

# PBMath

P. Baillehache

January 17, 2018

## Contents

|          |  |            |
|----------|--|------------|
| <b>1</b> | <b>Definitions</b>                     | <b>2</b>   |
| 1.1      | Vector . . . . .                       | 2          |
| 1.1.1    | Distance between two vectors . . . . . | 2          |
| 1.1.2    | Angle between two vectors . . . . .    | 2          |
| 1.2      | Matrix . . . . .                       | 3          |
| 1.2.1    | Inverse matrix . . . . .               | 3          |
| <b>2</b> | <b>Interface</b>                       | <b>4</b>   |
| <b>3</b> | <b>Code</b>                            | <b>24</b>  |
| 3.1      | pbmath.c . . . . .                     | 24         |
| 3.2      | pbmath-inline.c . . . . .              | 40         |
| <b>4</b> | <b>Makefile</b>                        | <b>73</b>  |
| <b>5</b> | <b>Unit tests</b>                      | <b>73</b>  |
| <b>6</b> | <b>Unit tests output</b>               | <b>108</b> |
| <b>7</b> | <b>Examples</b>                        | <b>109</b> |

## Introduction

PBMath is a 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 **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.

The **EqLinSys** structure and its functions can be used to solve systems of linear equation.

It uses the **PBErr** library.

# 1 Definitions

## 1.1 Vector

### 1.1.1 Distance between two vectors

For **VecShort**:

$$\begin{aligned} Dist(\vec{v}, \vec{w}) &= \sum_i |v_i - w_i| \\ HamiltonDist(\vec{v}, \vec{w}) &= \sum_i |v_i - w_i| \\ PixelDist(\vec{v}, \vec{w}) &= \sum_i |v_i - w_i| \end{aligned} \tag{1}$$

For **VecFloat**:

$$\begin{aligned} Dist(\vec{v}, \vec{w}) &= \sum_i (v_i - w_i)^2 \\ HamiltonDist(\vec{v}, \vec{w}) &= \sum_i |v_i - w_i| \\ PixelDist(\vec{v}, \vec{w}) &= \sum_i |[v_i] - [w_i]| \end{aligned} \tag{2}$$

### 1.1.2 Angle between two vectors

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

$$M = \begin{bmatrix} Ma & Mb \\ Mc & Md \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \quad (3)$$

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

$$\begin{cases} W_x = M_a V_x + M_b V_y \\ W_y = M_c V_x + M_d V_y \end{cases} \quad (4)$$

Equivalent to

$$\begin{cases} W_x = M_a V_x + M_b V_y \\ W_y = -M_b V_x + M_a V_y \end{cases} \quad (5)$$

where  $M_a = \cos(\theta)$  and  $M_b = -\sin(\theta)$ .

If  $V_x \neq 0.0$ , we can write

$$\begin{cases} M_b = \frac{M_a V_y - W_y}{V_x} \\ M_a = \frac{W_x + W_y V_y / V_x}{V_x + V_y^2 / V_x} \end{cases} \quad (6)$$

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

$$\begin{cases} M_a = \frac{W_y + M_b V_x}{V_y} \\ M_b = \frac{W_x - W_y V_x / V_y}{V_y + V_x^2 / V_y} \end{cases} \quad (7)$$

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 Matrix

### 1.2.1 Inverse matrix

The inverse of a matrix is only implemented for square matrices less than 3x3. It is computed directly, based on the determinant and the adjoint matrix.

For a 2x2 matrix  $M$ :

$$M^{-1} = \frac{1}{\det} \begin{bmatrix} M_3 & -M_2 \\ -M_1 & M_0 \end{bmatrix} \quad (8)$$

where

$$M = \begin{bmatrix} M_0 & M_2 \\ M_1 & M_3 \end{bmatrix} \quad (9)$$

and

$$\det = M_0 M_3 - M_1 M_2 \quad (10)$$

For a 3x3 matrix  $M$ :

$$M^{-1} = \frac{1}{\det} \begin{bmatrix} (M_4 M_8 - M_5 M_7) & -(M_3 M_8 - M_5 M_6) & (M_3 M_7 - M_4 M_6) \\ -(M_1 M_8 - M_2 M_7) & (M_0 M_8 - M_2 M_6) & -(M_0 M_7 - M_1 M_6) \\ (M_1 M_5 - M_2 M_4) & -(M_0 M_5 - M_2 M_3) & (M_0 M_4 - M_1 M_3) \end{bmatrix} \quad (11)$$

where

$$M = \begin{bmatrix} M_0 & M_3 & M_6 \\ M_1 & M_4 & M_7 \\ M_2 & M_5 & M_8 \end{bmatrix} \quad (12)$$

and

$$\det = M_0(M_4 M_8 - M_5 M_7) - M_3(M_1 M_8 - M_2 M_7) + M_6(M_1 M_5 - M_2 M_4) \quad (13)$$

## 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 "pberr.h"

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

#define PBMath_EPSILON 0.00001
#define PBMath_TWOPI 6.283185307
#define PBMath_TWOPI_DIV_360 0.01745329252
#define PBMath_PI 3.141592654
#define PBMath_HALFPI 1.570796327
#define PBMath_QUARTERPI 0.7853981634
#define PBMath_SQRTTWO 1.414213562
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))
```

```

#define ISEQUALF(a,b) (fabs((a)-(b))<PBMath_EPSILON)
#define SHORT(a) ((short)(round(a)))
#define INT(a) ((int)(round(a)))
#define rnd() (float)(rand())/(float)(RAND_MAX)

// ===== Polymorphism =====

#define VecClone(V) _Generic((V), \
    VecFloat*: VecFloatClone, \
    VecShort*: VecShortClone, \
    default: PBErrInvalidPolymorphism)(V)

#define VecLoad(V, S) _Generic((V), \
    VecFloat*: VecFloatLoad, \
    VecShort*: VecShortLoad, \
    default: PBErrInvalidPolymorphism)(V, S)

#define VecSave(V, S) _Generic((V), \
    VecFloat*: VecFloatSave, \
    VecFloat2D*: VecFloatSave, \
    VecFloat3D*: VecFloatSave, \
    VecShort*: VecShortSave, \
    VecShort2D*: VecShortSave, \
    VecShort3D*: VecShortSave, \
    VecShort4D*: VecShortSave, \
    default: PBErrInvalidPolymorphism)( \
    _Generic((V), \
        VecFloat2D*: (VecFloat*)(V), \
        VecFloat3D*: (VecFloat*)(V), \
        VecShort2D*: (VecShort*)(V), \
        VecShort3D*: (VecShort*)(V), \
        VecShort4D*: (VecShort*)(V), \
        default: (V)), \
    S)

#define VecFree(V) _Generic((V), \
    VecFloat*: VecFloatFree, \
    VecShort*: VecShortFree, \
    default: PBErrInvalidPolymorphism)(V)

#define VecPrint(V, S) _Generic((V), \
    VecFloat*: VecFloatPrintDef, \
    VecFloat2D*: VecFloatPrintDef, \
    VecFloat3D*: VecFloatPrintDef, \
    VecShort*: VecShortPrint, \
    VecShort2D*: VecShortPrint, \
    VecShort3D*: VecShortPrint, \
    VecShort4D*: VecShortPrint, \
    default: PBErrInvalidPolymorphism)( \
    _Generic((V), \
        VecFloat2D*: (VecFloat*)(V), \
        VecFloat3D*: (VecFloat*)(V), \
        VecShort2D*: (VecShort*)(V), \
        VecShort3D*: (VecShort*)(V), \
        VecShort4D*: (VecShort*)(V), \
        default: (V)), \
    S)

#define VecGet(V, I) _Generic((V), \
    VecFloat*: VecFloatGet, \
    VecFloat2D*: VecFloatGet2D, \
    VecFloat3D*: VecFloatGet3D, \

```

```

VecShort*: VecShortGet, \
VecShort2D*: VecShortGet2D, \
VecShort3D*: VecShortGet3D, \
VecShort4D*: VecShortGet4D, \
default: PBErrInvalidPolymorphism)(V, I)

#define VecSet(V, I, VAL) _Generic((V), \
    VecFloat*: VecFloatSet, \
    VecFloat2D*: VecFloatSet2D, \
    VecFloat3D*: VecFloatSet3D, \
    VecShort*: VecShortSet, \
    VecShort2D*: VecShortSet2D, \
    VecShort3D*: VecShortSet3D, \
    VecShort4D*: VecShortSet4D, \
    default: PBErrInvalidPolymorphism)(V, I, VAL)

#define VecSetNull(V) _Generic((V), \
    VecFloat*: VecFloatSetNull, \
    VecFloat2D*: VecFloatSetNull2D, \
    VecFloat3D*: VecFloatSetNull3D, \
    VecShort*: VecShortSetNull, \
    VecShort2D*: VecShortSetNull2D, \
    VecShort3D*: VecShortSetNull3D, \
    VecShort4D*: VecShortSetNull4D, \
    default: PBErrInvalidPolymorphism)(V)

#define VecCopy(V, W) _Generic((V), \
    VecFloat*: _Generic((W), \
        VecFloat*: VecFloatCopy, \
        VecFloat2D*: VecFloatCopy, \
        VecFloat3D*: VecFloatCopy, \
        default: PBErrInvalidPolymorphism), \
    VecFloat2D*: _Generic((W), \
        VecFloat*: VecFloatCopy, \
        VecFloat2D*: VecFloatCopy, \
        default: PBErrInvalidPolymorphism), \
    VecFloat3D*: _Generic((W), \
        VecFloat*: VecFloatCopy, \
        VecFloat3D*: VecFloatCopy, \
        default: PBErrInvalidPolymorphism), \
    VecShort*: _Generic((W), \
        VecShort*: VecShortCopy, \
        VecShort2D*: VecShortCopy, \
        VecShort3D*: VecShortCopy, \
        VecShort4D*: VecShortCopy, \
        default: PBErrInvalidPolymorphism), \
    VecShort2D*: _Generic((W), \
        VecShort*: VecShortCopy, \
        VecShort2D*: VecShortCopy, \
        default: PBErrInvalidPolymorphism), \
    VecShort3D*: _Generic((W), \
        VecShort*: VecShortCopy, \
        VecShort3D*: VecShortCopy, \
        default: PBErrInvalidPolymorphism), \
    VecShort4D*: _Generic((W), \
        VecShort*: VecShortCopy, \
        VecShort4D*: VecShortCopy, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)( \
    _Generic((V), \
        VecFloat2D*: (VecFloat*)(V), \
        VecFloat3D*: (VecFloat*)(V), \

```

```

    VecShort2D*: (VecShort*)(V), \
    VecShort3D*: (VecShort*)(V), \
    VecShort4D*: (VecShort*)(V), \
    default: (V)), \
    _Generic((W), \
        VecFloat2D*: (VecFloat*)(W), \
        VecFloat3D*: (VecFloat*)(W), \
        VecShort2D*: (VecShort*)(W), \
        VecShort3D*: (VecShort*)(W), \
        VecShort4D*: (VecShort*)(W), \
        default: (W)))

#define VecDim(V) _Generic((V), \
    VecFloat*: VecFloatDim, \
    VecShort*: VecShortDim, \
    default: PBErrInvalidPolymorphism)(V)

#define VecNorm(V) _Generic((V), \
    VecFloat*: VecFloatNorm, \
    VecFloat2D*: VecFloatNorm2D, \
    VecFloat3D*: VecFloatNorm3D, \
    default: PBErrInvalidPolymorphism)(V)

#define VecNormalise(V) _Generic((V), \
    VecFloat*: VecFloatNormalise, \
    VecFloat2D*: VecFloatNormalise2D, \
    VecFloat3D*: VecFloatNormalise3D, \
    default: PBErrInvalidPolymorphism)(V)

#define VecDist(V, W) _Generic((V), \
    VecFloat*: _Generic((W), \
        VecFloat*: VecFloatDist, \
        default: PBErrInvalidPolymorphism), \
    VecFloat2D*: _Generic((W), \
        VecFloat2D*: VecFloatDist2D, \
        default: PBErrInvalidPolymorphism), \
    VecFloat3D*: _Generic((W), \
        VecFloat3D*: VecFloatDist3D, \
        default: PBErrInvalidPolymorphism), \
    VecShort*: _Generic((W), \
        VecShort*: VecShortHamiltonDist, \
        default: PBErrInvalidPolymorphism), \
    VecShort2D*: _Generic((W), \
        VecShort2D*: VecShortHamiltonDist2D, \
        default: PBErrInvalidPolymorphism), \
    VecShort3D*: _Generic((W), \
        VecShort3D*: VecShortHamiltonDist3D, \
        default: PBErrInvalidPolymorphism), \
    VecShort4D*: _Generic((W), \
        VecShort4D*: VecShortHamiltonDist4D, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)(V, W)

#define VecHamiltonDist(V, W) _Generic((V), \
    VecFloat*: _Generic((W), \
        VecFloat*: VecFloatHamiltonDist, \
        default: PBErrInvalidPolymorphism), \
    VecFloat2D*: _Generic((W), \
        VecFloat2D*: VecFloatHamiltonDist2D, \
        default: PBErrInvalidPolymorphism), \
    VecFloat3D*: _Generic((W), \
        VecFloat3D*: VecFloatHamiltonDist3D, \

```

```

        default: PBErInvalidPolymorphism), \
VecShort*: _Generic((W), \
    VecShort*: VecShortHamiltonDist,\
    default: PBErInvalidPolymorphism), \
VecShort2D*: _Generic((W), \
    VecShort2D*: VecShortHamiltonDist2D,\
    default: PBErInvalidPolymorphism), \
VecShort3D*: _Generic((W), \
    VecShort3D*: VecShortHamiltonDist3D,\
    default: PBErInvalidPolymorphism), \
VecShort4D*: _Generic((W), \
    VecShort4D*: VecShortHamiltonDist4D,\
    default: PBErInvalidPolymorphism), \
default: PBErInvalidPolymorphism)(V, W)

#define VecPixelDist(V, W) _Generic((V), \
    VecFloat*: _Generic((W), \
        VecFloat*: VecFloatPixelDist, \
        default: PBErInvalidPolymorphism), \
    VecFloat2D*: _Generic((W), \
        VecFloat2D*: VecFloatPixelDist2D, \
        default: PBErInvalidPolymorphism), \
    VecFloat3D*: _Generic((W), \
        VecFloat3D*: VecFloatPixelDist3D, \
        default: PBErInvalidPolymorphism), \
    VecShort*: _Generic((W), \
        VecShort*: VecShortHamiltonDist,\
        default: PBErInvalidPolymorphism), \
    VecShort2D*: _Generic((W), \
        VecShort2D*: VecShortHamiltonDist2D,\
        default: PBErInvalidPolymorphism), \
    VecShort3D*: _Generic((W), \
        VecShort3D*: VecShortHamiltonDist3D,\
        default: PBErInvalidPolymorphism), \
    VecShort4D*: _Generic((W), \
        VecShort4D*: VecShortHamiltonDist4D,\
        default: PBErInvalidPolymorphism), \
    default: PBErInvalidPolymorphism)(V, W)

#define VecIsEqual(V, W) _Generic((V), \
    VecFloat*: _Generic((W), \
        VecFloat*: VecFloatIsEqual, \
        default: PBErInvalidPolymorphism), \
    VecFloat2D*: _Generic((W), \
        VecFloat2D*: VecFloatIsEqual2D, \
        default: PBErInvalidPolymorphism), \
    VecFloat3D*: _Generic((W), \
        VecFloat3D*: VecFloatIsEqual3D, \
        default: PBErInvalidPolymorphism), \
    VecShort*: _Generic((W), \
        VecShort*: VecShortIsEqual,\
        default: PBErInvalidPolymorphism), \
    VecShort2D*: _Generic((W), \
        VecShort2D*: VecShortIsEqual2D,\
        default: PBErInvalidPolymorphism), \
    VecShort3D*: _Generic((W), \
        VecShort3D*: VecShortIsEqual3D,\
        default: PBErInvalidPolymorphism), \
    VecShort4D*: _Generic((W), \
        VecShort4D*: VecShortIsEqual4D,\
        default: PBErInvalidPolymorphism), \
    default: PBErInvalidPolymorphism)(V, W)

```



```

#define VecOp(V, A, W, B) _Generic((V), \
    VecFloat*: _Generic((W), \
        VecFloat*: VecFloatOp, \
        default: PBErrInvalidPolymorphism), \
    VecFloat2D*: _Generic((W), \
        VecFloat2D*: VecFloatOp2D, \
        default: PBErrInvalidPolymorphism), \
    VecFloat3D*: _Generic((W), \
        VecFloat3D*: VecFloatOp3D, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)(V, A, W, B)

#define VecGetOp(V, A, W, B) _Generic((V), \
    VecFloat*: _Generic((W), \
        VecFloat*: VecFloatGetOp, \
        default: PBErrInvalidPolymorphism), \
    VecFloat2D*: _Generic((W), \
        VecFloat2D*: VecFloatGetOp2D, \
        default: PBErrInvalidPolymorphism), \
    VecFloat3D*: _Generic((W), \
        VecFloat3D*: VecFloatGetOp3D, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)(V, A, W, B)

#define VecScale(V, A) _Generic((V), \
    VecFloat*: VecFloatScale, \
    VecFloat2D*: VecFloatScale2D, \
    VecFloat3D*: VecFloatScale3D, \
    default: PBErrInvalidPolymorphism)(V, A)

#define VecGetScale(V, A) _Generic((V), \
    VecFloat*: VecFloatGetScale, \
    VecFloat2D*: VecFloatGetScale2D, \
    VecFloat3D*: VecFloatGetScale3D, \
    default: PBErrInvalidPolymorphism)(V, A)

#define VecRot(V, A) _Generic((V), \
    VecFloat*: VecFloatRot2D, \
    VecFloat2D*: VecFloatRot2D, \
    default: PBErrInvalidPolymorphism)((VecFloat2D*)V, A)

#define VecGetRot(V, A) _Generic((V), \
    VecFloat2D*: VecFloatGetRot2D, \
    default: PBErrInvalidPolymorphism)(V, A)

#define VecDotProd(V, W) _Generic((V), \
    VecShort*: VecShortDotProd, \
    VecShort2D*: VecShortDotProd2D, \
    VecShort3D*: VecShortDotProd3D, \
    VecShort4D*: VecShortDotProd4D, \
    VecFloat*: VecFloatDotProd, \
    VecFloat2D*: VecFloatDotProd2D, \
    VecFloat3D*: VecFloatDotProd3D, \
    default: PBErrInvalidPolymorphism)(V, W) \

#define VecAngleTo(V, W) _Generic((V), \
    VecFloat*: VecFloatAngleTo2D, \
    VecFloat2D*: VecFloatAngleTo2D, \
    default: PBErrInvalidPolymorphism)((VecFloat2D*)V, (VecFloat2D*)W)

#define VecStep(V, W) _Generic((V), \

```

```

VecShort*: VecShortStep, \
VecShort2D*: VecShortStep, \
VecShort3D*: VecShortStep, \
VecShort4D*: VecShortStep, \
default: PBErrInvalidPolymorphism)((VecShort*)V, (VecShort*)W)

#define VecPStep(V, W) _Generic((V), \
    VecShort*: VecShortPStep, \
    VecShort2D*: VecShortPStep, \
    VecShort3D*: VecShortPStep, \
    VecShort4D*: VecShortPStep, \
    default: PBErrInvalidPolymorphism)((VecShort*)V, (VecShort*)W)

#define MatClone(M) _Generic((M), \
    MatFloat*: MatFloatClone, \
    default: PBErrInvalidPolymorphism)(M)

#define MatLoad(M, S) _Generic((M), \
    MatFloat*: MatFloatLoad, \
    default: PBErrInvalidPolymorphism)(M, S)

#define MatSave(M, S) _Generic((M), \
    MatFloat*: MatFloatSave, \
    default: PBErrInvalidPolymorphism)(M, S)

#define MatFree(M) _Generic((M), \
    MatFloat*: MatFloatFree, \
    default: PBErrInvalidPolymorphism)(M)

#define MatPrintln(M, S) _Generic((M), \
    MatFloat*: MatFloatPrintlnDef, \
    default: PBErrInvalidPolymorphism)(M, S)

#define MatGet(M, I) _Generic((M), \
    MatFloat*: MatFloatGet, \
    default: PBErrInvalidPolymorphism)(M, I)

#define MatSet(M, I, VAL) _Generic((M), \
    MatFloat*: MatFloatSet, \
    default: PBErrInvalidPolymorphism)(M, I, VAL)

#define MatCopy(M, W) _Generic((M), \
    MatFloat*: _Generic ((W), \
        MatFloat*: MatFloatCopy, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)(M, W)

#define MatDim(M) _Generic((M), \
    MatFloat*: MatFloatDim, \
    default: PBErrInvalidPolymorphism)(M)

#define MatInv(M) _Generic((M), \
    MatFloat*: MatFloatInv, \
    default: PBErrInvalidPolymorphism)(M)

#define MatProdMat(A, B) _Generic(A, \
    MatFloat*: _Generic(B, \
        MatFloat*: MatFloatProdMatFloat, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)((A), (B))

#define MatProdVec(A, B) _Generic(A, \

```

```

MatFloat*: _Generic(B, \
    VecFloat*: MatFloatProdVecFloat, \
    VecFloat2D*: MatFloatProdVecFloat, \
    VecFloat3D*: MatFloatProdVecFloat, \
    default: PBErrInvalidPolymorphism), \
default: PBErrInvalidPolymorphism)((A), (VecFloat*)(B))

#define MatSetIdentity(M) _Generic((M), \
    MatFloat*: MatFloatSetIdentity, \
    default: PBErrInvalidPolymorphism)(M)

#define MatIsEqual(A, B) _Generic(A, \
    MatFloat*: _Generic(B, \
        MatFloat*: MatFloatIsEqual, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)((A), (B))

