PBMath

P. Baillehache

January 3, 2021

Contents

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.

The `Ratio` structure and its functions can be used to manipulate rationals.

It uses the `PBErr` library.

1 Definitions

1.1 Vector

1.1.1 Distance between two vectors

For `VecShort`:

$$\begin{array}{l} Dist(\overrightarrow{v}, \overrightarrow{w}) = \sum_i |v_i - w_i| \\ HamiltonDist(\overrightarrow{v}, \overrightarrow{w}) = \sum_i |v_i - w_i| \\ PixelDist(\overrightarrow{v}, \overrightarrow{w}) = \sum_i |v_i - w_i| \end{array} \quad (1)$$

For `VecFloat`:

$$\begin{array}{l} Dist(\overrightarrow{v}, \overrightarrow{w}) = \sum_i (v_i - w_i)^2 \\ HamiltonDist(\overrightarrow{v}, \overrightarrow{w}) = \sum_i |v_i - w_i| \\ PixelDist(\overrightarrow{v}, \overrightarrow{w}) = \sum_i |\lfloor v_i \rfloor - \lfloor w_i \rfloor| \end{array} \quad (2)$$

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 θ from $\vec{V}$ to $\vec{W}$.

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

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

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

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

Equivalent to

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

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

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

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

$$\begin{cases} Ma = \frac{W_y + M_b V_x}{V_y} \\ Mb = \frac{W_x - W_y V_x / V_y}{V_y + V_x^2 / V_y} \end{cases} \tag{7}$$

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 $\theta = cos^{-1}(M_a)$.

1.1.3 Rotation

Rotation if a vector is only defined in 2D and 3D. In 2D, for a right-handed rotation of angle θ the rotation matrix is equal to:

$$R = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{8}$$

In 3D, for a right-handed rotation of angle θ around axis $\overrightarrow{u}$ the rotation is equal to (to shorten notation θ is not written in the matrix below):

$$R = \begin{bmatrix} cos + u_x^2(1-cos) & u_x u_y(1-cos) - u_z sin & u_x u_z(1-cos) + u_y sin \\ u_x u_y(1-cos) + u_z sin & cos + u_y^2(1-cos) & u_y u_z(1-cos) - u_x sin \\ u_x u_z(1-cos) - u_y sin & u_y u_z(1-cos) + u_x sin & cos + u_z^2(1-cos) \end{bmatrix} \tag{9}$$

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:

$$M^{-1} = \frac{1}{det} \begin{bmatrix} M_3 & -M_2 \\ -M_1 & M_0 \end{bmatrix} \tag{10}$$

where

$$M = \begin{bmatrix} M_0 & M_2 \\ M_1 & M_3 \end{bmatrix} \tag{11}$$

and

$$det = M_0 M_3 - M_1 M_2 \tag{12}$$

For a 3x3 matrix M:

$$M^{-1} = \frac{1}{det} \begin{bmatrix} (M_4 M_8 - M_5 M_7) & -(M_3 M_8 - M_5 M_6) & (M_3 M_7 - M_4 M_6) \\ -(M_1 M_8 - M_2 M_7) & (M_0 M_8 - M_2 M_6) & -(M_0 M_7 - M_1 M_6) \\ (M_1 M_5 - M_2 M_4) & -(M_0 M_5 - M_2 M_3) & (M_0 M_4 - M_1 M_3) \end{bmatrix} \tag{13}$$

where

$$M = \begin{bmatrix} M_0 & M_3 & M_6 \\ M_1 & M_4 & M_7 \\ M_2 & M_5 & M_8 \end{bmatrix} \tag{14}$$

and

$$\begin{aligned} det = \; & M_0(M_4 M_8 - M_5 M_7) - \\ & M_3(M_1 M_8 - M_2 M_7) + \\ & M_6(M_1 M_5 - M_2 M_4) \end{aligned} \tag{15}$$

1.2.2 QR factorization

The QR factorization is performed using the Householder algorithm:

```
Compute Q and R such as A = QR
A[i][j] <=> value of matrix A at the i-th column and j-th row
Must have nbCol(A) <= nbRow(A)

QQtilde := matrix identity of dimensions (nbRow(A) columns, nbRow(A) rows)
for k := 0 to nbCol(A):

  w := vector null of dimension (nbRow(A) - k)
  for i := 0 to dim(w)
    w[i] := A[k][k + i]

  sign := if w[0] >= 0.0 then 1.0 else -1.0
  w[0] := w[0] + sign * norm(w)

  v = w / nom(w)

  H := matrix identity of dimensions (dim(v) columns, dim(v) rows)
  vvt := v * transpose(v)
  H := H - 2.0 * vvt

  reflector := matrix identity of dimensions (nbRow(A) columns, nbRow(A) rows)
  for i := 0 to nbCol(H)
    for j := 0 to nbRow(H)
      reflector[i + k][j + k] := H[i][j]

  A := reflector * A

  QQtilde := QQtilde * reflector

R := matrix of dimensions (nbCol(A) columns, nbCol(A) rows)
for i := 0 to nbCol(A)
  for j := 0 to nbCol(A)
    R[i][j] := A[i][j]

Q := matrix of dimensions (nbCol(A) columns, nbRow(A) rows)
for i := 0 to nbCol(A)
  for j := 0 to nbRow(A)
    Q[i][j] := QQtilde[i][j]
```

