ is outside the Shapoid. Depth is continuous and derivable on the volume of the Shapoid. It is defined by, for a Facoid:

$$\mathbf{D}_S(\overrightarrow{X})=\prod_{i\in I}\left(1.0-4.0*(0.5-X_i^S)^2\right) \tag{23}$$

for a Pyramidoid:

$$\mathbf{D}_S(\overrightarrow{X}) = \prod_{i \in I} \left(1.0 - 4.0 * \left(0.5 - \frac{X_i^S}{1.0 - \sum_{j \in I - \{i\}} X_j^S} \right)^2 \right) \tag{24}$$

and for a Spheroid:

$$\mathbf{D}_S(\overrightarrow{X}) = 1.0 - 2.0 * \left\| \overrightarrow{X^S} \right\| \tag{25}$$

The maximum depth is obtained at $\overrightarrow{C}$ such as, for a Facoid:

$$\forall i \in I, C_i^S = 0.5 \tag{26}$$

for a Pyramidoid:

$$\forall i \in I, C_i^S = \frac{1}{D+1} \tag{27}$$

for a Spheroid:

$$\forall i \in I, C_i^S = 0.0 \tag{28}$$

$\overrightarrow{C}$ is called the center of the Shapoid.

2 Interface

```
// ============ PBMATH.H ================

#ifndef PBMATH_H
#define PBMATH_H

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

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "gset.h"
struct VecFloat;
typedef struct VecFloat VecFloat;
struct Shapoid;
typedef struct Shapoid Shapoid;
struct SCurve;
typedef struct SCurve SCurve;
#include "bcurve.h"

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

#define PBMATH_EPSILON 0.0000001
#define PBMATH_TWOPI 6.28319
```

```
#define PBMATH_PI 3.14159
#define PBMATH_HALFPI 1.57080
#define PBMATH_QUARTERPI 0.78540

// ================= Generic functions ==================

void VecTypeUnsupported(void*t, ...);
#define VecClone(V) _Generic((V), \
  VecFloat*: VecFloatClone, \
  VecShort*: VecShortClone, \
  default: VecTypeUnsupported)(V)
#define VecLoad(V, S) _Generic((V), \
  VecFloat**: VecFloatLoad, \
  VecShort**: VecShortLoad, \
  default: VecTypeUnsupported)(V, S)
#define VecSave(V, S) _Generic((V), \
  VecFloat*: VecFloatSave, \
  VecShort*: VecShortSave, \
  default: VecTypeUnsupported)(V, S)
#define VecFree(V) _Generic((V), \
  VecFloat**: VecFloatFree, \
  VecShort**: VecShortFree, \
  default: VecTypeUnsupported)(V)
#define VecPrint(V, S) _Generic((V), \
  VecFloat*: VecFloatPrintDef, \
  VecShort*: VecShortPrint, \
  default: VecTypeUnsupported)(V, S)
#define VecGet(V, I) _Generic((V), \
  VecFloat*: VecFloatGet, \
  VecShort*: VecShortGet, \
  default: VecTypeUnsupported)(V, I)
#define VecSet(V, I, VAL) _Generic((V), \
  VecFloat*: VecFloatSet, \
  VecShort*: VecShortSet, \
  default: VecTypeUnsupported)(V, I, VAL)
#define VecCopy(V, W) _Generic((V), \
  VecFloat*: VecFloatCopy, \
  VecShort*: VecShortCopy, \
  default: VecTypeUnsupported)(V, W)
#define VecDim(V) _Generic((V), \
  VecFloat*: VecFloatDim, \
  VecShort*: VecShortDim, \
  default: VecTypeUnsupported)(V)
#define VecNorm(V) _Generic((V), \
  VecFloat*: VecFloatNorm, \
  default: VecTypeUnsupported)(V)
#define VecNormalise(V) _Generic((V), \
  VecFloat*: VecFloatNormalise, \
  default: VecTypeUnsupported)(V)
#define VecDist(V, W) _Generic((V), \
  VecFloat*: VecFloatDist, \
  VecShort*: VecShortHamiltonDist, \
  default: VecTypeUnsupported)(V, W)
#define VecHamiltonDist(V, W) _Generic((V), \
  VecFloat*: VecFloatHamiltonDist, \
  VecShort*: VecShortHamiltonDist, \
  default: VecTypeUnsupported)(V, W)
#define VecIsEqual(V, W) _Generic((V), \
  VecFloat*: VecFloatIsEqual, \
  default: VecTypeUnsupported)(V, W)
#define VecOp(V, A, W, B) _Generic((V), \
  VecFloat*: VecFloatOp, \
```

```
  default: VecTypeUnsupported)(V, A, W, B)
#define VecGetOp(V, A, W, B) _Generic((V), \
  VecFloat*: VecFloatGetOp, \
  default: VecTypeUnsupported)(V, A, W, B)
#define VecRot2D(V, A) _Generic((V), \
  VecFloat*: VecFloatRot2D, \
  default: VecTypeUnsupported)(V, A)
#define VecGetRot2D(V, A) _Generic((V), \
  VecFloat*: VecFloatGetRot2D, \
  default: VecTypeUnsupported)(V, A)
#define VecDotProd(V, W) _Generic((V), \
  VecFloat*: VecFloatDotProd, \
  default: VecTypeUnsupported)(V, W)
#define VecAngleTo2D(V, W) _Generic((V), \
  VecFloat*: VecFloatAngleTo2D, \
  default: VecTypeUnsupported)(V, W)

void MatTypeUnsupported(void*t, ...);
#define MatClone(M) _Generic((M), \
  MatFloat*: MatFloatClone, \
  default: MatTypeUnsupported)(M)
#define MatLoad(M, S) _Generic((M), \
  MatFloat**: MatFloatLoad, \
  default: MatTypeUnsupported)(M, S)
#define MatSave(M, S) _Generic((M), \
  MatFloat*: MatFloatSave, \
  default: MatTypeUnsupported)(M, S)
#define MatFree(M) _Generic((M), \
  MatFloat**: MatFloatFree, \
  default: MatTypeUnsupported)(M)
#define MatPrint(M, S) _Generic((M), \
  MatFloat*: MatFloatPrintDef, \
  default: MatTypeUnsupported)(M, S)
#define MatGet(M, I) _Generic((M), \
  MatFloat*: MatFloatGet, \
  default: MatTypeUnsupported)(M, I)
#define MatSet(M, I, VAL) _Generic((M), \
  MatFloat*: MatFloatSet, \
  default: MatTypeUnsupported)(M, I, VAL)
#define MatCopy(M, W) _Generic((M), \
  MatFloat*: MatFloatCopy, \
  default: MatTypeUnsupported)(M, W)
#define MatDim(M) _Generic((M), \
  MatFloat*: MatFloatDim, \
  default: MatTypeUnsupported)(M)
#define MatInv(M) _Generic((M), \
  MatFloat*: MatFloatInv, \
  default: MatTypeUnsupported)(M)
#define MatProd(A, B) _Generic(A, \
  MatFloat*: _Generic(B, \
    VecFloat*: MatFloatProdVecFloat, \
    MatFloat*: MatFloatProdMatFloat, \
    default: MatTypeUnsupported), \
  default: MatTypeUnsupported)(A, B)
#define MatSetIdentity(M) _Generic((M), \
  MatFloat*: MatFloatSetIdentity, \
  default: MatTypeUnsupported)(M)

void LinSysTypeUnsupported(void*t, ...);
#define LinSysFree(S) _Generic((S), \
  EqLinSys*: EqLinSysFree, \
  default: LinSysTypeUnsupported)(S)
```

```
#define LinSysSolve(S) _Generic((S), \
  EqLinSys*: EqLinSysSolve, \
  default: LinSysTypeUnsupported)(S)

void ShapoidGetBoundingBoxUnsupported(void*t, ...);
#define ShapoidGetBoundingBox(T) _Generic((T), \
  Shapoid*: ShapoidGetBoundingBoxThat, \
  GSet*: ShapoidGetBoundingBoxSet, \
  default: ShapoidGetBoundingBoxUnsupported)(T)


// ------------- VecShort

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

// Vector of short values
typedef struct VecShort {
  // Dimension
  int _dim;
  // Values
  short *_val;
} VecShort;

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

// Create a new VecShort of dimension 'dim'
// Values are initalized to 0.0
// Return NULL if we couldn't create the VecShort
VecShort* VecShortCreate(int dim);

// Clone the VecShort
// Return NULL if we couldn't clone the VecShort
VecShort* VecShortClone(VecShort *that);

// Load the VecShort from the stream
// If the VecShort is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int VecShortLoad(VecShort **that, FILE *stream);

// Save the VecShort to the stream
// Return 0 upon success, or
// 1: invalid arguments
// 2: fprintf error
int VecShortSave(VecShort *that, FILE *stream);

// Free the memory used by a VecShort
// Do nothing if arguments are invalid
void VecShortFree(VecShort **that);

// Print the VecShort on 'stream'
// Do nothing if arguments are invalid
void VecShortPrint(VecShort *that, FILE *stream);

// Return the i-th value of the VecShort
// Index starts at 0
// Return 0.0 if arguments are invalid
short VecShortGet(VecShort *that, int i);
```

```
// Set the i-th value of the VecShort to v
// Index starts at 0
// Do nothing if arguments are invalid
void VecShortSet(VecShort *that, int i, short v);

// Return the dimension of the VecShort
// Return 0 if arguments are invalid
int VecShortDim(VecShort *that);

// Return the Hamiltonian distance between the VecShort 'that' and 'tho'
// Return -1 if arguments are invalid
// If dimensions are different, missing ones are considered to
// be equal to 0
short VecShortHamiltonDist(VecShort *that, VecShort *tho);

// Copy the values of 'w' in 'that' (must have same dimensions)
// Do nothing if arguments are invalid
void VecShortCopy(VecShort *that, VecShort *w);

// ------------- VecFloat

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

// Vector of float values
typedef struct VecFloat {
  // Dimension
  int _dim;
  // Values
  float *_val;
} VecFloat;

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

// Create a new VecFloat of dimension 'dim'
// Values are initalized to 0.0
// Return NULL if we couldn't create the VecFloat
VecFloat* VecFloatCreate(int dim);

// Clone the VecFloat
// Return NULL if we couldn't clone the VecFloat
VecFloat* VecFloatClone(VecFloat *that);

// Load the VecFloat from the stream
// If the VecFloat is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int VecFloatLoad(VecFloat **that, FILE *stream);

// Save the VecFloat to the stream
// Return 0 upon success, or
// 1: invalid arguments
// 2: fprintf error
int VecFloatSave(VecFloat *that, FILE *stream);

// Free the memory used by a VecFloat
// Do nothing if arguments are invalid
void VecFloatFree(VecFloat **that);

// Print the VecFloat on 'stream' with 'prec' digit precision
```

```
// Do nothing if arguments are invalid
void VecFloatPrint(VecFloat *that, FILE *stream, int prec);
void VecFloatPrintDef(VecFloat *that, FILE *stream);

// Return the 'i'-th value of the VecFloat
// Index starts at 0
// Return 0.0 if arguments are invalid
float VecFloatGet(VecFloat *that, int i);

// Set the 'i'-th value of the VecFloat to 'v'
// Index starts at 0
// Do nothing if arguments are invalid
void VecFloatSet(VecFloat *that, int i, float v);

// Return the dimension of the VecFloat
// Return 0 if arguments are invalid
int VecFloatDim(VecFloat *that);

// Copy the values of 'w' in 'that' (must have same dimensions)
// Do nothing if arguments are invalid
void VecFloatCopy(VecFloat *that, VecFloat *w);

// Return the norm of the VecFloat
// Return 0.0 if arguments are invalid
float VecFloatNorm(VecFloat *that);

// Normalise the VecFloat
// Do nothing if arguments are invalid
void VecFloatNormalise(VecFloat *that);

// Return the distance between the VecFloat 'that' and 'tho'
// Return NaN if arguments are invalid
// If dimensions are different, missing ones are considered to
// be equal to 0.0
float VecFloatDist(VecFloat *that, VecFloat *tho);

// Return the Hamiltonian distance between the VecFloat 'that' and 'tho'
// Return NaN if arguments are invalid
// If dimensions are different, missing ones are considered to
// be equal to 0.0
float VecFloatHamiltonDist(VecFloat *that, VecFloat *tho);

// Return true if the VecFloat 'that' is equal to 'tho'
// Return false if arguments are invalid
// If dimensions are different, missing ones are considered to
// be equal to 0.0
bool VecFloatIsEqual(VecFloat *that, VecFloat *tho);

// Calculate (that * a + tho * b) and store the result in 'that'
// Do nothing if arguments are invalid
// 'tho' can be null, in which case it is consider to be the null vector
// If 'tho' is not null it must be of same dimension as 'that'
void VecFloatOp(VecFloat *that, float a, VecFloat *tho, float b);

// Return a VecFloat equal to (that * a + tho * b)
// Return NULL if arguments are invalid
// 'tho' can be null, in which case it is consider to be the null vector
// If 'tho' is not null it must be of same dimension as 'that'
VecFloat* VecFloatGetOp(VecFloat *that, float a,
  VecFloat *tho, float b);

// Rotate CCW 'that' by 'theta' radians and store the result in 'that'
```

```
// Do nothing if arguments are invalid
void VecFloatRot2D(VecFloat *that, float theta);

// Return a VecFloat equal to 'that' rotated CCW by 'theta' radians
// Return NULL if arguments are invalid
VecFloat* VecFloatGetRot2D(VecFloat *that, float theta);

// Return the dot product of 'that' and 'tho'
// Return 0.0 if arguments are invalid
float VecFloatDotProd(VecFloat *that, VecFloat *tho);

// Return the angle of the rotation making 'that' colinear to 'tho'
// Return 0.0 if arguments are invalid
float VecFloatAngleTo2D(VecFloat *that, VecFloat *tho);

// -------------- MatFloat

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

// Vector of float values
typedef struct MatFloat {
  // Dimension
  VecShort *_dim;
  // Values (memorized by lines)
  float *_val;
} MatFloat;

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

// Create a new MatFloat of dimension 'dim' (nbcol, nbline)
// Values are initalized to 0.0, 'dim' must be a VecShort of dimension 2
// Return NULL if we couldn't create the MatFloat
MatFloat* MatFloatCreate(VecShort *dim);

// Set the MatFloat to the identity matrix
// The matrix must be a square matrix
// Do nothing if arguments are invalid
void MatFloatSetIdentity(MatFloat *that);

// Clone the MatFloat
// Return NULL if we couldn't clone the MatFloat
MatFloat* MatFloatClone(MatFloat *that);

// Load the MatFloat from the stream
// If the MatFloat is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int MatFloatLoad(MatFloat **that, FILE *stream);

// Save the MatFloat to the stream
// Return 0 upon success, or
// 1: invalid arguments
// 2: fprintf error
int MatFloatSave(MatFloat *that, FILE *stream);

// Free the memory used by a MatFloat
// Do nothing if arguments are invalid
void MatFloatFree(MatFloat **that);
```

```
// Print the MatFloat on 'stream' with 'prec' digit precision
// Do nothing if arguments are invalid
void MatFloatPrint(MatFloat *that, FILE *stream, int prec);
void MatFloatPrintDef(MatFloat *that, FILE *stream);

// Return the value at index 'i' of the MatFloat
// Index starts at 0, i must be a VecShort of dimension 2
// Return 0.0 if arguments are invalid
float MatFloatGet(MatFloat *that, VecShort *i);

// Set the value at index 'i' of the MatFloat to 'v'
// Index starts at 0, 'i' must be a VecShort of dimension 2
// Do nothing if arguments are invalid
void MatFloatSet(MatFloat *that, VecShort *i, float v);

// Return a VecShort of dimension 2 containing the dimension of
// the MatFloat
// Return NULL if arguments are invalid
VecShort* MatFloatDim(MatFloat *that);

// Return the inverse matrix of 'that'
// The matrix must be a square matrix
// Return null if arguments are invalids
MatFloat* MatFloatInv(MatFloat *that);

// Return the product of matrix 'that' and vector 'v'
// Number of columns of 'that' must equal dimension of 'v'
// Return null if arguments are invalids
VecFloat* MatFloatProdVecFloat(MatFloat *that, VecFloat *v);

// Return the product of matrix 'that' by matrix 'tho'
// Number of columns of 'that' must equal number of line of 'tho'
// Return null if arguments are invalids
MatFloat* MatFloatProdMatFloat(MatFloat *that, MatFloat *tho);

// ------------- Gauss

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

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

// Vector of float values
typedef struct Gauss {
  // Mean
  float _mean;
  // Sigma
  float _sigma;
} Gauss;

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

// Create a new Gauss of mean 'mean' and sigma 'sigma'
// Return NULL if we couldn't create the Gauss
Gauss* GaussCreate(float mean, float sigma);

// Free the memory used by a Gauss
// Do nothing if arguments are invalid
void GaussFree(Gauss **that);

// Return the value of the Gauss 'that' at 'x'
// Return 0.0 if the arguments are invalid
float GaussGet(Gauss *that, float x);
```

```
// Return a random value according to the Gauss 'that'
// random() must have been called before calling this function
// Return 0.0 if the arguments are invalid
float GaussRnd(Gauss *that);

// -------------- Smoother

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

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

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

// Return the order 1 smooth value of 'x'
// if x < 0.0 return 0.0
// if x > 1.0 return 1.0
float SmoothStep(float x);

// Return the order 2 smooth value of 'x'
// if x < 0.0 return 0.0
// if x > 1.0 return 1.0
float SmootherStep(float x);

// -------------- Shapoid

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

#define SpheroidCreate(D) ShapoidCreate(D, ShapoidTypeSpheroid)
#define FacoidCreate(D) ShapoidCreate(D, ShapoidTypeFacoid)
#define PyramidoidCreate(D) ShapoidCreate(D, ShapoidTypePyramidoid)

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

typedef enum ShapoidType {
  ShapoidTypeInvalid, ShapoidTypeFacoid, ShapoidTypeSpheroid,
  ShapoidTypePyramidoid
} ShapoidType;
// Don't forget to update ShapoidTypeString in pbmath.c when adding
// new type

typedef struct Shapoid {
  // Position of origin
  VecFloat *_pos;
  // Dimension
  int _dim;
  // Vectors defining faces
  VecFloat **_axis;
  // Type of Shapoid
  ShapoidType _type;
} Shapoid;

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

// Create a Shapoid of dimension 'dim' and type 'type', default values:
// _pos = null vector
// _axis[d] = unit vector along dimension d
// Return NULL if arguments are invalid or malloc failed
Shapoid* ShapoidCreate(int dim, ShapoidType type);

// Clone a Shapoid
// Return NULL if couldn't clone
```

```
Shapoid* ShapoidClone(Shapoid *that);

// Free memory used by a Shapoid
// Do nothing if arguments are invalid
void ShapoidFree(Shapoid **that);

// Load the Shapoid from the stream
// If the VecFloat is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int ShapoidLoad(Shapoid **that, FILE *stream);

// Save the Shapoid to the stream
// Return 0 upon success, or
// 1: invalid arguments
// 2: fprintf error
int ShapoidSave(Shapoid *that, FILE *stream);

// Print the Shapoid on 'stream'
// Do nothing if arguments are invalid
void ShapoidPrint(Shapoid *that, FILE *stream);

// Get the dimension of the Shapoid
// Return 0 if arguments are invalid
int ShapoidGetDim(Shapoid *that);

// Get the type of the Shapoid
// Return ShapoidTypeInvalid if arguments are invalid
ShapoidType ShapoidGetType(Shapoid *that);

// Get the type of the Shapoid as a string
// Return a pointer to a constant string (not to be freed)
// Return the string for ShapoidTypeInvalid if arguments are invalid
const char* ShapoidGetTypeAsString(Shapoid *that);

// Return a VecFloat equal to the position of the Shapoid
// Return NULL if arguments are invalid
VecFloat* ShapoidGetPos(Shapoid *that);

// Return a VecFloat equal to the 'dim'-th axis of the Shapoid
// Return NULL if arguments are invalid
VecFloat* ShapoidGetAxis(Shapoid *that, int dim);

// Set the position of the Shapoid to 'pos'
// Do nothing if arguments are invalid
void ShapoidSetPos(Shapoid *that, VecFloat *pos);

// Set the 'dim'-th axis of the Shapoid to 'v'
// Do nothing if arguments are invalid
void ShapoidSetAxis(Shapoid *that, int dim, VecFloat *v);

// Translate the Shapoid by 'v'
// Do nothing if arguments are invalid
void ShapoidTranslate(Shapoid *that, VecFloat *v);

// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
// Do nothing if arguments are invalid
void ShapoidScale(Shapoid *that, VecFloat *v);
```

```
// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
// and translate the Shapoid such as its center after scaling
// is at the same position than before scaling
// Do nothing if arguments are invalid
void ShapoidGrow(Shapoid *that, VecFloat *v);

// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// Do nothing if arguments are invalid
void ShapoidRotate2D(Shapoid *that, float theta);

// Convert the coordinates of 'pos' from standard coordinate system
// toward the Shapoid coordinates system
// Return null if the arguments are invalid
VecFloat* ShapoidImportCoord(Shapoid *that, VecFloat *pos);

// Convert the coordinates of 'pos' from the Shapoid coordinates system
// toward standard coordinate system
// Return null if the arguments are invalid
VecFloat* ShapoidExportCoord(Shapoid *that, VecFloat *pos);

// Return true if 'pos' is inside the Shapoid
// Else return false
bool ShapoidIsPosInside(Shapoid *that, VecFloat *pos);

// Get a bounding box of the Shapoid. The bounding box is aligned
// on the standard coordinate system (its axis are colinear with
// the axis of the standard coordinate system).
// The bounding box is returned as a Facoid, which position is
// at the minimum value along each axis.
// Return null if the argument are invalid.
Shapoid* ShapoidGetBoundingBoxThat(Shapoid *that);

// Get the bounding box of a set of Facoid. The bounding box is aligned
// on the standard coordinate system (its axis are colinear with
// the axis of the standard coordinate system).
// The bounding box is returned as a Facoid, which position is
// at the minimum value along each axis.
// Return null if the arguments are invalid or the shapoid in the set
// don't have all the same dimension.
Shapoid* ShapoidGetBoundingBoxSet(GSet *set);

// Get a SCurve approximating the Shapoid 'that'
// 'that' must be of dimension 2
// Return null if arguments are invalid
SCurve* Shapoid2SCurve(Shapoid *that);

// Get the depth value in the Shapoid of 'pos'
// The depth is defined as follow: the point with depth equals 1.0 is
// the farthest point from the surface of the Shapoid (inside it),
// points with depth equals to 0.0 are point on the surface of the
// Shapoid. Depth is continuous and derivable over the volume of the
// Shapoid
// Return 0.0 if arguments are invalid, or pos is outside the Shapoid
float ShapoidGetPosDepth(Shapoid *that, VecFloat *pos);

// Get the center of the shapoid in standard coordinate system
// Return null if arguments are invalid
VecFloat* ShapoidGetCenter(Shapoid *that);

// Get the percentage of 'tho' included 'that' (in [0.0, 1.0])
// 0.0 -> 'tho' is completely outside of 'that'
// 1.0 -> 'tho' is completely inside of 'that'
```

```
// 'that' and 'tho' must me of same dimensions
// Return 0.0 if the arguments are invalid or something went wrong
float ShapoidGetCoverageRatio(Shapoid *that, Shapoid *tho);

// -------------- Conversion functions

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

// Convert radians to degrees
float ConvRad2Deg(float rad);

// Convert degrees to radians
float ConvDeg2Rad(float deg);

// -------------- EqLinSys

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

// Linear system of equalities
typedef struct EqLinSys {
  // Matrix
  MatFloat *_M;
  // Vector
  VecFloat *_V;
} EqLinSys;

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

// Create a new EqLinSys with matrix 'm' and vector 'v'
// The dimension of 'v' must be equal to the number of column of 'm'
// The matrix 'm' must be a square matrix
// Return NULL if we couldn't create the EqLinSys
EqLinSys* EqLinSysCreate(MatFloat *m, VecFloat *v);

// Free the memory used by the EqLinSys
// Do nothing if arguments are invalid
void EqLinSysFree(EqLinSys **that);

// Solve the EqLinSys _M.x = _V
// Return the solution vector, or null if there is no solution or the
// arguments are invalid
VecFloat* EqLinSysSolve(EqLinSys *that);

#endif
```