#define SysLinEqCreate(M, V) _Generic((V), \
    VecFloat*: SLECreate, \
    VecFloat2D*: SLECreate, \
    VecFloat3D*: SLECreate, \
    default: PBErrInvalidPolymorphism)(M, (VecFloat*)(V))

#define SysLinEqSetV(S, V) _Generic((V), \
    VecFloat*: SLESetV, \
    VecFloat2D*: SLESetV, \
    VecFloat3D*: SLESetV, \
    default: PBErrInvalidPolymorphism)(S, (VecFloat*)(V))

// ----- VecShort

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

// Vector of short values
typedef struct VecShort {
    // Dimension
    int _dim;
    // Values
    short _val[0];
} VecShort;

typedef struct VecShort2D {
    // Dimension
    int _dim;
    // Values
    short _val[2];
} VecShort2D;

typedef struct VecShort3D {
    // Dimension
    int _dim;
    // Values
    short _val[3];
} VecShort3D;

typedef struct VecShort4D {
    // Dimension
    int _dim;
    // Values
    short _val[4];
} VecShort4D;

```

```

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

// Create a new VecShort of dimension 'dim'
// Values are initialized to 0.0
VecShort* VecShortCreate(int dim);

// Static constructors for VecShort
#if BUILDMODE != 0
inline
#endif
VecShort2D VecShortCreateStatic2D();
#if BUILDMODE != 0
inline
#endif
VecShort3D VecShortCreateStatic3D();
#if BUILDMODE != 0
inline
#endif
VecShort4D VecShortCreateStatic4D();

// 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 true in case of success, else false
bool VecShortLoad(VecShort **that, FILE *stream);

// Save the VecShort to the stream
// Return true in case of success, else false
bool 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'
void VecShortPrint(VecShort *that, FILE *stream);

// Return the i-th value of the VecShort
#if BUILDMODE != 0
inline
#endif
short VecShortGet(VecShort *that, int i);
#if BUILDMODE != 0
inline
#endif
short VecShortGet2D(VecShort2D *that, int i);
#if BUILDMODE != 0
inline
#endif
short VecShortGet3D(VecShort3D *that, int i);
#if BUILDMODE != 0
inline
#endif
short VecShortGet4D(VecShort4D *that, int i);

// Set the i-th value of the VecShort to v
#if BUILDMODE != 0
inline
#endif

```

```

void VecShortSet(VecShort *that, int i, short v);
#if BUILDMODE != 0
inline
#endif
void VecShortSet2D(VecShort2D *that, int i, short v);
#if BUILDMODE != 0
inline
#endif
void VecShortSet3D(VecShort3D *that, int i, short v);
#if BUILDMODE != 0
inline
#endif
void VecShortSet4D(VecShort4D *that, int i, short v);

// Return the dimension of the VecShort
// Return 0 if arguments are invalid
#if BUILDMODE != 0
inline
#endif
int VecShortDim(VecShort *that);

// Return the Hamiltonian distance between the VecShort 'that' and 'tho'
#if BUILDMODE != 0
inline
#endif
short VecShortHamiltonDist(VecShort *that, VecShort *tho);
#if BUILDMODE != 0
inline
#endif
short VecShortHamiltonDist2D(VecShort2D *that, VecShort2D *tho);
#if BUILDMODE != 0
inline
#endif
short VecShortHamiltonDist3D(VecShort3D *that, VecShort3D *tho);
#if BUILDMODE != 0
inline
#endif
short VecShortHamiltonDist4D(VecShort4D *that, VecShort4D *tho);

// Return true if the VecShort 'that' is equal to 'tho', else false
#if BUILDMODE != 0
inline
#endif
bool VecShortIsEqual(VecShort *that, VecShort *tho);
#if BUILDMODE != 0
inline
#endif
bool VecShortIsEqual2D(VecShort2D *that, VecShort2D *tho);
#if BUILDMODE != 0
inline
#endif
bool VecShortIsEqual3D(VecShort3D *that, VecShort3D *tho);
#if BUILDMODE != 0
inline
#endif
bool VecShortIsEqual4D(VecShort4D *that, VecShort4D *tho);

// Copy the values of 'w' in 'that' (must have same dimensions)
#if BUILDMODE != 0
inline
#endif
void VecShortCopy(VecShort *that, VecShort *w);

```

```

// Return the dot product of 'that' and 'tho'
#if BUILDMODE != 0
inline
#endif
short VecShortDotProd(VecShort *that, VecShort *tho);
#if BUILDMODE != 0
inline
#endif
short VecShortDotProd2D(VecShort2D *that, VecShort2D *tho);
#if BUILDMODE != 0
inline
#endif
short VecShortDotProd3D(VecShort3D *that, VecShort3D *tho);
#if BUILDMODE != 0
inline
#endif
short VecShortDotProd4D(VecShort4D *that, VecShort4D *tho);

// Set all values of the vector 'that' to 0
#if BUILDMODE != 0
inline
#endif
void VecShortSetNull(VecShort *that);
#if BUILDMODE != 0
inline
#endif
void VecShortSetNull2D(VecShort2D *that);
#if BUILDMODE != 0
inline
#endif
void VecShortSetNull3D(VecShort3D *that);
#if BUILDMODE != 0
inline
#endif
void VecShortSetNull4D(VecShort4D *that);

// Step the values of the vector incrementally by 1 from 0
// in the following order (for example) :
// (0,0,0)->(0,0,1)->(0,0,2)->(0,1,0)->(0,1,1)->...
// The upper limit for each value is given by 'bound' (val[i] < dim[i])
// Return false if all values of 'that' have reached their upper limit
// (in which case 'that''s values are all set back to 0)
// Return true else
bool VecShortStep(VecShort *that, VecShort *bound);

// Step the values of the vector incrementally by 1 from 0
// in the following order (for example) :
// (0,0,0)->(1,0,0)->(2,0,0)->(0,1,0)->(1,1,0)->...
// The upper limit for each value is given by 'bound' (val[i] < dim[i])
// Return false if all values of 'that' have reached their upper limit
// (in which case 'that''s values are all set back to 0)
// Return true else
bool VecShortPStep(VecShort *that, VecShort *bound);

// ----- VecFloat

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

// Vector of float values
typedef struct VecFloat {
    // Dimension

```

```

    int _dim;
    // Values
    float _val[0];
} VecFloat;

typedef struct VecFloat2D {
    // Dimension
    int _dim;
    // Values
    float _val[2];
} VecFloat2D;

typedef struct VecFloat3D {
    // Dimension
    int _dim;
    // Values
    float _val[3];
} VecFloat3D;

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

// Create a new VecFloat of dimension 'dim'
// Values are initialized to 0.0
VecFloat* VecFloatCreate(int dim);

// Static constructors for VecFloat
#if BUILDMODE != 0
inline
#endif
VecFloat2D VecFloatCreateStatic2D();
#if BUILDMODE != 0
inline
#endif
VecFloat3D VecFloatCreateStatic3D();

// 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 true in case of success, else false
bool VecFloatLoad(VecFloat **that, FILE *stream);

// Save the VecFloat to the stream
// Return true in case of success, else false
bool 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, unsigned int prec);
inline void VecFloatPrintDef(VecFloat *that, FILE *stream) {
    VecFloatPrint(that, stream, 3);
}

// Return the 'i'-th value of the VecFloat
#if BUILDMODE != 0
inline
#endif

```

```

float VecFloatGet(VecFloat *that, int i);
#if BUILDMODE != 0
inline
#endif
float VecFloatGet2D(VecFloat2D *that, int i);
#if BUILDMODE != 0
inline
#endif
float VecFloatGet3D(VecFloat3D *that, int i);

// Set the 'i'-th value of the VecFloat to 'v'
#if BUILDMODE != 0
inline
#endif
void VecFloatSet(VecFloat *that, int i, float v);
#if BUILDMODE != 0
inline
#endif
void VecFloatSet2D(VecFloat2D *that, int i, float v);
#if BUILDMODE != 0
inline
#endif
void VecFloatSet3D(VecFloat3D *that, int i, float v);

// Set all values of the vector 'that' to 0
#if BUILDMODE != 0
inline
#endif
void VecFloatSetNull(VecFloat *that);
#if BUILDMODE != 0
inline
#endif
void VecFloatSetNull2D(VecFloat2D *that);
#if BUILDMODE != 0
inline
#endif
void VecFloatSetNull3D(VecFloat3D *that);

// Return the dimension of the VecFloat
// Return 0 if arguments are invalid
#if BUILDMODE != 0
inline
#endif
int VecFloatDim(VecFloat *that);

// Copy the values of 'w' in 'that' (must have same dimensions)
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void VecFloatCopy(VecFloat *that, VecFloat *w);

// Return the norm of the VecFloat
// Return 0.0 if arguments are invalid
#if BUILDMODE != 0
inline
#endif
float VecFloatNorm(VecFloat *that);
#if BUILDMODE != 0
inline
#endif
float VecFloatNorm2D(VecFloat2D *that);

```



```

#if BUILDMODE != 0
inline
#endif
float VecFloatNorm3D(VecFloat3D *that);

// Normalise the VecFloat
#if BUILDMODE != 0
inline
#endif
void VecFloatNormalise(VecFloat *that);
#if BUILDMODE != 0
inline
#endif
void VecFloatNormalise2D(VecFloat2D *that);
#if BUILDMODE != 0
inline
#endif
void VecFloatNormalise3D(VecFloat3D *that);

// Return the distance between the VecFloat 'that' and 'tho'
#if BUILDMODE != 0
inline
#endif
float VecFloatDist(VecFloat *that, VecFloat *tho);
#if BUILDMODE != 0
inline
#endif
float VecFloatDist2D(VecFloat2D *that, VecFloat2D *tho);
#if BUILDMODE != 0
inline
#endif
float VecFloatDist3D(VecFloat3D *that, VecFloat3D *tho);

// Return the Hamiltonian distance between the VecFloat 'that' and 'tho'
#if BUILDMODE != 0
inline
#endif
float VecFloatHamiltonDist(VecFloat *that, VecFloat *tho);
#if BUILDMODE != 0
inline
#endif
float VecFloatHamiltonDist2D(VecFloat2D *that, VecFloat2D *tho);
#if BUILDMODE != 0
inline
#endif
float VecFloatHamiltonDist3D(VecFloat3D *that, VecFloat3D *tho);

// Return the Pixel distance between the VecFloat 'that' and 'tho'
#if BUILDMODE != 0
inline
#endif
float VecFloatPixelDist(VecFloat *that, VecFloat *tho);
#if BUILDMODE != 0
inline
#endif
float VecFloatPixelDist2D(VecFloat2D *that, VecFloat2D *tho);
#if BUILDMODE != 0
inline
#endif
float VecFloatPixelDist3D(VecFloat3D *that, VecFloat3D *tho);

// Return true if the VecFloat 'that' is equal to 'tho', else false

```

```

#if BUILDMODE != 0
inline
#endif
bool VecFloatIsEqual(VecFloat *that, VecFloat *tho);
#if BUILDMODE != 0
inline
#endif
bool VecFloatIsEqual2D(VecFloat2D *that, VecFloat2D *tho);
#if BUILDMODE != 0
inline
#endif
bool VecFloatIsEqual3D(VecFloat3D *that, VecFloat3D *tho);

// Calculate (that * a) and store the result in 'that'
#if BUILDMODE != 0
inline
#endif
void VecFloatScale(VecFloat *that, float a);
#if BUILDMODE != 0
inline
#endif
void VecFloatScale2D(VecFloat2D *that, float a);
#if BUILDMODE != 0
inline
#endif
void VecFloatScale3D(VecFloat3D *that, float a);

// Return a VecFloat equal to (that * a)
#if BUILDMODE != 0
inline
#endif
VecFloat* VecFloatGetScale(VecFloat *that, float a);
#if BUILDMODE != 0
inline
#endif
VecFloat2D VecFloatGetScale2D(VecFloat2D *that, float a);
#if BUILDMODE != 0
inline
#endif
VecFloat3D VecFloatGetScale3D(VecFloat3D *that, float a);

// Calculate (that * a + tho * b) and store the result in 'that'
// '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'
#if BUILDMODE != 0
inline
#endif
void VecFloatOp(VecFloat *that, float a, VecFloat *tho, float b);
#if BUILDMODE != 0
inline
#endif
void VecFloatOp2D(VecFloat2D *that, float a, VecFloat2D *tho, float b);
#if BUILDMODE != 0
inline
#endif
void VecFloatOp3D(VecFloat3D *that, float a, VecFloat3D *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'
#if BUILDMODE != 0

```

```

inline
#endif
VecFloat* VecFloatGetOp(VecFloat *that, float a,
    VecFloat *tho, float b);
#if BUILDMODE != 0
inline
#endif
VecFloat2D VecFloatGetOp2D(VecFloat2D *that, float a,
    VecFloat2D *tho, float b);
#if BUILDMODE != 0
inline
#endif
VecFloat3D VecFloatGetOp3D(VecFloat3D *that, float a,
    VecFloat3D *tho, float b);

// Rotate CCW 'that' by 'theta' radians and store the result in 'that'
#if BUILDMODE != 0
inline
#endif
void VecFloatRot2D(VecFloat2D *that, float theta);

// Return a VecFloat equal to 'that' rotated CCW by 'theta' radians
#if BUILDMODE != 0
inline
#endif
VecFloat2D VecFloatGetRot2D(VecFloat2D *that, float theta);

// Return the dot product of 'that' and 'tho'
#if BUILDMODE != 0
inline
#endif
float VecFloatDotProd(VecFloat *that, VecFloat *tho);
#if BUILDMODE != 0
inline
#endif
float VecFloatDotProd2D(VecFloat2D *that, VecFloat2D *tho);
#if BUILDMODE != 0
inline
#endif
float VecFloatDotProd3D(VecFloat3D *that, VecFloat3D *tho);

// Return the angle of the rotation making 'that' colinear to 'tho'
// 'that' and 'tho' must be normalised
// Return a value in [-PI,PI]
float VecFloatAngleTo2D(VecFloat2D *that, VecFloat2D *tho);

// Return the conversion of VecFloat 'that' to a VecShort using round()
#if BUILDMODE != 0
inline
#endif
VecShort* VecFloatToShort(VecFloat *that);
#if BUILDMODE != 0
inline
#endif
VecShort2D VecFloatToShort2D(VecFloat2D *that);
#if BUILDMODE != 0
inline
#endif
VecShort3D VecFloatToShort3D(VecFloat3D *that);

// Return the conversion of VecShort 'that' to a VecFloat
#if BUILDMODE != 0

```

```

inline
#endif
VecFloat* VecShortToFloat(VecShort *that);
#if BUILDMODE != 0
inline
#endif
VecFloat2D VecShortToFloat2D(VecShort2D *that);
#if BUILDMODE != 0
inline
#endif
VecFloat3D VecShortToFloat3D(VecShort3D *that);

// ----- MatFloat

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

// Vector of float values
typedef struct MatFloat {
    // Dimension
    VecShort2D _dim;
    // Values (memorized by columns)
    float _val[0];
} MatFloat;

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

// Create a new MatFloat of dimension 'dim' (nbCol, nbLine)
// Values are initialized to 0.0
MatFloat* MatFloatCreate(VecShort2D *dim);

// Set the MatFloat to the identity matrix
// The matrix must be a square matrix
#if BUILDMODE != 0
inline
#endif
void MatFloatSetIdentity(MatFloat *that);

// Clone the MatFloat
MatFloat* MatFloatClone(MatFloat *that);

// Copy the values of 'w' in 'that' (must have same dimensions)
#if BUILDMODE != 0
inline
#endif
void MatFloatCopy(MatFloat *that, MatFloat *w);

// Load the MatFloat from the stream
// If the MatFloat is already allocated, it is freed before loading
// Return true upon success, else false
bool MatFloatLoad(MatFloat **that, FILE *stream);

// Save the MatFloat to the stream
// Return true upon success, else false
bool 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 MatFloatPrintln(MatFloat *that, FILE *stream, unsigned int prec);

```

```

inline void MatFloatPrintlnDef(MatFloat *that, FILE *stream) {
    MatFloatPrintln(that, stream, 3);
}

// Return the value at index (col, line) of the MatFloat
// Index starts at 0, index in matrix = line * nbCol + col
#if BUILDMODE != 0
inline
#endif
float MatFloatGet(MatFloat *that, VecShort2D *index);

// Set the value at index (col, line) of the MatFloat to 'v'
// Index starts at 0, index in matrix = line * nbCol + col
#if BUILDMODE != 0
inline
#endif
void MatFloatSet(MatFloat *that, VecShort2D *index, float v);

// Return a VecShort2D containing the dimension of the MatFloat
#if BUILDMODE != 0
inline
#endif
VecShort2D MatFloatDim(MatFloat *that);

// Return the inverse matrix of 'that'
// The matrix must be a square matrix
MatFloat* MatFloatInv(MatFloat *that);

// Return the product of matrix 'that' and vector 'v'
// Number of columns of 'that' must equal dimension of 'v'
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'
MatFloat* MatFloatProdMatFloat(MatFloat *that, MatFloat *tho);

// Return true if 'that' is equal to 'tho', false else
bool MatFloatIsEqual(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);
Gauss GaussCreateStatic(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'
#if BUILDMODE != 0
inline
#endif
float GaussGet(Gauss *that, float x);

// Return a random value according to the Gauss 'that'
// random() must have been called before calling this function
#if BUILDMODE != 0
inline
#endif
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
#if BUILDMODE != 0
inline
#endif
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
#if BUILDMODE != 0
inline
#endif
float SmootherStep(float x);

// ----- Conversion functions

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

// Convert radians to degrees
inline float ConvRad2Deg(float rad) {
    return rad / PBMath_TWOPI_DIV_360;
}

// Convert degrees to radians
inline float ConvDeg2Rad(float deg) {
    return PBMath_TWOPI_DIV_360 * deg;
}

// ----- SysLinEq

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

// Linear system of equalities
typedef struct SysLinEq {
    // Matrix
    MatFloat *_M;
    // Inverse of the matrix

```

```

    MatFloat *_Minv;
    // Vector
    VecFloat *_V;
} SysLinEq;

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

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

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

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

// Solve the SysLinEq  $M \cdot x = V$ 
// Return the solution vector, or null if there is no solution or the
// arguments are invalid
#if BUILDMODE != 0
inline
#endif
VecFloat* SysLinEqSolve(SysLinEq *that);

// Set the matrix of the SysLinEq to a clone of 'm'
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void SysLinEqSetM(SysLinEq *that, MatFloat *m);

// Set the vector of the SysLinEq to a clone of 'v'
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void SLESetV(SysLinEq *that, VecFloat *v);

// ----- Usefull basic functions

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

// Return  $x^y$  when x and y are int
// to avoid numerical imprecision from (pow(double,double)
// From https://stackoverflow.com/questions/29787310/
// does-pow-work-for-int-data-type-in-c
#if BUILDMODE != 0
inline
#endif
int powi(int base, int exp);

// Compute  $a^n$ , faster than std::pow for  $n < 100$ 
inline float fastpow(float a, int n) {
    double ret = 1.0;
    for (; n--;) ret *= (double)a;
    return (float)ret;
}

```

```

}

// Compute a^2
inline float fsquare(float a) {
    return a * a;
}

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

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

#endif

```

## 3 Code

### 3.1 pbmath.c

```

// ===== PBMath.C =====

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

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

// ----- VecShort

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

// Create a new Vec of dimension 'dim'
// Values are initialized to 0.0
VecShort* VecShortCreate(int dim) {
    #if BUILDMODE == 0
        if (dim <= 0) {
            PBMathErr->_type = PBErrTypeInvalidArg;
            sprintf(PBMathErr->_msg, "invalid 'dim' (%d)", dim);
            PBErrCatch(PBMathErr);
        }
    #endif
    // Allocate memory
    VecShort *that = PBErrMalloc(PBMathErr,
        sizeof(VecShort) + sizeof(short) * dim);
    // Set the default values
    that->_dim = dim;
    for (int i = dim; i--;)
        that->_val[i] = 0;
    // Return the new VecShort
    return that;
}

// Clone the VecShort
// Return NULL if we couldn't clone the VecShort
VecShort* VecShortClone(VecShort *that) {
    #if BUILDMODE == 0

```



```

    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create a clone
    VecShort *clone = VecShortCreate(that->_dim);
    // Copy the values
    memcpy(clone, that, sizeof(VecShort) + sizeof(short) * that->_dim);
    // Return the clone
    return clone;
}

// Load the VecShort from the stream
// If the VecShort is already allocated, it is freed before loading
// Return true in case of success, else false
bool VecShortLoad(VecShort **that, FILE *stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (stream == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'stream' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // 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 couldn't fscanf
    if (ret == EOF)
        return false;
    // Check the dimension
    if (dim <= 0)
        return false;
    // Allocate memory
    *that = VecShortCreate(dim);
    // Read the values
    for (int i = 0; i < dim; ++i) {
        ret = fscanf(stream, "%hi", (*that)->_val + i);
        // If we couldn't fscanf
        if (ret == EOF)
            return false;
    }
    // Return success code
    return true;
}

// Save the VecShort to the stream
// Return true in case of success, else false
bool VecShortSave(VecShort *that, FILE *stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }
    if (stream == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'stream' is null");
        PBErCatch(PBMathErr);
    }
#endif
    // Save the dimension
    int ret = fprintf(stream, "%d ", that->_dim);
    // If we couldn't fprintf
    if (ret < 0)
        return false;
    // Save the values
    for (int i = 0; i < that->_dim; ++i) {
        ret = fprintf(stream, "%hi ", that->_val[i]);
        // If we couldn't fprintf
        if (ret < 0)
            return false;
    }
    fprintf(stream, "\n");
    // If we couldn't fprintf
    if (ret < 0)
        return false;
    // Return success code
    return true;
}

// 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);
    *that = NULL;
}

// Print the VecShort on 'stream' with 'prec' digit precision
void VecShortPrint(VecShort *that, FILE *stream) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }
    if (stream == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'stream' is null");
        PBErCatch(PBMathErr);
    }
#endif
    // 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, ">");
}

```