Example of execution step by step:

```
A:
[-1.000, -1.000,  1.000
  1.000,  3.000,  3.000
 -1.000, -1.000,  5.000
  1.000,  3.000,  7.000]

k: 0
----
w: <-3.000,1.000,-1.000,1.000>
v: <-0.866,0.289,-0.289,0.289>
vvt:
[ 0.750, -0.250,  0.250, -0.250
```

```
 -0.250,  0.083, -0.083,  0.083
  0.250, -0.083,  0.083, -0.083
 -0.250,  0.083, -0.083,  0.083]
H:
[-0.500,  0.500, -0.500,  0.500
  0.500,  0.833,  0.167, -0.167
 -0.500,  0.167,  0.833,  0.167
  0.500, -0.167,  0.167,  0.833]
reflector:
[-0.500,  0.500, -0.500,  0.500
  0.500,  0.833,  0.167, -0.167
 -0.500,  0.167,  0.833,  0.167
  0.500, -0.167,  0.167,  0.833]
reflector.A:
[ 2.000,  4.000,  2.000
  0.000,  1.333,  2.667
 -0.000,  0.667,  5.333
  0.000,  1.333,  6.667]
QQtilde:
[-0.500,  0.500, -0.500,  0.500
  0.500,  0.833,  0.167, -0.167
 -0.500,  0.167,  0.833,  0.167
  0.500, -0.167,  0.167,  0.833]

k: 1
----
w: <3.333,0.667,1.333>
v: <0.913,0.183,0.365>
vvt:
[ 0.833,  0.167,  0.333
  0.167,  0.033,  0.067
  0.333,  0.067,  0.133]
H:
[-0.667, -0.333, -0.667
 -0.333,  0.933, -0.133
 -0.667, -0.133,  0.733]
reflector:
[ 1.000,  0.000,  0.000,  0.000
  0.000, -0.667, -0.333, -0.667
  0.000, -0.333,  0.933, -0.133
  0.000, -0.667, -0.133,  0.733]
reflector.A:
[ 2.000,  4.000,  2.000
 -0.000, -2.000, -8.000
 -0.000,  0.000,  3.200
  0.000, -0.000,  2.400]
QQtilde:
[-0.500, -0.500, -0.700,  0.100
  0.500, -0.500, -0.100, -0.700
 -0.500, -0.500,  0.700, -0.100
  0.500, -0.500,  0.100,  0.700]

k: 2
----
w: <7.200,2.400>
v: <0.949,0.316>
vvt:
[ 0.900,  0.300
  0.300,  0.100]
H:
[-0.800, -0.600
 -0.600,  0.800]
```

```
reflector:
[ 1.000,  0.000,  0.000,  0.000
  0.000,  1.000,  0.000,  0.000
  0.000,  0.000, -0.800, -0.600
  0.000,  0.000, -0.600,  0.800]
reflector.A:
[ 2.000,  4.000,  2.000
 -0.000, -2.000, -8.000
  0.000, -0.000, -4.000
  0.000, -0.000, -0.000]
QQtilde:
[-0.500, -0.500,  0.500,  0.500
  0.500, -0.500,  0.500, -0.500
 -0.500, -0.500, -0.500, -0.500
  0.500, -0.500, -0.500,  0.500]

Q:
[-0.500, -0.500,  0.500
  0.500, -0.500,  0.500
 -0.500, -0.500, -0.500
  0.500, -0.500, -0.500]
R:
[ 2.000,  4.000,  2.000
 -0.000, -2.000, -8.000
  0.000, -0.000, -4.000]
QR:
[-1.000, -1.000,  1.000
  1.000,  3.000,  3.000
 -1.000, -1.000,  5.000
  1.000,  3.000,  7.000]
```

1.2.3 Eigen values and vectors

The Eigen values and vectors are obtained using the QR algorithm:

```
Compute Eigen values and vectors of A
A[i][j] <=> value of matrix A at the i-th column and j-th row
Must have nbCol(A) = nbRow(A)

err := 1.0
M := matrix identity of dimensions(nbCol(A) columns, nbRow(A) rows)
loop until err < epsilon
  Q,R := QRDecomposition(A)
  A := R * Q
  M := M * Q
  err := max(abs(non diagonal values of A))
for i := 0 to NbCol(A)
  eigenValue[i] := A[i][i]
  for j := 0 to nbRow(A)
    eigenVector_i[j] := M[i][j]
```