3 Code

```
// ============ PBMATH.C ================

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

#include "pbmath.h"

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

#define rnd() (float)(rand())/(float)(RAND_MAX)

// -------------- VecShort
```

```
// ================= Define ==================

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

// Create a new Vec of dimension 'dim'
// Values are initalized to 0.0
// Return NULL if we couldn't create the Vec
VecShort* VecShortCreate(int dim) {
  // Check argument
  if (dim <= 0)
    return NULL;
  // Allocate memory
  VecShort *that = (VecShort*)malloc(sizeof(VecShort));
  //If we could allocate memory
  if (that != NULL) {
    // Allocate memory for values
    that->_val = (short*)malloc(sizeof(short) * dim);
    // If we couldn't allocate memory
    if (that->_val == NULL) {
      // Free memory
      free(that);
      // Stop here
      return NULL;
    }
    // Set the default values
    that->_dim = dim;
    for (int i = dim; i--;)
      that->_val[i] = 0.0;
  }
  // Return the new VecShort
  return that;
}

// Clone the VecShort
// Return NULL if we couldn't clone the VecShort
VecShort* VecShortClone(VecShort *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Create a clone
  VecShort *clone = VecShortCreate(that->_dim);
  // If we could create the clone
  if (clone != NULL) {
    // Clone the properties
    for (int i = that->_dim; i--;)
      clone->_val[i] = that->_val[i];
  }
  // Return the clone
  return clone;
}

// Load the VecShort from the stream
// If the VecShort is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int VecShortLoad(VecShort **that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
```

```
  // If 'that' is already allocated
  if (*that != NULL) {
    // Free memory
    VecShortFree(that);
  }
  // Read the number of dimension
  int dim;
  int ret = fscanf(stream, "%d", &dim);
  // If we coudln't fscanf
  if (ret == EOF)
    return 4;
  if (dim <= 0)
    return 3;
  // Allocate memory
  *that = VecShortCreate(dim);
  // If we coudln't allocate memory
  if (*that == NULL) {
    return 2;
  }
  // Read the values
  for (int i = 0; i < dim; ++i) {
    ret = fscanf(stream, "%hi", (*that)->_val + i);
    // If we coudln't fscanf
    if (ret == EOF)
      return 4;
  }
  // Return success code
  return 0;
}

// Save the VecShort to the stream
// Return 0 upon success, or:
// 1: invalid arguments
// 2: fprintf error
int VecShortSave(VecShort *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // Save the dimension
  int ret = fprintf(stream, "%d ", that->_dim);
  // If we coudln't fprintf
  if (ret < 0)
    return 2;
  // Save the values
  for (int i = 0; i < that->_dim; ++i) {
    ret = fprintf(stream, "%hi ", that->_val[i]);
    // If we coudln't fprintf
    if (ret < 0)
      return 2;
  }
  fprintf(stream, "\n");
  // If we coudln't fprintf
  if (ret < 0)
    return 2;
  // Return success code
  return 0;
}

// Free the memory used by a VecShort
// Do nothing if arguments are invalid
void VecShortFree(VecShort **that) {
  // Check argument
```

```
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  free((*that)->_val);
  free(*that);
  *that = NULL;
}

// Print the VecShort on 'stream' with 'prec' digit precision
// Do nothing if arguments are invalid
void VecShortPrint(VecShort *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return;
  // Print the values
  fprintf(stream, "<");
  for (int i = 0; i < that->_dim; ++i) {
    fprintf(stream, "%hi", that->_val[i]);
    if (i < that->_dim - 1)
      fprintf(stream, ",");
  }
  fprintf(stream, ">");
}

// Return the i-th value of the VecShort
// Index starts at 0
// Return 0.0 if arguments are invalid
short VecShortGet(VecShort *that, int i) {
  // Check argument
  if (that == NULL || i < 0 || i >= that->_dim)
    return 0.0;
  // Return the value
  return that->_val[i];
}

// Set the i-th value of the VecShort to v
// Index starts at 0
// Do nohting if arguments are invalid
void VecShortSet(VecShort *that, int i, short v) {
  // Check argument
  if (that == NULL || i < 0 || i >= that->_dim)
    return;
  // Set the value
  that->_val[i] = v;
}

// Return the dimension of the VecShort
// Return 0 if arguments are invalid
int VecShortDim(VecShort *that) {
  // Check argument
  if (that == NULL)
    return 0;
  // Return the dimension
  return that->_dim;
}

// Return the Hamiltonian distance between the VecShort 'that' and 'tho'
// Return -1 if arguments are invalid
// If dimensions are different, missing ones are considered to
// be equal to 0
short VecShortHamiltonDist(VecShort *that, VecShort *tho) {
  // Check argument
```

```
  if (that == NULL || tho == NULL)
    return -1;
  // Declare a variable to calculate the distance
  short ret = 0;
  for (int iDim = that->_dim; iDim--;)
    ret += (short)floor(fabs(VecGet(that, iDim) - VecGet(tho, iDim)));
  // Return the distance
  return ret;
}

// Copy the values of 'w' in 'that' (must have same dimensions)
// Do nothing if arguments are invalid
void VecShortCopy(VecShort *that, VecShort *w) {
  // Check argument
  if (that == NULL || w == NULL || that->_dim != w->_dim)
    return;
  // Copy the values
  memcpy(that->_val, w->_val, sizeof(short) * that->_dim);
}

// -------------- VecFloat

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

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

// Create a new Vec of dimension 'dim'
// Values are initalized to 0.0
// Return NULL if we couldn't create the Vec
VecFloat* VecFloatCreate(int dim) {
  // Check argument
  if (dim <= 0)
    return NULL;
  // Allocate memory
  VecFloat *that = (VecFloat*)malloc(sizeof(VecFloat));
  //If we could allocate memory
  if (that != NULL) {
    // Allocate memory for values
    that->_val = (float*)malloc(sizeof(float) * dim);
    // If we couldn't allocate memory
    if (that->_val == NULL) {
      // Free memory
      free(that);
      // Stop here
      return NULL;
    }
    // Set the default values
    that->_dim = dim;
    for (int i = dim; i--;)
      that->_val[i] = 0.0;
  }
  // Return the new VecFloat
  return that;
}

// Clone the VecFloat
// Return NULL if we couldn't clone the VecFloat
VecFloat* VecFloatClone(VecFloat *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Create a clone
```

```
  VecFloat *clone = VecFloatCreate(that->_dim);
  // If we could create the clone
  if (clone != NULL) {
    // Clone the properties
    for (int i = that->_dim; i--;)
      clone->_val[i] = that->_val[i];
  }
  // Return the clone
  return clone;
}

// Load the VecFloat from the stream
// If the VecFloat is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int VecFloatLoad(VecFloat **that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // If 'that' is already allocated
  if (*that != NULL) {
    // Free memory
    VecFloatFree(that);
  }
  // Read the number of dimension
  int dim;
  int ret = fscanf(stream, "%d", &dim);
  // If we coudln't fscanf
  if (ret == EOF)
    return 4;
  if (dim <= 0)
    return 3;
  // Allocate memory
  *that = VecFloatCreate(dim);
  // If we coudln't allocate memory
  if (*that == NULL) {
    return 2;
  }
  // Read the values
  for (int i = 0; i < dim; ++i) {
    ret = fscanf(stream, "%f", (*that)->_val + i);
    // If we coudln't fscanf
    if (ret == EOF)
      return 4;
  }
  // Return success code
  return 0;
}

// Save the VecFloat to the stream
// Return 0 upon success, or:
// 1: invalid arguments
// 2: fprintf error
int VecFloatSave(VecFloat *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // Save the dimension
  int ret = fprintf(stream, "%d ", that->_dim);
```

```
  // If we coudln't fprintf
  if (ret < 0)
    return 2;
  // Save the values
  for (int i = 0; i < that->_dim; ++i) {
    ret = fprintf(stream, "%f ", that->_val[i]);
    // If we coudln't fprintf
    if (ret < 0)
      return 2;
  }
  fprintf(stream, "\n");
  // If we coudln't fprintf
  if (ret < 0)
    return 2;
  // Return success code
  return 0;
}

// Free the memory used by a VecFloat
// Do nothing if arguments are invalid
void VecFloatFree(VecFloat **that) {
  // Check argument
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  free((*that)->_val);
  free(*that);
  *that = NULL;
}

// Print the VecFloat on 'stream' with 'prec' digit precision
// Do nothing if arguments are invalid
void VecFloatPrint(VecFloat *that, FILE *stream, int prec) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return;
  // Create the format string
  char format[20] = {'\0'};
  sprintf(format, "%%.%df", prec);
  // Print the values
  fprintf(stream, "<");
  for (int i = 0; i < that->_dim; ++i) {
    fprintf(stream, format, that->_val[i]);
    if (i < that->_dim - 1)
      fprintf(stream, ",");
  }
  fprintf(stream, ">");
}
void VecFloatPrintDef(VecFloat *that, FILE *stream) {
  VecFloatPrint(that, stream, 3);
}

// Return the i-th value of the VecFloat
// Index starts at 0
// Return 0.0 if arguments are invalid
float VecFloatGet(VecFloat *that, int i) {
  // Check argument
  if (that == NULL || i < 0 || i >= that->_dim)
    return 0.0;
  // Return the value
  return that->_val[i];
}
```

```
// Set the i-th value of the VecFloat to v
// Index starts at 0
// Do nohting if arguments are invalid
void VecFloatSet(VecFloat *that, int i, float v) {
  // Check argument
  if (that == NULL || i < 0 || i >= that->_dim)
    return;
  // Set the value
  that->_val[i] = v;
}

// Return the dimension of the VecFloat
// Return 0 if arguments are invalid
int VecFloatDim(VecFloat *that) {
  // Check argument
  if (that == NULL)
    return 0;
  // Return the dimension
  return that->_dim;
}

// Copy the values of 'w' in 'that' (must have same dimensions)
// Do nothing if arguments are invalid
void VecFloatCopy(VecFloat *that, VecFloat *w) {
  // Check argument
  if (that == NULL || w == NULL || that->_dim != w->_dim)
    return;
  // Copy the values
  memcpy(that->_val, w->_val, sizeof(float) * that->_dim);
}

// Return the norm of the VecFloat
// Return 0.0 if arguments are invalid
float VecFloatNorm(VecFloat *that) {
  // Check argument
  if (that == NULL)
    return 0.0;
  // Declare a variable to calculate the norm
  float ret = 0.0;
  // Calculate the norm
  for (int iDim = that->_dim; iDim--;)
    ret += pow(that->_val[iDim], 2.0);
  ret = sqrt(ret);
  // Return the result
  return ret;
}

// Normalise the VecFloat
// Do nothing if arguments are invalid
void VecFloatNormalise(VecFloat *that) {
  // Check argument
  if (that == NULL)
    return;
  // Normalise
  float norm = VecNorm(that);
  for (int iDim = that->_dim; iDim--;)
    that->_val[iDim] /= norm;
}

// Return the distance between the VecFloat 'that' and 'tho'
// Return NaN if arguments are invalid
```

```
// If dimensions are different, missing ones are considered to
// be equal to 0.0
float VecFloatDist(VecFloat *that, VecFloat *tho) {
  // Check argument
  if (that == NULL || tho == NULL)
    return NAN;
  // Declare a variable to calculate the distance
  float ret = 0.0;
  for (int iDim = that->_dim; iDim--;)
    ret += pow(VecGet(that, iDim) - VecGet(tho, iDim), 2.0);
  ret = sqrt(ret);
  // Return the distance
  return ret;
}

// Return the Hamiltonian distance between the VecFloat 'that' and 'tho'
// Return NaN if arguments are invalid
// If dimensions are different, missing ones are considered to
// be equal to 0.0
float VecFloatHamiltonDist(VecFloat *that, VecFloat *tho) {
  // Check argument
  if (that == NULL || tho == NULL)
    return NAN;
  // Declare a variable to calculate the distance
  float ret = 0.0;
  for (int iDim = that->_dim; iDim--;)
    ret += floor(fabs(VecGet(that, iDim) - VecGet(tho, iDim)));
  // Return the distance
  return ret;
}

// Return true if the VecFloat 'that' is equal to 'tho'
// Return false if arguments are invalid
// If dimensions are different, missing ones are considered to
// be equal to 0.0
bool VecFloatIsEqual(VecFloat *that, VecFloat *tho) {
  // Check argument
  if (that == NULL || tho == NULL)
    return false;
  // For each component
  for (int iDim = that->_dim; iDim--;)
    // If the values of this components are different
    if (fabs(VecGet(that, iDim) - VecGet(tho, iDim)) > PBMATH_EPSILON)
      // Return false
      return false;
  // Return true
  return true;
}

// Calculate (that * a + tho * b) and store the result in 'that'
// Do nothing if arguments are invalid
// 'tho' can be null, in which case it is consider to be the null vector
// If 'tho' is not null it must be of same dimension as 'that'
void VecFloatOp(VecFloat *that, float a, VecFloat *tho, float b) {
  // Check argument
  if (that == NULL)
    return;
  // Calculate
  VecFloat *res = VecFloatGetOp(that, a, tho, b);
  // If we could calculate
  if (res != NULL) {
    // Copy the result in 'that'
```

```
    VecFloatCopy(that, res);
    // Free memory
    VecFloatFree(&res);
  }
}

// Return a VecFloat equal to (that * a + tho * b)
// Return NULL if arguments are invalid
// 'tho' can be null, in which case it is consider to be the null vector
// If 'tho' is not null it must be of same dimension as 'that'
VecFloat* VecFloatGetOp(VecFloat *that, float a,
  VecFloat *tho, float b) {
  // Check argument
  if (that == NULL || (tho != NULL && that->_dim != tho->_dim))
    return NULL;
  // Declare a variable to memorize the result
  VecFloat *res = VecFloatCreate(that->_dim);
  // If we could allocate memory
  if (res != NULL) {
    // For each component
    for (int iDim = that->_dim; iDim--;) {
      // Calculate
      res->_val[iDim] = a * that->_val[iDim];
      if (tho != NULL)
        res->_val[iDim] += b * tho->_val[iDim];
    }
  }
  // Return the result
  return res;
}

// Rotate CCW 'that' by 'theta' radians and store the result in 'that'
// Do nothing if arguments are invalid
void VecFloatRot2D(VecFloat *that, float theta) {
  // Check argument
  if (that == NULL || that->_dim != 2)
    return;
  // Calculate
  VecFloat *res = VecFloatGetRot2D(that, theta);
  // If we could calculate
  if (res != NULL) {
    // Copy the result in 'that'
    VecFloatCopy(that, res);
    // Free memory
    VecFloatFree(&res);
  }
}

// Return a VecFloat equal to 'that' rotated CCW by 'theta' radians
// Return NULL if arguments are invalid
VecFloat* VecFloatGetRot2D(VecFloat *that, float theta) {
  // Check argument
  if (that == NULL || that->_dim != 2)
    return NULL;
  // Declare a variable to memorize the result
  VecFloat *res = VecFloatCreate(that->_dim);
  // If we could allocate memory
  if (res != NULL) {
    // Calculate
    res->_val[0] =
      cos(theta) * that->_val[0] - sin(theta) * that->_val[1];
    res->_val[1] =
```

```
      sin(theta) * that->_val[0] + cos(theta) * that->_val[1];
  }
  // Return the result
  return res;
}

// Return the dot product of 'that' and 'tho'
// Return 0.0 if arguments are invalid
float VecFloatDotProd(VecFloat *that, VecFloat *tho) {
  // Check arguments
  if (that == NULL || tho == NULL || that->_dim != tho->_dim)
    return 0.0;
  // Declare a variable to memorize the result
  float res = 0.0;
  // Calculate
  for (int iDim = that->_dim; iDim--;)
    res += that->_val[iDim] * tho->_val[iDim];
  // Return the result
  return res;
}

// Return the angle of the rotation making 'that' colinear to 'tho'
// Return 0.0 if arguments are invalid
float VecFloatAngleTo2D(VecFloat *that, VecFloat *tho) {
  // Check arguments
  if (that == NULL || tho == NULL ||
    VecDim(that) != 2 || VecDim(tho) != 2)
    return 0.0;
  // Declare a variable to memorize the result
  float theta = 0.0;
  // Calculate the angle
  VecFloat *v = VecClone(that);
  if (v == NULL)
    return 0.0;
  VecFloat *w = VecClone(tho);
  if (w == NULL) {
    VecFree(&v);
    return 0.0;
  }
  if (VecNorm(v) < PBMATH_EPSILON || VecNorm(v) < PBMATH_EPSILON) {
    VecFree(&v);
    VecFree(&w);
    return 0.0;
  }
  VecNormalise(v);
  VecNormalise(w);
  float m[2];
  if (fabs(VecGet(v, 0)) > fabs(VecGet(v, 1))) {
    m[0] = (VecGet(w, 0) + VecGet(w, 1) * VecGet(v, 1) / VecGet(v, 0))/
      (VecGet(v, 0) + pow(VecGet(v, 1), 2.0) / VecGet(v, 0));
    m[1] = (m[0] * VecGet(v, 1) - VecGet(w, 1)) / VecGet(v, 0);
  } else {
    m[1] = (VecGet(w, 0) - VecGet(w, 1) * VecGet(v, 0) / VecGet(v, 1))/
      (VecGet(v, 1) + pow(VecGet(v, 0), 2.0) / VecGet(v, 1));
    m[0] = (VecGet(w, 1) + m[1] * VecGet(v, 0)) / VecGet(v, 1);
  }