```

}

// Step the values of the vector incrementally by 1 from 0
// in the following order (for example) :
// (0,0,0)->(0,0,1)->(0,0,2)->(0,1,0)->(0,1,1)->...
// The upper limit for each value is given by 'bound' (val[i] < dim[i])
// Return false if all values of 'that' have reached their upper limit
// (in which case 'that''s values are all set back to 0)
// Return true else
bool VecShortStep(VecShort *that, VecShort *bound) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (bound == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'bound' is null");
        PBErrCatch(PBMathErr);
    }
    if (that->_dim != bound->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
            that->_dim, bound->_dim);
        PBErrCatch(PBMathErr);
    }
#endif
    // Declare a variable for the returned flag
    bool ret = true;
    // Declare a variable to memorise the dimension currently increasing
    int iDim = that->_dim - 1;
    // Declare a flag for the loop condition
    bool flag = true;
    // Increment
    do {
        ++(that->_val[iDim]);
        if (that->_val[iDim] >= bound->_val[iDim]) {
            that->_val[iDim] = 0;
            --iDim;
        } else {
            flag = false;
        }
    } while (iDim >= 0 && flag == true);
    if (iDim == -1)
        ret = false;
    // Return the flag
    return ret;
}

// Step the values of the vector incrementally by 1 from 0
// in the following order (for example) :
// (0,0,0)->(1,0,0)->(2,0,0)->(0,1,0)->(1,1,0)->...
// The upper limit for each value is given by 'bound' (val[i] < dim[i])
// Return false if all values of 'that' have reached their upper limit
// (in which case 'that''s values are all set back to 0)
// Return true else
bool VecShortPStep(VecShort *that, VecShort *bound) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");

```

```

        PBErCatch(PBMathErr);
    }
    if (bound == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'bound' is null");
        PBErCatch(PBMathErr);
    }
    if (that->_dim != bound->_dim) {
        PBMathErr->_type = PBErTypeInvalidArg;
        sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
            that->_dim, bound->_dim);
        PBErCatch(PBMathErr);
    }
#endif
    // Declare a variable for the returned flag
    bool ret = true;
    // Declare a variable to memorise the dimension currently increasing
    int iDim = 0;
    // Declare a flag for the loop condition
    bool flag = true;
    // Increment
    do {
        ++(that->_val[iDim]);
        if (that->_val[iDim] >= bound->_val[iDim]) {
            that->_val[iDim] = 0;
            ++iDim;
        } else {
            flag = false;
        }
    } while (iDim < that->_dim && flag == true);
    if (iDim == that->_dim)
        ret = false;
    // Return the flag
    return ret;
}

// ----- VecFloat

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

// Create a new Vec of dimension 'dim'
// Values are initialized to 0.0
VecFloat* VecFloatCreate(int dim) {
    #if BUILDMODE == 0
        if (dim <= 0) {
            PBMathErr->_type = PBErTypeInvalidArg;
            sprintf(PBMathErr->_msg, "invalid 'dim' (%d)", dim);
            PBErCatch(PBMathErr);
        }
    #endif
    // Allocate memory
    VecFloat *that = PBErMalloc(PBMathErr,
        sizeof(VecFloat) + sizeof(float) * dim);
    // 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
VecFloat* VecFloatClone(VecFloat *that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create a clone
    VecFloat *clone = VecFloatCreate(that->_dim);
    // Clone the properties
    memcpy(clone, that, sizeof(VecFloat) + sizeof(float) * that->_dim);
    // Return the clone
    return clone;
}

// Load the VecFloat from the stream
// If the VecFloat is already allocated, it is freed before loading
bool VecFloatLoad(VecFloat **that, FILE *stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (stream == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'stream' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // 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 couldn't fscanf
    if (ret == EOF)
        return false;
    // Check the dimension
    if (dim <= 0)
        return false;
    // Allocate memory
    *that = VecFloatCreate(dim);
    // Read the values
    for (int i = 0; i < dim; ++i) {
        ret = fscanf(stream, "%f", (*that)->_val + i);
        // If we couldn't fscanf
        if (ret == EOF)
            return false;
    }
    // Return success code
    return true;
}

// Save the VecFloat to the stream
// Return true in case of success, else false
bool VecFloatSave(VecFloat *that, FILE *stream) {

```

```

#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (stream == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'stream' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Save the dimension
    int ret = fprintf(stream, "%d ", that->_dim);
    // If we couldn't fprintf
    if (ret < 0)
        return false;
    // Save the values
    for (int i = 0; i < that->_dim; ++i) {
        ret = fprintf(stream, "%f ", that->_val[i]);
        // If we couldn't fprintf
        if (ret < 0)
            return false;
    }
    fprintf(stream, "\n");
    // If we couldn't fprintf
    if (ret < 0)
        return false;
    // Return success code
    return true;
}

// 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);
    *that = NULL;
}

// Print the VecFloat on 'stream' with 'prec' digit precision
// Do nothing if arguments are invalid
void VecFloatPrint(VecFloat *that, FILE *stream, unsigned int prec) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (stream == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'stream' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create the format string
    char format[100] = {'\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, ">");
}

// Return the angle of the rotation making 'that' colinear to 'tho'
// 'that' and 'tho' must be normalised
// Return a value in [-PI,PI]
float VecFloatAngleTo2D(VecFloat2D *that, VecFloat2D *tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
    if (!ISEQUALF(VecNorm(that), 1.0)) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'that' is not a normed vector");
        PBErrCatch(PBMathErr);
    }
    if (!ISEQUALF(VecNorm(tho), 1.0)) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'tho' is not a normed vector");
        PBErrCatch(PBMathErr);
    }
}
#endif
// Declare a variable to memorize the result
float theta = 0.0;
// Calculate the angle
VecFloat2D m = VecFloatCreateStatic2D();
if (fabs(VecGet(that, 0)) > fabs(VecGet(that, 1))) {
    VecSet(&m, 0,
        (VecGet(tho, 0) + VecGet(tho, 1) * VecGet(that, 1) /
         VecGet(that, 0)) /
        (VecGet(that, 0) + fsquare(VecGet(that, 1)) / VecGet(that, 0)));
    VecSet(&m, 1,
        (VecGet(&m, 0) * VecGet(that, 1) - VecGet(tho, 1)) /
        VecGet(that, 0));
} else {
    VecSet(&m, 1,
        (VecGet(tho, 0) - VecGet(tho, 1) * VecGet(that, 0) /
         VecGet(that, 1)) /
        (VecGet(that, 1) + fsquare(VecGet(that, 0)) / VecGet(that, 1)));
    VecSet(&m, 0,
        (VecGet(tho, 1) + VecGet(&m, 1) * VecGet(that, 0)) /
        VecGet(that, 1));
}
// Due to numerical imprecision m[0] may be slightly out of [-1,1]
// which makes acos return NaN, prevent this
if (VecGet(&m, 0) < -1.0)
    theta = PB_MATH_PI;
else if (VecGet(&m, 0) > 1.0)
    theta = 0.0;

```

```

    else
        theta = acos(VecGet(&m, 0));
        if (sin(theta) * VecGet(&m, 1) > 0.0)
            theta *= -1.0;
        // Return the result
        return theta;
}

// ----- MatFloat

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

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

// Create a new MatFloat of dimension 'dim' (nbc, nbl)
// Values are initialized to 0.0
MatFloat* MatFloatCreate(VecShort2D *dim) {
    #if BUILDMODE == 0
        if (dim == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'dim' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
    // Allocate memory
    int d = VecGet(dim, 0) * VecGet(dim, 1);
    MatFloat *that = PBErrMalloc(PBMathErr, sizeof(MatFloat) +
        sizeof(float) * d);
    // Set the dimension
    that->_dim = *dim;
    // Set the default values
    for (int i = d; i--;)
        that->_val[i] = 0.0;
    // Return the new MatFloat
    return that;
}

// Clone the MatFloat
MatFloat* MatFloatClone(MatFloat *that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
    // Create a clone
    MatFloat *clone = MatFloatCreate(&(that->_dim));
    // Copy the values
    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 true upon success, else false
bool MatFloatLoad(MatFloat **that, FILE *stream) {
    #if BUILDMODE == 0
        if (that == NULL) {

```



```

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
}
if (stream == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'stream' is null");
    PBErrCatch(PBMathErr);
}
#endif
// If 'that' is already allocated
if (*that != NULL)
    // Free memory
    MatFloatFree(that);
// Read the number of dimension
VecShort2D dim = VecShortCreateStatic2D();
int ret = fscanf(stream, "%hi %hi", dim._val, dim._val + 1);
// If we couldn't fscanf
if (ret == EOF)
    return false;
if (VecGet(&dim, 0) <= 0 || VecGet(&dim, 1) <= 0)
    return false;
// Allocate memory
*that = MatFloatCreate(&dim);
// Read the values
VecShort2D index = VecShortCreateStatic2D();
do {
    float v;
    ret = fscanf(stream, "%f", &v);
    // If we couldn't fscanf
    if (ret == EOF)
        return false;
    MatSet(*that, &index, v);
} while (VecPStep(&index, &dim));
// Return success code
return true;
}

// Save the MatFloat to the stream
// Return true upon success, else false
bool MatFloatSave(MatFloat *that, FILE *stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (stream == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'stream' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Save the dimension
    int ret = fprintf(stream, "%hi %hi\n",
        VecGet(&(that->_dim), 0), VecGet(&(that->_dim), 1));
    if (ret < 0)
        return false;
    // Save the values
    VecShort2D index = VecShortCreateStatic2D();
    do {
        ret = fprintf(stream, "%f ", MatGet(that, &index));

```

```

    // If we couldn't fprintf
    if (ret < 0)
        return false;
    if (VecGet(&index, 0) == VecGet(&(that->_dim), 0) - 1) {
        ret = fprintf(stream, "\n");
        // If we couldn't fprintf
        if (ret < 0)
            return false;
    }
} while (VecPStep(&index, &(that->_dim)));
// Return success code
return true;
}

// 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
    free(*that);
    *that = NULL;
}

// Print the MatFloat on 'stream' with 'prec' digit precision
// Do nothing if arguments are invalid
void MatFloatPrintln(MatFloat *that, FILE *stream, unsigned int prec) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (stream == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'stream' is null");
        PBErrCatch(PBMathErr);
    }
}
#endif
// Create the format string
char format[100] = {'\0'};
sprintf(format, "%%.%df", prec);
// Print the values
fprintf(stream, "[");
VecShort2D index = VecShortCreateStatic2D();
do {
    if (VecGet(&index, 1) != 0 || VecGet(&index, 0) != 0)
        fprintf(stream, " ");
    fprintf(stream, format, MatGet(that, &index));
    if (VecGet(&index, 0) < VecGet(&(that->_dim), 0) - 1)
        fprintf(stream, ",");
    if (VecGet(&index, 0) == VecGet(&(that->_dim), 0) - 1) {
        if (VecGet(&index, 1) == VecGet(&(that->_dim), 1) - 1)
            fprintf(stream, "]");
        fprintf(stream, "\n");
    }
} while (VecPStep(&index, &(that->_dim)));
}

// Return the inverse matrix of 'that'
// The matrix must be a square matrix

```

```

MatFloat* MatFloatInv(MatFloat *that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (VecGet(&(that->_dim), 0) != VecGet(&(that->_dim), 1)) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "the matrix is not square (%dx%d)",
            VecGet(&(that->_dim), 0), VecGet(&(that->_dim), 1));
        PBErrCatch(PBMathErr);
    }
    if (VecGet(&(that->_dim), 0) > 3) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg,
            "MatFloatInv is defined only for matrix of dim <= 3x3 (%dx%d)",
            VecGet(&(that->_dim), 0), VecGet(&(that->_dim), 1));
        PBErrCatch(PBMathErr);
    }
#endif
    // Allocate memory for the result
    MatFloat *res = MatFloatCreate(&(that->_dim));
    // If the matrix is of dimension 1x1
    if (VecGet(&(that->_dim), 0) == 1) {
#ifdef BUILDMODE == 0
        if (that->_val[0] < PBMath_EPSILON) {
            PBMathErr->_type = PBErrTypeOther;
            sprintf(PBMathErr->_msg, "the matrix is not inversible");
            PBErrCatch(PBMathErr);
        }
#endif
        res->_val[0] = 1.0 / that->_val[0];
        // If the matrix is of dimension 2x2
    } else if (VecGet(&(that->_dim), 0) == 2) {
        float det = that->_val[0] * that->_val[3] -
            that->_val[2] * that->_val[1];
#ifdef BUILDMODE == 0
        if (ISEQUALF(det, 0.0)) {
            PBMathErr->_type = PBErrTypeOther;
            sprintf(PBMathErr->_msg, "the matrix is not inversible");
            PBErrCatch(PBMathErr);
        }
#endif
        res->_val[0] = that->_val[3] / det;
        res->_val[1] = -1.0 * that->_val[1] / det;
        res->_val[2] = -1.0 * that->_val[2] / det;
        res->_val[3] = that->_val[0] / det;
        // Else, the matrix dimension is 3x3
    } else if (VecGet(&(that->_dim), 0) == 3) {
        float det =
            that->_val[0] *
                (that->_val[4] * that->_val[8] -
                    that->_val[5] * that->_val[7]) -
            that->_val[3] *
                (that->_val[1] * that->_val[8] -
                    that->_val[2] * that->_val[7]) +
            that->_val[6] *
                (that->_val[1] * that->_val[5] -
                    that->_val[2] * that->_val[4]);
#ifdef BUILDMODE == 0
        if (ISEQUALF(det, 0.0)) {

```

```

        PBMathErr->_type = PBErrTypeOther;
        sprintf(PBMathErr->_msg, "the matrix is not inversible");
        PBErrCatch(PBMathErr);
    }
#endif
    res->_val[0] = (that->_val[4] * that->_val[8] -
        that->_val[5] * that->_val[7]) / det;
    res->_val[1] = -(that->_val[1] * that->_val[8] -
        that->_val[2] * that->_val[7]) / det;
    res->_val[2] = (that->_val[1] * that->_val[5] -
        that->_val[2] * that->_val[4]) / det;
    res->_val[3] = -(that->_val[3] * that->_val[8] -
        that->_val[5] * that->_val[6]) / det;
    res->_val[4] = (that->_val[0] * that->_val[8] -
        that->_val[2] * that->_val[6]) / det;
    res->_val[5] = -(that->_val[0] * that->_val[5] -
        that->_val[2] * that->_val[3]) / det;
    res->_val[6] = (that->_val[3] * that->_val[7] -
        that->_val[4] * that->_val[6]) / det;
    res->_val[7] = -(that->_val[0] * that->_val[7] -
        that->_val[1] * that->_val[6]) / det;
    res->_val[8] = (that->_val[0] * that->_val[4] -
        that->_val[1] * that->_val[3]) / det;
}
// Return the result
return res;
}

// Return the product of matrix 'that' and vector 'v'
// Number of column of 'that' must equal dimension of 'v'
VecFloat* MatFloatProdVecFloat(MatFloat *that, VecFloat *v) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (v == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'v' is null");
            PBErrCatch(PBMathErr);
        }
        if (VecGet(&(that->_dim), 0) != VecDim(v)) {
            PBMathErr->_type = PBErrTypeInvalidArg;
            sprintf(PBMathErr->_msg,
                "the matrix and vector have incompatible dimensions (%d==%d)",
                VecGet(&(that->_dim), 0), VecDim(v));
            PBErrCatch(PBMathErr);
        }
    #endif
    // Declare a variable to memorize the index in the matrix
    VecShort2D i = VecShortCreateStatic2D();
    // 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));
}

```

```

    // 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'
MatFloat* MatFloatProdMatFloat(MatFloat *that, MatFloat *tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
    if (VecGet(&(that->_dim), 0) != VecGet(&(tho->_dim), 1)) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg,
            "the matrices have incompatible dimensions (%d==%d)",
            VecGet(&(that->_dim), 0), VecGet(&(tho->_dim), 1));
        PBErrCatch(PBMathErr);
    }
#endif
    // Declare 3 variables to memorize the index in the matrix
    VecShort2D i = VecShortCreateStatic2D();
    VecShort2D j = VecShortCreateStatic2D();
    VecShort2D k = VecShortCreateStatic2D();
    // Allocate memory for the solution
    VecSet(&i, 0, VecGet(&(tho->_dim), 0));
    VecSet(&i, 1, VecGet(&(that->_dim), 1));
    MatFloat *ret = MatFloatCreate(&i);
    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));
            }
    // Return the result
    return ret;
}

// Return true if 'that' is equal to 'tho', false else
bool MatFloatIsEqual(MatFloat *that, MatFloat *tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }

```

```

    }
#endif
    if (!VecIsEqual(&(that->_dim), &(tho->_dim)))
        return false;
    VecShort2D v = VecShortCreateStatic2D();
    do {
        if (!ISEQUALF(MatGet(that, &v), MatGet(tho, &v)))
            return false;
    } while (VecStep(&v, &(that->_dim)));
    return true;
}

// ----- 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 = PBErrMalloc(PBMathErr, sizeof(Gauss));
    // Set properties
    that->_mean = mean;
    that->_sigma = sigma;
    // Return the new Gauss
    return that;
}

Gauss GaussCreateStatic(float mean, float sigma) {
    // Allocate memory
    Gauss that = {._mean = mean, ._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;
}

// ----- SysLinEq

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

// Create a new SysLinEq with matrix 'm' and vector 'v'
// The dimension of 'v' must be equal to the number of column of 'm'
// If 'v' is null the vector null is used instead
// The matrix 'm' must be a square matrix
// Return NULL if we couldn't create the SysLinEq
SysLinEq* SLECreate(MatFloat *m, VecFloat *v) {
#if BUILDMODE == 0
    if (m == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'m' is null");
        PBErrCatch(PBMathErr);
    }
#endif
}

```

```

    }
    if (VecGet(&(m->_dim), 0) != VecGet(&(m->_dim), 1)) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "the matrix is not square (%dx%d)",
            VecGet(&(m->_dim), 0), VecGet(&(m->_dim), 1));
        PBErrCatch(PBMathErr);
    }
    if (v != NULL) {
        if (VecGet(&(m->_dim), 0) != VecDim(v)) {
            PBMathErr->_type = PBErrTypeInvalidArg;
            sprintf(PBMathErr->_msg,
                "the matrix and vector have incompatible dimensions (%d==%d)",
                VecGet(&(m->_dim), 0), VecDim(v));
            PBErrCatch(PBMathErr);
        }
    }
}
#endif
// Allocate memory
SysLinEq *that = PBErrMalloc(PBMathErr, sizeof(SysLinEq));
that->_M = MatClone(m);
that->_Minv = MatInv(that->_M);
if (v != NULL)
    that->_V = VecClone(v);
else
    that->_V = VecFloatCreate(VecGet(&(m->_dim), 0));
if (that->_M == NULL || that->_V == NULL || that->_Minv == NULL) {
#ifdef BUILDMODE == 0
    if (that->_M == NULL) {
        PBMathErr->_type = PBErrTypeOther;
        sprintf(PBMathErr->_msg, "couldn't create the matrix");
        PBErrCatch(PBMathErr);
    }
    if (that->_Minv == NULL) {
        PBMathErr->_type = PBErrTypeOther;
        sprintf(PBMathErr->_msg, "couldn't inverse the matrix");
        PBErrCatch(PBMathErr);
    }
    if (that->_V == NULL) {
        PBMathErr->_type = PBErrTypeOther;
        sprintf(PBMathErr->_msg, "couldn't create the vector");
        PBErrCatch(PBMathErr);
    }
}
#endif
SysLinEqFree(&that);
}
// Return the new SysLinEq
return that;
}

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

```

```

// Clone the SysLinEq 'that'
// Return NULL if we couldn't clone the SysLinEq
SysLinEq* SysLinEqClone(SysLinEq *that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Declare a variable for the result
    SysLinEq *ret = PBErrMalloc(PBMathErr, sizeof(SysLinEq));
    ret->_M = MatClone(that->_M);
    ret->_Minv = MatClone(that->_Minv);
    ret->_V = VecClone(that->_V);
    if (ret->_M == NULL || ret->_V == NULL || ret->_Minv == NULL)
        SysLinEqFree(&ret);
    // Return the new SysLinEq
    return ret;
}

```

## 3.2 pbmath-inline.c

```

// ===== PBMath_INLINE.C =====

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

// Static constructors for VecShort
#ifdef BUILDMODE != 0
inline
#endif
VecShort2D VecShortCreateStatic2D() {
    VecShort2D v = {._val = {0, 0}, ._dim = 2};
    return v;
}
#ifdef BUILDMODE != 0
inline
#endif
VecShort3D VecShortCreateStatic3D() {
    VecShort3D v = {._val = {0, 0, 0}, ._dim = 3};
    return v;
}
#ifdef BUILDMODE != 0
inline
#endif
VecShort4D VecShortCreateStatic4D() {
    VecShort4D v = {._val = {0, 0, 0, 0}, ._dim = 4};
    return v;
}

// Return the i-th value of the VecShort
#ifdef BUILDMODE != 0
inline
#endif
short VecShortGet(VecShort *that, int i) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;

```



```

        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= that->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<%d)", i,
            that->_dim);
        PBErrCatch(PBMathErr);
    }
#endif
    return ((short*)((void*)that) + sizeof(int))[i];
}

#if BUILDMODE != 0
inline
#endif
short VecShortGet2D(VecShort2D *that, int i) {
    if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (i < 0 || i >= 2) {
            PBMathErr->_type = PBErrTypeInvalidArg;
            sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<2)", i);
            PBErrCatch(PBMathErr);
        }
    }
#endif
    return that->_val[i];
}

#if BUILDMODE != 0
inline
#endif
short VecShortGet3D(VecShort3D *that, int i) {
    if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (i < 0 || i >= 3) {
            PBMathErr->_type = PBErrTypeInvalidArg;
            sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<3)", i);
            PBErrCatch(PBMathErr);
        }
    }
#endif
    return that->_val[i];
}