Example of execution step by step:

```
A:
[ 2.920000,  0.860000, -1.150000
  0.860000,  6.510000,  3.320000
```

```
 -1.150000,  3.320000,  4.570000]
k: 0
----
A:
[ 3.893980,  0.051622, -0.542073
  0.051622,  8.916388,  0.802225
 -0.542073,  0.802225,  1.189634]
M:
[-0.897358,  0.044696, -0.439034
 -0.264290, -0.851147,  0.453541
  0.353411, -0.523021, -0.775596]
k: 1
----
A:
[ 3.989910, -0.007497, -0.141512
 -0.007497,  8.997707,  0.092890
 -0.141512,  0.092890,  1.012387]
M:
[ 0.827596, -0.005324,  0.561300
  0.335447,  0.806451, -0.486943
 -0.450068,  0.591278,  0.669201]
k: 2
----
A:
[ 3.996253, -0.020544, -0.035648
 -0.020544,  8.998704,  0.010273
 -0.035648,  0.010273,  1.005045]
M:
[-0.807189, -0.002264, -0.590289
 -0.350981, -0.802184,  0.483025
  0.474614, -0.597072, -0.646720]
k: 3
----
A:
[ 3.996995, -0.046356, -0.008962
 -0.046356,  8.998370,  0.001080
 -0.008962,  0.001080,  1.004634]
M:
[ 0.801869,  0.007056,  0.597458
  0.351147,  0.803462, -0.480776
 -0.483427,  0.595315,  0.641793]
k: 4
----
A:
[ 3.998767, -0.104343, -0.002252
 -0.104343,  8.996623,  0.000083
 -0.002252,  0.000083,  1.004609]
M:
[-0.800392, -0.016405, -0.599253
 -0.342883, -0.807440,  0.480075
  0.491736, -0.589721, -0.640644]
k: 5
----
A:
[ 4.007611, -0.234525, -0.000566
 -0.234525,  8.987781, -0.000012
 -0.000566, -0.000012,  1.004607]
M:
[ 0.799354,  0.037270,  0.599703
  0.321974,  0.816115, -0.479884
 -0.507312,  0.576686,  0.640365]
k: 6
```

```
----
A:
[ 4.051963, -0.523369, -0.000142
 -0.523369,  8.943428, -0.000013
 -0.000142, -0.000013,  1.004607]
M:
[-0.795727, -0.083897, -0.599817
 -0.273814, -0.833538,  0.479834
  0.540226, -0.546055, -0.640296]
k: 7
----
A:
[ 4.265212, -1.127628, -0.000035
 -1.127628,  8.730177, -0.000008
 -0.000035, -0.000008,  1.004607]
M:
[ 0.778403,  0.185133,  0.599845
  0.164799,  0.861750, -0.479822
 -0.605747,  0.472348,  0.640279]
k: 8
----
A:
[ 5.114194, -2.083614, -0.000008
 -2.083614,  7.881196, -0.000004
 -0.000008, -0.000004,  1.004607]
M:
[-0.705223, -0.377938, -0.599852
  0.060932, -0.875249,  0.479818
  0.706362, -0.301829, -0.640275]
k: 9
----
A:
[ 6.964207, -2.457212, -0.000001
 -2.457211,  6.031182, -0.000002
 -0.000001, -0.000002,  1.004607]
M:
[ 0.510500,  0.616088,  0.599854
 -0.386664,  0.787570, -0.479818
 -0.768037,  0.013005,  0.640274]
k: 10
----
A:
[ 8.402923, -1.620381, -0.000000
 -1.620380,  4.592468, -0.000001
 -0.000000, -0.000001,  1.004607]
M:
[-0.276422, -0.750843, -0.599854
  0.626681, -0.614041,  0.479817
  0.728603,  0.243285, -0.640273]
k: 11
----
A:
[ 8.868839, -0.795746,  0.000000
 -0.795745,  4.126554, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[ 0.129251,  0.789600,  0.599855
 -0.731611,  0.484273, -0.479817
 -0.669357, -0.376843,  0.640273]
k: 12
----
A:
```

```
[ 8.972622, -0.360938,  0.000000
 -0.360938,  4.022771, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[-0.058172, -0.797991, -0.599855
  0.771961, -0.416956,  0.479817
  0.633003,  0.435152, -0.640273]
k: 13
----
A:
[ 8.993616, -0.160978,  0.000000
 -0.160978,  4.001777, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[ 0.026050,  0.799685,  0.599855
 -0.788096,  0.385590, -0.479817
 -0.615001, -0.460244,  0.640273]
k: 14
----
A:
[ 8.997777, -0.071554,  0.000000
 -0.071554,  3.997614, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[-0.011735, -0.800023, -0.599855
  0.794871, -0.371425,  0.479817
  0.606666,  0.471176, -0.640273]
k: 15
----
A:
[ 8.998597, -0.031784,  0.000000
 -0.031784,  3.996793, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[ 0.005373,  0.800091,  0.599855
 -0.797799,  0.365092, -0.479817
 -0.602899, -0.475986,  0.640273]
k: 16
----
A:
[ 8.998760, -0.014117,  0.000000
 -0.014117,  3.996630, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[-0.002547, -0.800105, -0.599855
  0.799084, -0.362272,  0.479817
  0.601215,  0.478112, -0.640273]
k: 17
----
A:
[ 8.998793, -0.006270,  0.000000
 -0.006270,  3.996598, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[ 0.001291,  0.800108,  0.599855
 -0.799651,  0.361018, -0.479817
 -0.600464, -0.479055,  0.640273]
k: 18
----
A:
[ 8.998798, -0.002785,  0.000000
 -0.002784,  3.996592, -0.000000
```

```
 -0.000000, -0.000000,  1.004607]
M:
[-0.000734, -0.800108, -0.599855
  0.799902, -0.360460,  0.479817
  0.600130,  0.479473, -0.640273]
k: 19
----
A:
[ 8.998800, -0.001237,  0.000000
 -0.001237,  3.996591, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[ 0.000486,  0.800109,  0.599855
 -0.800014,  0.360213, -0.479817
 -0.599982, -0.479659,  0.640273]
k: 20
----
A:
[ 8.998800, -0.000550,  0.000000
 -0.000549,  3.996590, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[-0.000376, -0.800109, -0.599855
  0.800063, -0.360103,  0.479817
  0.599916,  0.479741, -0.640273]
k: 21
----
A:
[ 8.998800, -0.000244,  0.000000
 -0.000244,  3.996590, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[ 0.000328,  0.800109,  0.599855
 -0.800085,  0.360054, -0.479817
 -0.599886, -0.479778,  0.640273]
k: 22
----
A:
[ 8.998800, -0.000109,  0.000000
 -0.000108,  3.996590, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[-0.000306, -0.800109, -0.599855
  0.800095, -0.360032,  0.479817
  0.599873,  0.479794, -0.640273]
k: 23
----
A:
[ 8.998800, -0.000048,  0.000000
 -0.000048,  3.996590, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[ 0.000296,  0.800109,  0.599855
 -0.800099,  0.360023, -0.479817
 -0.599868, -0.479801,  0.640273]
k: 24
----
A:
[ 8.998800, -0.000022,  0.000000
 -0.000021,  3.996590, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
```

```
[-0.000292, -0.800109, -0.599855
  0.800101, -0.360019,  0.479817
  0.599865,  0.479804, -0.640273]
k: 25
----
A:
[ 8.998800, -0.000010,  0.000000
 -0.000009,  3.996590, -0.000000
 -0.000000, -0.000000,  1.004607]
M:
[ 0.000290,  0.800109,  0.599855
 -0.800102,  0.360017, -0.479817
 -0.599864, -0.479806,  0.640273]
Eigen values: <8.999,3.997,1.005>
Eigen vector 1: <0.000,-0.800,-0.600>
Eigen vector 2: <0.800,0.360,-0.480>
Eigen vector 3: <0.600,-0.480,0.640>
```

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 <stddef.h>
#include <float.h>
#include <stdint.h>
#include "pberr.h"
#include "pbjson.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 PBMATH_SQRTONEHALF 0.707106781
#ifndef MAX
  #define MAX(a,b) ((a)>(b)?(a):(b))
#endif
#ifndef MIN
  #define MIN(a,b) ((a)<(b)?(a):(b))
#endif
#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)
```

```
// -------------- VecLong

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

// Vector of long values
typedef struct VecLong {
  // Dimension
  long _dim;
  // Values
  long _val[0];
} VecLong;

typedef struct VecLong2D {
  // Dimension
  long _dim;
  // Values
  long _val[2];
} VecLong2D;

typedef struct VecLong3D {
  // Dimension
  long _dim;
  // Values
  long _val[3];
} VecLong3D;

typedef struct VecLong4D {
  // Dimension
  long _dim;
  // Values
  long _val[4];
} VecLong4D;