  // Due to numerical imprecision m[0] may be slightly out of [-1,1]
  // which makes acos return NaN, prevent this
  if (m[0] < -1.0)
    theta = PBMATH_PI;
  else if (m[0] > 1.0)
    theta = 0.0;
```

```
  else
    theta = acos(m[0]);
  if (sin(theta) * m[1] > 0.0)
    theta *= -1.0;
  // Free memory
  VecFree(&v);
  VecFree(&w);
  // Return the result
  return theta;
}

// ------------- MatFloat

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

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

// Create a new MatFloat of dimension 'dim' (nbcol, nbline)
// Values are initalized to 0.0, 'dim' must be a VecShort of dimension 2
// Return NULL if we couldn't create the MatFloat
MatFloat* MatFloatCreate(VecShort *dim) {
  // Check argument
  if (dim == NULL || VecDim(dim) != 2)
    return NULL;
  // Allocate memory
  MatFloat *that = (MatFloat*)malloc(sizeof(MatFloat));
  //If we could allocate memory
  if (that != NULL) {
    // Set the dimension
    that->_dim = VecClone(dim);
    if (that->_dim == NULL) {
      // Free memory
      free(that);
      // Stop here
      return NULL;
    }
    // Allocate memory for values
    int d = VecGet(dim, 0) * VecGet(dim, 1);
    that->_val = (float*)malloc(sizeof(float) * d);
    // If we couldn't allocate memory
    if (that->_val == NULL) {
      // Free memory
      free(that);
      // Stop here
      return NULL;
    }
    // Set the default values
    for (int i = d; i--;)
      that->_val[i] = 0.0;
  }
  // Return the new MatFloat
  return that;
}

// Set the MatFloat to the identity matrix
// The matrix must be a square matrix
// Do nothing if arguments are invalid
void MatFloatSetIdentity(MatFloat *that) {
  // Check argument
  if (that == NULL || VecGet(that->_dim, 0) != VecGet(that->_dim, 1))
    return;
  // Set the values
```

```
  VecShort *i = VecShortCreate(2);
  if (i != NULL) {
    for (VecSet(i, 0, 0); VecGet(i, 0) < VecGet(that->_dim, 0);
      VecSet(i, 0, VecGet(i, 0) + 1)) {
      for (VecSet(i, 1, 0); VecGet(i, 1) < VecGet(that->_dim, 1);
        VecSet(i, 1, VecGet(i, 1) + 1)) {
        if (VecGet(i, 0) == VecGet(i, 1))
          MatSet(that, i, 1.0);
        else
          MatSet(that, i, 0.0);
      }
    }
  }
  VecFree(&i);
}

// Clone the MatFloat
// Return NULL if we couldn't clone the MatFloat
MatFloat* MatFloatClone(MatFloat *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Create a clone
  MatFloat *clone = MatFloatCreate(that->_dim);
  // If we could create the clone
  if (clone != NULL) {
    // Clone the properties
    VecCopy(clone->_dim, that->_dim);
    int d = VecGet(that->_dim, 0) * VecGet(that->_dim, 1);
    for (int i = d; i--;)
      clone->_val[i] = that->_val[i];
  }
  // Return the clone
  return clone;
}

// Load the MatFloat from the stream
// If the MatFloat is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int MatFloatLoad(MatFloat **that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // If 'that' is already allocated
  if (*that != NULL) {
    // Free memory
    MatFloatFree(that);
  }
  // Read the number of dimension
  int dim[2];
  int ret = fscanf(stream, "%d %d", dim , dim + 1);
  // If we coudln't fscanf
  if (ret == EOF)
    return 4;
  if (dim[0] <= 0 || dim[1] <= 0)
    return 3;
  // Allocate memory
  VecShort *d = VecShortCreate(2);
```

```
  VecSet(d, 0, dim[0]);
  VecSet(d, 1, dim[1]);
  *that = MatFloatCreate(d);
  // If we coudln't allocate memory
  if (*that == NULL)
    return 2;
  // Read the values
  int nbVal = dim[0] * dim[1];
  for (int i = 0; i < nbVal; ++i) {
    ret = fscanf(stream, "%f", (*that)->_val + i);
    // If we coudln't fscanf
    if (ret == EOF)
      return 4;
  }
  // Free memory
  VecFree(&d);
  // Return success code
  return 0;
}

// Save the MatFloat to the stream
// Return 0 upon success, or
// 1: invalid arguments
// 2: fprintf error
int MatFloatSave(MatFloat *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // Save the dimension
  int ret = fprintf(stream, "%d %d\n", VecGet(that->_dim, 0),
    VecGet(that->_dim, 1));
  // If we coudln't fprintf
  if (ret < 0)
    return 2;
  // Save the values
  for (int i = 0; i < VecGet(that->_dim, 1); ++i) {
    for (int j = 0; j < VecGet(that->_dim, 0); ++j) {
      ret = fprintf(stream, "%f ",
        that->_val[i * VecGet(that->_dim, 0) + j]);
      // If we coudln't fprintf
      if (ret < 0)
        return 2;
    }
    ret = fprintf(stream, "\n");
    // If we coudln't fprintf
    if (ret < 0)
      return 2;
  }
  // Return success code
  return 0;
}

// Free the memory used by a MatFloat
// Do nothing if arguments are invalid
void MatFloatFree(MatFloat **that) {
  // Check argument
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  VecFree(&((*that)->_dim));
  free((*that)->_val);
  free(*that);
```

```
  *that = NULL;
}

// Print the MatFloat on 'stream' with 'prec' digit precision
// Do nothing if arguments are invalid
void MatFloatPrint(MatFloat *that, FILE *stream, int prec) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return;
  // Create the format string
  char format[20] = {'\0'};
  sprintf(format, "%%.%df", prec);
  // Print the values
  fprintf(stream, "[");
  for (int i = 0; i < VecGet(that->_dim, 1); ++i) {
    if (i > 0)
      fprintf(stream, " ");
    for (int j = 0; j < VecGet(that->_dim, 0); ++j) {
      fprintf(stream, format,
        that->_val[i * VecGet(that->_dim, 0) + j]);
      if (j < VecGet(that->_dim, 0) - 1)
        fprintf(stream, ",");
    }
    if (i < VecGet(that->_dim, 1) - 1)
      fprintf(stream, "\n");
  }
  fprintf(stream, "]");
}
void MatFloatPrintDef(MatFloat *that, FILE *stream) {
  MatFloatPrint(that, stream, 3);
}

// Return the value at index 'i' of the MatFloat
// Index starts at 0, i must be a VecShort of dimension 2
// Return 0.0 if arguments are invalid
float MatFloatGet(MatFloat *that, VecShort *i) {
  // Check argument
  if (that == NULL || i == NULL || VecDim(i) != 2 ||
    VecGet(i, 0) < 0 || VecGet(i, 0) >= VecGet(that->_dim, 0) ||
    VecGet(i, 1) < 0 || VecGet(i, 1) >= VecGet(that->_dim, 1))
    return 0.0;
  // Return the value
  return
    that->_val[VecGet(i, 1) * VecGet(that->_dim, 0) + VecGet(i, 0)];
}

// Set the value at index 'i' of the MatFloat to 'v'
// Index starts at 0, 'i' must be a VecShort of dimension 2
// Do nothing if arguments are invalid
void MatFloatSet(MatFloat *that, VecShort *i, float v) {
  // Check argument
  if (that == NULL || i == NULL || VecDim(i) != 2 ||
    VecGet(i, 0) < 0 || VecGet(i, 0) >= VecGet(that->_dim, 0) ||
    VecGet(i, 1) < 0 || VecGet(i, 1) >= VecGet(that->_dim, 1))
    return;
  // Set the value
  that->_val[VecGet(i, 1) * VecGet(that->_dim, 0) + VecGet(i, 0)] = v;
}

// Return a VecShort of dimension 2 containing the dimension of
// the MatFloat
// Return NULL if arguments are invalid
```

```
VecShort* MatFloatDim(MatFloat *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Return the dimension
  return VecClone(that->_dim);
}

// Return the inverse matrix of 'that'
// The matrix must be a square matrix
// Return null if arguments are invalids
MatFloat* MatFloatInv(MatFloat *that) {
  // Check arguments
  if (that == NULL || VecGet(that->_dim, 0) != VecGet(that->_dim, 1))
    return NULL;
  // Allocate memory for the pivot
  VecShort *pivot = VecShortCreate(2);
  if (pivot == NULL)
    return NULL;
  // Allocate memory for the result
  MatFloat *res = MatFloatCreate(that->_dim);
  // If we could allocate memory
  if (res != NULL) {
    // If the matrix is of dimension 1x1
    if (VecGet(that->_dim, 0) == 1) {
      MatSet(res, pivot, 1.0 / MatGet(that, pivot));
    // Else, the matrix dimension is greater than 1x1
    } else {
      // Set the result to the identity
      MatSetIdentity(res);
      // Clone the original matrix
      MatFloat *copy = MatClone(that);
      // If we couldn't clone
      if (copy == NULL) {
        MatFree(&res);
        return NULL;
      }
      // Allocate memory for the index to manipulate the matrix
      VecShort *index = VecShortCreate(2);
      // If we couldn't allocate memory
      if (index == NULL) {
        MatFree(&res);
        MatFree(&copy);
        return NULL;
      }
      // For each pivot
      for (VecSet(pivot, 0, 0), VecSet(pivot, 1, 0);
        VecGet(pivot, 0) < VecGet(that->_dim, 0);
        VecSet(pivot, 0, VecGet(pivot, 0) + 1),
        VecSet(pivot, 1, VecGet(pivot, 1) + 1)) {
        // If the pivot is null
        if (MatGet(copy, pivot) < PBMATH_EPSILON) {
          // Search a line where the value under the pivot is not null
          VecCopy(index, pivot);
          VecSet(index, 1, 0);
          while (VecGet(index, 1) < VecGet(that->_dim, 1) &&
            fabs(MatGet(copy, index)) < PBMATH_EPSILON)
            VecSet(index, 1, VecGet(index, 1) + 1);
          // If there is no line where the pivot is not null
          if (VecGet(index, 1) >= VecGet(that->_dim, 1)) {
            // The system has no solution
            // Free memory
```

```
      MatFree(&copy);
      VecFree(&index);
      MatFree(&res);
      MatFree(&copy);
      // Stop here
      return NULL;
    }
    // Add the line where the pivot is not null to the line
    // of the pivot to un-nullify it
    VecSet(index, 0, 0);
    VecSet(pivot, 0, 0);
    while (VecGet(index, 0) < VecGet(that->_dim, 0)) {
      MatSet(copy, pivot,
        MatGet(copy, pivot) + MatGet(copy, index));
      MatSet(res, pivot,
        MatGet(res, pivot) + MatGet(res, index));
      VecSet(index, 0, VecGet(index, 0) + 1);
      VecSet(pivot, 0, VecGet(pivot, 0) + 1);
    }
    // Reposition the pivot
    VecSet(pivot, 0, VecGet(pivot, 1));
  }
  // Divide the values by the pivot
  float p = MatGet(copy, pivot);
  VecSet(pivot, 0, 0);
  while (VecGet(pivot, 0) < VecGet(that->_dim, 0)) {
    MatSet(copy, pivot, MatGet(copy, pivot) / p);
    MatSet(res, pivot, MatGet(res, pivot) / p);
    VecSet(pivot, 0, VecGet(pivot, 0) + 1);
  }
  // Reposition the pivot
  VecSet(pivot, 0, VecGet(pivot, 1));
  // Nullify the values below the pivot
  VecSet(pivot, 0, 0);
  VecSet(index, 1, VecGet(pivot, 1) + 1);
  while (VecGet(index, 1) < VecGet(that->_dim, 1)) {
    VecSet(index, 0, VecGet(pivot, 1));
    p = MatGet(copy, index);
    VecSet(index, 0, 0);
    while (VecGet(index, 0) < VecGet(that->_dim, 0)) {
      MatSet(copy, index,
        MatGet(copy, index) - MatGet(copy, pivot) * p);
      MatSet(res, index,
        MatGet(res, index) - MatGet(res, pivot) * p);
      VecSet(pivot, 0, VecGet(pivot, 0) + 1);
      VecSet(index, 0, VecGet(index, 0) + 1);
    }
    VecSet(pivot, 0, 0);
    VecSet(index, 0, 0);
    VecSet(index, 1, VecGet(index, 1) + 1);
  }
  // Reposition the pivot
  VecSet(pivot, 0, VecGet(pivot, 1));
}
// Now the matrix is triangular, move back through the pivots
// to make it diagonal
for (; VecGet(pivot, 0) >= 0;
  VecSet(pivot, 0, VecGet(pivot, 0) - 1),
  VecSet(pivot, 1, VecGet(pivot, 1) - 1)) {
  // Nullify the values above the pivot by substracting the line
  // of the pivot
  VecSet(pivot, 0, 0);
```

```
        VecSet(index, 1, VecGet(pivot, 1) - 1);
        while (VecGet(index, 1) >= 0) {
          VecSet(index, 0, VecGet(pivot, 1));
          float p = MatGet(copy, index);
          VecSet(index, 0, 0);
          while (VecGet(index, 0) < VecGet(that->_dim, 0)) {
            MatSet(copy, index,
              MatGet(copy, index) - MatGet(copy, pivot) * p);
            MatSet(res, index,
              MatGet(res, index) - MatGet(res, pivot) * p);
            VecSet(pivot, 0, VecGet(pivot, 0) + 1);
            VecSet(index, 0, VecGet(index, 0) + 1);
          }
          VecSet(pivot, 0, 0);
          VecSet(index, 0, 0);
          VecSet(index, 1, VecGet(index, 1) - 1);
        }
        // Reposition the pivot
        VecSet(pivot, 0, VecGet(pivot, 1));
      }
      // Free memory
      MatFree(&copy);
      VecFree(&index);
    }
  }
  // Free memory
  VecShortFree(&pivot);
  // Return the result
  return res;
}

// Return the product of matrix 'that' and vector 'v'
// Number of colum of 'that' must equal dimension of 'v'
// Return null if arguments are invalids
VecFloat* MatFloatProdVecFloat(MatFloat *that, VecFloat *v) {
  // Check arguments
  if (that == NULL || v == NULL || VecGet(that->_dim, 0) != VecDim(v))
    return NULL;
  // Declare a variable to memorize the index in the matrix
  VecShort *i = VecShortCreate(2);
  if (i == NULL)
    return NULL;
  // Allocate memory for the solution
  VecFloat *ret = VecFloatCreate(VecGet(that->_dim, 1));
  // If we could allocate memory
  if (ret != NULL) {
    for (VecSet(i, 0, 0); VecGet(i, 0) < VecGet(that->_dim, 0);
      VecSet(i, 0, VecGet(i, 0) + 1)) {
      for (VecSet(i, 1, 0); VecGet(i, 1) < VecGet(that->_dim, 1);
        VecSet(i, 1, VecGet(i, 1) + 1)) {
        VecSet(ret, VecGet(i, 1), VecGet(ret,
          VecGet(i, 1)) + VecGet(v, VecGet(i, 0)) * MatGet(that, i));
      }
    }
  }
  // Free memory
  VecFree(&i);
  // Return the result
  return ret;
}

// Return the product of matrix 'that' by matrix 'tho'
```

```
// Number of columns of 'that' must equal number of line of 'tho'
// Return null if arguments are invalids
MatFloat* MatFloatProdMatFloat(MatFloat *that, MatFloat *tho) {
  // Check arguments
  if (that == NULL || tho == NULL ||
    VecGet(that->_dim, 0) != VecGet(tho->_dim, 1))
    return NULL;
  // Declare 3 variables to memorize the index in the matrix
  VecShort *i = VecShortCreate(2);
  if (i == NULL)
    return NULL;
  VecShort *j = VecShortCreate(2);
  if (j == NULL) {
    VecFree(&i);
    return NULL;
  }
  VecShort *k = VecShortCreate(2);
  if (k == NULL) {
    VecFree(&i);
    VecFree(&j);
    return NULL;
  }
  // Allocate memory for the solution
  VecSet(i, 0, VecGet(tho->_dim, 0));
  VecSet(i, 1, VecGet(that->_dim, 1));
  MatFloat *ret = MatFloatCreate(i);
  // If we could allocate memory
  if (ret != NULL) {
    for (VecSet(i, 0, 0); VecGet(i, 0) < VecGet(tho->_dim, 0);
      VecSet(i, 0, VecGet(i, 0) + 1)) {
      for (VecSet(i, 1, 0); VecGet(i, 1) < VecGet(that->_dim, 1);
        VecSet(i, 1, VecGet(i, 1) + 1)) {
        for (VecSet(j, 0, 0), VecSet(j, 1, VecGet(i, 1)),
          VecSet(k, 0, VecGet(i, 0)), VecSet(k, 1, 0);
          VecGet(j, 0) < VecGet(that->_dim, 0);
          VecSet(j, 0, VecGet(j, 0) + 1),
          VecSet(k, 1, VecGet(k, 1) + 1)) {
          MatSet(ret, i, MatGet(ret, i) +
            MatGet(that, j) * MatGet(tho, k));
        }
      }
    }
  }
  // Free memory
  VecFree(&i);
  VecFree(&j);
  VecFree(&k);
  // Return the result
  return ret;
}

// -------------- Gauss

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

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

// Create a new Gauss of mean 'mean' and sigma 'sigma'
// Return NULL if we couldn't create the Gauss
Gauss* GaussCreate(float mean, float sigma) {
  // Allocate memory
  Gauss *that = (Gauss*)malloc(sizeof(Gauss));
```

```
  // If we could allocate memory
  if (that != NULL) {
    // Set properties
    that->_mean = mean;
    that->_sigma = sigma;
  }
  // REturn the new Gauss
  return that;
}

// Free the memory used by a Gauss
// Do nothing if arguments are invalid
void GaussFree(Gauss **that) {
  // Check argument
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  free(*that);
  *that = NULL;
}

// Return the value of the Gauss 'that' at 'x'
// Return 0.0 if the arguments are invalid
float GaussGet(Gauss *that, float x) {
  // Check arguments
  if (that == NULL)
    return 0.0;
  // Calculate the value
  float a = 1.0 / (that->_sigma * sqrt(2.0 * PBMATH_PI));
  float ret = a * exp(-1.0 * pow(x - that->_mean, 2.0) /
    (2.0 * pow(that->_sigma, 2.0)));
  // Return the value
  return ret;
}

// Return a random value (in ]0.0, 1.0[)according to the
// Gauss distribution 'that'
// random() must have been called before calling this function
// Return 0.0 if the arguments are invalid
float GaussRnd(Gauss *that) {
  // Check arguments
  if (that == NULL)
    return 0.0;
  // Declare variable for calcul
  float v1,v2,s;
  // Calculate the value
  do {
    v1 = (rnd() - 0.5) * 2.0;
    v2 = (rnd() - 0.5) * 2.0;
    s = v1 * v1 + v2 * v2;
  } while (s >= 1.0);
  // Return the value
  float ret = 0.0;
  if (s > PBMATH_EPSILON)
    ret = v1 * sqrt(-2.0 * log(s) / s);
  return ret * that->_sigma + that->_mean;
}

// ------------- Smoother

// ================ Define =================
```