#if BUILDMODE != 0
inline
#endif
short VecShortGet4D(VecShort4D *that, int i) {
    if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (i < 0 || i >= 4) {
            PBMathErr->_type = PBErrTypeInvalidArg;
            sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<4)", i);
            PBErrCatch(PBMathErr);
        }
    }
}

```

```

    }
#endif
    return that->_val[i];
}

// Set the i-th value of the VecShort to v
#if BUILDMODE != 0
inline
#endif
void VecShortSet(VecShort *that, int i, short v) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= that->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<%d)", i,
            that->_dim);
        PBErrCatch(PBMathErr);
    }
#endif
    ((short*)((void*)that) + sizeof(int))[i] = v;
}
#if BUILDMODE != 0
inline
#endif
void VecShortSet2D(VecShort2D *that, int i, short v) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= 2) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<2)", i);
        PBErrCatch(PBMathErr);
    }
#endif
    that->_val[i] = v;
}
#if BUILDMODE != 0
inline
#endif
void VecShortSet3D(VecShort3D *that, int i, short v) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= 3) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<3)", i);
        PBErrCatch(PBMathErr);
    }
#endif
    that->_val[i] = v;
}
#if BUILDMODE != 0

```

```

inline
#endif
void VecShortSet4D(VecShort4D *that, int i, short v) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= 4) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<4)", i);
        PBErrCatch(PBMathErr);
    }
}
#endif
    that->_val[i] = v;
}

// Set all values of the vector 'that' to 0
#if BUILDMODE != 0
inline
#endif
void VecShortSetNull(VecShort *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
}
#endif
    // Set values
    for (int iDim = that->_dim; iDim--;)
        that->_val[iDim] = 0;
}

#if BUILDMODE != 0
inline
#endif
void VecShortSetNull2D(VecShort2D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
}
#endif
    // Set values
    that->_val[0] = 0;
    that->_val[1] = 0;
}

#if BUILDMODE != 0
inline
#endif
void VecShortSetNull3D(VecShort3D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
}
#endif
    // Set values
    that->_val[0] = 0;

```

```

        that->_val[1] = 0;
        that->_val[2] = 0;
    }
    #if BUILDMODE != 0
    inline
    #endif
    void VecShortSetNull4D(VecShort4D *that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
        // Set values
        that->_val[0] = 0;
        that->_val[1] = 0;
        that->_val[2] = 0;
        that->_val[3] = 0;
    }

    // Return the dimension of the VecShort
    #if BUILDMODE != 0
    inline
    #endif
    int VecShortDim(VecShort *that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
        return that->_dim;
    }

    // Return the Hamiltonian distance between the VecShort 'that' and 'tho'
    #if BUILDMODE != 0
    inline
    #endif
    short VecShortHamiltonDist(VecShort *that, VecShort *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
        if (that->_dim != tho->_dim) {
            PBMathErr->_type = PBErrTypeInvalidArg;
            sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
                that->_dim, tho->_dim);
            PBErrCatch(PBMathErr);
        }
    #endif
        // Declare a variable to calculate the distance
        short ret = 0;
        for (int iDim = VecDim(that); iDim--;)
            ret += abs(VecGet(that, iDim) - VecGet(tho, iDim));
    }

```

```

        // Return the distance
        return ret;
    }
    #if BUILDMODE != 0
    inline
    #endif
    short VecShortHamiltonDist2D(VecShort2D *that, VecShort2D *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
        // Return the distance
        return abs(VecGet(that, 0) - VecGet(tho, 0)) +
            abs(VecGet(that, 1) - VecGet(tho, 1));
    }
    #if BUILDMODE != 0
    inline
    #endif
    short VecShortHamiltonDist3D(VecShort3D *that, VecShort3D *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
        // Return the distance
        return abs(VecGet(that, 0) - VecGet(tho, 0)) +
            abs(VecGet(that, 1) - VecGet(tho, 1)) +
            abs(VecGet(that, 2) - VecGet(tho, 2));
    }
    #if BUILDMODE != 0
    inline
    #endif
    short VecShortHamiltonDist4D(VecShort4D *that, VecShort4D *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
        // Return the distance
        return abs(VecGet(that, 0) - VecGet(tho, 0)) +

```

```

        abs(VecGet(that, 1) - VecGet(tho, 1)) +
        abs(VecGet(that, 2) - VecGet(tho, 2)) +
        abs(VecGet(that, 3) - VecGet(tho, 3));
    }

    // Return true if the VecShort 'that' is equal to 'tho', else false
    #if BUILDMODE != 0
    inline
    #endif
    bool VecShortIsEqual(VecShort *that, VecShort *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
        if (that->_dim != tho->_dim) {
            PBMathErr->_type = PBErrTypeInvalidArg;
            sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
                that->_dim, tho->_dim);
            PBErrCatch(PBMathErr);
        }
    #endif
        return
            (memcmp(that->_val, tho->_val, sizeof(short) * that->_dim) == 0);
    }
    #if BUILDMODE != 0
    inline
    #endif
    bool VecShortIsEqual2D(VecShort2D *that, VecShort2D *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
        return (memcmp(that->_val, tho->_val, sizeof(short) * 2) == 0);
    }
    #if BUILDMODE != 0
    inline
    #endif
    bool VecShortIsEqual3D(VecShort3D *that, VecShort3D *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
        }
    #endif

```

```

        PBErCatch(PBMathErr);
    }
#endif
    return (memcmp(that->_val, tho->_val, sizeof(short) * 3) == 0);
}
#endif
inline
#endif
bool VecShortIsEqual4D(VecShort4D *that, VecShort4D *tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErCatch(PBMathErr);
    }
#endif
    return (memcmp(that->_val, tho->_val, sizeof(short) * 4) == 0);
}

// Copy the values of 'tho' in 'that'
#ifdef BUILDMODE != 0
inline
#endif
void VecShortCopy(VecShort *that, VecShort *tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErCatch(PBMathErr);
    }
    if (that->_dim != tho->_dim) {
        PBMathErr->_type = PBErTypeInvalidArg;
        sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
            that->_dim, tho->_dim);
        PBErCatch(PBMathErr);
    }
#endif
    // Copy the values
    memcpy(that->_val, tho->_val, sizeof(short) * that->_dim);
}

// Return the dot product of 'that' and 'tho'
#ifdef BUILDMODE != 0
inline
#endif
short VecShortDotProd(VecShort *that, VecShort *tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }

```

```

    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
    if (that->_dim != tho->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
            that->_dim, tho->_dim);
        PBErrCatch(PBMathErr);
    }
}
#endif
// Declare a variable to memorise the result
short res = 0;
// For each component
for (int iDim = that->_dim; iDim--;)
    // Calculate the product
    res += VecGet(that, iDim) * VecGet(tho, iDim);
// Return the result
return res;
}
#endif
inline
#endif
short VecShortDotProd2D(VecShort2D *that, VecShort2D *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
    return VecGet(that, 0) * VecGet(tho, 0) +
        VecGet(that, 1) * VecGet(tho, 1);
}
#endif
inline
#endif
short VecShortDotProd3D(VecShort3D *that, VecShort3D *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
    return VecGet(that, 0) * VecGet(tho, 0) +
        VecGet(that, 1) * VecGet(tho, 1) +
        VecGet(that, 2) * VecGet(tho, 2);
}
#endif
inline

```



```

#endif
short VecShortDotProd4D(VecShort4D *that, VecShort4D *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    return VecGet(that, 0) * VecGet(tho, 0) +
        VecGet(that, 1) * VecGet(tho, 1) +
        VecGet(that, 2) * VecGet(tho, 2) +
        VecGet(that, 3) * VecGet(tho, 3);
}

// Set all values of the vector 'that' to 0
#if BUILDMODE != 0
inline
#endif
void VecSetNull2D(VecShort2D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Set values
    that->_val[0] = that->_val[1] = 0;
}

#if BUILDMODE != 0
inline
#endif
void VecSetNull3D(VecShort3D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Set values
    that->_val[0] = that->_val[1] = that->_val[2] = 0;
}

#if BUILDMODE != 0
inline
#endif
void VecSetNull4D(VecShort4D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Set values
    that->_val[0] = that->_val[1] = that->_val[2] = that->_val[3] = 0;
}

```

```

}

// Static constructors for VecFloat
#if BUILDMODE != 0
inline
#endif
VecFloat2D VecFloatCreateStatic2D() {
    VecFloat2D v = {._val = {0.0, 0.0}, ._dim = 2};
    return v;
}
#if BUILDMODE != 0
inline
#endif
VecFloat3D VecFloatCreateStatic3D() {
    VecFloat3D v = {._val = {0.0, 0.0, 0.0}, ._dim = 3};
    return v;
}

// Return the i-th value of the VecFloat
#if BUILDMODE != 0
inline
#endif
float VecFloatGet(VecFloat *that, int i) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= that->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg,
            "'i' is invalid (0<=%d<%d)", i, that->_dim);
        PBErrCatch(PBMathErr);
    }
#endif
    // Return the value
    return that->_val[i];
}
#if BUILDMODE != 0
inline
#endif
float VecFloatGet2D(VecFloat2D *that, int i) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= 2) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<2)", i);
        PBErrCatch(PBMathErr);
    }
#endif
    // Return the value
    return that->_val[i];
}
#if BUILDMODE != 0
inline
#endif
float VecFloatGet3D(VecFloat3D *that, int i) {

```

```

#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= 3) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<3)", i);
        PBErrCatch(PBMathErr);
    }
#endif
    // Return the value
    return that->_val[i];
}

// Set the i-th value of the VecFloat to v
#if BUILDMODE != 0
inline
#endif
void VecFloatSet(VecFloat *that, int i, float v) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= that->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg,
            "'i' is invalid (0<=%d<%d)", i, that->_dim);
        PBErrCatch(PBMathErr);
    }
#endif
    // Set the value
    that->_val[i] = v;
}

#if BUILDMODE != 0
inline
#endif
void VecFloatSet2D(VecFloat2D *that, int i, float v) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (i < 0 || i >= 2) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<2)", i);
        PBErrCatch(PBMathErr);
    }
#endif
    // Set the value
    that->_val[i] = v;
}

#if BUILDMODE != 0
inline
#endif
void VecFloatSet3D(VecFloat3D *that, int i, float v) {
#if BUILDMODE == 0
    if (that == NULL) {

```

```

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
}
if (i < 0 || i >= 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'i' is invalid (0<=%d<3)", i);
    PBErrCatch(PBMathErr);
}
#endif
// Set the value
that->_val[i] = v;
}

// Set all values of the vector 'that' to 0.0
#if BUILDMODE != 0
inline
#endif
void VecFloatSetNull(VecFloat *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Set values
    for (int iDim = that->_dim; iDim--;)
        that->_val[iDim] = 0.0;
}
#if BUILDMODE != 0
inline
#endif
void VecFloatSetNull2D(VecFloat2D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Set values
    that->_val[0] = 0.0;
    that->_val[1] = 0.0;
}
#if BUILDMODE != 0
inline
#endif
void VecFloatSetNull3D(VecFloat3D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Set values
    that->_val[0] = 0.0;
    that->_val[1] = 0.0;
    that->_val[2] = 0.0;
}

```

```

// Return the dimension of the VecFloat
#if BUILDMODE != 0
inline
#endif
int VecFloatDim(VecFloat *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    return that->_dim;
}

// Copy the values of 'tho' in 'that'
#if BUILDMODE != 0
inline
#endif
void VecFloatCopy(VecFloat *that, VecFloat *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
    if (that->_dim != tho->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
            that->_dim, tho->_dim);
        PBErrCatch(PBMathErr);
    }
#endif
    // Copy the values
    memcpy(that->_val, tho->_val, sizeof(float) * that->_dim);
}

// Return the norm of the VecFloat
#if BUILDMODE != 0
inline
#endif
float VecFloatNorm(VecFloat *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Declare a variable to calculate the norm
    float ret = 0.0;
    // Calculate the norm
    for (int iDim = that->_dim; iDim--;)
        ret += fsquare(VecGet(that, iDim));
    ret = sqrt(ret);
    // Return the result
    return ret;
}

```

```

}
#if BUILDMODE != 0
inline
#endif
float VecFloatNorm2D(VecFloat2D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Return the result
    return sqrt(fsquare(VecGet(that, 0)) + fsquare(VecGet(that, 1)));
}
#if BUILDMODE != 0
inline
#endif
float VecFloatNorm3D(VecFloat3D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Return the result
    return sqrt(fsquare(VecGet(that, 0)) + fsquare(VecGet(that, 1)) +
        fsquare(VecGet(that, 2)));
}

// Normalise the VecFloat
#if BUILDMODE != 0
inline
#endif
void VecFloatNormalise(VecFloat *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Normalise
    float norm = VecNorm(that);
    for (int iDim = that->_dim; iDim--;)
        VecSet(that, iDim, VecGet(that, iDim) / norm);
}

#if BUILDMODE != 0
inline
#endif
void VecFloatNormalise2D(VecFloat2D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Normalise
    float norm = VecFloatNorm2D(that);

```

```

    VecSet(that, 0, VecGet(that, 0) / norm);
    VecSet(that, 1, VecGet(that, 1) / norm);
}
#if BUILDMODE != 0
inline
#endif
void VecFloatNormalise3D(VecFloat3D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'that' is null");
        PBErrCatch(PBMATHERR);
    }
#endif
    // Normalise
    float norm = VecFloatNorm3D(that);
    VecSet(that, 0, VecGet(that, 0) / norm);
    VecSet(that, 1, VecGet(that, 1) / norm);
    VecSet(that, 2, VecGet(that, 2) / norm);
}

// Return the distance between the VecFloat 'that' and 'tho'
#if BUILDMODE != 0
inline
#endif
float VecFloatDist(VecFloat *that, VecFloat *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'that' is null");
        PBErrCatch(PBMATHERR);
    }
    if (tho == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'tho' is null");
        PBErrCatch(PBMATHERR);
    }
    if (that->_dim != tho->_dim) {
        PBMATHERR->_type = PBErrTypeInvalidArg;
        sprintf(PBMATHERR->_msg, "dimensions don't match (%d==%d)",
            that->_dim, tho->_dim);
        PBErrCatch(PBMATHERR);
    }
#endif
    // Declare a variable to calculate the distance
    float ret = 0.0;
    for (int iDim = that->_dim; iDim--;)
        ret += fsquare(VecGet(that, iDim) - VecGet(tho, iDim));
    ret = sqrt(ret);
    // Return the distance
    return ret;
}
#if BUILDMODE != 0
inline
#endif
float VecFloatDist2D(VecFloat2D *that, VecFloat2D *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'that' is null");
        PBErrCatch(PBMATHERR);
    }
}

```

```

    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Return the distance
    return sqrt(fsquare(VecGet(that, 0) - VecGet(tho, 0)) +
        fsquare(VecGet(that, 1) - VecGet(tho, 1)));
}
#endif
inline
#endif
float VecFloatDist3D(VecFloat3D *that, VecFloat3D *tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Return the distance
    return sqrt(fsquare(VecGet(that, 0) - VecGet(tho, 0)) +
        fsquare(VecGet(that, 1) - VecGet(tho, 1)) +
        fsquare(VecGet(that, 2) - VecGet(tho, 2)));
}

// Return the Hamiltonian distance between the VecFloat 'that' and 'tho'
#ifdef BUILDMODE != 0
inline
#endif
float VecFloatHamiltonDist(VecFloat *that, VecFloat *tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
    if (that->_dim != tho->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
            that->_dim, tho->_dim);
        PBErrCatch(PBMathErr);
    }
#endif
    // Declare a variable to calculate the distance
    float ret = 0.0;
    for (int iDim = that->_dim; iDim--;)
        ret += fabs(VecGet(that, iDim) - VecGet(tho, iDim));
    // Return the distance
    return ret;
}

```



```

#if BUILDMODE != 0
inline
#endif
float VecFloatHamiltonDist2D(VecFloat2D *that, VecFloat2D *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'that' is null");
        PBErrCatch(PBMATHERR);
    }
    if (tho == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'tho' is null");
        PBErrCatch(PBMATHERR);
    }
#endif
    // Return the distance
    return fabs(VecGet(that, 0) - VecGet(tho, 0)) +
        fabs(VecGet(that, 1) - VecGet(tho, 1));
}

#if BUILDMODE != 0
inline
#endif
float VecFloatHamiltonDist3D(VecFloat3D *that, VecFloat3D *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'that' is null");
        PBErrCatch(PBMATHERR);
    }
    if (tho == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'tho' is null");
        PBErrCatch(PBMATHERR);
    }
#endif
    // Return the distance
    return fabs(VecGet(that, 0) - VecGet(tho, 0)) +
        fabs(VecGet(that, 1) - VecGet(tho, 1)) +
        fabs(VecGet(that, 2) - VecGet(tho, 2));
}

// Return the Pixel distance between the VecFloat 'that' and 'tho'
#if BUILDMODE != 0
inline
#endif
float VecFloatPixelDist(VecFloat *that, VecFloat *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'that' is null");
        PBErrCatch(PBMATHERR);
    }
    if (tho == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'tho' is null");
        PBErrCatch(PBMATHERR);
    }
    if (that->_dim != tho->_dim) {
        PBMATHERR->_type = PBErrTypeInvalidArg;
        sprintf(PBMATHERR->_msg, "dimensions don't match (%d==%d)",
            that->_dim, tho->_dim);
    }
#endif
}

```

```

        PBErCatch(PBMathErr);
    }
#endif
// Declare a variable to calculate the distance
float ret = 0.0;
for (int iDim = that->_dim; iDim--;)
    ret += fabs(floor(VecGet(that, iDim)) - floor(VecGet(tho, iDim)));
// Return the distance
return ret;
}
#if BUILDMODE != 0
inline
#endif
float VecFloatPixelDist2D(VecFloat2D *that, VecFloat2D *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErCatch(PBMathErr);
    }
#endif
// Return the distance
return fabs(floor(VecGet(that, 0)) - floor(VecGet(tho, 0))) +
        fabs(floor(VecGet(that, 1)) - floor(VecGet(tho, 1)));
}
#if BUILDMODE != 0
inline
#endif
float VecFloatPixelDist3D(VecFloat3D *that, VecFloat3D *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErCatch(PBMathErr);
    }
#endif
// Return the distance
return fabs(floor(VecGet(that, 0)) - floor(VecGet(tho, 0))) +
        fabs(floor(VecGet(that, 1)) - floor(VecGet(tho, 1))) +
        fabs(floor(VecGet(that, 2)) - floor(VecGet(tho, 2)));
}

// Return true if the VecFloat 'that' is equal to 'tho', else false
#if BUILDMODE != 0
inline
#endif
bool VecFloatIsEqual(VecFloat *that, VecFloat *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }

```

```

    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
    if (that->_dim != tho->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
            that->_dim, tho->_dim);
        PBErrCatch(PBMathErr);
    }
}
#endif
// For each component
for (int iDim = that->_dim; iDim--;)
    // If the values of this components are different
    if (!ISEQUALF(VecGet(that, iDim), VecGet(tho, iDim)))
        // Return false
        return false;
// Return true
return true;
}
#if BUILDMODE != 0
inline
#endif
bool VecFloatIsEqual2D(VecFloat2D *that, VecFloat2D *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
    }
    #endif
    return ISEQUALF(VecGet(that, 0), VecGet(tho, 0)) &&
        ISEQUALF(VecGet(that, 1), VecGet(tho, 1));
}
#if BUILDMODE != 0
inline
#endif
bool VecFloatIsEqual3D(VecFloat3D *that, VecFloat3D *tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErrCatch(PBMathErr);
        }
    }
    #endif
    return ISEQUALF(VecGet(that, 0), VecGet(tho, 0)) &&
        ISEQUALF(VecGet(that, 1), VecGet(tho, 1)) &&
        ISEQUALF(VecGet(that, 2), VecGet(tho, 2));
}

```

```

// Calculate (that * a + tho * b) and store the result in 'that'
#if BUILDMODE != 0
inline
#endif
void VecFloatOp(VecFloat *that, float a, VecFloat *tho, float b) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
    if (that->_dim != tho->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
            that->_dim, tho->_dim);
        PBErrCatch(PBMathErr);
    }
#endif
    for (int iDim = that->_dim; iDim--;)
        VecSet(that, iDim,
            a * VecGet(that, iDim) + b * VecGet(tho, iDim));
}
#if BUILDMODE != 0
inline
#endif
void VecFloatOp2D(VecFloat2D *that, float a, VecFloat2D *tho, float b) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    VecSet(that, 0, a * VecGet(that, 0) + b * VecGet(tho, 0));
    VecSet(that, 1, a * VecGet(that, 1) + b * VecGet(tho, 1));
}
#if BUILDMODE != 0
inline
#endif
void VecFloatOp3D(VecFloat3D *that, float a, VecFloat3D *tho, float b) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
#endif
}

```

```

    VecSet(that, 0, a * VecGet(that, 0) + b * VecGet(tho, 0));
    VecSet(that, 1, a * VecGet(that, 1) + b * VecGet(tho, 1));
    VecSet(that, 2, a * VecGet(that, 2) + b * VecGet(tho, 2));
}

// Return a VecFloat equal to (that * a + tho * b)
#if BUILDMODE != 0
inline
#endif
VecFloat* VecFloatGetOp(VecFloat *that, float a,
    VecFloat *tho, float b) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
    if (that->_dim != tho->_dim) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
            that->_dim, tho->_dim);
        PBErrCatch(PBMathErr);
    }
#endif
    VecFloat *res = VecFloatCreate(that->_dim);
    for (int iDim = that->_dim; iDim--;)
        VecSet(res, iDim,
            a * VecGet(that, iDim) + b * VecGet(tho, iDim));
    return res;
}