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

// Create a new VecLong of dimension 'dim'
// Values are initalized to 0.0
VecLong* VecLongCreate(const long dim);

// Static constructors for VecLong
#if BUILDMODE != 0
static inline
#endif
VecLong2D VecLongCreateStatic2D();
#if BUILDMODE != 0
static inline
#endif
VecLong3D VecLongCreateStatic3D();
#if BUILDMODE != 0
static inline
#endif
VecLong4D VecLongCreateStatic4D();

// Clone the VecLong
// Return NULL if we couldn't clone the VecLong
VecLong* _VecLongClone(const VecLong* const that);

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

// Function which decode from JSON encoding 'json' to 'that'
bool _VecLongDecodeAsJSON(VecLong** that, const JSONNode* const json);
```

```
// Load the VecLong from the stream
// If the VecLong is already allocated, it is freed before loading
// Return true in case of success, else false
bool _VecLongLoad(VecLong** that, FILE* const stream);

// Save the VecLong to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true in case of success, else false
bool _VecLongSave(const VecLong* const that,
  FILE* const stream, const bool compact);

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

// Print the VecLong on 'stream'
void _VecLongPrint(const VecLong* const that,
  FILE* const stream);

// Return the i-th value of the VecLong
#if BUILDMODE != 0
static inline
#endif
long _VecLongGet(const VecLong* const that, const long i);
#if BUILDMODE != 0
static inline
#endif
long _VecLongGet2D(const VecLong2D* const that, const long i);
#if BUILDMODE != 0
static inline
#endif
long _VecLongGet3D(const VecLong3D* const that, const long i);
#if BUILDMODE != 0
static inline
#endif
long _VecLongGet4D(const VecLong4D* const that, const long i);

// Set the i-th value of the VecLong to v
#if BUILDMODE != 0
static inline
#endif
void _VecLongSet(VecLong* const that, const long i, const long v);
#if BUILDMODE != 0
static inline
#endif
void _VecLongSet2D(VecLong2D* const that, const long i, const long v);
#if BUILDMODE != 0
static inline
#endif
void _VecLongSet3D(VecLong3D* const that, const long i, const long v);
#if BUILDMODE != 0
static inline
#endif
void _VecLongSet4D(VecLong4D* const that, const long i, const long v);

// Set the i-th value of the VecLong to v plus its current value
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetAdd(VecLong* const that, const long i, const long v);
```

```
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetAdd2D(VecLong2D* const that, const long i, const long v);
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetAdd3D(VecLong3D* const that, const long i, const long v);
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetAdd4D(VecLong4D* const that, const long i, const long v);

// Return the dimension of the VecLong
// Return 0 if arguments are invalid
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetDim(const VecLong* const that);

// Return the Hamiltonian distance between the VecLong 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
long _VecLongHamiltonDist(const VecLong* const that, const VecLong* const tho);
#if BUILDMODE != 0
static inline
#endif
long _VecLongHamiltonDist2D(const VecLong2D* const that, const VecLong2D* const tho);
#if BUILDMODE != 0
static inline
#endif
long _VecLongHamiltonDist3D(const VecLong3D* const that, const VecLong3D* const tho);
#if BUILDMODE != 0
static inline
#endif
long _VecLongHamiltonDist4D(const VecLong4D* const that, const VecLong4D* const tho);

// Return true if the VecLong 'that' is equal to 'tho', else false
#if BUILDMODE != 0
static inline
#endif
bool _VecLongIsEqual(const VecLong* const that,
  const VecLong* const tho);

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

// Return the dot product of 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
long _VecLongDotProd(const VecLong* const that,
  const VecLong* const tho);
#if BUILDMODE != 0
static inline
#endif
long _VecLongDotProd2D(const VecLong2D* const that,
  const VecLong2D* const tho);
```

```
#if BUILDMODE != 0
static inline
#endif
long _VecLongDotProd3D(const VecLong3D* const that,
  const VecLong3D* const tho);
#if BUILDMODE != 0
static inline
#endif
long _VecLongDotProd4D(const VecLong4D* const that,
  const VecLong4D* const tho);

// Set all values of the vector 'that' to 0
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetNull(VecLong* const that);

// Set all values of the vector 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetAll(VecLong* const that, long v);

// 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 _VecLongStep(VecLong* const that, const VecLong* const 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 _VecLongPStep(VecLong* const that, const VecLong* const bound);

// Step the values of the vector incrementally by 1
// in the following order (for example) :
// (0,0,0)->(0,0,1)->(0,0,2)->(0,1,0)->(0,1,1)->...
// The lower limit for each value is given by 'from' (val[i] >= from[i])
// The upper limit for each value is given by 'to' (val[i] < to[i])
// 'that' must be initialised to 'from' before the first call of this
// function
// Return false if all values of 'that' have reached their upper limit
// (in which case 'that''s values are all set back to from)
// Return true else
bool _VecLongShiftStep(VecLong* const that,
  const VecLong* const from, const VecLong* const to);

// Step the values of the vector incrementally by delta 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 after 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 _VecLongStepDelta(VecLong* const that,
```

```
  const VecLong* const bound, const VecLong* const delta);

// Step the values of the vector incrementally by delta 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 after 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 _VecLongPStepDelta(VecLong* const that,
  const VecLong* const bound, const VecLong* const delta);

// 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
static inline
#endif
void _VecLongOp(VecLong* const that, const long a,
  const VecLong* const tho, const long b);
#if BUILDMODE != 0
static inline
#endif
void _VecLongOp2D(VecLong2D* const that, const long a,
  const VecLong2D* const tho, const long b);
#if BUILDMODE != 0
static inline
#endif
void _VecLongOp3D(VecLong3D* const that, const long a,
  const VecLong3D* const tho, const long b);
#if BUILDMODE != 0
static inline
#endif
void _VecLongOp4D(VecLong4D* const that, const long a,
  const VecLong4D* const tho, const long b);