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

// Return the order 1 smooth value of 'x'
// if x < 0.0 return 0.0
// if x > 1.0 return 1.0
float SmoothStep(float x) {
  if (x <= 0.0)
    return 0.0;
  else if (x >= 1.0)
    return 1.0;
  else
    return x * x * (3.0 - 2.0 * x);
}

// Return the order 2 smooth value of 'x'
// if x < 0.0 return 0.0
// if x > 1.0 return 1.0
float SmootherStep(float x) {
  if (x <= 0.0)
    return 0.0;
  else if (x >= 1.0)
    return 1.0;
  else
    return x * x * x * (x * (x * 6.0 - 15.0) + 10.0);
}

// -------------- Shapoid

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

const char *ShapoidTypeString[4] = {
  (const char*)"InvalidShapoid", (const char*)"Facoid",
  (const char*)"Spheroid", (const char*)"Pyramidoid"};

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

// Create a Shapoid of dimension 'dim' and type 'type', default values:
// _pos = null vector
// _axis[d] = unit vector along dimension d
// Return NULL if arguments are invalid or malloc failed
Shapoid* ShapoidCreate(int dim, ShapoidType type) {
  // Check argument
  if (dim < 0 || type == ShapoidTypeInvalid)
    return NULL;
  // Allocate memory
  Shapoid *that = (Shapoid*)malloc(sizeof(Shapoid));
  // If we could allocate memory
  if (that != NULL) {
    // Set the dimension and type
    that->_type = type;
    that->_dim = dim;
    // Allocate memory for position
    that->_pos = VecFloatCreate(dim);
    // If we couldn't allocate memory
    if (that->_pos == NULL) {
      free(that);
      return NULL;
    }
    // Allocate memory for array of axis
    that->_axis = (VecFloat**)malloc(sizeof(VecFloat*) * dim);
    if (that->_axis == NULL) {
      free(that);
```

```
      return NULL;
    }
    // Allocate memory for each axis
    for (int iAxis = 0; iAxis < dim; ++iAxis) {
      // Allocate memory for position
      that->_axis[iAxis] = VecFloatCreate(dim);
      // If we couldn't allocate memory
      if (that->_axis[iAxis] == NULL) {
        ShapoidFree(&that);
        return NULL;
      }
      // Set value of the axis
      VecSet(that->_axis[iAxis], iAxis, 1.0);
    }
  }
  return that;
}

// Clone a Shapoid
// Return NULL if couldn't clone
Shapoid* ShapoidClone(Shapoid *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Create a clone
  Shapoid *clone = ShapoidCreate(that->_dim, that->_type);
  if (clone != NULL) {
    // Set the position and axis of the clone
    ShapoidSetPos(clone, that->_pos);
    for (int iAxis = clone->_dim; iAxis--;)
      ShapoidSetAxis(clone, iAxis, that->_axis[iAxis]);
  }
  // Return the clone
  return clone;
}

// Free memory used by a Shapoid
// Do nothing if arguments are invalid
void ShapoidFree(Shapoid **that) {
  // Check argument
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  for (int iAxis = 0; iAxis < (*that)->_dim; ++iAxis)
    VecFree((*that)->_axis + iAxis);
  free((*that)->_axis);
  VecFree(&((*that)->_pos));
  free(*that);
  *that = NULL;
}

// Load the Shapoid from the stream
// If the VecFloat is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int ShapoidLoad(Shapoid **that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
```

```
  // If 'that' is already allocated
  if (*that != NULL) {
    // Free memory
    ShapoidFree(that);
  }
  // Read the dimension and type
  int dim;
  int ret = fscanf(stream, "%d", &dim);
  // If we coudln't fscanf
  if (ret == EOF)
    return 4;
  if (dim <= 0)
    return 3;
  ShapoidType type;
  ret = fscanf(stream, "%u", &type);
  // If we coudln't fscanf
  if (ret == EOF)
    return 4;
  // Allocate memory
  *that = ShapoidCreate(dim, type);
  // If we coudln't allocate memory
  if (*that == NULL) {
    return 2;
  }
  // Read the values
  ret = VecFloatLoad(&((*that)->_pos), stream);
  if (ret != 0)
    return ret;
  for (int iAxis = 0; iAxis < dim; ++iAxis) {
    ret = VecFloatLoad((*that)->_axis + iAxis, stream);
    if (ret != 0)
      return ret;
  }
  // Return success code
  return 0;
}

// Save the Shapoid to the stream
// Return 0 upon success, or
// 1: invalid arguments
// 2: fprintf error
int ShapoidSave(Shapoid *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // Save the dimension and type
  int ret = fprintf(stream, "%d %u\n", that->_dim, that->_type);
  // If we coudln't fprintf
  if (ret < 0)
    return 2;
  // Save the position and axis
  ret = VecFloatSave(that->_pos, stream);
  if (ret != 0)
    return ret;
  for (int iAxis = 0; iAxis < that->_dim; ++iAxis) {
    ret = VecFloatSave(that->_axis[iAxis], stream);
    if (ret != 0)
      return ret;
  }
  // Return success code
  return 0;
}
```

```
// Print the Shapoid on 'stream'
// Do nothing if arguments are invalid
void ShapoidPrint(Shapoid *that, FILE *stream) {
  // Check arugments
  if (that == NULL || stream == NULL)
    return;
  // Print the Shapoid
  fprintf(stream, "Type: %s\n", ShapoidTypeString[that->_type]);
  fprintf(stream, "Dim: %d\n", that->_dim);
  fprintf(stream, "Pos: ");
  VecPrint(that->_pos, stream);
  fprintf(stream, "\n");
  for (int iAxis = 0; iAxis < that->_dim; ++iAxis) {
    fprintf(stream, "Axis(%d): ", iAxis);
    VecPrint(that->_axis[iAxis], stream);
    fprintf(stream, "\n");
  }
}