#if BUILDMODE != 0
inline
#endif
VecFloat2D VecFloatGetOp2D(VecFloat2D *that, float a,
    VecFloat2D *tho, float b) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    VecFloat2D res = VecFloatCreateStatic2D();
    VecSet(&res, 0, a * VecGet(that, 0) + b * VecGet(tho, 0));
    VecSet(&res, 1, a * VecGet(that, 1) + b * VecGet(tho, 1));
    return res;
}

#if BUILDMODE != 0
inline
#endif
VecFloat3D VecFloatGetOp3D(VecFloat3D *that, float a,
    VecFloat3D *tho, float b) {

```

```

#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
#endif
VecFloat3D res = VecFloatCreateStatic3D();
VecSet(&res, 0, a * VecGet(that, 0) + b * VecGet(tho, 0));
VecSet(&res, 1, a * VecGet(that, 1) + b * VecGet(tho, 1));
VecSet(&res, 2, a * VecGet(that, 2) + b * VecGet(tho, 2));
return res;
}

// Calculate (that * a) and store the result in 'that'
#if BUILDMODE != 0
inline
#endif
void VecFloatScale(VecFloat *that, float a) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    for (int iDim = that->_dim; iDim--;)
        VecSet(that, iDim, a * VecGet(that, iDim));
}

#if BUILDMODE != 0
inline
#endif
void VecFloatScale2D(VecFloat2D *that, float a) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    VecSet(that, 0, a * VecGet(that, 0));
    VecSet(that, 1, a * VecGet(that, 1));
}

#if BUILDMODE != 0
inline
#endif
void VecFloatScale3D(VecFloat3D *that, float a) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    VecSet(that, 0, a * VecGet(that, 0));
    VecSet(that, 1, a * VecGet(that, 1));
    VecSet(that, 2, a * VecGet(that, 2));
}

```

```

}

// Return a VecFloat equal to (that * a)
#if BUILDMODE != 0
inline
#endif
VecFloat* VecFloatGetScale(VecFloat *that, float a) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    VecFloat *res = VecFloatCreate(that->_dim);
    for (int iDim = that->_dim; iDim--;)
        VecSet(res, iDim, a * VecGet(that, iDim));
    return res;
}

#if BUILDMODE != 0
inline
#endif
VecFloat2D VecFloatGetScale2D(VecFloat2D *that, float a) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    VecFloat2D res = VecFloatCreateStatic2D();
    VecSet(&res, 0, a * VecGet(that, 0));
    VecSet(&res, 1, a * VecGet(that, 1));
    return res;
}

#if BUILDMODE != 0
inline
#endif
VecFloat3D VecFloatGetScale3D(VecFloat3D *that, float a) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    VecFloat3D res = VecFloatCreateStatic3D();
    VecSet(&res, 0, a * VecGet(that, 0));
    VecSet(&res, 1, a * VecGet(that, 1));
    VecSet(&res, 2, a * VecGet(that, 2));
    return res;
}

// Rotate CCW 'that' by 'theta' radians and store the result in 'that'
#if BUILDMODE != 0
inline
#endif
void VecFloatRot2D(VecFloat2D *that, float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
    }
#endif
}

```

```

        PBErCatch(PBMathErr);
    }
#endif
    VecFloat2D v = *that;
    VecSet(that, 0,
        cos(theta) * VecGet(&v, 0) - sin(theta) * VecGet(&v, 1));
    VecSet(that, 1,
        sin(theta) * VecGet(&v, 0) + cos(theta) * VecGet(&v, 1));
}

// Return a VecFloat equal to 'that' rotated CCW by 'theta' radians
// Return NULL if arguments are invalid
#if BUILDMODE != 0
inline
#endif
VecFloat2D VecFloatGetRot2D(VecFloat2D *that, float theta) {
    if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErCatch(PBMathErr);
        }
    #endif
    // Declare a variable to memorize the result
    VecFloat2D res = VecFloatCreateStatic2D();
    // Calculate
    VecSet(&res, 0,
        cos(theta) * VecGet(that, 0) - sin(theta) * VecGet(that, 1));
    VecSet(&res, 1,
        sin(theta) * VecGet(that, 0) + cos(theta) * VecGet(that, 1));
    // Return the result
    return res;
}

// Return the dot product of 'that' and 'tho'
#if BUILDMODE != 0
inline
#endif
float VecFloatDotProd(VecFloat *that, VecFloat *tho) {
    if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErCatch(PBMathErr);
        }
        if (tho == NULL) {
            PBMathErr->_type = PBErTypeNullPointer;
            sprintf(PBMathErr->_msg, "'tho' is null");
            PBErCatch(PBMathErr);
        }
        if (that->_dim != tho->_dim) {
            PBMathErr->_type = PBErTypeInvalidArg;
            sprintf(PBMathErr->_msg, "dimensions don't match (%d==%d)",
                that->_dim, tho->_dim);
            PBErCatch(PBMathErr);
        }
    #endif
    // 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;
}
#if BUILDMODE != 0
inline
#endif
float VecFloatDotProd2D(VecFloat2D *that, VecFloat2D *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    return that->_val[0] * tho->_val[0] + that->_val[1] * tho->_val[1];
}
#if BUILDMODE != 0
inline
#endif
float VecFloatDotProd3D(VecFloat3D *that, VecFloat3D *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    return that->_val[0] * tho->_val[0] + that->_val[1] * tho->_val[1] +
        that->_val[2] * tho->_val[2];
}

// Return the conversion of VecFloat 'that' to a VecShort using round()
#if BUILDMODE != 0
inline
#endif
VecShort* VecFloatToShort(VecFloat *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create the result
    VecShort *res = VecShortCreate(that->_dim);
    for (int iDim = that->_dim; iDim--;)
        VecSet(res, iDim, SHORT(VecGet(that, iDim)));
    // Return the result
    return res;
}
#if BUILDMODE != 0
inline

```

```

#endif
VecShort2D VecFloatToShort2D(VecFloat2D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create the result
    VecShort2D res = VecShortCreateStatic2D();
    VecSet(&res, 0, SHORT(VecGet(that, 0)));
    VecSet(&res, 1, SHORT(VecGet(that, 1)));
    // Return the result
    return res;
}

#if BUILDMODE != 0
inline
#endif
VecShort3D VecFloatToShort3D(VecFloat3D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create the result
    VecShort3D res = VecShortCreateStatic3D();
    VecSet(&res, 0, SHORT(VecGet(that, 0)));
    VecSet(&res, 1, SHORT(VecGet(that, 1)));
    VecSet(&res, 2, SHORT(VecGet(that, 2)));
    // Return the result
    return res;
}

// Return the conversion of VecShort 'that' to a VecFloat
#if BUILDMODE != 0
inline
#endif
VecFloat* VecShortToFloat(VecShort *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create the result
    VecFloat *res = VecFloatCreate(that->_dim);
    for (int iDim = that->_dim; iDim--;)
        VecSet(res, iDim, (float)VecGet(that, iDim));
    // Return the result
    return res;
}

#if BUILDMODE != 0
inline
#endif
VecFloat2D VecShortToFloat2D(VecShort2D *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }
#endif
    // Create the result
    VecFloat2D res = VecFloatCreateStatic2D();
    VecSet(&res, 0, (float)VecGet(that, 0));
    VecSet(&res, 1, (float)VecGet(that, 1));
    // Return the result
    return res;
}
#if BUILDMODE != 0
inline
#endif
VecFloat3D VecShortToFloat3D(VecShort3D *that) {
    if (BUILDMODE == 0)
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErCatch(PBMathErr);
        }
    // Create the result
    VecFloat3D res = VecFloatCreateStatic3D();
    VecSet(&res, 0, (float)VecGet(that, 0));
    VecSet(&res, 1, (float)VecGet(that, 1));
    VecSet(&res, 2, (float)VecGet(that, 2));
    // Return the result
    return res;
}

// Set the MatFloat to the identity matrix
// The matrix must be a square matrix
#if BUILDMODE != 0
inline
#endif
void MatFloatSetIdentity(MatFloat *that) {
    if (BUILDMODE == 0)
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErCatch(PBMathErr);
        }
        if (VecGet(&(that->_dim), 0) != VecGet(&(that->_dim), 1)) {
            PBMathErr->_type = PBErrTypeInvalidArg;
            sprintf(PBMathErr->_msg, "the matrix is not square (%dx%d)",
                    VecGet(&(that->_dim), 0), VecGet(&(that->_dim), 1));
            PBErCatch(PBMathErr);
        }
    // Set the values
    VecShort2D i = VecShortCreateStatic2D();
    do {
        if (VecGet(&i, 0) == VecGet(&i, 1))
            MatSet(that, &i, 1.0);
        else
            MatSet(that, &i, 0.0);
    } while (VecStep(&i, &(that->_dim)));
}

// Copy the values of 'w' in 'that' (must have same dimensions)
#if BUILDMODE != 0

```

```

inline
#endif
void MatFloatCopy(MatFloat *that, MatFloat *tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (tho == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'tho' is null");
        PBErrCatch(PBMathErr);
    }
    if (!VecIsEqual(&(that->_dim), &(tho->_dim))) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg,
            "'that' and 'tho' have different dimensions (%dx%d==%dx%d)",
            VecGet(&(that->_dim), 0), VecGet(&(that->_dim), 1),
            VecGet(&(tho->_dim), 0), VecGet(&(tho->_dim), 1));
        PBErrCatch(PBMathErr);
    }
#endif
    // Copy the matrix values
    int d = VecGet(&(that->_dim), 0) * VecGet(&(that->_dim), 1);
    memcpy(that->_val, tho->_val, d * sizeof(float));
}

// Return the value at index 'i' (col, line) of the MatFloat
// Index starts at 0, index in matrix = line * nbCol + col
#if BUILDMODE != 0
inline
#endif
float MatFloatGet(MatFloat *that, VecShort2D *index) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (index == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'index' is null");
        PBErrCatch(PBMathErr);
    }
    if (VecGet(index, 0) < 0 ||
        VecGet(index, 0) >= VecGet(&(that->_dim), 0) ||
        VecGet(index, 1) < 0 ||
        VecGet(index, 1) >= VecGet(&(that->_dim), 1)) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg,
            "'index' is invalid (0,0 <= %d,%d < %d,%d)",
            VecGet(index, 0), VecGet(index, 1),
            VecGet(&(that->_dim), 0), VecGet(&(that->_dim), 1));
        PBErrCatch(PBMathErr);
    }
#endif
    // Return the value
    return that->_val[VecGet(index, 1) * VecGet(&(that->_dim), 0) +
        VecGet(index, 0)];
}

```

```

// Set the value at index 'i' (col, line) of the MatFloat to 'v'
// Index starts at 0, index in matrix = line * nbCol + col
#if BUILDMODE != 0
inline
#endif
void MatFloatSet(MatFloat *that, VecShort2D *index, float v) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'that' is null");
        PBErrCatch(PBMATHERR);
    }
    if (index == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'index' is null");
        PBErrCatch(PBMATHERR);
    }
    if (VecGet(index, 0) < 0 ||
        VecGet(index, 0) >= VecGet(&(that->_dim), 0) ||
        VecGet(index, 1) < 0 ||
        VecGet(index, 1) >= VecGet(&(that->_dim), 1)) {
        PBMATHERR->_type = PBErrTypeInvalidArg;
        sprintf(PBMATHERR->_msg,
            "'index' is invalid (0,0 <= %d,%d < %d,%d)",
            VecGet(index, 0), VecGet(index, 1),
            VecGet(&(that->_dim), 0), VecGet(&(that->_dim), 1));
        PBErrCatch(PBMATHERR);
    }
#endif
    // Set the value
    that->_val[VecGet(index, 1) * VecGet(&(that->_dim), 0) +
        VecGet(index, 0)] = v;
}

// Return a VecShort2D containing the dimension of the MatFloat
#if BUILDMODE != 0
inline
#endif
VecShort2D MatFloatDim(MatFloat *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'that' is null");
        PBErrCatch(PBMATHERR);
    }
#endif
    // Return the dimension
    return that->_dim;
}

// Return the value of the Gauss 'that' at 'x'
#if BUILDMODE != 0
inline
#endif
float GaussGet(Gauss *that, float x) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMATHERR->_type = PBErrTypeNullPointer;
        sprintf(PBMATHERR->_msg, "'that' is null");
        PBErrCatch(PBMATHERR);
    }
#endif
}

```

```

    // Calculate the value
    float a = 1.0 / (that->_sigma * sqrt(2.0 * PBMath_PI));
    float ret = a * exp(-1.0 * fsquare(x - that->_mean) /
        (2.0 * fsquare(that->_sigma)));
    // 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
#if BUILDMODE != 0
inline
#endif
float GaussRnd(Gauss *that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
    // 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;
}

// Return the order 1 smooth value of 'x'
// if x < 0.0 return 0.0
// if x > 1.0 return 1.0
#if BUILDMODE != 0
inline
#endif
float SmoothStep(float x) {
    if (x > 0.0)
        if (x < 1.0)
            return x * x * (3.0 - 2.0 * x);
        else
            return 1.0;
    else
        return 0.0;
}

// Return the order 2 smooth value of 'x'
// if x < 0.0 return 0.0
// if x > 1.0 return 1.0
#if BUILDMODE != 0
inline
#endif
float SmootherStep(float x) {
    if (x > 0.0)
        if (x < 1.0)

```

```

        return x * x * x * (x * (x * 6.0 - 15.0) + 10.0);
    else
        return 1.0;
    else
        return 0.0;
}

// Solve the SysLinEq _M.x = _V
// Return the solution vector, or null if there is no solution or the
// arguments are invalid
#if BUILDMODE != 0
inline
#endif
VecFloat* SysLinEqSolve(SysLinEq *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Declare a variable to memorize the solution
    VecFloat *ret = NULL;
    // Calculate the solution
    ret = MatProdVec(that->_Minv, that->_V);
    // Return the solution vector
    return ret;
}

// Set the matrix of the SysLinEq to a copy of 'm'
// 'm' must have same dimensions has the current matrix
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void SysLinEqSetM(SysLinEq *that, MatFloat *m) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (m == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'m' is null");
        PBErrCatch(PBMathErr);
    }
    if (!VecIsEqual(&(m->_dim), &(that->_M->_dim))) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'m' has invalid dimension (%dx%d==%dx%d)",
            VecGet(&(m->_dim), 0), VecGet(&(m->_dim), 1),
            VecGet(&(that->_M->_dim), 0), VecGet(&(that->_M->_dim), 1));
        PBErrCatch(PBMathErr);
    }
#endif
    // Update the matrix values
    MatCopy(that->_M, m);
    // Update the inverse matrix
    MatFree(&(that->_Minv));
    that->_Minv = MatInv(that->_M);
#if BUILDMODE == 0
    if (that->_Minv == NULL) {

```

```

        PBMathErr->_type = PBErrTypeOther;
        sprintf(PBMathErr->_msg, "couldn't inverse the matrix");
        PBErrCatch(PBMathErr);
    }
#endif
}

// Set the vector of the SysLinEq to a copy of 'v'
// 'v' must have same dimensions has the current vector
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void SLESetV(SysLinEq *that, VecFloat *v) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (v == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'v' is null");
        PBErrCatch(PBMathErr);
    }
    if (VecDim(v) != VecDim(that->_V)) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'v' has invalid dimension (%d==%d)",
            VecDim(v), VecDim(that->_V));
        PBErrCatch(PBMathErr);
    }
#endif
    // Update the vector values
    VecCopy(that->_V, v);
}

// Return x^y when x and y are int
// to avoid numerical imprecision from (pow(double,double)
// From https://stackoverflow.com/questions/29787310/
// does-pow-work-for-int-data-type-in-c
#if BUILDMODE != 0
inline
#endif
int powi(int base, int exp) {
    // Declare a variable to memorize the result and init to 1
    int res = 1;
    // Loop on exponent
    while (exp) {
        // Do some magic trick
        if (exp & 1)
            res *= base;
        exp /= 2;
        base *= base;
    }
    // Return the result
    return res;
}

```



## 4 Makefile

```
#directory
PBERRDIR=../PBErr

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

include $(PBERRDIR)/Makefile.inc

INCPATH=-I./ -I$(PBERRDIR)/
BUILDOPTIONS=$(BUILDPARAM) $(INCPATH)

# compiler
COMPILER=gcc

#rules
all : main

main: main.o pberr.o pbmath.o Makefile
$(COMPILER) main.o pberr.o pbmath.o $(LINKOPTIONS) -o main

main.o : main.c $(PBERRDIR)/pberr.h pbmath.h pbmath-inline.c Makefile
$(COMPILER) $(BUILDOPTIONS) -c main.c

pbmath.o : pbmath.c pbmath.h pbmath-inline.c Makefile
$(COMPILER) $(BUILDOPTIONS) -c pbmath.c

pberr.o : $(PBERRDIR)/pberr.c $(PBERRDIR)/pberr.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(PBERRDIR)/pberr.c

clean :
rm -rf *.o main

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

unitTest :
main > unitTest.txt; diff unitTest.txt unitTestRef.txt
```

## 5 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "pbmath.h"

#define RANDOMSEED 0

void UnitTestPowi() {
    int a;
```

```

int n;
for (n = 1; n <= 5; ++n) {
    for (a = 0; a <= 10; ++a) {
        int b = powi(a, n);
        int c = 1;
        int m = n;
        for (; m--;) c *= a;
        if (b != c) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg,
                "powi(%d, %d) = %d , %d^%d = %d",
                a, n, b, a, n, c);
            PBErrCatch(PBMathErr);
        }
    }
}
printf("powi OK\n");
}

void UnitTestFastPow() {
    srandom(RANDOMSEED);
    int nbTest = 1000;
    float sumErr = 0.0;
    float maxErr = 0.0;
    int i = nbTest;
    for (; i--;) {
        float a = (rnd() - 0.5) * 1000.0;
        int n = INT(rnd() * 5.0);
        float b = fastpow(a, n);
        float c = pow(a, n);
        float err = fabs(b - c);
        sumErr += err;
        if (maxErr < err)
            maxErr = err;
    }
    float avgErr = sumErr / (float)nbTest;
    printf("average error: %f < %f, max error: %f < %f\n",
        avgErr, PBMath_EPSILON, maxErr, PBMath_EPSILON * 10.0);
    if (avgErr >= PBMath_EPSILON ||
        maxErr >= PBMath_EPSILON * 10.0) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "fastpow NOK");
        PBErrCatch(PBMathErr);
    }
    printf("fastpow OK\n");
}

void UnitTestSpeedFastPow() {
    srandom(RANDOMSEED);
    int nbTest = 1000;
    int i = nbTest;
    clock_t clockBefore = clock();
    for (; i--;) {
        float a = (rnd() - 0.5) * 1000.0;
        int n = INT(rnd() * 5.0);
        float b = fastpow(a, n);
        b = b;
    }
    clock_t clockAfter = clock();
    double timeFastpow = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    srandom(RANDOMSEED);

```

```

i = nbTest;
clockBefore = clock();
for (; i--;) {
    float a = (rnd() - 0.5) * 1000.0;
    int n = INT(rnd() * 5.0);
    float c = pow(a, n);
    c = c;
}
clockAfter = clock();
double timePow = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
printf("fastpow: %fms, pow: %fms\n",
    timeFastpow / (float)nbTest, timePow / (float)nbTest);
if (timeFastpow >= timePow) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    PBMathErr->_fatal = false;
    sprintf(PBMathErr->_msg, "speed fastpow NOK");
    PBErrCatch(PBMathErr);
}
printf("speed fastpow OK\n");
}

void UnitTestFSquare() {
    srandom(RANDOMSEED);
    int nbTest = 1000;
    for (; nbTest--;) {
        float a = (rnd() - 0.5) * 2000.0;
        float b = fsquare(a);
        float c = a * a;
        if (!ISEQUALF(b, c)) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            PBMathErr->_fatal = false;
            sprintf(PBMathErr->_msg,
                "fsquare(%f) = %f , %f*f = %f",
                a, b, a, a, c);
            PBErrCatch(PBMathErr);
        }
    }
    printf("fsquare OK\n");
}

void UnitTestVecShortCreateFree() {
    VecShort *v = VecShortCreate(5);
    VecShort2D v2 = VecShortCreateStatic2D();
    VecShort3D v3 = VecShortCreateStatic3D();
    VecShort4D v4 = VecShortCreateStatic4D();
    VecPrint(v, stdout);printf("\n");
    VecPrint(&v2, stdout);printf("\n");
    VecPrint(&v3, stdout);printf("\n");
    VecPrint(&v4, stdout);printf("\n");
    VecFree(&v);
    if (v != NULL) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShort is not null after VecFree");
        PBErrCatch(PBMathErr);
    }
    printf("VecShortCreateFree OK\n");
}

void UnitTestVecShortClone() {
    VecShort *v = VecShortCreate(5);
    for (int i = 5; i--;) VecSet(v, i, i + 1);
}

```

```

VecShort *w = VecClone(v);
if (memcmp(v, w, sizeof(VecShort) + sizeof(short) * 5) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortClone NOK");
    PBErrCatch(PBMathErr);
}
VecFree(&v);
VecFree(&w);
printf("VecShortClone OK\n");
}

void UnitTestVecShortLoadSave() {
    VecShort *v = VecShortCreate(5);
    VecShort2D v2 = VecShortCreateStatic2D();
    VecShort3D v3 = VecShortCreateStatic3D();
    VecShort4D v4 = VecShortCreateStatic4D();
    for (int i = 5; i--;) VecSet(v, i, i + 1);
    for (int i = 2; i--;) VecSet(&v2, i, i + 1);
    for (int i = 3; i--;) VecSet(&v3, i, i + 1);
    for (int i = 4; i--;) VecSet(&v4, i, i + 1);
    FILE *f = fopen("./UnitTestVecShortLoadSave.txt", "w");
    if (f == NULL) {
        PBMathErr->_type = PBErrTypeOther;
        sprintf(PBMathErr->_msg,
            "Can't open ./UnitTestVecShortLoadSave.txt for writing");
        PBErrCatch(PBMathErr);
    }
    if (!VecSave(v, f)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortSave NOK");
        PBErrCatch(PBMathErr);
    }
    if (!VecSave(&v2, f)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortSave NOK");
        PBErrCatch(PBMathErr);
    }
    if (!VecSave(&v3, f)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortSave NOK");
        PBErrCatch(PBMathErr);
    }
    if (!VecSave(&v4, f)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortSave NOK");
        PBErrCatch(PBMathErr);
    }
    fclose(f);
    VecShort *w = VecShortCreate(2);
    f = fopen("./UnitTestVecShortLoadSave.txt", "r");
    if (f == NULL) {
        PBMathErr->_type = PBErrTypeOther;
        sprintf(PBMathErr->_msg,
            "Can't open ./UnitTestVecShortLoadSave.txt for reading");
        PBErrCatch(PBMathErr);
    }
    if (!VecLoad(&w, f)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortLoad NOK");
        PBErrCatch(PBMathErr);
    }
    if (memcmp(v, w, sizeof(VecShort) + sizeof(short) * 5) != 0) {