// Return a VecLong 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
static inline
#endif
VecLong* _VecLongGetOp(const VecLong* const that, const long a,
  const VecLong* const tho, const long b);
#if BUILDMODE != 0
static inline
#endif
VecLong2D _VecLongGetOp2D(const VecLong2D* const that, const long a,
  const VecLong2D* const tho, const long b);
#if BUILDMODE != 0
static inline
#endif
VecLong3D _VecLongGetOp3D(const VecLong3D* const that, const long a,
  const VecLong3D* const tho, const long b);
#if BUILDMODE != 0
static inline
#endif
VecLong4D _VecLongGetOp4D(const VecLong4D* const that, const long a,
  const VecLong4D* const tho, const long b);

// Calculate the Hadamard product of that by tho and store the
```

```
// result in 'that'
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
void _VecLongHadamardProd(VecLong* const that,
  const VecLong* const tho);
#if BUILDMODE != 0
static inline
#endif
void _VecLongHadamardProd2D(VecLong2D* const that,
  const VecLong2D* const tho);
#if BUILDMODE != 0
static inline
#endif
void _VecLongHadamardProd3D(VecLong3D* const that,
  const VecLong3D* const tho);
#if BUILDMODE != 0
static inline
#endif
void _VecLongHadamardProd4D(VecLong4D* const that,
  const VecLong4D* const tho);

// Return a VecLong equal to the hadamard product of 'that' and 'tho'
// Return NULL if arguments are invalid
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
VecLong* _VecLongGetHadamardProd(const VecLong* const that,
  const VecLong* const tho);
#if BUILDMODE != 0
static inline
#endif
VecLong2D _VecLongGetHadamardProd2D(const VecLong2D* const that,
  const VecLong2D* const tho);
#if BUILDMODE != 0
static inline
#endif
VecLong3D _VecLongGetHadamardProd3D(const VecLong3D* const that,
  const VecLong3D* const tho);
#if BUILDMODE != 0
static inline
#endif
VecLong4D _VecLongGetHadamardProd4D(const VecLong4D* const that,
  const VecLong4D* const tho);

// Get the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetMaxVal(const VecLong* const that);

// Get the min value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetMinVal(const VecLong* const that);

// Get the max value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
```

```
static inline
#endif
long _VecLongGetMaxValAbs(const VecLong* const that);

// Get the min value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetMinValAbs(const VecLong* const that);

// Get the index of the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetIMaxVal(const VecLong* const that);

// Return a new VecLong as a copy of the VecLong 'that' with
// dimension changed to 'dim'
// if it is extended, the values of new components are 0
// If it is shrinked, values are discarded from the end of the vector
VecLong* _VecLongGetNewDim(const VecLong* const that, const long dim);

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

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

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

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

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

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

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

// Create a new VecShort of dimension 'dim'
// Values are initalized to 0.0
VecShort* VecShortCreate(const long dim);
```

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

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

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

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

// 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* const stream);

// Save the VecShort to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true in case of success, else false
bool _VecShortSave(const VecShort* const that,
  FILE* const stream, const bool compact);

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

// Print the VecShort on 'stream'
void _VecShortPrint(const VecShort* const that,
  FILE* const stream);

// Return the i-th value of the VecShort
#if BUILDMODE != 0
static inline
#endif
short _VecShortGet(const VecShort* const that, const long i);
#if BUILDMODE != 0
static inline
#endif
short _VecShortGet2D(const VecShort2D* const that, const long i);
#if BUILDMODE != 0
static inline
#endif
short _VecShortGet3D(const VecShort3D* const that, const long i);
#if BUILDMODE != 0
static inline
#endif
short _VecShortGet4D(const VecShort4D* const that, const long i);
```

```
// Set the i-th value of the VecShort to v
#if BUILDMODE != 0
static inline
#endif
void _VecShortSet(VecShort* const that, const long i, const short v);
#if BUILDMODE != 0
static inline
#endif
void _VecShortSet2D(VecShort2D* const that, const long i, const short v);
#if BUILDMODE != 0
static inline
#endif
void _VecShortSet3D(VecShort3D* const that, const long i, const short v);
#if BUILDMODE != 0
static inline
#endif
void _VecShortSet4D(VecShort4D* const that, const long i, const short v);

// Set the i-th value of the VecShort to v plus its current value
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetAdd(VecShort* const that, const long i, const short v);
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetAdd2D(VecShort2D* const that, const long i, const short v);
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetAdd3D(VecShort3D* const that, const long i, const short v);
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetAdd4D(VecShort4D* const that, const long i, const short v);

// Return the dimension of the VecShort
// Return 0 if arguments are invalid
#if BUILDMODE != 0
static inline
#endif
long _VecShortGetDim(const VecShort* const that);

// Return the Hamiltonian distance between the VecShort 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
short _VecShortHamiltonDist(const VecShort* const that, const VecShort* const tho);
#if BUILDMODE != 0
static inline
#endif
short _VecShortHamiltonDist2D(const VecShort2D* const that, const VecShort2D* const tho);
#if BUILDMODE != 0
static inline
#endif
short _VecShortHamiltonDist3D(const VecShort3D* const that, const VecShort3D* const tho);
#if BUILDMODE != 0
static inline
#endif
short _VecShortHamiltonDist4D(const VecShort4D* const that, const VecShort4D* const tho);

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

```
#if BUILDMODE != 0
static inline
#endif
bool _VecShortIsEqual(const VecShort* const that,
  const VecShort* const tho);

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

// Return the dot product of 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
short _VecShortDotProd(const VecShort* const that,
  const VecShort* const tho);
#if BUILDMODE != 0
static inline
#endif
short _VecShortDotProd2D(const VecShort2D* const that,
  const VecShort2D* const tho);
#if BUILDMODE != 0
static inline
#endif
short _VecShortDotProd3D(const VecShort3D* const that,
  const VecShort3D* const tho);
#if BUILDMODE != 0
static inline
#endif
short _VecShortDotProd4D(const VecShort4D* const that,
  const VecShort4D* const tho);

// Set all values of the vector 'that' to 0
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetNull(VecShort* const that);

// Set all values of the vector 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetAll(VecShort* const that, short v);

// 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* const that, const VecShort* const 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* const that, const VecShort* const bound);