// Get the dimension of the Shapoid
// Return 0 if arguments are invalid
int ShapoidGetDim(Shapoid *that) {
  // Check arugments
  if (that == NULL)
    return 0;
  // Return the dimension
  return that->_dim;
}

// Get the dimension of the Shapoid
// Return 0 if arguments are invalid
ShapoidType ShapoidGetType(Shapoid *that) {
  // Check arugments
  if (that == NULL)
    return 0;
  // Return the type
  return that->_type;
}

// Get the type of the Shapoid as a string
// Return a pointer to a constant string (not to be freed)
// Return the string for ShapoidTypeInvalid if arguments are invalid
const char* ShapoidGetTypeAsString(Shapoid *that) {
  // Check arugments
  if (that == NULL)
    return ShapoidTypeString[ShapoidTypeInvalid];
  // Return the type
  return ShapoidTypeString[that->_type];
}

// Return a VecFloat equal to the position of the Shapoid
// Return NULL if arguments are invalid
VecFloat* ShapoidGetPos(Shapoid *that) {
  // Check arugments
  if (that == NULL)
    return NULL;
  // Return a clone of the position
  return VecClone(that->_pos);
}

// Return a VecFloat equal to the 'dim'-th axis of the Shapoid
```

```
// Return NULL if arguments are invalid
VecFloat* ShapoidGetAxis(Shapoid *that, int dim) {
  // Check arugments
  if (that == NULL || dim < 0 || dim >= that->_dim)
    return NULL;
  // Return a clone of the axis
  return VecClone(that->_axis[dim]);
}

// Set the position of the Shapoid to 'pos'
// Do nothing if arguments are invalid
void ShapoidSetPos(Shapoid *that, VecFloat *pos) {
  // Check arugments
  if (that == NULL || pos == NULL)
    return;
  // Set the position
  VecCopy(that->_pos, pos);
}

// Set the 'dim'-th axis of the Shapoid to 'v'
// Do nothing if arguments are invalid
void ShapoidSetAxis(Shapoid *that, int dim, VecFloat *v) {
  // Check arugments
  if (that == NULL || v == NULL)
    return;
  // Set the axis
  VecCopy(that->_axis[dim], v);
}

// Translate the Shapoid by 'v'
// Do nothing if arguments are invalid
void ShapoidTranslate(Shapoid *that, VecFloat *v) {
  // Check arugments
  if (that == NULL || v == NULL)
    return;
  // Translate the position
  VecOp(that->_pos, 1.0, v, 1.0);
}

// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
// Do nothing if arguments are invalid
void ShapoidScale(Shapoid *that, VecFloat *v) {
  // Check arugments
  if (that == NULL || v == NULL)
    return;
  // Scale each axis
  for (int iAxis = that->_dim; iAxis--;)
    VecOp(that->_axis[iAxis], VecGet(v, iAxis), NULL, 0.0);
}

// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
// and translate the Shapoid such as its center after scaling
// is at the same position than before scaling
// Do nothing if arguments are invalid
void ShapoidGrow(Shapoid *that, VecFloat *v) {
  // Check arugments
  if (that == NULL || v == NULL)
    return;
  // Scale
  ShapoidScale(that, v);
  // If the shapoid is a Facoid or Pyramidoid
  if (that->_type == ShapoidTypeFacoid ||
```

```
    that->_type == ShapoidTypePyramidoid) {
    // Reposition to keep center at the same position
    for (int iAxis = that->_dim; iAxis--;)
      VecOp(that->_pos, 1.0,
        that->_axis[iAxis], -0.5 * (1.0 - 1.0 / VecGet(v, iAxis)));
  }
}


// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// Do nothing if arguments are invalid
void ShapoidRotate2D(Shapoid *that, float theta) {
  // Check arugments
  if (that == NULL)
    return;
  // Rotate each axis
  for (int iAxis = that->_dim; iAxis--;)
    VecRot2D(that->_axis[iAxis], theta);
}

// Convert the coordinates of 'pos' from standard coordinate system
// toward the Shapoid coordinates system
// Return null if the arguments are invalid
VecFloat* ShapoidImportCoord(Shapoid *that, VecFloat *pos) {
  // Check arguments
  if (that == NULL || pos == NULL || ShapoidGetDim(that) != VecDim(pos))
    return NULL;
  // Create a matrix for the linear system solver
  VecShort *dim = VecShortCreate(2);
  if (dim == NULL)
    return NULL;
  VecSet(dim, 0, that->_dim);
  VecSet(dim, 1, that->_dim);
  MatFloat *mat = MatFloatCreate(dim);
  if (mat == NULL) {
    VecFree(&dim);
    return NULL;
  }
  // Set the values of the matrix
  for (VecSet(dim, 0, 0); VecGet(dim, 0) < that->_dim;
    VecSet(dim, 0, VecGet(dim, 0) + 1)) {
    for (VecSet(dim, 1, 0); VecGet(dim, 1) < that->_dim;
      VecSet(dim, 1, VecGet(dim, 1) + 1)) {
      MatSet(mat, dim, VecGet(that->_axis[VecGet(dim, 0)],
        VecGet(dim, 1)));
    }
  }
  VecFloat *v = VecGetOp(pos, 1.0, that->_pos, -1.0);
  if (v == NULL) {
    VecFree(&dim);
    MatFree(&mat);
    return NULL;
  }
  // Create the linear system solver and solve it
  EqLinSys *sys = EqLinSysCreate(mat, v);
  VecFloat *res = EqLinSysSolve(sys);
  // Free memory
  VecFree(&v);
  VecFree(&dim);
  EqLinSysFree(&sys);
  MatFree(&mat);
  // return the result
```

```
  return res;
}

// Convert the coordinates of 'pos' from the Shapoid coordinates system
// toward standard coordinate system
// Return null if the arguments are invalid
VecFloat* ShapoidExportCoord(Shapoid *that, VecFloat *pos) {
  // Check arguments
  if (that == NULL || pos == NULL || ShapoidGetDim(that) != VecDim(pos))
    return NULL;
  // Allocate memory for the result
  VecFloat *res = VecClone(that->_pos);
  // If we could allocate memory
  if (res != NULL) {
    for (int dim = that->_dim; dim--;)
      VecOp(res, 1.0, that->_axis[dim], VecGet(pos, dim));
  }
  // Return the result
  return res;
}

// Return true if 'pos' is inside the Shapoid
// Else return false
bool ShapoidIsPosInside(Shapoid *that, VecFloat *pos) {
  // Check arguments
  if (that == NULL || pos == NULL || ShapoidGetDim(that) != VecDim(pos))
    return false;
  // Get the coordinates of pos in the Shapoid coordinate system
  VecFloat *coord = ShapoidImportCoord(that, pos);
  // If we couldn't get the coordinates
  if (coord == NULL)
    // Stop here
    return false;
  // Declare a variable to memorize the result
  bool ret = false;
  // If the Shapoid is a Facoid
  if (that->_type == ShapoidTypeFacoid) {
    // pos is in the Shapoid if all the coord in Shapoid coord
    // system are in [0.0, 1.0]
    ret = true;
    for (int dim = that->_dim; dim-- && ret == true;) {
      float v = VecGet(coord, dim);
      if (v < 0.0 || v > 1.0)
        ret = false;
    }
  // Else, if the Shapoid is a Pyramidoid
  } else if (that->_type == ShapoidTypePyramidoid) {
    // pos is in the Shapoid if all the coord in Shapoid coord
    // system are in [0.0, 1.0] and their sum is in [0.0, 1.0]
    ret = true;
    float sum = 0.0;
    for (int dim = that->_dim; dim-- && ret == true;) {
      float v = VecGet(coord, dim);
      sum += v;
      if (v < 0.0 || v > 1.0)
        ret = false;
    }
    if (ret == true && sum > 1.0)
      ret = false;
  // Else, if the Shapoid is a Spheroid
  } else if (that->_type == ShapoidTypeSpheroid) {
    // pos is in the Shapoid if its norm is in [0.0, 0.5]
```

```
    float norm = VecNorm(coord);
    if (norm <= 0.5)
      ret = true;
  }
  // Free memory
  VecFloatFree(&coord);
  // Return the result
  return ret;
}

// Get a bounding box of the Shapoid. The bounding box is aligned
// on the standard coordinate system (its axis are colinear with
// the axis of the standard coordinate system).
// The bounding box is returned as a Facoid, which position is
// at the minimum value along each axis.
// Return null if the argument are invalid.
Shapoid* ShapoidGetBoundingBoxThat(Shapoid *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Declare a variable to memorize the result
  Shapoid *res = FacoidCreate(ShapoidGetDim(that));
  if (res != NULL) {
    // If the Shapoid is a Facoid
    if (that->_type == ShapoidTypeFacoid) {
      // For each axis
      for (int dim = that->_dim; dim--;) {
        // Declare a variable to memorize the bound of the interval on
        // this axis
        float bound[2];
        bound[0] = bound[1] = VecGet(that->_pos, dim);
        // For each parameter
        for (int param = that->_dim; param--;) {
          // Get the value of the axis influencing the current dimension
          float v = VecGet(that->_axis[param], dim);
          // If the value is negative, update the minimum bound
          if (v < 0.0)
            bound[0] += v;
          // Else, if the value is negative, update the minimum bound
          else
            bound[1] += v;
        }
        // Memorize the result
        VecSet(res->_pos, dim, bound[0]);
        VecSet(res->_axis[dim], dim, bound[1] - bound[0]);
      }
    // Else, if the Shapoid is a Pyramidoid
    } else  if (that->_type == ShapoidTypePyramidoid) {
      // For each axis
      for (int dim = that->_dim; dim--;) {
        // Declare a variable to memorize the bound of the interval on
        // this axis
        float bound[2];
        bound[0] = bound[1] = VecGet(that->_axis[0], dim);
        // For each parameter
        for (int param = that->_dim; param--;) {
          // Get the value of the axis influencing the current dimension
          float v = VecGet(that->_axis[param], dim);
          // Search the min and max values
          if (v < bound[0])
            bound[0] = v;
          if (v > bound[1])
```

```
          bound[1] = v;
        }
        // Memorize the result
        if (bound[0] < 0.0) {
          VecSet(res->_pos, dim, VecGet(that->_pos, dim) + bound[0]);
          VecSet(res->_axis[dim], dim, bound[1] - bound[0]);
        } else {
          VecSet(res->_pos, dim, VecGet(that->_pos, dim));
          VecSet(res->_axis[dim], dim, bound[1]);
        }
      }
    // Else, if the Shapoid is a Spheroid
    } else  if (that->_type == ShapoidTypeSpheroid) {
      // In case of a Spheroid, things get complicate
      // We'll approximate the bounding box of the Spheroid
      // with the one of the same Spheroid viewed as a Facoid
      // and simply take care that the _pos is at the center of the
      // Spheroid
      // For each axis
      for (int dim = that->_dim; dim--;) {
        // Declare a variable to memorize the bound of the interval on
        // this axis
        float bound[2];
        bound[0] = VecGet(that->_pos, dim);
        // Correct position
        // For each parameter
        for (int param = that->_dim; param--;) {
          // Get the value of the axis influencing the current dimension
          float v = VecGet(that->_axis[param], dim);
          // Correct the pos
          bound[0] -= 0.5 * v;
        }
        bound[1] = bound[0];
        // For each parameter
        for (int param = that->_dim; param--;) {
          // Get the value of the axis influencing the current dimension
          float v = VecGet(that->_axis[param], dim);
          // If the value is negative, update the minimum bound
          if (v < 0.0)
            bound[0] += v;
          // Else, if the value is negative, update the minimum bound
          else
            bound[1] += v;
        }
        // Memorize the result
        VecSet(res->_pos, dim, bound[0]);
        VecSet(res->_axis[dim], dim, bound[1] - bound[0]);
      }
    } else {
      // In any case of invalid shapoid type return NULL
      ShapoidFree(&res);
    }
  }
  // Return the result
  return res;
}

// Get the bounding box of a set of Facoid. The bounding box is aligned
// on the standard coordinate system (its axis are colinear with
// the axis of the standard coordinate system).
// The bounding box is returned as a Facoid, which position is
// at the minimum value along each axis.
```

```
// Return null if the arguments are invalid or the shapoid in the set
// don't have all the same dimension.
Shapoid* ShapoidGetBoundingBoxSet(GSet *set) {
  // Check arguments
  if (set == NULL)
    return NULL;
  // Declare a variable for the result
  Shapoid *res = NULL;
  // Declare a pointer to the elements of the set
  GSetElem *elem = set->_head;
  // Loop on element of the set
  while (elem != NULL) {
    // Declare a pointer to the Facoid
    Shapoid *shapoid = (Shapoid*)(elem->_data);
    // If it's the first Facoid in the set
    if (res == NULL) {
      // Get the bounding box of this shapoid
      res = ShapoidGetBoundingBox(shapoid);
      // If we couldn't get the bounding box
      if (res == NULL)
        return NULL;
    // Else, this is not the first Shapoid in the set
    } else {
      // Ensure the Facoids have all the same dimension
      if (shapoid->_dim != res->_dim) {
        ShapoidFree(&res);
        return NULL;
      }
      // Get the bounding box of this shapoid
      Shapoid *bound = ShapoidGetBoundingBox(shapoid);
      // If we couldn't get the bounding box
      if (bound == NULL) {
        ShapoidFree(&res);
        return NULL;
      }
      // For each dimension
      for (int iDim = res->_dim; iDim--;) {
        // Update the bounding box
        if (VecGet(bound->_pos, iDim) < VecGet(res->_pos, iDim)) {
          VecSet(res->_axis[iDim], iDim,
            VecGet(res->_axis[iDim], iDim) +
            VecGet(res->_pos, iDim) -
            VecGet(bound->_pos, iDim));
          VecSet(res->_pos, iDim, VecGet(bound->_pos, iDim));
        }
        if (VecGet(bound->_pos, iDim) +
          VecGet(bound->_axis[iDim], iDim) >
          VecGet(res->_pos, iDim) +
          VecGet(res->_axis[iDim], iDim))
          VecSet(res->_axis[iDim], iDim,
          VecGet(bound->_pos, iDim) +
          VecGet(bound->_axis[iDim], iDim) -
          VecGet(res->_pos, iDim));
      }
      // Free memory used by the bounding box
      ShapoidFree(&bound);
    }
    // Move to the next element
    elem = elem->_next;
  }
  // Return the result
  return res;
```

```
}

// Get a SCurve approximating the Shapoid 'that'
// 'that' must be of dimension 2
// Return null if arguments are invalid
SCurve* Shapoid2SCurve(Shapoid *that) {
  // Check arguments
  if (that == NULL || ShapoidGetDim(that) != 2)
    return NULL;
  // Declare a SCurve to memorize the result
  SCurve *ret = SCurveCreate(ShapoidGetDim(that));
  // If we couldn't allocate memory
  if (ret == NULL)
    return NULL;
  // Declare a pointer to the GSet of the SCurve
  GSet *set = ret->_curves;
  // If the shapoid is a Facoid
  if (ShapoidGetType(that) == ShapoidTypeFacoid) {
    VecFloat *A = VecGetOp(that->_pos, 1.0, that->_axis[0], 1.0);
    VecFloat *B = VecGetOp(that->_pos, 1.0, that->_axis[1], 1.0);
    VecFloat *C = VecGetOp(A, 1.0, that->_axis[1], 1.0);
    BCurve *curve = NULL;
    if (A != NULL && B != NULL && C != NULL) {
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, that->_pos);
        BCurveSet(curve, 1, A);
        GSetAppend(set, curve);
      }
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, A);
        BCurveSet(curve, 1, C);
        GSetAppend(set, curve);
      }
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, C);
        BCurveSet(curve, 1, B);
        GSetAppend(set, curve);
      }
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, B);
        BCurveSet(curve, 1, that->_pos);
        GSetAppend(set, curve);
      }
    }
    VecFree(&A);
    VecFree(&B);
    VecFree(&C);
  // Else, if the shapoid is a Pyramidoid
  } else if (ShapoidGetType(that) == ShapoidTypePyramidoid) {
    VecFloat *A = VecGetOp(that->_pos, 1.0, that->_axis[0], 1.0);
    VecFloat *B = VecGetOp(that->_pos, 1.0, that->_axis[1], 1.0);
    BCurve *curve = NULL;
    if (A != NULL && B != NULL) {
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, that->_pos);
        BCurveSet(curve, 1, A);
        GSetAppend(set, curve);
```