```

```

    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortLoadSave NOK");
    PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortLoad NOK");
    PBErrCatch(PBMathErr);
}
if (memcmp(&v2, w, sizeof(VecShort) + sizeof(short) * 2) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortLoadSave NOK");
    PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortLoad NOK");
    PBErrCatch(PBMathErr);
}
if (memcmp(&v3, w, sizeof(VecShort) + sizeof(short) * 3) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortLoadSave NOK");
    PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortLoad NOK");
    PBErrCatch(PBMathErr);
}
if (memcmp(&v4, w, sizeof(VecShort) + sizeof(short) * 4) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortLoadSave NOK");
    PBErrCatch(PBMathErr);
}
fclose(f);
VecFree(&v);
VecFree(&w);
int ret = system("cat ./UnitTestVecShortLoadSave.txt");
printf("VecShortLoadSave OK\n");
ret = system("rm ./UnitTestVecShortLoadSave.txt");
ret = ret;
}

void UnitTestVecShortGetSetDim() {
    VecShort *v = VecShortCreate(5);
    VecShort2D v2 = VecShortCreateStatic2D();
    VecShort3D v3 = VecShortCreateStatic3D();
    VecShort4D v4 = VecShortCreateStatic4D();
    if (VecDim(v) != 5) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortDim NOK");
        PBErrCatch(PBMathErr);
    }
    for (int i = 5; i--;) VecSet(v, i, i + 1);
    for (int i = 2; i--;) VecSet(&v2, i, i + 1);
    for (int i = 3; i--;) VecSet(&v3, i, i + 1);
    for (int i = 4; i--;) VecSet(&v4, i, i + 1);
    for (int i = 5; i--;)
        if (v->_val[i] != i + 1) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortSet NOK");
            PBErrCatch(PBMathErr);
        }
}

```

```

    }
    for (int i = 2; i--;)
        if (v2._val[i] != i + 1) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortSet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 3; i--;)
        if (v3._val[i] != i + 1) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortSet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 4; i--;)
        if (v4._val[i] != i + 1) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortSet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 5; i--;)
        if (VecGet(v, i) != i + 1) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortGet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 2; i--;)
        if (VecGet(&v2, i) != i + 1) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortGet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 3; i--;)
        if (VecGet(&v3, i) != i + 1) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortGet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 4; i--;)
        if (VecGet(&v4, i) != i + 1) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortGet NOK");
            PBErrCatch(PBMathErr);
        }
    VecSetNull(v);
    VecSetNull(&v2);
    VecSetNull(&v3);
    VecSetNull(&v4);
    for (int i = 5; i--;)
        if (VecGet(v, i) != 0) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortGet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 2; i--;)
        if (VecGet(&v2, i) != 0) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecShortGet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 3; i--;)
        if (VecGet(&v3, i) != 0) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;

```



```
PBErrCatch(PBMathErr);
}
++iCheck;
} while (VecStep(&v3, &bv3));
iCheck = 0;
do {
    int a = VecGet(&v4, 0);
    for (int i = 1; i < 4; ++i)
        a = a * b[i] + VecGet(&v4, i);
    if (a != acheck[iCheck]) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortStep NOK");
        PBErrCatch(PBMathErr);
    }
    ++iCheck;
} while (VecStep(&v4, &bv4));
iCheck = 0;
do {
    int a = VecGet(v, VecDim(v) - 1);
    for (int i = VecDim(v) - 2; i >= 0; --i)
        a = a * b[i] + VecGet(v, i);
    if (a != acheck[iCheck]) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortPStep NOK");
        PBErrCatch(PBMathErr);
    }
    ++iCheck;
} while (VecPStep(v, bv));
iCheck = 0;
do {
    int a = VecGet(&v2, 1);
    a = a * b[0] + VecGet(&v2, 0);
    if (a != acheck[iCheck]) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortPStep NOK");
        PBErrCatch(PBMathErr);
    }
    ++iCheck;
} while (VecPStep(&v2, &bv2));
iCheck = 0;
do {
    int a = VecGet(&v3, 2);
    for (int i = 1; i >= 0; --i)
        a = a * b[i] + VecGet(&v3, i);
    if (a != acheck[iCheck]) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortPStep NOK");
        PBErrCatch(PBMathErr);
    }
    ++iCheck;
} while (VecPStep(&v3, &bv3));
iCheck = 0;
do {
    int a = VecGet(&v4, 3);
    for (int i = 2; i >= 0; --i)
        a = a * b[i] + VecGet(&v4, i);
    if (a != acheck[iCheck]) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortPStep NOK");
        PBErrCatch(PBMathErr);
    }
    ++iCheck;
```



```

    } while (VecPStep(&v4, &bv4));
    VecFree(&v);
    VecFree(&bv);
    printf("UnitTestVecShortStep OK\n");
}

void UnitTestVecShortHamiltonDist() {
    VecShort *v = VecShortCreate(5);
    VecShort2D v2 = VecShortCreateStatic2D();
    VecShort3D v3 = VecShortCreateStatic3D();
    VecShort4D v4 = VecShortCreateStatic4D();
    VecShort *w = VecShortCreate(5);
    VecShort2D w2 = VecShortCreateStatic2D();
    VecShort3D w3 = VecShortCreateStatic3D();
    VecShort4D w4 = VecShortCreateStatic4D();
    short b[5] = {-2, -1, 0, 1, 2};
    for (int i = 5; i--;) VecSet(v, i, b[i]);
    for (int i = 2; i--;) VecSet(&v2, i, b[i]);
    for (int i = 3; i--;) VecSet(&v3, i, b[i]);
    for (int i = 4; i--;) VecSet(&v4, i, b[i]);
    for (int i = 5; i--;) VecSet(w, i, b[4 - i] + 1);
    for (int i = 2; i--;) VecSet(&w2, i, b[1 - i] + 1);
    for (int i = 3; i--;) VecSet(&w3, i, b[2 - i] + 1);
    for (int i = 4; i--;) VecSet(&w4, i, b[3 - i] + 1);
    short dist = VecHamiltonDist(v, w);
    if (dist != 13) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortHamiltonDist NOK");
        PBErrCatch(PBMathErr);
    }
    dist = VecHamiltonDist(&v2, &w2);
    if (dist != 2) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortHamiltonDist NOK");
        PBErrCatch(PBMathErr);
    }
    dist = VecHamiltonDist(&v3, &w3);
    if (dist != 5) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortHamiltonDist NOK");
        PBErrCatch(PBMathErr);
    }
    dist = VecHamiltonDist(&v4, &w4);
    if (dist != 8) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortHamiltonDist NOK");
        PBErrCatch(PBMathErr);
    }
    VecFree(&v);
    VecFree(&w);
    printf("UnitTestVecShortHamiltonDist OK\n");
}

void UnitTestVecShortIsEqual() {
    VecShort *v = VecShortCreate(5);
    VecShort2D v2 = VecShortCreateStatic2D();
    VecShort3D v3 = VecShortCreateStatic3D();
    VecShort4D v4 = VecShortCreateStatic4D();
    for (int i = 5; i--;) VecSet(v, i, i + 1);
    for (int i = 2; i--;) VecSet(&v2, i, i + 1);
    for (int i = 3; i--;) VecSet(&v3, i, i + 1);
    for (int i = 4; i--;) VecSet(&v4, i, i + 1);

```

```

VecShort *w = VecShortCreate(5);
VecShort2D w2 = VecShortCreateStatic2D();
VecShort3D w3 = VecShortCreateStatic3D();
VecShort4D w4 = VecShortCreateStatic4D();
if (VecIsEqual(v, w)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortIsEqual NOK");
    PBErrCatch(PBMathErr);
}
if (VecIsEqual(&v2, &w2)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortIsEqual NOK");
    PBErrCatch(PBMathErr);
}
if (VecIsEqual(&v3, &w3)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortIsEqual NOK");
    PBErrCatch(PBMathErr);
}
if (VecIsEqual(&v4, &w4)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortIsEqual NOK");
    PBErrCatch(PBMathErr);
}
for (int i = 5; i--;) VecSet(w, i, i + 1);
for (int i = 2; i--;) VecSet(&w2, i, i + 1);
for (int i = 3; i--;) VecSet(&w3, i, i + 1);
for (int i = 4; i--;) VecSet(&w4, i, i + 1);
if (!VecIsEqual(v, w)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortIsEqual NOK");
    PBErrCatch(PBMathErr);
}
if (!VecIsEqual(&v2, &w2)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortIsEqual NOK");
    PBErrCatch(PBMathErr);
}
if (!VecIsEqual(&v3, &w3)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortIsEqual NOK");
    PBErrCatch(PBMathErr);
}
if (!VecIsEqual(&v4, &w4)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortIsEqual NOK");
    PBErrCatch(PBMathErr);
}
VecFree(&v);
VecFree(&w);
printf("UnitTestVecShortIsEqual OK\n");
}

void UnitTestVecShortCopy() {
    VecShort *v = VecShortCreate(5);
    VecShort2D v2 = VecShortCreateStatic2D();
    VecShort3D v3 = VecShortCreateStatic3D();
    VecShort4D v4 = VecShortCreateStatic4D();
    for (int i = 5; i--;) VecSet(v, i, i + 1);
    for (int i = 2; i--;) VecSet(&v2, i, i + 1);
    for (int i = 3; i--;) VecSet(&v3, i, i + 1);
    for (int i = 4; i--;) VecSet(&v4, i, i + 1);
}

```

```

VecShort *w = VecShortCreate(5);
VecShort2D w2 = VecShortCreateStatic2D();
VecShort3D w3 = VecShortCreateStatic3D();
VecShort4D w4 = VecShortCreateStatic4D();
VecCopy(w, v);
VecCopy(&w2, &v2);
VecCopy(&w3, &v3);
VecCopy(&w4, &v4);
if (!VecIsEqual(v, w)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortCopy NOK");
    PBErrCatch(PBMathErr);
}
if (!VecIsEqual(&v2, &w2)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortCopy NOK");
    PBErrCatch(PBMathErr);
}
if (!VecIsEqual(&v3, &w3)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortCopy NOK");
    PBErrCatch(PBMathErr);
}
if (!VecIsEqual(&v4, &w4)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecShortCopy NOK");
    PBErrCatch(PBMathErr);
}
VecFree(&v);
VecFree(&w);
printf("UnitTestVecShortCopy OK\n");
}

void UnitTestVecShortDotProd() {
    VecShort *v = VecShortCreate(5);
    VecShort2D v2 = VecShortCreateStatic2D();
    VecShort3D v3 = VecShortCreateStatic3D();
    VecShort4D v4 = VecShortCreateStatic4D();
    VecShort *w = VecShortCreate(5);
    VecShort2D w2 = VecShortCreateStatic2D();
    VecShort3D w3 = VecShortCreateStatic3D();
    VecShort4D w4 = VecShortCreateStatic4D();
    short b[5] = {-2, -1, 0, 1, 2};
    for (int i = 5; i--;) VecSet(v, i, b[i]);
    for (int i = 2; i--;) VecSet(&v2, i, b[i]);
    for (int i = 3; i--;) VecSet(&v3, i, b[i]);
    for (int i = 4; i--;) VecSet(&v4, i, b[i]);
    for (int i = 5; i--;) VecSet(w, i, b[4 - i] + 1);
    for (int i = 2; i--;) VecSet(&w2, i, b[1 - i] + 1);
    for (int i = 3; i--;) VecSet(&w3, i, b[2 - i] + 1);
    for (int i = 4; i--;) VecSet(&w4, i, b[3 - i] + 1);
    short prod = VecDotProd(v, w);
    if (prod != -10) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortDotProd NOK");
        PBErrCatch(PBMathErr);
    }
    prod = VecDotProd(&v2, &w2);
    if (prod != 1) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortDotProd NOK");
        PBErrCatch(PBMathErr);
    }
}

```

```

    }
    prod = VecDotProd(&v3, &w3);
    if (prod != -2) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortDotProd NOK");
        PBErrCatch(PBMathErr);
    }
    prod = VecDotProd(&v4, &w4);
    if (prod != -6) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShortDotProd NOK");
        PBErrCatch(PBMathErr);
    }
    VecFree(&v);
    VecFree(&w);
    printf("UnitTestVecShortDotProd OK\n");
}

void UnitTestSpeedVecShort() {
    VecShort *v = VecShortCreate(5);
    VecShort2D v2 = VecShortCreateStatic2D();
    VecShort3D v3 = VecShortCreateStatic3D();
    VecShort4D v4 = VecShortCreateStatic4D();
    int nbTest = 100000;

    srandom(RANDOMSEED);
    int i = nbTest;
    clock_t clockBefore = clock();
    for (; i--;) {
        int j = INT(rnd() * ((float)(VecDim(v) - 1) - PBMath_EPSILON));
        short val = 1;
        VecSet(v, j, val);
        short valb = VecGet(v, j);
        valb = valb;
    }
    clock_t clockAfter = clock();
    double timeV = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    srandom(RANDOMSEED);
    i = nbTest;
    clockBefore = clock();
    short* array = malloc(sizeof(short) * 5);
    for (; i--;) {
        int j = INT(rnd() * ((float)(VecDim(v) - 1) - PBMath_EPSILON));
        short val = 1;
        array[j] = val;
        short valb = array[j];
        valb = valb;
    }
    clockAfter = clock();
    double timeRef = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    printf("VecShort: %fms, array: %fms\n",
        timeV / (float)nbTest, timeRef / (float)nbTest);
    if (timeV / (float)nbTest > 2.0 * timeRef / (float)nbTest) {
#ifdef BUILDMODE == 0
        PBMathErr->_fatal = false;
#endif
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestSpeedVecShort NOK");
        PBErrCatch(PBMathErr);
    }
}

```

```

srandom(RANDOMSEED);
i = nbTest;
clockBefore = clock();
for (; i--;) {
    int j = INT(rnd() * (1.0 - PBMath_EPSILON));
    short val = 1;
    VecSet(&v2, j, val);
    short valb = VecGet(&v2, j);
    valb = valb;
}
clockAfter = clock();
timeV = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
srandom(RANDOMSEED);
i = nbTest;
clockBefore = clock();
short array2[2];
for (; i--;) {
    int j = INT(rnd() * (1.0 - PBMath_EPSILON));
    short val = 1;
    array2[j] = val;
    short valb = array2[j];
    valb = valb;
}
clockAfter = clock();
timeRef = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
printf("VecShort2D: %fms, array: %fms\n",
    timeV / (float)nbTest, timeRef / (float)nbTest);
if (timeV / (float)nbTest > 2.0 * timeRef / (float)nbTest) {
#if BUILDMODE == 0
    PBMathErr->_fatal = false;
#endif
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "UnitTestSpeedVecShort NOK");
    PBErrCatch(PBMathErr);
}

srandom(RANDOMSEED);
i = nbTest;
clockBefore = clock();
for (; i--;) {
    int j = INT(rnd() * (2.0 - PBMath_EPSILON));
    short val = 1;
    VecSet(&v3, j, val);
    short valb = VecGet(&v3, j);
    valb = valb;
}
clockAfter = clock();
timeV = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
srandom(RANDOMSEED);
i = nbTest;
clockBefore = clock();
short array3[3];
for (; i--;) {
    int j = INT(rnd() * (2.0 - PBMath_EPSILON));
    short val = 1;
    array3[j] = val;
    short valb = array3[j];
    valb = valb;
}

```

```

    }
    clockAfter = clock();
    timeRef = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    printf("VecShort3D: %fms, array: %fms\n",
        timeV / (float)nbTest, timeRef / (float)nbTest);
    if (timeV / (float)nbTest > 2.0 * timeRef / (float)nbTest) {
#if BUILDMODE == 0
        PBMathErr->_fatal = false;
#endif
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestSpeedVecShort NOK");
        PBErrCatch(PBMathErr);
    }

    srandom(RANDOMSEED);
    i = nbTest;
    clockBefore = clock();
    for (; i--;) {
        int j = INT(rnd() * (3.0 - PBMath_EPSILON));
        short val = 1;
        VecSet(&v4, j, val);
        short valb = VecGet(&v4, j);
        valb = valb;
    }
    clockAfter = clock();
    timeV = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    srandom(RANDOMSEED);
    i = nbTest;
    clockBefore = clock();
    short array4[4];
    for (; i--;) {
        int j = INT(rnd() * (3.0 - PBMath_EPSILON));
        short val = 1;
        array4[j] = val;
        short valb = array4[j];
        valb = valb;
    }
    clockAfter = clock();
    timeRef = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    printf("VecShort4D: %fms, array: %fms\n",
        timeV / (float)nbTest, timeRef / (float)nbTest);
    if (timeV / (float)nbTest > 2.0 * timeRef / (float)nbTest) {
#if BUILDMODE == 0
        PBMathErr->_fatal = false;
#endif
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestSpeedVecShort NOK");
        PBErrCatch(PBMathErr);
    }

    VecFree(&v);
    free(array);
    printf("UnitTestSpeedVecShort OK\n");
}

void UnitTestVecShortToFloat() {
    VecShort *v = VecShortCreate(5);
    VecShort2D v2 = VecShortCreateStatic2D();
    VecShort3D v3 = VecShortCreateStatic3D();

```

```

VecShort4D v4 = VecShortCreateStatic4D();
for (int i = 5; i--;) VecSet(v, i, i + 1);
for (int i = 2; i--;) VecSet(&v2, i, i + 1);
for (int i = 3; i--;) VecSet(&v3, i, i + 1);
for (int i = 4; i--;) VecSet(&v4, i, i + 1);
VecFloat *w = VecShortToFloat(v);
VecFloat2D w2 = VecShortToFloat2D(&v2);
VecFloat3D w3 = VecShortToFloat3D(&v3);
VecPrint(w, stdout); printf("\n");
VecPrint(&w2, stdout); printf("\n");
VecPrint(&w3, stdout); printf("\n");
VecFree(&v);
VecFree(&w);
printf("UnitTestVecShortToFloat OK\n");
}

void UnitTestVecShort() {
    UnitTestVecShortCreateFree();
    UnitTestVecShortClone();
    UnitTestVecShortLoadSave();
    UnitTestVecShortGetSetDim();
    UnitTestVecShortStep();
    UnitTestVecShortHamiltonDist();
    UnitTestVecShortIsEqual();
    UnitTestVecShortDotProd();
    UnitTestVecShortCopy();
    UnitTestSpeedVecShort();
    UnitTestVecShortToFloat();
    printf("UnitTestVecShort OK\n");
}

void UnitTestVecFloatCreateFree() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();
    VecPrint(v, stdout); printf("\n");
    VecPrint(&v2, stdout); printf("\n");
    VecPrint(&v3, stdout); printf("\n");
    VecFloatFree(&v);
    if (v != NULL) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloat is not null after VecFloatFree");
        PBErrCatch(PBMathErr);
    }
    printf("VecFloatCreateFree OK\n");
}

void UnitTestVecFloatClone() {
    VecFloat *v = VecFloatCreate(5);
    for (int i = 5; i--;) VecSet(v, i, i + 1);
    VecFloat *w = VecClone(v);
    if (memcmp(v, w, sizeof(VecFloat) + sizeof(float) * 5) != 0) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatClone NOK");
        PBErrCatch(PBMathErr);
    }
    VecFloatFree(&v);
    VecFloatFree(&w);
    printf("VecFloatClone OK\n");
}

void UnitTestVecFloatLoadSave() {

```

```

VecFloat *v = VecFloatCreate(5);
VecFloat2D v2 = VecFloatCreateStatic2D();
VecFloat3D v3 = VecFloatCreateStatic3D();
for (int i = 5; i--;) VecSet(v, i, i + 1);
for (int i = 2; i--;) VecSet(&v2, i, i + 1);
for (int i = 3; i--;) VecSet(&v3, i, i + 1);
FILE *f = fopen("./UnitTestVecFloatLoadSave.txt", "w");
if (f == NULL) {
    PBMathErr->_type = PBErrTypeOther;
    sprintf(PBMathErr->_msg,
        "Can't open ./UnitTestVecFloatLoadSave.txt for writing");
    PBErrCatch(PBMathErr);
}
if (!VecSave(v, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatSave NOK");
    PBErrCatch(PBMathErr);
}
if (!VecSave(&v2, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatSave NOK");
    PBErrCatch(PBMathErr);
}
if (!VecSave(&v3, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatSave NOK");
    PBErrCatch(PBMathErr);
}
fclose(f);
VecFloat *w = VecFloatCreate(2);
f = fopen("./UnitTestVecFloatLoadSave.txt", "r");
if (f == NULL) {
    PBMathErr->_type = PBErrTypeOther;
    sprintf(PBMathErr->_msg,
        "Can't open ./UnitTestVecFloatLoadSave.txt for reading");
    PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatLoad NOK");
    PBErrCatch(PBMathErr);
}
if (memcmp(v, w, sizeof(VecFloat) + sizeof(float) * 5) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatLoadSave NOK");
    PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatLoad NOK");
    PBErrCatch(PBMathErr);
}
if (memcmp(&v2, w, sizeof(VecFloat) + sizeof(float) * 2) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatLoadSave NOK");
    PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatLoad NOK");
    PBErrCatch(PBMathErr);
}
}