// Step the values of the vector incrementally by 1
// in the following order (for example) :
// (0,0,0)->(0,0,1)->(0,0,2)->(0,1,0)->(0,1,1)->...
// The lower limit for each value is given by 'from' (val[i] >= from[i])
// The upper limit for each value is given by 'to' (val[i] < to[i])
// 'that' must be initialised to 'from' before the first call of this
// function
// Return false if all values of 'that' have reached their upper limit
// (in which case 'that''s values are all set back to from)
// Return true else
bool _VecShortShiftStep(VecShort* const that,
  const VecShort* const from, const VecShort* const to);

// Step the values of the vector incrementally by delta 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 after 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 _VecShortStepDelta(VecShort* const that,
  const VecShort* const bound, const VecShort* const delta);

// Step the values of the vector incrementally by delta 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 after 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 _VecShortPStepDelta(VecShort* const that,
  const VecShort* const bound, const VecShort* const delta);

// 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
static inline
#endif
void _VecShortOp(VecShort* const that, const short a,
  const VecShort* const tho, const short b);
#if BUILDMODE != 0
static inline
#endif
void _VecShortOp2D(VecShort2D* const that, const short a,
  const VecShort2D* const tho, const short b);
#if BUILDMODE != 0
static inline
#endif
void _VecShortOp3D(VecShort3D* const that, const short a,
  const VecShort3D* const tho, const short b);
#if BUILDMODE != 0
static inline
#endif
void _VecShortOp4D(VecShort4D* const that, const short a,
  const VecShort4D* const tho, const short b);

// Return a VecShort 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
static inline
#endif
VecShort* _VecShortGetOp(const VecShort* const that, const short a,
  const VecShort* const tho, const short b);
#if BUILDMODE != 0
static inline
#endif
VecShort2D _VecShortGetOp2D(const VecShort2D* const that, const short a,
  const VecShort2D* const tho, const short b);
#if BUILDMODE != 0
static inline
#endif
VecShort3D _VecShortGetOp3D(const VecShort3D* const that, const short a,
  const VecShort3D* const tho, const short b);
#if BUILDMODE != 0
static inline
#endif
VecShort4D _VecShortGetOp4D(const VecShort4D* const that, const short a,
  const VecShort4D* const tho, const short b);

// Calculate the Hadamard product of that by tho and store the
// result in 'that'
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
void _VecShortHadamardProd(VecShort* const that,
  const VecShort* const tho);
#if BUILDMODE != 0
static inline
#endif
void _VecShortHadamardProd2D(VecShort2D* const that,
  const VecShort2D* const tho);
#if BUILDMODE != 0
static inline
#endif
void _VecShortHadamardProd3D(VecShort3D* const that,
  const VecShort3D* const tho);
#if BUILDMODE != 0
static inline
#endif
void _VecShortHadamardProd4D(VecShort4D* const that,
  const VecShort4D* const tho);

// Return a VecShort equal to the hadamard product of 'that' and 'tho'
// Return NULL if arguments are invalid
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
VecShort* _VecShortGetHadamardProd(const VecShort* const that,
  const VecShort* const tho);
#if BUILDMODE != 0
static inline
#endif
VecShort2D _VecShortGetHadamardProd2D(const VecShort2D* const that,
  const VecShort2D* const tho);
#if BUILDMODE != 0
static inline
#endif
```

```
VecShort3D _VecShortGetHadamardProd3D(const VecShort3D* const that,
  const VecShort3D* const tho);
#if BUILDMODE != 0
static inline
#endif
VecShort4D _VecShortGetHadamardProd4D(const VecShort4D* const that,
  const VecShort4D* const tho);

// Get the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
short _VecShortGetMaxVal(const VecShort* const that);

// Get the min value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
short _VecShortGetMinVal(const VecShort* const that);

// Get the max value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
short _VecShortGetMaxValAbs(const VecShort* const that);

// Get the min value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
short _VecShortGetMinValAbs(const VecShort* const that);

// Get the index of the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecShortGetIMaxVal(const VecShort* const that);

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

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

// Vector of float values
typedef struct VecFloat {
  // Dimension
  long _dim;
  // Values
  float _val[0];
} VecFloat;

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

typedef struct VecFloat3D {
  // Dimension
  long _dim;
```

```
  // Values
  float _val[3];
} VecFloat3D;

typedef struct VecFloat4D {
  // Dimension
  long _dim;
  // Values
  float _val[4];
} VecFloat4D;

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

// Create a new VecFloat of dimension 'dim'
// Values are initalized to 0.0
VecFloat* VecFloatCreate(const long dim);

// Static constructors for VecFloat
#if BUILDMODE != 0
static inline
#endif
VecFloat2D VecFloatCreateStatic2D();
#if BUILDMODE != 0
static inline
#endif
VecFloat3D VecFloatCreateStatic3D();
#if BUILDMODE != 0
static inline
#endif
VecFloat4D VecFloatCreateStatic4D();

// Clone the VecFloat
VecFloat* _VecFloatClone(const VecFloat* const that);

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

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

// 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* const stream);

// Save the VecFloat to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true in case of success, else false
bool _VecFloatSave(const VecFloat* const that,
  FILE* const stream, const bool compact);

// 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(const VecFloat* const that, FILE* const stream,
  const unsigned int prec);
static inline void _VecFloatPrintDef(const VecFloat* const that,
  FILE* const stream) {
```

```
  VecFloatPrint(that, stream, 3);
}

// Return the 'i'-th value of the VecFloat
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGet(const VecFloat* const that, const long i);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGet2D(const VecFloat2D* const that, const long i);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGet3D(const VecFloat3D* const that, const long i);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGet4D(const VecFloat4D* const that, const long i);

// Set the 'i'-th value of the VecFloat to 'v'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSet(VecFloat* const that, const long i, const float v);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSet2D(VecFloat2D* const that, const long i, const float v);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSet3D(VecFloat3D* const that, const long i, const float v);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSet4D(VecFloat4D* const that, const long i, const float v);

// Set the 'i'-th value of the VecFloat to 'v' plus its current value
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAdd(VecFloat* const that, const long i, const float v);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAdd2D(VecFloat2D* const that, const long i,
  const float v);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAdd3D(VecFloat3D* const that, const long i,
  const float v);

// Set all values of the vector 'that' to 0
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetNull(VecFloat* const that);
#if BUILDMODE != 0
static inline
```

```
#endif
void _VecFloatSetNull2D(VecFloat2D* const that);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetNull3D(VecFloat3D* const that);