```
    }
    curve = BCurveCreate(1, 2);
    if (curve != NULL) {
      BCurveSet(curve, 0, A);
      BCurveSet(curve, 1, B);
      GSetAppend(set, curve);
    }
    curve = BCurveCreate(1, 2);
    if (curve != NULL) {
      BCurveSet(curve, 0, B);
      BCurveSet(curve, 1, that->_pos);
      GSetAppend(set, curve);
    }
  }
  VecFree(&A);
  VecFree(&B);
// Else, if the shapoid is a Spheroid
} else if (ShapoidGetType(that) == ShapoidTypeSpheroid) {
  // Approximate each quarter of the Spheroid with BCurves
  // Declare a variable to memorize the angular position on the
  // Spheroid surface
  float theta = 0.0;
  // Declare a variable to memorize the delta of angular position
  float deltaTheta = PBMATH_HALFPI / 3.0;
  // Declare a GSet to memorize the 4 points of the point cloud
  // used to calculate the BCurve approximating the quarter of
  // Spheroid
  GSet *pointCloud = GSetCreate();
  if (pointCloud != NULL) {
    // Loop until we have made a full turn around the Spheroid
    for (int iCurve = 4; iCurve--;) {
      // For each point of the point cloud
      for (int iPoint = 4; iPoint--;) {
        // Declare a variable to memorize the coordinates of the point
        VecFloat *point = VecFloatCreate(2);
        // If we could allocate memory
        if (point != NULL) {
          // Calculate the coordinates of the current point in the
          // Spheroid coordinate system
          VecSet(point, 0, 0.5 * cos(theta));
          VecSet(point, 1, 0.5 * sin(theta));
          // Add the point converted to standard coordinate system
          // to the point cloud
          VecFloat *pointConvert = ShapoidExportCoord(that, point);
          GSetAppend(pointCloud, pointConvert);
          VecFree(&point);
        }
        // Increment the angular position
        theta += deltaTheta;
      }
      BCurve *curve = BCurveFromCloudPoint(pointCloud);
      // If we could get the BCurve
      if (curve != NULL)
        // Append the curve to the set of curve to be drawm
        GSetAppend(set, curve);
      // Free memory
      GSetElem *elem = pointCloud->_head;
      while (elem != NULL) {
        VecFloatFree((VecFloat**)(&(elem->_data)));
        elem = elem->_next;
      }
      // Empty the point cloud
```

```
        GSetFlush(pointCloud);
        // We need to decrement theta because the first point of the
        // next curve will be the last point of the current curve
        theta -= deltaTheta;
      }
      GSetFree(&pointCloud);
    }
  }
  // Return the result
  return ret;
}

// Get the depth value in the Shapoid of 'pos'
// The depth is defined as follow: the point with depth equals 1.0 is
// the farthest point from the surface of the Shapoid (inside it),
// points with depth equals to 0.0 are point on the surface of the
// Shapoid. Depth is continuous and derivable over the volume of the
// Shapoid
// Return 0.0 if arguments are invalid, or pos is outside the Shapoid
float ShapoidGetPosDepth(Shapoid *that, VecFloat *pos) {
  // Check arguments
  if (that == NULL || pos == NULL || ShapoidGetDim(that) != VecDim(pos))
    return 0.0;
  // Get the coordinates of pos in the Shapoid coordinate system
  VecFloat *coord = ShapoidImportCoord(that, pos);
  // If we couldn't get the coordinates
  if (coord == NULL)
    // Stop here
    return 0.0;
  // Declare a variable to memorize the result
  float ret = 0.0;
  // If the Shapoid is a Facoid
  if (that->_type == ShapoidTypeFacoid) {
    ret = 1.0;
    for (int dim = that->_dim; dim-- && ret > PBMATH_EPSILON;) {
      float v = VecGet(coord, dim);
      if (v < 0.0 || VecGet(coord, dim) > 1.0)
        ret = 0.0;
      else
        ret *= 1.0 - pow(0.5 - v, 2.0) * 4.0;
    }
  // Else, if the Shapoid is a Pyramidoid
  } else if (that->_type == ShapoidTypePyramidoid) {
    ret = 1.0;
    float sum = 0.0;
    bool flag = true;
    for (int dim = that->_dim; dim-- && ret > PBMATH_EPSILON;) {
      float v = VecGet(coord, dim);
      sum += v;
      if (v < 0.0 || v > 1.0)
        flag = false;
    }
    if (flag == true && sum > 1.0)
      flag = false;
    if (flag == false)
      ret = 0.0;
    else {
      ret = 1.0;
      for (int dim = ShapoidGetDim(that); dim--;) {
        float z = 0.0;
        for (int d = ShapoidGetDim(that); d--;)
          if (d != dim)
```

```
            z += VecGet(coord, d);
        ret *=
          (1.0 - 4.0 * pow(0.5 - VecGet(coord, dim) / (1.0 - z), 2.0));
      }
    }
  // Else, if the Shapoid is a Spheroid
  } else if (that->_type == ShapoidTypeSpheroid) {
    float norm = VecNorm(coord);
    if (norm <= 0.5)
      ret = 1.0 - norm * 2.0;
  }
  // Free memory
  VecFloatFree(&coord);
  // Return the result
  return ret;
}

// Get the center of the shapoid in standard coordinate system
// Return null if arguments are invalid
VecFloat* ShapoidGetCenter(Shapoid *that) {
  // Check arguments
  if (that == NULL)
    return NULL;
  // Declare a variable to memorize the result in Shapoid
  // coordinate system
  VecFloat *coord = VecFloatCreate(ShapoidGetDim(that));
  // If we could allocate memory
  if (coord != NULL) {
    // For each dimension
    for (int dim = ShapoidGetDim(that); dim--;) {
      if (ShapoidGetType(that) == ShapoidTypeFacoid)
        VecSet(coord, dim, 0.5);
      else if (ShapoidGetType(that) == ShapoidTypePyramidoid)
        VecSet(coord, dim, 1.0 / (1.0 + ShapoidGetDim(that)));
      else if (ShapoidGetType(that) == ShapoidTypeSpheroid)
        VecSet(coord, dim, 0.0);
    }
  }
  // Convert the cooridnate in standard cooridnate system
  VecFloat *res = ShapoidExportCoord(that, coord);
  // Return the result
  return res;
}

// Get the percentage of 'tho' included 'that' (in [0.0, 1.0])
// 0.0 -> 'tho' is completely outside of 'that'
// 1.0 -> 'tho' is completely inside of 'that'
// 'that' and 'tho' must me of same dimensions
// Return 0.0 if the arguments are invalid or something went wrong
float ShapoidGetCoverageRatio(Shapoid *that, Shapoid *tho) {
  // Check arguments
  if (that == NULL || tho == NULL)
    return 0.0;
  // Declare a variable to memorize the result
  float ratio = 0.0;
  // Declare variables for the relative and absolute position in 'tho'
  VecFloat *pRel = VecFloatCreate(ShapoidGetDim(that));
  VecFloat *pAbs = NULL;
  // If we couldn't allocate memory
  if (pRel == NULL) {
    // Free memory and stop here
    VecFree(&pRel);
```

```
    return 0.0;
  }
  // Declare a variable to memorize the step in relative coordinates
  float delta = 0.1;
  // Declare a variable to memorize the last index in dimension
  int lastI = VecDim(pRel) - 1;
  // Loop on relative coordinates
  while (VecGet(pRel, lastI) < 1.0 + delta - PBMATH_EPSILON) {
    // Get the absolute coordinates
    pAbs = ShapoidExportCoord(tho, pRel);
    // If we could get the position
    if (pAbs != NULL) {
      // If this position is inside 'that'
      if (ShapoidIsPosInside(that, pAbs) == true)
        // Increment the ratio
        ratio += 1.0;
      // Free memory
      VecFree(&pAbs);
    }
    // Step the relative coordinates
    int iDim = 0;
    while (iDim >= 0) {
      VecSet(pRel, iDim, VecGet(pRel, iDim) + delta);
      if (iDim != lastI &&
        VecGet(pRel, iDim) >= 1.0 + delta - PBMATH_EPSILON) {
        VecSet(pRel, iDim, 0.0);
        ++iDim;
      } else {
        iDim = -1;
      }
    }
  }
  // Finish the computation of the ratio
  ratio /= pow(1.0 / delta + 1.0, (float)VecDim(pRel));
  // Free memory
  VecFree(&pRel);
  // Return the result
  return ratio;
}

// -------------- Conversion functions

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

// Convert radians to degrees
float ConvRad2Deg(float rad) {
  return 360.0 * rad / PBMATH_TWOPI;
}

// Convert degrees to radians
float ConvDeg2Rad(float deg) {
  return PBMATH_TWOPI * deg / 360.0;
}

// -------------- EqLinSys

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

// Create a new EqLinSys with matrix 'm' and vector 'v'
// The dimension of 'v' must be equal to the number of column of 'm'
// The matrix 'm' must be a square matrix
// Return NULL if we couldn't create the EqLinSys
```

```
EqLinSys* EqLinSysCreate(MatFloat *m, VecFloat *v) {
  // Check arguments
  if (m == NULL || v == NULL || VecGet(m->_dim, 0) != VecDim(v) ||
    VecGet(m->_dim, 0) != VecGet(m->_dim, 1))
    return NULL;
  // Allocate memory
  EqLinSys *that = (EqLinSys*)malloc(sizeof(EqLinSys));
  // If we could allocate memory
  if (that != NULL) {
    that->_M = MatClone(m);
    that->_V = VecClone(v);
    if (that->_M == NULL || that->_V == NULL) {
      EqLinSysFree(&that);
      return NULL;
    }
  }
  // Return the new EqLinSys
  return that;
}

// Free the memory used by the EqLinSys
// Do nothing if arguments are invalid
void EqLinSysFree(EqLinSys **that) {
  // Check arguments
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  MatFree(&((*that)->_M));
  VecFree(&((*that)->_V));
  free(*that);
  *that = NULL;
}

// Solve the EqLinSys _M.x = _V
// Return the solution vector, or null if there is no solution or the
// arguments are invalid
VecFloat* EqLinSysSolve(EqLinSys *that) {
  // Check the argument
  if (that == NULL)
    return NULL;
  // Declare a variable to memorize the solution
  VecFloat *ret = NULL;
  // Inverse the matrix
  MatFloat *inv = MatInv(that->_M);
  // If we could inverse the matrix
  if (inv != NULL) {
    // Calculate the solution
    ret = MatProd(inv, that->_V);
  }
  // Free memory
  MatFree(&inv);
  // Return the solution vector
  return ret;
}
```

4 Makefile

```
OPTIONS_DEBUG=-ggdb -g3 -Wall
```

```
OPTIONS_RELEASE=-O3
OPTIONS=$(OPTIONS_RELEASE)
INCPATH=/home/bayashi/Coding/Include
LIBPATH=/home/bayashi/Coding/Include

all : main

main: main.o pbmath.o Makefile $(LIBPATH)/bcurve.o $(LIBPATH)/gset.o
gcc $(OPTIONS) main.o pbmath.o $(LIBPATH)/bcurve.o $(LIBPATH)/gset.o -o main -lm

main.o : main.c pbmath.h Makefile
gcc $(OPTIONS) -I$(INCPATH) -c main.c

pbmath.o : pbmath.c pbmath.h Makefile $(INCPATH)/bcurve.h $(INCPATH)/gset.h
gcc $(OPTIONS) -I$(INCPATH) -c pbmath.c

clean :
rm -rf *.o main

valgrind :
valgrind -v --track-origins=yes --leak-check=full --gen-suppressions=yes --show-leak-kinds=all ./main