```



```

if (memcmp(&v3, w, sizeof(VecFloat) + sizeof(float) * 3) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatLoadSave NOK");
    PBErrCatch(PBMathErr);
}
fclose(f);
VecFree(&v);
VecFree(&w);
int ret = system("cat ./UnitTestVecFloatLoadSave.txt");
printf("VecFloatLoadSave OK\n");
ret = system("rm ./UnitTestVecFloatLoadSave.txt");
ret = ret;
}

void UnitTestVecFloatGetSetDim() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();
    if (VecDim(v) != 5) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatDim NOK");
        PBErrCatch(PBMathErr);
    }
    for (int i = 5; i--;) VecSet(v, i, (float)(i + 1));
    for (int i = 2; i--;) VecSet(&v2, i, (float)(i + 1));
    for (int i = 3; i--;) VecSet(&v3, i, (float)(i + 1));
    for (int i = 5; i--;)
        if (!ISEQUALF(v->_val[i], (float)(i + 1))) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatSet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 2; i--;)
        if (!ISEQUALF(v2._val[i], (float)(i + 1))) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatSet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 3; i--;)
        if (!ISEQUALF(v3._val[i], (float)(i + 1))) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatSet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 5; i--;)
        if (!ISEQUALF(VecGet(v, i), (float)(i + 1))) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatGet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 2; i--;)
        if (!ISEQUALF(VecGet(&v2, i), (float)(i + 1))) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatGet NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 3; i--;)
        if (!ISEQUALF(VecGet(&v3, i), (float)(i + 1))) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatGet NOK");
            PBErrCatch(PBMathErr);
        }
}

```

```

VecSetNull(v);
VecSetNull(&v2);
VecSetNull(&v3);
for (int i = 5; i--;)
    if (!ISEQUALF(VecGet(v, i), 0.0)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatGet NOK");
        PBErrCatch(PBMathErr);
    }
for (int i = 2; i--;)
    if (!ISEQUALF(VecGet(&v2, i), 0.0)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatGet NOK");
        PBErrCatch(PBMathErr);
    }
for (int i = 3; i--;)
    if (!ISEQUALF(VecGet(&v3, i), 0.0)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatGet NOK");
        PBErrCatch(PBMathErr);
    }
VecFree(&v);
printf("VecFloatGetSetDim OK\n");
}

void UnitTestVecFloatCopy() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();
    for (int i = 5; i--;) VecSet(v, i, i + 1);
    for (int i = 2; i--;) VecSet(&v2, i, i + 1);
    for (int i = 3; i--;) VecSet(&v3, i, i + 1);
    VecFloat *w = VecFloatCreate(5);
    VecFloat2D w2 = VecFloatCreateStatic2D();
    VecFloat3D w3 = VecFloatCreateStatic3D();
    VecCopy(w, v);
    VecCopy(&w2, &v2);
    VecCopy(&w3, &v3);
    if (!VecIsEqual(v, w)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatCopy NOK");
        PBErrCatch(PBMathErr);
    }
    if (!VecIsEqual(&v2, &w2)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatCopy NOK");
        PBErrCatch(PBMathErr);
    }
    if (!VecIsEqual(&v3, &w3)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatCopy NOK");
        PBErrCatch(PBMathErr);
    }
    VecFree(&v);
    VecFree(&w);
    printf("UnitTestVecFloatCopy OK\n");
}

void UnitTestVecFloatNorm() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();

```

```

for (int i = 5; i--;) VecSet(v, i, i + 1);
for (int i = 2; i--;) VecSet(&v2, i, i + 1);
for (int i = 3; i--;) VecSet(&v3, i, i + 1);
if (!ISEQUALF(VecNorm(v), 7.416198)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatNorm NOK");
    PBErrCatch(PBMathErr);
}
if (!ISEQUALF(VecNorm(&v2), 2.236068)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatNorm NOK");
    PBErrCatch(PBMathErr);
}
if (!ISEQUALF(VecNorm(&v3), 3.741657)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatNorm NOK");
    PBErrCatch(PBMathErr);
}
VecNormalise(v);
VecNormalise(&v2);
VecNormalise(&v3);
if (!ISEQUALF(VecNorm(v), 1.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatNormalise NOK");
    PBErrCatch(PBMathErr);
}
if (!ISEQUALF(VecNorm(&v2), 1.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatNormalise NOK");
    PBErrCatch(PBMathErr);
}
if (!ISEQUALF(VecNorm(&v3), 1.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatNormalise NOK");
    PBErrCatch(PBMathErr);
}
VecFree(&v);
printf("UnitTestVecFloatNorm OK\n");
}

void UnitTestVecFloatDist() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();
    VecFloat *w = VecFloatCreate(5);
    VecFloat2D w2 = VecFloatCreateStatic2D();
    VecFloat3D w3 = VecFloatCreateStatic3D();
    float b[5] = {-2.0, -1.0, 0.0, 1.0, 2.0};
    for (int i = 5; i--;) VecSet(v, i, b[i]);
    for (int i = 2; i--;) VecSet(&v2, i, b[i]);
    for (int i = 3; i--;) VecSet(&v3, i, b[i]);
    for (int i = 5; i--;) VecSet(w, i, b[4 - i] + 1.5);
    for (int i = 2; i--;) VecSet(&w2, i, b[1 - i] + 1.5);
    for (int i = 3; i--;) VecSet(&w3, i, b[2 - i] + 1.5);
    if (!ISEQUALF(VecDist(v, w), 7.158911)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatDist NOK");
        PBErrCatch(PBMathErr);
    }
    if (!ISEQUALF(VecDist(&v2, &w2), 2.549510)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatDist NOK");
    }
}

```

```

    PBErriCatch(PBMathErr);
}
if (!ISEQUALF(VecDist(&v3, &w3), 3.840573)) {
    PBMathErr->_type = PBErriTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatDist NOK");
    PBErriCatch(PBMathErr);
}
if (!ISEQUALF(VecHamiltonDist(v, w), 13.5)) {
    PBMathErr->_type = PBErriTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatHamiltonDist NOK");
    PBErriCatch(PBMathErr);
}
if (!ISEQUALF(VecHamiltonDist(&v2, &w2), 3.0)) {
    PBMathErr->_type = PBErriTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatHamiltonDist NOK");
    PBErriCatch(PBMathErr);
}
if (!ISEQUALF(VecHamiltonDist(&v3, &w3), 5.5)) {
    PBMathErr->_type = PBErriTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatHamiltonDist NOK");
    PBErriCatch(PBMathErr);
}
if (!ISEQUALF(VecPixelDist(v, w), 13.0)) {
    PBMathErr->_type = PBErriTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatPixelDist NOK");
    PBErriCatch(PBMathErr);
}
if (!ISEQUALF(VecPixelDist(&v2, &w2), 2.0)) {
    PBMathErr->_type = PBErriTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatPixelDist NOK");
    PBErriCatch(PBMathErr);
}
if (!ISEQUALF(VecPixelDist(&v3, &w3), 5.0)) {
    PBMathErr->_type = PBErriTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecFloatPixelDist NOK");
    PBErriCatch(PBMathErr);
}
VecFree(&v);
VecFree(&w);
printf("UnitTestVecFloatDist OK\n");
}

void UnitTestVecFloatIsEqual() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();
    for (int i = 5; i--;) VecSet(v, i, i + 1);
    for (int i = 2; i--;) VecSet(&v2, i, i + 1);
    for (int i = 3; i--;) VecSet(&v3, i, i + 1);
    VecFloat *w = VecFloatCreate(5);
    VecFloat2D w2 = VecFloatCreateStatic2D();
    VecFloat3D w3 = VecFloatCreateStatic3D();
    if (VecIsEqual(v, w)) {
        PBMathErr->_type = PBErriTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatIsEqual NOK");
        PBErriCatch(PBMathErr);
    }
    if (VecIsEqual(&v2, &w2)) {
        PBMathErr->_type = PBErriTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatIsEqual NOK");
        PBErriCatch(PBMathErr);
    }
}

```

```

    if (VecIsEqual(&v3, &w3)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatIsEqual NOK");
        PBErrCatch(PBMathErr);
    }
    for (int i = 5; i--;) VecSet(w, i, i + 1);
    for (int i = 2; i--;) VecSet(&w2, i, i + 1);
    for (int i = 3; i--;) VecSet(&w3, i, i + 1);
    if (!VecIsEqual(v, w)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatIsEqual NOK");
        PBErrCatch(PBMathErr);
    }
    if (!VecIsEqual(&v2, &w2)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatIsEqual NOK");
        PBErrCatch(PBMathErr);
    }
    if (!VecIsEqual(&v3, &w3)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatIsEqual NOK");
        PBErrCatch(PBMathErr);
    }
    VecFree(&v);
    VecFree(&w);
    printf("UnitTestVecFloatIsEqual OK\n");
}

void UnitTestVecFloatScale() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();
    for (int i = 5; i--;) VecSet(v, i, i + 1);
    for (int i = 2; i--;) VecSet(&v2, i, i + 1);
    for (int i = 3; i--;) VecSet(&v3, i, i + 1);
    float a = 0.1;
    VecFloat *w = VecGetScale(v, a);
    VecFloat2D w2 = VecGetScale(&v2, a);
    VecFloat3D w3 = VecGetScale(&v3, a);
    VecScale(v, a);
    VecScale(&v2, a);
    VecScale(&v3, a);
    for (int i = 5; i--;)
        if (!ISEQUALF(VecGet(w, i), (float)(i + 1) * a)) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatGetScale NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 2; i--;)
        if (!ISEQUALF(VecGet(&w2, i), (float)(i + 1) * a)) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatGetScale NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 3; i--;)
        if (!ISEQUALF(VecGet(&w3, i), (float)(i + 1) * a)) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatGetScale NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 5; i--;)
        if (!ISEQUALF(VecGet(v, i), (float)(i + 1) * a)) {

```

```

        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatScale NOK");
        PBErrCatch(PBMathErr);
    }
    for (int i = 2; i--;)
        if (!ISEQUALF(VecGet(&v2, i), (float)(i + 1) * a)) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatScale NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 3; i--;)
        if (!ISEQUALF(VecGet(&v3, i), (float)(i + 1) * a)) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatScale NOK");
            PBErrCatch(PBMathErr);
        }
    VecFree(&v);
    VecFree(&w);
    printf("UnitTestVecFloatScale OK\n");
}

void UnitTestVecFloatOp() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();
    VecFloat *w = VecFloatCreate(5);
    VecFloat2D w2 = VecFloatCreateStatic2D();
    VecFloat3D w3 = VecFloatCreateStatic3D();
    for (int i = 5; i--;) VecSet(v, i, i + 1);
    for (int i = 2; i--;) VecSet(&v2, i, i + 1);
    for (int i = 3; i--;) VecSet(&v3, i, i + 1);
    float a[2] = {-0.1, 2.0};
    float b[5] = {-2.0, -1.0, 0.0, 1.0, 2.0};
    for (int i = 5; i--;) VecSet(v, i, b[i]);
    for (int i = 2; i--;) VecSet(&v2, i, b[i]);
    for (int i = 3; i--;) VecSet(&v3, i, b[i]);
    for (int i = 5; i--;) VecSet(w, i, b[4 - i] + 0.5);
    for (int i = 2; i--;) VecSet(&w2, i, b[1 - i] + 0.5);
    for (int i = 3; i--;) VecSet(&w3, i, b[2 - i] + 0.5);
    VecFloat *u = VecGetOp(v, a[0], w, a[1]);
    VecFloat2D u2 = VecGetOp(&v2, a[0], &w2, a[1]);
    VecFloat3D u3 = VecGetOp(&v3, a[0], &w3, a[1]);
    float checku[5] = {5.2, 3.1, 1.0, -1.1, -3.2};
    float checku2[2] = {-0.8, -2.9};
    float checku3[3] = {1.2, -0.9, -3.0};
    for (int i = 5; i--;)
        if (!ISEQUALF(VecGet(u, i), checku[i])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatGetOp NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 2; i--;)
        if (!ISEQUALF(VecGet(&u2, i), checku2[i])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatGetOp NOK");
            PBErrCatch(PBMathErr);
        }
    for (int i = 3; i--;)
        if (!ISEQUALF(VecGet(&u3, i), checku3[i])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "VecFloatGetOp NOK");
            PBErrCatch(PBMathErr);
        }
}

```

```

    }
    VecOp(v, a[0], w, a[1]);
    VecOp(&v2, a[0], &w2, a[1]);
    VecOp(&v3, a[0], &w3, a[1]);
    if (!VecIsEqual(v, u)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatOp NOK");
        PBErrCatch(PBMathErr);
    }
    if (!VecIsEqual(&v2, &u2)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatOp NOK");
        PBErrCatch(PBMathErr);
    }
    if (!VecIsEqual(&v3, &u3)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatOp NOK");
        PBErrCatch(PBMathErr);
    }
    VecFree(&v);
    VecFree(&w);
    VecFree(&u);
    printf("UnitTestVecFloatOp OK\n");
}

void UnitTestVecFloatDotProd() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();
    VecFloat *w = VecFloatCreate(5);
    VecFloat2D w2 = VecFloatCreateStatic2D();
    VecFloat3D w3 = VecFloatCreateStatic3D();
    float b[5] = {-2.0, -1.0, 0.0, 1.0, 2.0};
    for (int i = 5; i--;) VecSet(v, i, b[i]);
    for (int i = 2; i--;) VecSet(&v2, i, b[i]);
    for (int i = 3; i--;) VecSet(&v3, i, b[i]);
    for (int i = 5; i--;) VecSet(w, i, b[4 - i] + 1.5);
    for (int i = 2; i--;) VecSet(&w2, i, b[1 - i] + 1.5);
    for (int i = 3; i--;) VecSet(&w3, i, b[2 - i] + 1.5);
    float prod = VecDotProd(v, w);
    if (!ISEQUALF(prod, -10.0)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatDotProd NOK");
        PBErrCatch(PBMathErr);
    }
    prod = VecDotProd(&v2, &w2);
    if (!ISEQUALF(prod, -0.5)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatDotProd NOK");
        PBErrCatch(PBMathErr);
    }
    prod = VecDotProd(&v3, &w3);
    if (!ISEQUALF(prod, -3.5)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatDotProd NOK");
        PBErrCatch(PBMathErr);
    }
    VecFree(&v);
    VecFree(&w);
    printf("UnitTestVecFloatDotProd OK\n");
}

```

```

void UnitTestVecFloatRotAngleTo() {
    VecFloat *v = VecFloatCreate(2);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat *w = VecFloatCreate(2);
    VecFloat2D w2 = VecFloatCreateStatic2D();
    VecSet(v, 0, 1.0);
    VecSet(&v2, 0, 1.0);
    VecSet(w, 0, 1.0);
    VecSet(&w2, 0, 1.0);
    float a = 0.0;
    float da = PBMath_TWOPI_DIV_360;
    for (int i = 360; i--;) {
        VecRot(v, da);
        VecNormalise(v);
        VecRot(&v2, da);
        VecNormalise(&v2);
        a += da;
        if (ISEQUALF(a, PBMath_PI)) {
            a = -PBMath_PI;
            if (!ISEQUALF(fabs(VecAngleTo(w, v)), fabs(a))) {
                PBMathErr->_type = PBErrTypeUnitTestFailed;
                sprintf(PBMathErr->_msg, "VecFloatAngleTo NOK");
                PBErrCatch(PBMathErr);
            }
            if (!ISEQUALF(fabs(VecAngleTo(&w2, &v2)), fabs(a))) {
                PBMathErr->_type = PBErrTypeUnitTestFailed;
                sprintf(PBMathErr->_msg, "VecFloatAngleTo NOK");
                PBErrCatch(PBMathErr);
            }
        } else {
            if (!ISEQUALF(VecAngleTo(w, v), a)) {
                PBMathErr->_type = PBErrTypeUnitTestFailed;
                sprintf(PBMathErr->_msg, "VecFloatAngleTo NOK");
                PBErrCatch(PBMathErr);
            }
            if (!ISEQUALF(VecAngleTo(&w2, &v2), a)) {
                PBMathErr->_type = PBErrTypeUnitTestFailed;
                sprintf(PBMathErr->_msg, "VecFloatAngleTo NOK");
                PBErrCatch(PBMathErr);
            }
        }
    }
    VecSet(v, 0, 1.0);
    VecSet(v, 1, 0.0);
    VecRot(v, PBMath_QUARTERPI);
    VecNormalise(v);
    VecPrint(v, stdout); printf("\n");
    if (!ISEQUALF(VecGet(v, 0), 0.70711) ||
        !ISEQUALF(VecGet(v, 1), 0.70711)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecFloatRot NOK");
        PBErrCatch(PBMathErr);
    }
    VecFree(&v);
    VecFree(&w);
    printf("UnitTestVecFloatAngleTo OK\n");
}

void UnitTestVecFloatToShort() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();

```



```

    for (int i = 5; i--;) VecSet(v, i, i + 1);
    for (int i = 2; i--;) VecSet(&v2, i, i + 1);
    for (int i = 3; i--;) VecSet(&v3, i, i + 1);
    VecShort *w = VecFloatToShort(v);
    VecShort2D w2 = VecFloatToShort2D(&v2);
    VecShort3D w3 = VecFloatToShort3D(&v3);
    VecPrint(w, stdout); printf("\n");
    VecPrint(&w2, stdout); printf("\n");
    VecPrint(&w3, stdout); printf("\n");
    VecFree(&v);
    VecFree(&w);
    printf("UnitTestVecFloatToShort OK\n");
}

void UnitTestSpeedVecFloat() {
    VecFloat *v = VecFloatCreate(5);
    VecFloat2D v2 = VecFloatCreateStatic2D();
    VecFloat3D v3 = VecFloatCreateStatic3D();
    int nbTest = 100000;

    srand(RANDOMSEED);
    int i = nbTest;
    clock_t clockBefore = clock();
    for (; i--;) {
        int j = INT(rnd() * ((float)(VecDim(v) - 1) - PBMath_EPSILON));
        float val = 1.0;
        VecSet(v, j, val);
        float valb = VecGet(v, j);
        valb = valb;
    }
    clock_t clockAfter = clock();
    double timeV = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    srand(RANDOMSEED);
    i = nbTest;
    clockBefore = clock();
    float* array = malloc(sizeof(float) * 5);
    for (; i--;) {
        int j = INT(rnd() * ((float)(VecDim(v) - 1) - PBMath_EPSILON));
        float val = 1.0;
        array[j] = val;
        float valb = array[j];
        valb = valb;
    }
    clockAfter = clock();
    double timeRef = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    printf("VecFloat: %fms, array: %fms\n",
        timeV / (float)nbTest, timeRef / (float)nbTest);
    if (timeV / (float)nbTest > 2.0 * timeRef / (float)nbTest) {
#ifdef BUILDMODE == 0
        PBMathErr->_fatal = false;
#endif
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestSpeedVecFloat NOK");
        PBErrCatch(PBMathErr);
    }

    srand(RANDOMSEED);
    i = nbTest;
    clockBefore = clock();
    for (; i--;) {

```

```

        int j = INT(rnd() * (1.0 - PBMath_EPSILON));
        float val = 1.0;
        VecSet(&v2, j, val);
        float valb = VecGet(&v2, j);
        valb = valb;
    }
    clockAfter = clock();
    timeV = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    srandom(RANDOMSEED);
    i = nbTest;
    clockBefore = clock();
    float array2[2];
    for (; i--;) {
        int j = INT(rnd() * (1.0 - PBMath_EPSILON));
        float val = 1.0;
        array2[j] = val;
        float valb = array2[j];
        valb = valb;
    }
    clockAfter = clock();
    timeRef = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    printf("VecFloat2D: %fms, array: %fms\n",
        timeV / (float)nbTest, timeRef / (float)nbTest);
    if (timeV / (float)nbTest > 2.0 * timeRef / (float)nbTest) {
#ifdef BUILDMODE == 0
        PBMathErr->_fatal = false;
#endif
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestSpeedVecFloat NOK");
        PBErrCatch(PBMathErr);
    }

    srandom(RANDOMSEED);
    i = nbTest;
    clockBefore = clock();
    for (; i--;) {
        int j = INT(rnd() * (2.0 - PBMath_EPSILON));
        float val = 1.0;
        VecSet(&v3, j, val);
        float valb = VecGet(&v3, j);
        valb = valb;
    }
    clockAfter = clock();
    timeV = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    srandom(RANDOMSEED);
    i = nbTest;
    clockBefore = clock();
    float array3[3];
    for (; i--;) {
        int j = INT(rnd() * (2.0 - PBMath_EPSILON));
        float val = 1.0;
        array3[j] = val;
        float valb = array3[j];
        valb = valb;
    }
    clockAfter = clock();
    timeRef = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    printf("VecFloat3D: %fms, array: %fms\n",