// Set all values of the vector 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAll(VecFloat* const that, float v);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAll2D(VecFloat2D* const that, float v);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAll3D(VecFloat3D* const that, float v);

// Return the dimension of the VecFloat
// Return 0 if arguments are invalid
#if BUILDMODE != 0
static inline
#endif
long _VecFloatGetDim(const VecFloat* const that);

// Return a new VecFloat as a copy of the VecFloat 'that' with
// dimension changed to 'dim'
// if it is extended, the values of new components are 0.0
// If it is shrinked, values are discarded from the end of the vector
VecFloat* _VecFloatGetNewDim(const VecFloat* const that, const long dim);

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

// Return the norm of the VecFloat
// Return 0.0 if arguments are invalid
#if BUILDMODE != 0
static inline
#endif
float _VecFloatNorm(const VecFloat* const that);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatNorm2D(const VecFloat2D* const that);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatNorm3D(const VecFloat3D* const that);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatNorm4D(const VecFloat4D* const that);

// Normalise the VecFloat
#if BUILDMODE != 0
```

```
static inline
#endif
void _VecFloatNormalise(VecFloat* const that);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatNormalise2D(VecFloat2D* const that);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatNormalise3D(VecFloat3D* const that);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatNormalise4D(VecFloat4D* const that);

// Return the distance between the VecFloat 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDist(const VecFloat* const that,
  const VecFloat* const tho);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDist2D(const VecFloat2D* const that,
  const VecFloat2D* const tho);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDist3D(const VecFloat3D* const that,
  const VecFloat3D* const tho);

// Return the Hamiltonian distance between the VecFloat 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatHamiltonDist(const VecFloat* const that,
  const VecFloat* const tho);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatHamiltonDist2D(const VecFloat2D* const that,
  const VecFloat2D* const tho);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatHamiltonDist3D(const VecFloat3D* const that,
  const VecFloat3D* const tho);

// Return the Pixel distance between the VecFloat 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatPixelDist(const VecFloat* const that,
  const VecFloat* const tho);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatPixelDist2D(const VecFloat2D* const that,
  const VecFloat2D* const tho);
#if BUILDMODE != 0
```

```
static inline
#endif
float _VecFloatPixelDist3D(const VecFloat3D* const that,
  const VecFloat3D* const tho);

// Return true if the VecFloat 'that' is equal to 'tho', else false
#if BUILDMODE != 0
static inline
#endif
bool _VecFloatIsEqual(const VecFloat* const that,
  const VecFloat* const tho);

// Calculate (that * a) and store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatScale(VecFloat* const that, const float a);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatScale2D(VecFloat2D* const that, const float a);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatScale3D(VecFloat3D* const that, const float a);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatScale4D(VecFloat4D* const that, const float a);

// Return a VecFloat equal to (that * a)
#if BUILDMODE != 0
static inline
#endif
VecFloat* _VecFloatGetScale(const VecFloat* const that, const float a);
#if BUILDMODE != 0
static inline
#endif
VecFloat2D _VecFloatGetScale2D(const VecFloat2D* const that,
  const float a);
#if BUILDMODE != 0
static inline
#endif
VecFloat3D _VecFloatGetScale3D(const VecFloat3D* const that,
  const 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
static inline
#endif
void _VecFloatOp(VecFloat* const that, const float a,
  const VecFloat* const tho, const float b);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatOp2D(VecFloat2D* const that, const float a,
  const VecFloat2D* const tho, const float b);
#if BUILDMODE != 0
static inline
#endif
```

```
void _VecFloatOp3D(VecFloat3D* const that, const float a,
  const VecFloat3D* const tho, const float b);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatOp4D(VecFloat4D* const that, const float a,
  const VecFloat4D* const tho, const 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
static inline
#endif
VecFloat* _VecFloatGetOp(const VecFloat* const that, const float a,
  const VecFloat* const tho, const float b);
#if BUILDMODE != 0
static inline
#endif
VecFloat2D _VecFloatGetOp2D(const VecFloat2D* const that, const float a,
  const VecFloat2D* const tho, const float b);
#if BUILDMODE != 0
static inline
#endif
VecFloat3D _VecFloatGetOp3D(const VecFloat3D* const that, const float a,
  const VecFloat3D* const tho, const float b);

// Calculate the Hadamard product of that by tho and store the
// result in 'that'
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
void _VecFloatHadamardProd(VecFloat* const that,
  const VecFloat* const tho);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatHadamardProd2D(VecFloat2D* const that,
  const VecFloat2D* const tho);
#if BUILDMODE != 0
static inline
#endif
void _VecFloatHadamardProd3D(VecFloat3D* const that,
  const VecFloat3D* const tho);

// Return a VecFloat equal to the hadamard product of 'that' and 'tho'
// Return NULL if arguments are invalid
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
VecFloat* _VecFloatGetHadamardProd(const VecFloat* const that,
  const VecFloat* const tho);
#if BUILDMODE != 0
static inline
#endif
VecFloat2D _VecFloatGetHadamardProd2D(const VecFloat2D* const that,
  const VecFloat2D* const tho);
#if BUILDMODE != 0
static inline
```

```
#endif
VecFloat3D _VecFloatGetHadamardProd3D(const VecFloat3D* const that,
  const VecFloat3D* const tho);

// Rotate CCW 'that' by 'theta' radians and store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatRot2D(VecFloat2D* const that, const float theta);

// Return a VecFloat2D equal to 'that' rotated CCW by 'theta' radians
#if BUILDMODE != 0
static inline
#endif
VecFloat2D _VecFloatGetRot2D(const VecFloat2D* const that,
  const float theta);

// Rotate right-hand 'that' by 'theta' radians around 'axis' and
// store the result in 'that'
// 'axis' must be normalized
// https://en.wikipedia.org/wiki/Rotation_matrix
#if BUILDMODE != 0
static inline
#endif
void _VecFloatRotAxis(VecFloat3D* const that,
  const VecFloat3D* const axis, const float theta);

// Return a VecFloat3D equal to 'that' rotated right-hand by 'theta'
// radians around 'axis'
// 'axis' must be normalized
// https://en.wikipedia.org/wiki/Rotation_matrix
VecFloat3D _VecFloatGetRotAxis(const VecFloat3D* const that,
  const VecFloat3D* const axis, const float theta);