install :
cp pbmath.h ../Include; cp pbmath.o ../Include
```

5 Usage

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "pbmath.h"

int main(int argc, char **argv) {
  // Initialise the random generator
  srandom(time(NULL));

  // -------------- VecShort
  fprintf(stdout, "-------- VecShort\n");
  // Create a vector of dimension 3
  VecShort *a = VecShortCreate(3);
  // If we couldn't create the vector
  if (a == NULL) {
    fprintf(stderr, "VecCreate failed\n");
    return 1;
  }
  // Print the vector
  fprintf(stdout, "a: ");
  VecPrint(a, stdout);
  fprintf(stdout, "\n");
  // Set the 2nd value to 1
  VecSet(a, 1, 1);
  // Print the vector
  fprintf(stdout, "a: ");
  VecPrint(a, stdout);
  fprintf(stdout, "\n");
  // Clone the vector
  VecShort *cloneShort = VecClone(a);
```

```
if (cloneShort == NULL) {
  fprintf(stderr, "VecClone failed\n");
  return 2;
}
// Print the vector
fprintf(stdout, "cloneShort: ");
VecPrint(cloneShort, stdout);
fprintf(stdout, "\n");
VecFree(&cloneShort);
// Save the vector
FILE *f = fopen("./vecshort.txt", "w");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 3;
}
int ret = VecSave(a, f);
if (ret != 0) {
  fprintf(stderr, "VecSave failed (%d)\n", ret);
  return 4;
}
fclose(f);
// Load the vector
f = fopen("./vecshort.txt", "r");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 5;
}
VecShort *b = NULL;
ret = VecLoad(&b, f);
if (ret != 0) {
  fprintf(stderr, "VecLoad failed (%d)\n", ret);
  return 6;
}
fclose(f);
// Get the dimension and values of the loaded vector
fprintf(stdout, "b: %d ", VecDim(b));
for (int i = 0; i < VecDim(b); ++i)
  fprintf(stdout, "%d ", VecGet(b, i));
fprintf(stdout, "\n");
// Change the values of the loaded vector and print it
VecSet(b, 0, 2);
VecSet(b, 2, 3);
fprintf(stdout, "b: ");
VecPrint(b, stdout);
fprintf(stdout, "\n");
// Copy the loaded vector into the first one and print the first one
VecCopy(a, b);
fprintf(stdout, "a: ");
VecPrint(a, stdout);
fprintf(stdout, "\n");
// Free memory
VecFree(&a);
VecFree(&b);

// ------------- VecFloat
fprintf(stdout, "-------- VecFloat\n");
// Create a vector of dimension 3
VecFloat *v = VecFloatCreate(3);
// If we couldn't create the vector
if (v == NULL) {
  fprintf(stderr, "VecCreate failed\n");
  return 7;
```

```
}
// Print the vector
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
// Set the 2nd value to 1.0
VecSet(v, 1, 1.0);
// Print the vector
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
// Clone the vector
VecFloat *cloneFloat = VecClone(v);
if (cloneFloat == NULL) {
  fprintf(stderr, "VecClone failed\n");
  return 8;
}
// Print the vector
fprintf(stdout, "cloneFloat: ");
VecPrint(cloneFloat, stdout);
fprintf(stdout, "\n");
VecFree(&cloneFloat);
// Save the vector
f = fopen("./vecfloat.txt", "w");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 9;
}
ret = VecSave(v, f);
if (ret != 0) {
  fprintf(stderr, "VecSave failed (%d)\n", ret);
  return 10;
}
fclose(f);
// Load the vector
f = fopen("./vecfloat.txt", "r");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 11;
}
VecFloat *w = NULL;
ret = VecLoad(&w, f);
if (ret != 0) {
  fprintf(stderr, "VecLoad failed (%d)\n", ret);
  return 12;
}
fclose(f);
// Get the dimension and values of the loaded vector
fprintf(stdout, "w: %d ", VecDim(w));
for (int i = 0; i < VecDim(w); ++i)
  fprintf(stdout, "%f ", VecGet(w, i));
fprintf(stdout, "\n");
// Change the values of the loaded vector and print it
VecSet(w, 0, 2.0);
VecSet(w, 2, 3.0);
fprintf(stdout, "w: ");
VecPrint(w, stdout);
fprintf(stdout, "\n");
// Copy the loaded vector into the first one and print the first one
VecCopy(v, w);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
```

```
fprintf(stdout, "\n");
// Get the norm
float norm = VecNorm(v);
fprintf(stdout, "Norm of v: %.3f\n", norm);
// Normalise
VecNormalise(v);
fprintf(stdout, "Normalized v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
// Distance between v and w
fprintf(stdout, "Distance between v and w: %.3f\n", VecDist(v, w));
fprintf(stdout, "Hamiltonian distance between v and w: %.3f\n",
  VecHamiltonDist(v, w));
// Equality
if (VecIsEqual(v, w) == true)
  fprintf(stdout, "v = w\n");
else
  fprintf(stdout, "v != w\n");
if (VecIsEqual(v, v) == true)
  fprintf(stdout, "v = v\n");
else
  fprintf(stdout, "v != v\n");
// Op
VecFloat *x = VecGetOp(v, norm, w, 2.0);
if (x == NULL) {
  fprintf(stderr, "VecGetOp failed\n");
  return 13;
}
fprintf(stdout, "x: ");
VecPrint(x, stdout);
fprintf(stdout, "\n");
VecOp(v, norm, NULL, 0.0);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
// Dot prod
fprintf(stdout, "dot prod v.x: %.3f\n", VecDotProd(v, x));
// Rotate
VecFree(&v);
v = VecFloatCreate(2);
if (v == NULL) {
  fprintf(stderr, "malloc failed\n");
  return 14;
}
VecSet(v, 0, 1.0);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
VecRot2D(v, PBMATH_QUARTERPI);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
VecFree(&x);
x = VecGetRot2D(v, PBMATH_QUARTERPI);
if (v == NULL) {
  fprintf(stderr, "VecGetRot2D failed\n");
  return 15;
}
fprintf(stdout, "x: ");
VecPrint(x, stdout);
fprintf(stdout, "\n");
// AngleTo
```

```
fprintf(stdout, "Angle between vector:\n");
float dtheta = PBMATH_PI / 6.0;
VecSet(x, 0, 1.0); VecSet(x, 1, 0.0);
for (int i = 0; i < 12; ++i) {
  VecSet(v, 0, 1.0); VecSet(v, 1, 0.0);
  for (int j = 0; j < 12; ++j) {
    VecPrint(x, stdout);
    fprintf(stdout, " ");
    VecPrint(v, stdout);
    fprintf(stdout, " %.3f\n", ConvRad2Deg(VecAngleTo2D(x, v)));
    VecRot2D(v, dtheta);
  }
  VecRot2D(x, dtheta);
}
// Free memory
VecFree(&x);
VecFree(&w);
VecFree(&v);

// -------------- MatFloat
fprintf(stdout, "-------- MatFloat\n");
// Create a matrix of dimension 3,2
VecShort *dimMat = VecShortCreate(2);
VecSet(dimMat, 0, 3);
VecSet(dimMat, 1, 2);
MatFloat *mat = MatFloatCreate(dimMat);
// If we couldn't create the matrix
if (mat == NULL) {
  fprintf(stderr, "MatCreate failed\n");
  return 16;
}
// Print the matrix
fprintf(stdout, "mat: \n");
MatPrint(mat, stdout);
fprintf(stdout, "\n");
// Set some values
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 0);
MatSet(mat, dimMat, 0.5);
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 1);
MatSet(mat, dimMat, 2.0);
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 2);
MatSet(mat, dimMat, 1.0);
VecSet(dimMat, 0, 1);
VecSet(dimMat, 1, 0);
MatSet(mat, dimMat, 2.0);
VecSet(dimMat, 0, 2);
VecSet(dimMat, 1, 1);
MatSet(mat, dimMat, 1.0);
// Print the matrix
fprintf(stdout, "mat: \n");
MatPrint(mat, stdout);
fprintf(stdout, "\n");
// Clone the matrix
MatFloat *cloneMatFloat = MatClone(mat);
if (cloneMatFloat == NULL) {
  fprintf(stderr, "MatClone failed\n");
  return 17;
}
// Print the matrix
```

```
fprintf(stdout, "cloneMatFloat:\n");
MatPrint(cloneMatFloat, stdout);
fprintf(stdout, "\n");
MatFree(&cloneMatFloat);
// Save the matrix
f = fopen("./matfloat.txt", "w");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 18;
}
ret = MatSave(mat, f);
if (ret != 0) {
  fprintf(stderr, "MatSave failed (%d)\n", ret);
  return 19;
}
fclose(f);
// Load the matrix
f = fopen("./matfloat.txt", "r");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 20;
}
MatFloat *matb = NULL;
ret = MatLoad(&matb, f);
if (ret != 0) {
  fprintf(stderr, "MatLoad failed (%d)\n", ret);
  return 21;
}
fclose(f);
// Get the dimension and values of the loaded matrix
VecShort *dimMatb = MatDim(matb);
fprintf(stdout, "dim loaded matrix: ");
VecPrint(dimMatb, stdout);
fprintf(stdout, "\n");
for (VecSet(dimMat, 1, 0); VecGet(dimMat, 1) < VecGet(dimMatb, 1);
  VecSet(dimMat, 1, VecGet(dimMat, 1) + 1)) {
  for (VecSet(dimMat, 0, 0); VecGet(dimMat, 0) < VecGet(dimMatb, 0);
    VecSet(dimMat, 0, VecGet(dimMat, 0) + 1))
    fprintf(stdout, "%f ", MatGet(matb, dimMat));
  fprintf(stdout, "\n");
}
// MatProdVec
v = VecFloatCreate(3);
if (v == NULL) {
  fprintf(stderr, "VecFloatCreate failed\n");
  return 22;
}
VecSet(v, 0, 2.0);
VecSet(v, 1, 3.0);
VecSet(v, 2, 4.0);
w = MatProd(matb, v);
if (w == NULL) {
  fprintf(stderr, "MatProd failed\n");
  return 23;
}
fprintf(stdout, "Mat prod of\n");
MatPrint(matb, stdout);
fprintf(stdout, "\nand\n");
VecPrint(v, stdout);
fprintf(stdout, "\nequals\n");
VecPrint(w, stdout);
fprintf(stdout, "\n");
```

```
VecFree(&v);
VecFree(&w);
// MatProdMat
VecSet(dimMat, 0, VecGet(dimMatb, 1));
VecSet(dimMat, 1, VecGet(dimMatb, 0));
MatFloat *matc = MatFloatCreate(dimMat);
if (matc == NULL) {
  fprintf(stderr, "MatFloatCreate failed\n");
  return 24;
}
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 0);
MatSet(matc, dimMat, 1.0);
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 1);
MatSet(matc, dimMat, 2.0);
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 2);
MatSet(matc, dimMat, 3.0);
VecSet(dimMat, 0, 1);
VecSet(dimMat, 1, 0);
MatSet(matc, dimMat, 4.0);
VecSet(dimMat, 0, 1);
VecSet(dimMat, 1, 1);
MatSet(matc, dimMat, 5.0);
VecSet(dimMat, 0, 1);
VecSet(dimMat, 1, 2);
MatSet(matc, dimMat, 6.0);
fprintf(stdout, "Mat prod of\n");
MatPrint(mat, stdout);
fprintf(stdout, "\nand\n");
MatPrint(matc, stdout);
fprintf(stdout, "\nequals\n");
MatFloat *matd = MatProd(mat, matc);
if (matd == NULL) {
  fprintf(stderr, "MatProd failed\n");
  return 25;
}
MatPrint(matd, stdout);
fprintf(stdout, "\n");
// Create a matrix and set it to identity
VecSet(dimMat, 0, 3);
VecSet(dimMat, 1, 3);
MatFloat *squareMat = MatFloatCreate(dimMat);
MatSetIdentity(squareMat);
fprintf(stdout, "identity:\n");
MatPrint(squareMat, stdout);
fprintf(stdout, "\n");
// Matrix inverse
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 0);
MatSet(squareMat, dimMat, 3.0);
VecSet(dimMat, 0, 1);
VecSet(dimMat, 1, 0);
MatSet(squareMat, dimMat, 0.0);
VecSet(dimMat, 0, 2);
VecSet(dimMat, 1, 0);
MatSet(squareMat, dimMat, 2.0);
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 1);
MatSet(squareMat, dimMat, 2.0);
VecSet(dimMat, 0, 1);
```

```
VecSet(dimMat, 1, 1);
MatSet(squareMat, dimMat, 0.0);
VecSet(dimMat, 0, 2);
VecSet(dimMat, 1, 1);
MatSet(squareMat, dimMat, -2.0);
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 2);
MatSet(squareMat, dimMat, 0.0);
VecSet(dimMat, 0, 1);
VecSet(dimMat, 1, 2);
MatSet(squareMat, dimMat, 1.0);
VecSet(dimMat, 0, 2);
VecSet(dimMat, 1, 2);
MatSet(squareMat, dimMat, 1.0);
MatFloat *matinv = MatInv(squareMat);
if (matinv == NULL) {
  fprintf(stderr, "MatInv failed\n");
  return 26;
}
fprintf(stdout, "inverse of:\n");
MatPrint(squareMat, stdout);
fprintf(stdout, "\nequals\n");
MatPrint(matinv, stdout);
fprintf(stdout, "\n");
MatFloat *checkinv = MatProd(squareMat, matinv);
fprintf(stdout, "check of inverse:\n");
MatPrint(checkinv, stdout);
fprintf(stdout, "\n");
// Free memory
VecFree(&dimMat);
VecFree(&dimMatb);
MatFree(&mat);
MatFree(&matb);
MatFree(&matc);
MatFree(&matd);
MatFree(&checkinv);
MatFree(&squareMat);
MatFree(&matinv);

// ------------- Gauss
fprintf(stdout, "-------- Gauss\n");
// Create a Gauss function
float mean = 0.0;
float sigma = 1.0;
Gauss *gauss = GaussCreate(mean, sigma);
// If we couldn't create the Gauss
if (gauss == NULL) {
  fprintf(stderr, "Couldn't create the Gauss\n");
  return 27;
}
// Get some values of the Gauss function
fprintf(stdout, "Gauss function (mean:0.0, sigma:1.0):\n");
for (float x = -2.0; x <= 2.01; x += 0.2)
  fprintf(stdout, "%.3f %.3f\n", x, GaussGet(gauss, x));
// Change the mean
gauss->_mean = 1.0;
gauss->_sigma = 0.5;
// Get some random values according to the Gauss function
fprintf(stdout, "Gauss rnd (mean:1.0, sigma:0.5):\n");
for (int iVal = 0; iVal < 10; ++iVal)
  fprintf(stdout, "%.3f %.3f\n", GaussRnd(gauss), GaussRnd(gauss));
//Free memory
```

```
GaussFree(&gauss);

// -------------- Smoother
fprintf(stdout, "-------- Smoother\n");
for (float x = 0.0; x <= 1.01; x += 0.1)
  fprintf(stdout, "%.3f %.3f %.3f\n", x, SmoothStep(x),
    SmootherStep(x));

// -------------- Shapoid
fprintf(stdout, "-------- Shapoid\n");
Shapoid* facoidA = FacoidCreate(2);
ShapoidPrint(facoidA, stdout);
v = VecFloatCreate(2);
if (v == NULL) {
  fprintf(stderr, "malloc failed\n");
  return 28;
}
VecSet(v, 0, 2.0);
VecSet(v, 1, 3.0);
ShapoidScale(facoidA, v);
fprintf(stdout, "scale by ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
ShapoidPrint(facoidA, stdout);
ShapoidRotate2D(facoidA, PBMATH_PI * 0.5);
fprintf(stdout, "rotate by %.3f\n", PBMATH_PI * 0.5);
ShapoidPrint(facoidA, stdout);
VecSet(v, 0, 1.0);
VecSet(v, 1, -1.0);
ShapoidTranslate(facoidA, v);
fprintf(stdout, "translate by ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
ShapoidPrint(facoidA, stdout);
VecSet(v, 0, 0.0);
VecSet(v, 1, 1.0);
ShapoidSetAxis(facoidA, 0, v);
fprintf(stdout, "set axis 0 to ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
ShapoidPrint(facoidA, stdout);
fprintf(stdout, "save shapoid to shapoid.txt\n");
f = fopen("./shapoid.txt", "w");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 29;
}
ShapoidSave(facoidA, f);
fclose(f);
fprintf(stdout, "load shapoid from shapoid.txt\n");
Shapoid* facoidB = NULL;
f = fopen("./shapoid.txt", "r");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 30;
}
ShapoidLoad(&facoidB, f);
fclose(f);
ShapoidPrint(facoidB, stdout);
VecSet(v, 0, 1.0);
VecSet(v, 1, 1.0);
VecFloat *coordEx = ShapoidExportCoord(facoidB, v);
```

```
if (coordEx == NULL) {
  fprintf(stderr, "ShapoidExport failed\n");
  return 31;
}
fprintf(stdout,"coordinates ");
VecPrint(v, stdout);
fprintf(stdout," in the shapoid becomes ");
VecPrint(coordEx, stdout);
fprintf(stdout," in the standard coordinate system\n");
VecCopy(v, coordEx);
VecFloat *coordIm = ShapoidImportCoord(facoidB, v);
if (coordIm == NULL) {
  fprintf(stderr, "ShapoidImport failed\n");
  return 32;
}
fprintf(stdout,"coordinates ");
VecPrint(v, stdout);
fprintf(stdout," in the standard coordinate system becomes ");
VecPrint(coordIm, stdout);
fprintf(stdout," in the shapoid\n");
VecSet(v, 0, 0.0);
VecSet(v, 1, 0.0);
VecPrint(v, stdout);
if (ShapoidIsPosInside(facoidB, v) == true)
  fprintf(stdout, " is in the facoid\n");
else
  fprintf(stdout, " is not in the facoid\n");
VecSet(v, 0, 1.0);
VecSet(v, 1, -4.0);
VecPrint(v, stdout);
if (ShapoidIsPosInside(facoidB, v) == true)
  fprintf(stdout, " is in the facoid\n");
else
  fprintf(stdout, " is not in the facoid\n");
ShapoidRotate2D(facoidB, -PBMATH_QUARTERPI);
Shapoid *bounding = ShapoidGetBoundingBox(facoidB);
if (bounding == NULL) {
  fprintf(stderr, "ShapoidGetBoundingBox failed\n");
  return 33;
}
fprintf(stdout, "bounding box of\n");
ShapoidPrint(facoidB, stdout);
fprintf(stdout, "is\n");
ShapoidPrint(bounding, stdout);
ShapoidFree(&bounding);
VecFloat *center = ShapoidGetCenter(facoidB);
if (center == NULL) {
  fprintf(stderr, "ShapoidGetCenter failed\n");
  return 34;
}
fprintf(stdout, "center of the facoid is ");
VecPrint(center, stdout);
fprintf(stdout, "\n");
VecFree(&center);
facoidB->_type = ShapoidTypePyramidoid;
bounding = ShapoidGetBoundingBox(facoidB);
if (bounding == NULL) {
  fprintf(stderr, "ShapoidGetBoundingBox failed\n");
  return 35;
}
fprintf(stdout, "bounding box of\n");
ShapoidPrint(facoidB, stdout);
```

```
fprintf(stdout, "is\n");
ShapoidPrint(bounding, stdout);
ShapoidFree(&bounding);
center = ShapoidGetCenter(facoidB);
if (center == NULL) {
  fprintf(stderr, "ShapoidGetCenter failed\n");
  return 36;
}
fprintf(stdout, "center of the facoid is ");
VecPrint(center, stdout);
fprintf(stdout, "\n");
VecFree(&center);
facoidB->_type = ShapoidTypeSpheroid;
bounding = ShapoidGetBoundingBox(facoidB);
if (bounding == NULL) {
  fprintf(stderr, "ShapoidGetBoundingBox failed\n");
  return 37;
}
fprintf(stdout, "bounding box of\n");
ShapoidPrint(facoidB, stdout);
fprintf(stdout, "is\n");
ShapoidPrint(bounding, stdout);
center = ShapoidGetCenter(facoidB);
if (center == NULL) {
  fprintf(stderr, "ShapoidGetCenter failed\n");
  return 38;
}
fprintf(stdout, "center of the shapoid is ");
VecPrint(center, stdout);
fprintf(stdout, "\n");
VecFree(&center);

GSet *setBounding = GSetCreate();
if (setBounding == NULL) {
  fprintf(stderr, "GSetCreate failed\n");
  return 39;
}
GSetAppend(setBounding, facoidA);
GSetAppend(setBounding, facoidB);
facoidB->_type = ShapoidTypeFacoid;
VecSet(facoidB->_pos, 0, 2.0);
fprintf(stdout, "bounding box of\n");
ShapoidPrint(facoidA, stdout);
fprintf(stdout, "and\n");
ShapoidPrint(facoidB, stdout);
fprintf(stdout, "is\n");
bounding = ShapoidGetBoundingBox(setBounding);
if (bounding == NULL) {
  fprintf(stderr, "ShapoidGetBoundingBox failed\n");
  return 40;
}
ShapoidPrint(bounding, stdout);
// Grow
fprintf(stdout, "Grow the facoid:\n");
ShapoidPrint(facoidA, stdout);
fprintf(stdout, "by 2.0:\n");
VecSet(v, 0, 2.0); VecSet(v, 1, 2.0);
ShapoidGrow(facoidA, v);
ShapoidPrint(facoidA, stdout);
// Coverage ratio
fprintf(stdout, "Percentage of :\n");
ShapoidPrint(facoidB, stdout);
```

```
  fprintf(stdout, "included in :\n");
  ShapoidPrint(facoidA, stdout);
  float ratio = ShapoidGetCoverageRatio(facoidA, facoidB);
  fprintf(stdout, "is %f\n", ratio);
  // Free memory
  ShapoidFree(&bounding);
  GSetFree(&setBounding);
  VecFree(&coordEx);
  VecFree(&coordIm);
  VecFree(&v);
  ShapoidFree(&facoidA);
  ShapoidFree(&facoidB);
  ShapoidFree(&bounding);

  // -------------- Conversion functions
  fprintf(stdout, "-------- Conversion functions\n");
  fprintf(stdout, "%f radians -> %f degrees\n", PBMATH_QUARTERPI,
    ConvRad2Deg(PBMATH_QUARTERPI));
  fprintf(stdout, "%f radians -> %f degrees\n", 90.0,
    ConvDeg2Rad(90.0));

  // Return success code
  return 0;
}
```