```

```

        timeV / (float)nbTest, timeRef / (float)nbTest);
    if (timeV / (float)nbTest > 2.0 * timeRef / (float)nbTest) {
#if BUILDMODE == 0
        PBMathErr->_fatal = false;
#endif
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestSpeedVecFloat NOK");
        PBErrCatch(PBMathErr);
    }

    VecFree(&v);
    free(array);
    printf("UnitTestSpeedVecFloat OK\n");
}

void UnitTestVecFloat() {
    UnitTestVecFloatCreateFree();
    UnitTestVecFloatClone();
    UnitTestVecFloatLoadSave();
    UnitTestVecFloatGetSetDim();
    UnitTestVecFloatCopy();
    UnitTestVecFloatNorm();
    UnitTestVecFloatDist();
    UnitTestVecFloatIsEqual();
    UnitTestVecFloatScale();
    UnitTestVecFloatOp();
    UnitTestVecFloatDotProd();
    UnitTestVecFloatRotAngleTo();
    UnitTestVecFloatToShort();
    UnitTestSpeedVecFloat();
    printf("UnitTestVecFloat OK\n");
}

void UnitTestMatFloatCreateFree() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2);
    VecSet(&dim, 1, 3);
    MatFloat *mat = MatFloatCreate(&dim);
    for (int i = VecGet(&dim, 0) * VecGet(&dim, 1); i--;) {
        if (!ISEQUALF(mat->_val[i], 0.0)) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestMatFloatCreateFree NOK");
            PBErrCatch(PBMathErr);
        }
    }
    MatFree(&mat);
    if (mat != NULL) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "mat is not null after MatFree");
        PBErrCatch(PBMathErr);
    }
    printf("UnitTestMatFloatCreateFree OK\n");
}

void UnitTestMatFloatGetSetDim() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2);
    VecSet(&dim, 1, 3);
    MatFloat *mat = MatFloatCreate(&dim);
    if (!VecIsEqual(&(mat->_dim), &dim)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestMatFloatGetSetDim NOK");
    }
}

```

```

    PBErCatch(PBMathErr);
}
VecShort2D i = VecShortCreateStatic2D();
float v = 1.0;
do {
    MatSet(mat, &i, v);
    v += 1.0;
} while(VecStep(&i, &dim));
v = 1.0;
for (int j = 0; j < VecGet(&dim, 0); ++j) {
    for (int k = 0; k < VecGet(&dim, 1); ++k) {
        if (!ISEQUALF(mat->_val[k * VecGet(&dim, 0) + j], v)) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestMatFloatGetSetDim NOK");
            PBErCatch(PBMathErr);
        }
        v += 1.0;
    }
}
VecSetNull(&i);
v = 1.0;
do {
    float w = MatGet(mat, &i);
    if (!ISEQUALF(v, w)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestMatFloatGetSetDim NOK");
        PBErCatch(PBMathErr);
    }
    v += 1.0;
} while(VecStep(&i, &dim));
MatFree(&mat);
printf("UnitTestMatFloatGetSetDim OK\n");
}

void UnitTestMatFloatCloneIsEqual() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2);
    VecSet(&dim, 1, 3);
    MatFloat *mat = MatFloatCreate(&dim);
    VecShort2D i = VecShortCreateStatic2D();
    float v = 1.0;
    do {
        MatSet(mat, &i, v);
        v += 1.0;
    } while(VecStep(&i, &dim));
    MatFloat *clone = MatClone(mat);
    if (!VecIsEqual(&(mat->_dim), &(clone->_dim))) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestMatFloatClone NOK");
        PBErCatch(PBMathErr);
    }
    VecSetNull(&i);
    do {
        if (!ISEQUALF(MatGet(mat, &i), MatGet(clone, &i))) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestMatFloatClone NOK");
            PBErCatch(PBMathErr);
        }
    } while(VecStep(&i, &dim));
    if (MatIsEqual(mat, clone) == false) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestMatFloatIsEqual NOK1");
    }
}

```

```

        PBErCatch(PBMathErr);
    }
    VecSet(&i, 0, 0); VecSet(&i, 1, 0);
    MatSet(clone, &i, -1.0);
    if (MatIsEqual(mat, clone) == true) {
        PBMathErr->_type = PBErTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestMatFloatIsEqual NOK2");
        PBErCatch(PBMathErr);
    }
    MatFree(&mat);
    MatFree(&clone);
    printf("UnitTestMatFloatCloneIsEqual OK\n");
}

void UnitTestMatFloatLoadSave() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2);
    VecSet(&dim, 1, 3);
    MatFloat *mat = MatFloatCreate(&dim);
    VecShort2D i = VecShortCreateStatic2D();
    float v = 1.0;
    do {
        MatSet(mat, &i, v);
        v += 1.0;
    } while(VecStep(&i, &dim));
    FILE *f = fopen("./UnitTestMatFloatLoadSave.txt", "w");
    if (f == NULL) {
        PBMathErr->_type = PBErTypeOther;
        sprintf(PBMathErr->_msg,
            "Can't open ./UnitTestMatFloatLoadSave.txt for writing");
        PBErCatch(PBMathErr);
    }
    if (!MatSave(mat, f)) {
        PBMathErr->_type = PBErTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "MatFloatSave NOK");
        PBErCatch(PBMathErr);
    }
    fclose(f);
    MatFloat *clone = MatFloatCreate(&dim);
    f = fopen("./UnitTestMatFloatLoadSave.txt", "r");
    if (f == NULL) {
        PBMathErr->_type = PBErTypeOther;
        sprintf(PBMathErr->_msg,
            "Can't open ./UnitTestMatFloatLoadSave.txt for reading");
        PBErCatch(PBMathErr);
    }
    if (!MatLoad(&clone, f)) {
        PBMathErr->_type = PBErTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "MatFloatLoad NOK");
        PBErCatch(PBMathErr);
    }
    if (!VecIsEqual(&(mat->_dim), &(clone->_dim))) {
        PBMathErr->_type = PBErTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestMatFloatLoadSave NOK");
        PBErCatch(PBMathErr);
    }
    VecSetNull(&i);
    do {
        if (!ISEQUALF(MatGet(mat, &i), MatGet(clone, &i))) {
            PBMathErr->_type = PBErTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestMatFloatLoadSave NOK");
            PBErCatch(PBMathErr);
        }
    }
}

```

```

    }
} while(VecStep(&i, &dim));
fclose(f);
MatFree(&mat);
MatFree(&clone);
int ret = system("cat ./UnitTestMatFloatLoadSave.txt");
ret = system("rm ./UnitTestMatFloatLoadSave.txt");
ret = ret;
printf("UnitTestMatFloatLoadSave OK\n");
}

void UnitTestMatFloatInv() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 3);
    VecSet(&dim, 1, 3);
    MatFloat *mat = MatFloatCreate(&dim);
    VecShort2D i = VecShortCreateStatic2D();
    float v[9] = {3.0, 2.0, 0.0, 0.0, 0.0, 1.0, 2.0, -2.0, 1.0};
    int j = 0;
    do {
        MatSet(mat, &i, v[j]);
        ++j;
    } while(VecStep(&i, &dim));
    MatFloat *inv = MatInv(mat);
    float w[9] = {0.2, -0.2, 0.2, 0.2, 0.3, -0.3, 0.0, 1.0, 0.0};
    VecSetNull(&i);
    j = 0;
    do {
        if (!ISEQUALF(MatGet(inv, &i), w[j])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK");
            PBErrCatch(PBMathErr);
        }
        ++j;
    } while(VecStep(&i, &dim));
    MatFree(&mat);
    MatFree(&inv);
    VecSet(&dim, 0, 2);
    VecSet(&dim, 1, 2);
    mat = MatFloatCreate(&dim);
    float vb[4] = {4.0, 2.0, 7.0, 6.0};
    VecSetNull(&i);
    j = 0;
    do {
        MatSet(mat, &i, vb[j]);
        ++j;
    } while(VecStep(&i, &dim));
    inv = MatInv(mat);
    float wb[4] = {0.6, -0.2, -0.7, 0.4};
    VecSetNull(&i);
    j = 0;
    do {
        if (!ISEQUALF(MatGet(inv, &i), wb[j])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK");
            PBErrCatch(PBMathErr);
        }
        ++j;
    } while(VecStep(&i, &dim));
    MatFree(&mat);
    MatFree(&inv);
    printf("UnitTestMatFloatInv OK\n");
}

```

```

}

void UnitTestMatFloatProdVecFloat() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2);
    VecSet(&dim, 1, 3);
    MatFloat *mat = MatFloatCreate(&dim);
    VecShort2D i = VecShortCreateStatic2D();
    float v = 1.0;
    do {
        MatSet(mat, &i, v);
        v += 1.0;
    } while(VecStep(&i, &dim));
    VecFloat2D u = VecFloatCreateStatic2D();
    for (int j = 2; j--;)
        VecSet(&u, j, (float)j + 1.0);
    VecFloat *w = MatProdVec(mat, &u);
    float b[3] = {9.0, 12.0, 15.0};
    for (int j = 3; j--;) {
        if (!ISEQUALF(VecGet(w, j), b[j])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestMatFloatProdVecFloat NOK");
            PBErrCatch(PBMathErr);
        }
    }
    MatFree(&mat);
    VecFree(&w);
    printf("UnitTestMatFloatProdVecFloat OK\n");
}

void UnitTestMatFloatProdMatFloat() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 3);
    VecSet(&dim, 1, 2);
    MatFloat *mat = MatFloatCreate(&dim);
    VecShort2D i = VecShortCreateStatic2D();
    float v = 1.0;
    do {
        MatSet(mat, &i, v);
        v += 1.0;
    } while(VecStep(&i, &dim));
    VecSet(&dim, 0, 2);
    VecSet(&dim, 1, 3);
    MatFloat *matb = MatFloatCreate(&dim);
    VecSetNull(&i);
    v = 1.0;
    do {
        MatSet(matb, &i, v);
        v += 1.0;
    } while(VecStep(&i, &dim));
    MatFloat *matc = MatProdMat(mat, matb);
    float w[4] = {22.0, 28.0, 49.0, 64.0};
    VecSetNull(&i);
    int j = 0;
    VecSet(&dim, 0, 2);
    VecSet(&dim, 1, 2);
    if (!VecIsEqual(&dim, &(matc->_dim))) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestMatFloatProdMatFloat NOK");
        PBErrCatch(PBMathErr);
    }
    do {

```

```

        if (!ISEQUALF(MatGet(matc, &i), w[j])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestMatFloatProdMatFloat NOK");
            PBErrCatch(PBMathErr);
        }
        ++j;
    } while(VecStep(&i, &dim));
    MatFree(&mat);
    MatFree(&matb);
    MatFree(&matc);
    printf("UnitTestMatFloatProdMatFloat OK\n");
}

void UnitTestSpeedMatFloat() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 3);
    VecSet(&dim, 1, 3);
    MatFloat *mat = MatFloatCreate(&dim);
    int nbTest = 100000;
    srandom(RANDOMSEED);
    int i = nbTest;
    clock_t clockBefore = clock();
    VecShort2D j = VecShortCreateStatic2D();
    for (; i--;) {
        float val = 1.0;
        MatSet(mat, &j, val);
        float valb = MatGet(mat, &j);
        valb = valb;
        VecStep(&j, &dim);
    }
    clock_t clockAfter = clock();
    double timeV = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    srandom(RANDOMSEED);
    i = nbTest;
    clockBefore = clock();
    float* array = malloc(sizeof(float) * 9);
    short *ptr = j._val;
    for (; i--;) {
        float val = 1.0;
        int k = ptr[1] * 3 + ptr[0];
        array[k] = val;
        float valb = array[k];
        valb = valb;
        VecStep(&j, &dim);
    }
    clockAfter = clock();
    double timeRef = ((double)(clockAfter - clockBefore)) /
        CLOCKS_PER_SEC * 1000.0;
    printf("MatFloat: %fms, array: %fms\n",
        timeV / (float)nbTest, timeRef / (float)nbTest);
    if (timeV / (float)nbTest > 2.0 * timeRef / (float)nbTest) {
#ifdef BUILDMODE == 0
        PBMathErr->_fatal = false;
#endif
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestSpeedMatFloat NOK");
        PBErrCatch(PBMathErr);
    }
    MatFree(&mat);
    free(array);
    printf("UnitTestSpeedMatFloat OK\n");
}

```



```

}

void UnitTestMatFloat() {
    UnitTestMatFloatCreateFree();
    UnitTestMatFloatGetSetDim();
    UnitTestMatFloatCloneIsEqual();
    UnitTestMatFloatLoadSave();
    UnitTestMatFloatInv();
    UnitTestMatFloatProdVecFloat();
    UnitTestMatFloatProdMatFloat();
    UnitTestSpeedMatFloat();
    printf("UnitTestMatFloat OK\n");
}

void UnitTestSysLinEq() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 3);
    VecSet(&dim, 1, 3);
    MatFloat *mat = MatFloatCreate(&dim);
    float a[9] = {2.0, 2.0, 6.0, 1.0, 6.0, 8.0, 3.0, 8.0, 18.0};
    VecShort2D index = VecShortCreateStatic2D();
    int j = 0;
    do {
        MatSet(mat, &index, a[j]);
        ++j;
    } while(VecStep(&index, &dim));
    VecFloat3D v = VecFloatCreateStatic3D();
    float b[3] = {1.0, 3.0, 5.0};
    for (int i = 3; i--;)
        VecSet(&v, i, b[i]);
    SysLinEq *sys = SysLinEqCreate(mat, &v);
    VecFloat *res = SysLinEqSolve(sys);
    float c[3] = {0.3, 0.4, 0};
    for (int i = 3; i--;) {
        if (!ISEQUALF(c[i], VecGet(res, i))) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "SysLinEqSolve NOK");
            PBErrCatch(PBMathErr);
        }
    }
    float ab[9] = {3.0, 2.0, -1.0, 2.0, -2.0, 0.5, -1.0, 4.0, -1.0};
    VecSetNull(&index);
    j = 0;
    do {
        MatSet(mat, &index, ab[j]);
        ++j;
    } while(VecStep(&index, &dim));
    SysLinEqSetM(sys, mat);
    float bb[3] = {1.0, -2.0, 0.0};
    for (int i = 3; i--;)
        VecSet(&v, i, bb[i]);
    SysLinEqSetV(sys, &v);
    VecFree(&res);
    res = SysLinEqSolve(sys);
    float cb[3] = {1.0, -2.0, -2.0};
    for (int i = 3; i--;) {
        if (!ISEQUALF(cb[i], VecGet(res, i))) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "SysLinEqSolve NOK");
            PBErrCatch(PBMathErr);
        }
    }
}

```

```

VecFree(&res);
SysLinEqFree(&sys);
if (sys != NULL) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "sys is not null after free");
    PBErrCatch(PBMathErr);
}
MatFree(&mat);
printf("UnitTestSysLinEq OK\n");
}

void UnitTestGauss() {
    srandom(RANDOMSEED);
    float mean = 1.0;
    float sigma = 0.5;
    Gauss *gauss = GaussCreate(mean, sigma);
    if (!ISEQUALF(gauss->_mean, mean) ||
        !ISEQUALF(gauss->_sigma, sigma)) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestGaussCreate NOK");
        PBErrCatch(PBMathErr);
    }
    float a[22] = {0.000268, 0.001224, 0.004768, 0.015831, 0.044789,
        0.107982, 0.221842, 0.388372, 0.579383, 0.736540, 0.797885,
        0.736540, 0.579383, 0.388372, 0.221842, 0.107982, 0.044789,
        0.015831, 0.004768, 0.001224, 0.000268};
    for (int i = -5; i <= 15; ++i) {
        if (!ISEQUALF(GaussGet(gauss, (float)i * 0.2), a[i + 5])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestGaussGet NOK");
            PBErrCatch(PBMathErr);
        }
    }
    int nbsample = 1000000;
    double sum = 0.0;
    double sumsquare = 0.0;
    for (int i = nbsample; i--;) {
        float v = GaussRnd(gauss);
        sum += v;
        sumsquare += fsquare(v);
    }
    double avg = sum / (double)nbsample;
    double sig = sqrtf(sumsquare / (double)nbsample - fsquare(avg));
    if (fabs(avg - mean) > 0.001 ||
        fabs(sig - sigma) > 0.001) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestGaussRnd NOK");
        PBErrCatch(PBMathErr);
    }
    GaussFree(&gauss);
    if (gauss != NULL) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "gauss is not null after free");
        PBErrCatch(PBMathErr);
    }
    printf("UnitTestGauss OK\n");
}

void UnitTestSmoother() {
    float smooth[11] = {0.0, 0.028, 0.104, 0.216, 0.352, 0.5, 0.648,
        0.784, 0.896, 0.972, 1.0};
    float smoother[11] = {0.0, 0.00856, 0.05792, 0.16308, 0.31744, 0.5,

```

```

    0.68256, 0.83692, 0.94208, 0.99144, 1.0};
for (int i = 0; i <= 10; ++i) {
    if (!ISEQUALF(SmoothStep((float)i * 0.1), smooth[i])) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestSmooth NOK");
        PBErrCatch(PBMathErr);
    }
    if (!ISEQUALF(SmootherStep((float)i * 0.1), smoother[i])) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "UnitTestSmoother NOK");
        PBErrCatch(PBMathErr);
    }
}
printf("UnitTestSmoother OK\n");
}

void UnitTestConv() {
    float rad[5] = {0.0, PBMath_TWOPI, PBMath_PI, PBMath_HALFPI, 3.0 * PBMath_HALFPI};
    float deg[5] = {0.0, 360.0, 180.0, 90.0, 270.0};
    for (int i = 5; i--;) {
        if (!ISEQUALF(ConvRad2Deg(rad[i]), deg[i])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestConvRad2Deg NOK");
            PBErrCatch(PBMathErr);
        }
        if (!ISEQUALF(ConvDeg2Rad(deg[i]), rad[i])) {
            PBMathErr->_type = PBErrTypeUnitTestFailed;
            sprintf(PBMathErr->_msg, "UnitTestConvDeg2Rad NOK");
            PBErrCatch(PBMathErr);
        }
    }
    printf("UnitTestConv OK\n");
}

void UnitTestBasicFunctions() {
    UnitTestConv();
    UnitTestPowi();
    UnitTestFastPow();
    UnitTestSpeedFastPow();
    UnitTestFSquare();
    UnitTestConv();
    printf("UnitTestBasicFunctions OK\n");
}

void UnitTestAll() {
    UnitTestVecShort();
    UnitTestVecFloat();
    UnitTestMatFloat();
    UnitTestSysLinEq();
    UnitTestGauss();
    UnitTestSmoother();
    UnitTestBasicFunctions();
    printf("UnitTestAll OK\n");
}

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

```

## 6 Unit tests output

```
5 1 2 3 4 5
2 1 2
3 1 2 3
4 1 2 3 4
5 1.000000 2.000000 3.000000 4.000000 5.000000
2 1.000000 2.000000
3 1.000000 2.000000 3.000000
2 3
1.000000 4.000000
2.000000 5.000000
3.000000 6.000000
<0,0,0,0,0>
<0,0>
<0,0,0>
<0,0,0,0>
VecShortCreateFree OK
VecShortClone OK
VecShortLoadSave OK
VecShortGetSetDim OK
UnitTestVecShortStep OK
UnitTestVecShortHamiltonDist OK
UnitTestVecShortIsEqual OK
UnitTestVecShortDotProd OK
UnitTestVecShortCopy OK
VecShort: 0.000063ms, array: 0.000033ms
VecShort2D: 0.000013ms, array: 0.000014ms
VecShort3D: 0.000014ms, array: 0.000014ms
VecShort4D: 0.000014ms, array: 0.000014ms
UnitTestSpeedVecShort OK
<1.000,2.000,3.000,4.000,5.000>
<1.000,2.000>
<1.000,2.000,3.000>
UnitTestVecShortToFloat OK
UnitTestVecShort OK
<0.000,0.000,0.000,0.000,0.000>
<0.000,0.000>
<0.000,0.000,0.000>
VecFloatCreateFree OK
VecFloatClone OK
VecFloatLoadSave OK
VecFloatGetSetDim OK
UnitTestVecFloatCopy OK
UnitTestVecFloatNorm OK
UnitTestVecFloatDist OK
UnitTestVecFloatIsEqual OK
UnitTestVecFloatScale OK
UnitTestVecFloatOp OK
UnitTestVecFloatDotProd OK
<0.707,0.707>
UnitTestVecFloatAngleTo OK
<1,2,3,4,5>
<1,2>
<1,2,3>
UnitTestVecFloatToShort OK
VecFloat: 0.000029ms, array: 0.000028ms
VecFloat2D: 0.000013ms, array: 0.000013ms
VecFloat3D: 0.000013ms, array: 0.000013ms
UnitTestSpeedVecFloat OK
UnitTestVecFloat OK
```

```

UnitTestMatFloatCreateFree OK
UnitTestMatFloatGetSetDim OK
UnitTestMatFloatCloneIsEqual OK
UnitTestMatFloatLoadSave OK
UnitTestMatFloatInv OK
UnitTestMatFloatProdVecFloat OK
UnitTestMatFloatProdMatFloat OK
MatFloat: 0.000007ms, array: 0.000007ms
UnitTestSpeedMatFloat OK
UnitTestMatFloat OK
UnitTestSysLinEq OK
UnitTestGauss OK
UnitTestSmoother OK
UnitTestConv OK
powi OK
average error: 0.000000 < 0.000010, max error: 0.000000 < 0.000100
fastpow OK
fastpow: 0.000026ms, pow: 0.000076ms
speed fastpow OK
fsquare OK
UnitTestConv OK
UnitTestBasicFunctions OK
UnitTestAll OK

```

## 7 Examples

vecshort.txt:

```
3 0 1 0
```

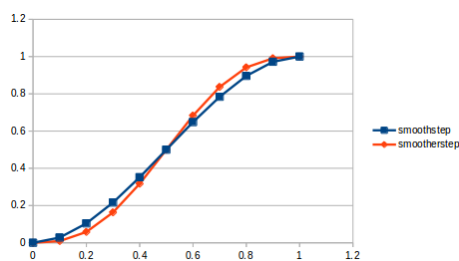
vecfloat.txt:

```
3 0.000000 1.000000 0.000000
```

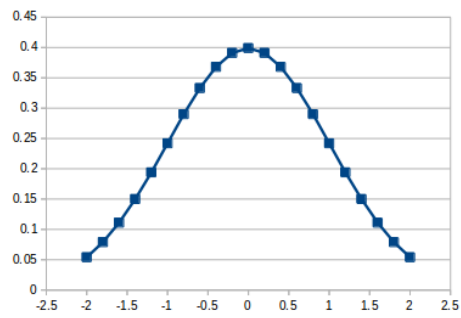
matfloat.txt:

```
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):