Output:

```
-------- VecShort
a: <0,0,0>
a: <0,1,0>
cloneShort: <0,1,0>
b: 3 0 1 0
b: <2,1,3>
a: <2,1,3>
-------- VecFloat
v: <0.000,0.000,0.000>
v: <0.000,1.000,0.000>
cloneFloat: <0.000,1.000,0.000>
w: 3 0.000000 1.000000 0.000000
w: <2.000,1.000,3.000>
v: <2.000,1.000,3.000>
Norm of v: 3.742
Normalized v: <0.535,0.267,0.802>
Distance between v and w: 2.742
Hamiltonian distance between v and w: 3.000
v != w
v = v
x: <6.000,3.000,9.000>
v: <2.000,1.000,3.000>
dot prod v.x: 42.000
v: <1.000,0.000>
v: <0.707,0.707>
x: <-0.000,1.000>
Angle between vector:
<1.000,0.000> <1.000,0.000> 0.000
<1.000,0.000> <0.866,0.500> 30.000
<1.000,0.000> <0.500,0.866> 60.000
<1.000,0.000> <0.000,1.000> 90.000
<1.000,0.000> <-0.500,0.866> 120.000
<1.000,0.000> <-0.866,0.500> 150.000
```

```
<1.000,0.000> <-1.000,0.000> -180.000
<1.000,0.000> <-0.866,-0.500> -150.000
<1.000,0.000> <-0.500,-0.866> -120.000
<1.000,0.000> <-0.000,-1.000> -90.000
<1.000,0.000> <0.500,-0.866> -60.000
<1.000,0.000> <0.866,-0.500> -30.000
<0.866,0.500> <1.000,0.000> -30.000
<0.866,0.500> <0.866,0.500> 0.000
<0.866,0.500> <0.500,0.866> 30.000
<0.866,0.500> <0.000,1.000> 60.000
<0.866,0.500> <-0.500,0.866> 90.000
<0.866,0.500> <-0.866,0.500> 120.000
<0.866,0.500> <-1.000,0.000> 150.000
<0.866,0.500> <-0.866,-0.500> -180.000
<0.866,0.500> <-0.500,-0.866> -150.000
<0.866,0.500> <-0.000,-1.000> -120.000
<0.866,0.500> <0.500,-0.866> -90.000
<0.866,0.500> <0.866,-0.500> -60.000
<0.500,0.866> <1.000,0.000> -60.000
<0.500,0.866> <0.866,0.500> -30.000
<0.500,0.866> <0.500,0.866> 0.000
<0.500,0.866> <0.000,1.000> 30.000
<0.500,0.866> <-0.500,0.866> 60.000
<0.500,0.866> <-0.866,0.500> 90.000
<0.500,0.866> <-1.000,0.000> 120.000
<0.500,0.866> <-0.866,-0.500> 150.000
<0.500,0.866> <-0.500,-0.866> 179.989
<0.500,0.866> <-0.000,-1.000> -150.000
<0.500,0.866> <0.500,-0.866> -120.000
<0.500,0.866> <0.866,-0.500> -90.000
<0.000,1.000> <1.000,0.000> -90.000
<0.000,1.000> <0.866,0.500> -60.000
<0.000,1.000> <0.500,0.866> -30.000
<0.000,1.000> <0.000,1.000> 0.000
<0.000,1.000> <-0.500,0.866> 30.000
<0.000,1.000> <-0.866,0.500> 60.000
<0.000,1.000> <-1.000,0.000> 90.000
<0.000,1.000> <-0.866,-0.500> 120.000
<0.000,1.000> <-0.500,-0.866> 150.000
<0.000,1.000> <-0.000,-1.000> 180.000
<0.000,1.000> <0.500,-0.866> -150.000
<0.000,1.000> <0.866,-0.500> -120.000
<-0.500,0.866> <1.000,0.000> -120.000
<-0.500,0.866> <0.866,0.500> -90.000
<-0.500,0.866> <0.500,0.866> -60.000
<-0.500,0.866> <0.000,1.000> -30.000
<-0.500,0.866> <-0.500,0.866> 0.000
<-0.500,0.866> <-0.866,0.500> 30.000
<-0.500,0.866> <-1.000,0.000> 60.000
<-0.500,0.866> <-0.866,-0.500> 90.000
<-0.500,0.866> <-0.500,-0.866> 120.000
<-0.500,0.866> <-0.000,-1.000> 150.000
<-0.500,0.866> <0.500,-0.866> 180.000
<-0.500,0.866> <0.866,-0.500> -150.000
<-0.866,0.500> <1.000,0.000> -150.000
<-0.866,0.500> <0.866,0.500> -120.000
<-0.866,0.500> <0.500,0.866> -90.000
<-0.866,0.500> <0.000,1.000> -60.000
<-0.866,0.500> <-0.500,0.866> -30.000
<-0.866,0.500> <-0.866,0.500> 0.000
<-0.866,0.500> <-1.000,0.000> 30.000
<-0.866,0.500> <-0.866,-0.500> 60.000
```

```
<-0.866,0.500> <-0.500,-0.866> 90.000
<-0.866,0.500> <-0.000,-1.000> 120.000
<-0.866,0.500> <0.500,-0.866> 150.000
<-0.866,0.500> <0.866,-0.500> -180.000
<-1.000,0.000> <1.000,0.000> 180.000
<-1.000,0.000> <0.866,0.500> -150.000
<-1.000,0.000> <0.500,0.866> -120.000
<-1.000,0.000> <0.000,1.000> -90.000
<-1.000,0.000> <-0.500,0.866> -60.000
<-1.000,0.000> <-0.866,0.500> -30.000
<-1.000,0.000> <-1.000,0.000> 0.000
<-1.000,0.000> <-0.866,-0.500> 30.000
<-1.000,0.000> <-0.500,-0.866> 60.000
<-1.000,0.000> <-0.000,-1.000> 90.000
<-1.000,0.000> <0.500,-0.866> 120.000
<-1.000,0.000> <0.866,-0.500> 150.000
<-0.866,-0.500> <1.000,0.000> 150.000
<-0.866,-0.500> <0.866,0.500> 180.000
<-0.866,-0.500> <0.500,0.866> -150.000
<-0.866,-0.500> <0.000,1.000> -120.000
<-0.866,-0.500> <-0.500,0.866> -90.000
<-0.866,-0.500> <-0.866,0.500> -60.000
<-0.866,-0.500> <-1.000,0.000> -30.000
<-0.866,-0.500> <-0.866,-0.500> 0.000
<-0.866,-0.500> <-0.500,-0.866> 30.000
<-0.866,-0.500> <-0.000,-1.000> 60.000
<-0.866,-0.500> <0.500,-0.866> 90.000
<-0.866,-0.500> <0.866,-0.500> 120.000
<-0.500,-0.866> <1.000,0.000> 120.000
<-0.500,-0.866> <0.866,0.500> 150.000
<-0.500,-0.866> <0.500,0.866> -180.000
<-0.500,-0.866> <0.000,1.000> -150.000
<-0.500,-0.866> <-0.500,0.866> -120.000
<-0.500,-0.866> <-0.866,0.500> -90.000
<-0.500,-0.866> <-1.000,0.000> -60.000
<-0.500,-0.866> <-0.866,-0.500> -30.000
<-0.500,-0.866> <-0.500,-0.866> 0.000
<-0.500,-0.866> <-0.000,-1.000> 30.000
<-0.500,-0.866> <0.500,-0.866> 60.000
<-0.500,-0.866> <0.866,-0.500> 90.000
<-0.000,-1.000> <1.000,0.000> 90.000
<-0.000,-1.000> <0.866,0.500> 120.000
<-0.000,-1.000> <0.500,0.866> 150.000
<-0.000,-1.000> <0.000,1.000> -180.000
<-0.000,-1.000> <-0.500,0.866> -150.000
<-0.000,-1.000> <-0.866,0.500> -120.000
<-0.000,-1.000> <-1.000,0.000> -90.000
<-0.000,-1.000> <-0.866,-0.500> -60.000
<-0.000,-1.000> <-0.500,-0.866> -30.000
<-0.000,-1.000> <-0.000,-1.000> 0.000
<-0.000,-1.000> <0.500,-0.866> 30.000
<-0.000,-1.000> <0.866,-0.500> 60.000
<0.500,-0.866> <1.000,0.000> 60.000
<0.500,-0.866> <0.866,0.500> 90.000
<0.500,-0.866> <0.500,0.866> 120.000
<0.500,-0.866> <0.000,1.000> 150.000
<0.500,-0.866> <-0.500,0.866> -179.982
<0.500,-0.866> <-0.866,0.500> -150.000
<0.500,-0.866> <-1.000,0.000> -120.000
<0.500,-0.866> <-0.866,-0.500> -90.000
<0.500,-0.866> <-0.500,-0.866> -60.000
<0.500,-0.866> <-0.000,-1.000> -30.000
```

```
<0.500,-0.866> <0.500,-0.866> 0.007
<0.500,-0.866> <0.866,-0.500> 30.000
<0.866,-0.500> <1.000,0.000> 30.000
<0.866,-0.500> <0.866,0.500> 60.000
<0.866,-0.500> <0.500,0.866> 90.000
<0.866,-0.500> <0.000,1.000> 120.000
<0.866,-0.500> <-0.500,0.866> 150.000
<0.866,-0.500> <-0.866,0.500> 180.000
<0.866,-0.500> <-1.000,0.000> -150.000
<0.866,-0.500> <-0.866,-0.500> -120.000
<0.866,-0.500> <-0.500,-0.866> -90.000
<0.866,-0.500> <-0.000,-1.000> -60.000
<0.866,-0.500> <0.500,-0.866> -30.000
<0.866,-0.500> <0.866,-0.500> 0.000
-------- MatFloat
mat:
[0.000,0.000,0.000
 0.000,0.000,0.000]
mat:
[0.500,2.000,0.000
 2.000,0.000,1.000]
cloneMatFloat:
[0.500,2.000,0.000
 2.000,0.000,1.000]
dim loaded matrix: <3,2>
0.500000 2.000000 0.000000
2.000000 0.000000 1.000000
Mat prod of
[0.500,2.000,0.000
 2.000,0.000,1.000]
and
<2.000,3.000,4.000>
equals
<7.000,8.000>
Mat prod of
[0.500,2.000,0.000
 2.000,0.000,1.000]
and
[1.000,4.000
 2.000,5.000
 3.000,6.000]
equals
[4.500,12.000
 5.000,14.000]
identity:
[1.000,0.000,0.000
 0.000,1.000,0.000
 0.000,0.000,1.000]
inverse of:
[3.000,0.000,2.000
 2.000,0.000,-2.000
 0.000,1.000,1.000]
equals
[0.200,0.200,0.000
 -0.200,0.300,1.000
 0.200,-0.300,0.000]
check of inverse:
[1.000,0.000,0.000
 0.000,1.000,0.000
 0.000,0.000,1.000]
-------- Gauss
Gauss function (mean:0.0, sigma:1.0):
```

```
-2.000 0.054
-1.800 0.079
-1.600 0.111
-1.400 0.150
-1.200 0.194
-1.000 0.242
-0.800 0.290
-0.600 0.333
-0.400 0.368
-0.200 0.391
0.000 0.399
0.200 0.391
0.400 0.368
0.600 0.333
0.800 0.290
1.000 0.242
1.200 0.194
1.400 0.150
1.600 0.111
1.800 0.079
2.000 0.054
Gauss rnd (mean:1.0, sigma:0.5):
0.287 0.564
0.887 0.237
1.482 1.580
1.625 0.441
0.863 1.325
0.765 1.440
1.172 1.070
0.833 0.801
1.156 0.548
0.901 2.097
-------- Smoother
0.000 0.000 0.000
0.100 0.028 0.009
0.200 0.104 0.058
0.300 0.216 0.163
0.400 0.352 0.317
0.500 0.500 0.500
0.600 0.648 0.683
0.700 0.784 0.837
0.800 0.896 0.942
0.900 0.972 0.991
1.000 1.000 1.000
-------- Shapoid
Type: Facoid
Dim: 2
Pos: <0.000,0.000>
Axis(0): <1.000,0.000>
Axis(1): <0.000,1.000>
scale by <2.000,3.000>
Type: Facoid
Dim: 2
Pos: <0.000,0.000>
Axis(0): <2.000,0.000>
Axis(1): <0.000,3.000>
rotate by 1.571
Type: Facoid
Dim: 2
Pos: <0.000,0.000>
Axis(0): <0.000,2.000>
Axis(1): <-3.000,0.000>
```

```
translate by <1.000,-1.000>
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,2.000>
Axis(1): <-3.000,0.000>
set axis 0 to <0.000,1.000>
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,1.000>
Axis(1): <-3.000,0.000>
save shapoid to shapoid.txt
load shapoid from shapoid.txt
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,1.000>
Axis(1): <-3.000,0.000>
coordinates <1.000,1.000> in the shapoid becomes <-2.000,0.000> in the standard coordinate system
coordinates <-2.000,0.000> in the standard coordinate system becomes <1.000,1.000> in the shapoid
<0.000,0.000> is in the facoid
<1.000,-4.000> is not in the facoid
bounding box of
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
is
Type: Facoid
Dim: 2
Pos: <-1.121,-1.000>
Axis(0): <2.828,0.000>
Axis(1): <0.000,2.828>
center of the facoid is <0.293,0.414>
bounding box of
Type: Pyramidoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
is
Type: Facoid
Dim: 2
Pos: <-1.121,-1.000>
Axis(0): <2.828,0.000>
Axis(1): <0.000,2.121>
center of the facoid is <0.529,-0.057>
bounding box of
Type: Spheroid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
is
Type: Facoid
Dim: 2
Pos: <-0.414,-2.414>
Axis(0): <2.828,0.000>
Axis(1): <0.000,2.828>
center of the shapoid is <1.000,-1.000>
```

```
bounding box of
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,1.000>
Axis(1): <-3.000,0.000>
and
Type: Facoid
Dim: 2
Pos: <2.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
is
Type: Facoid
Dim: 2
Pos: <-2.000,-1.000>
Axis(0): <4.707,0.000>
Axis(1): <0.000,2.828>
Grow the facoid:
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,1.000>
Axis(1): <-3.000,0.000>
by 2.0:
Type: Facoid
Dim: 2
Pos: <2.500,-1.500>
Axis(0): <0.000,2.000>
Axis(1): <-6.000,0.000>
Percentage of :
Type: Facoid
Dim: 2
Pos: <2.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
included in :
Type: Facoid
Dim: 2
Pos: <2.500,-1.500>
Axis(0): <0.000,2.000>
Axis(1): <-6.000,0.000>
is 0.520661
-------- Conversion functions
0.785400 radians -> 45.000069 degrees
90.000000 radians -> 1.570797 degrees
```

vecshort.txt:

```
3 0 1 0
```

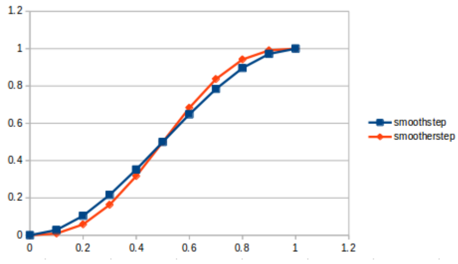
vecfloat.txt:

```
3 0.000000 1.000000 0.000000
```

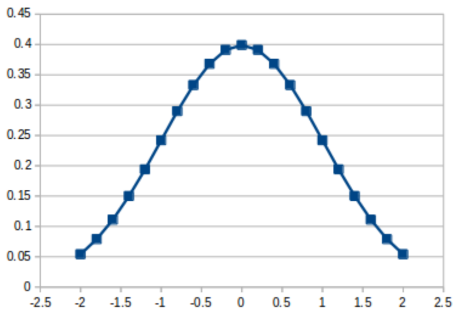
matfloat.txt:

```
3 2
0.500000 2.000000 0.000000
2.000000 0.000000 1.000000
```

smoother functions:



gauss function (mean:0.0, sigma:1.0):



gauss rand function (mean:1.0, sigma:0.5):