// Rotate right-hand 'that' by 'theta' radians around X and
// store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatRotX(VecFloat3D* const that, const float theta);

// Rotate right-hand 'that' by 'theta' radians around Y and
// store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatRotY(VecFloat3D* const that, const float theta);

// Rotate right-hand 'that' by 'theta' radians around Z and
// store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatRotZ(VecFloat3D* const that, const float theta);

// Return a VecFloat3D equal to 'that' rotated right-hand by 'theta'
// radians around X
VecFloat3D _VecFloatGetRotX(const VecFloat3D* const that,
  const float theta);

// Return a VecFloat3D equal to 'that' rotated right-hand by 'theta'
// radians around Y
```

```
VecFloat3D _VecFloatGetRotY(const VecFloat3D* const that,
  const float theta);

// Return a VecFloat3D equal to 'that' rotated right-hand by 'theta'
// radians around Z
VecFloat3D _VecFloatGetRotZ(const VecFloat3D* const that,
  const float theta);

// Return the dot product of 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDotProd(const VecFloat* const that,
  const VecFloat* const tho);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDotProd2D(const VecFloat2D* const that,
  const VecFloat2D* const tho);
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDotProd3D(const VecFloat3D* const that,
  const VecFloat3D* const tho);

// Return the cross product of 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
VecFloat* _VecFloatGetCrossProd(const VecFloat* const that,
  const VecFloat* const tho);
#if BUILDMODE != 0
static inline
#endif
VecFloat3D _VecFloatGetCrossProd3D(const VecFloat3D* const that,
  const VecFloat3D* const 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(const VecFloat2D* const that,
  const VecFloat2D* const tho);

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

// Return the conversion of VecShort 'that' to a VecFloat
#if BUILDMODE != 0
static inline
#endif
VecFloat* VecShortToFloat(const VecShort* const that);
```

```
#if BUILDMODE != 0
static inline
#endif
VecFloat2D VecShortToFloat2D(const VecShort2D* const that);
#if BUILDMODE != 0
static inline
#endif
VecFloat3D VecShortToFloat3D(const VecShort3D* const that);

// Return the conversion of VecLong 'that' to a VecFloat
#if BUILDMODE != 0
static inline
#endif
VecFloat* VecLongToFloat(const VecLong* const that);
#if BUILDMODE != 0
static inline
#endif
VecFloat2D VecLongToFloat2D(const VecLong2D* const that);
#if BUILDMODE != 0
static inline
#endif
VecFloat3D VecLongToFloat3D(const VecLong3D* const that);

// Get the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGetMaxVal(const VecFloat* const that);

// Get the min value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGetMinVal(const VecFloat* const that);

// Get the max value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGetMaxValAbs(const VecFloat* const that);

// Get the min value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGetMinValAbs(const VecFloat* const that);

// Step the values of the vector incrementally by delta 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 after 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 _VecFloatStepDelta(VecFloat* const that,
  const VecFloat* const bound, const VecFloat* const delta);

// Step the values of the vector incrementally by delta
// in the following order (for example) :
// (0.,0.,0.)->(0.,0.,1.)->(0.,0.,2.)->(0.,1.,0.)->(0.,1.,1.)->...
```

```
// The lower limit for each value is given by 'from' (val[i] >= from[i])
// The upper limit for each value is given by 'to' (val[i] <= to[i])
// 'that' must be initialised to 'from' before the first call of this
// function
// Return false after all values of 'that' have reached their upper
// limit (in which case 'that''s values are all set back to from)
// Return true else
bool _VecFloatShiftStepDelta(VecFloat* const that,
  const VecFloat* const from, const VecFloat* const to,
  const VecFloat* const delta);

// Get the index of the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecFloatGetIMaxVal(const VecFloat* const that);

// Return a set of two vectors containing the bounds of the vectors in
// the GSet 'that'
// The set must have at least one element
// The returned set is ordered as follow: (boundMin, boundMax)
GSetVecFloat _GSetVecFloatGetBounds(const GSetVecFloat* const that);

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

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

// Vector of float values
typedef struct MatFloat {
  // Dimension (nbCol, nbLine)
  const VecShort2D _dim;
  // Values (memorized by lines)
  float _val[0];
} MatFloat;

// Simple pod to hold the result of a QR decomposition
typedef struct QRDecomp {
  MatFloat* _Q;
  MatFloat* _R;
} QRDecomp;

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

// Free memory used by the QRDecomp 'that'
#if BUILDMODE != 0
static inline
#endif
void QRDecompFreeStatic(QRDecomp* const that);

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

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

// Clone the MatFloat
MatFloat* _MatFloatClone(const MatFloat* const that);
```

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

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

// Function which decode from JSON encoding 'json' to 'that'
bool _MatFloatDecodeAsJSON(MatFloat** that, JSONNode* json);

// 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
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true upon success, else false
bool _MatFloatSave(MatFloat* const that, FILE* stream, bool compact);

// 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* const that, FILE* stream, unsigned int prec);
static inline void _MatFloatPrintlnDef(MatFloat* const 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
static inline
#endif
float _MatFloatGet(const MatFloat* const 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
static inline
#endif
void _MatFloatSet(MatFloat* const that, VecShort2D* index, float v);

// Return the dimension of the MatFloat
#if BUILDMODE != 0
static inline
#endif
const VecShort2D* _MatFloatDim(const MatFloat* const that);

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

```
// Return the number of rows of the MatFloat 'that'
#if BUILDMODE != 0
static inline
#endif
short _MatFloatGetNbRow(const MatFloat* const that);

// Return the number of columns of the MatFloat 'that'
#if BUILDMODE != 0
static inline
#endif
short _MatFloatGetNbCol(const MatFloat* const that);

// Return the inverse matrix of 'that'
// The matrix must be a square matrix
// Return NULL if the matrix is not invertible, or in some case when
// the matrix's diagonal contains null values and the matrix's size
// is greater than 3
MatFloat* _MatFloatGetInv(const MatFloat* const that);

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

// Return the product of vector 'v' and transpose of vector 'w'
MatFloat* _MatFloatGetProdVecVecTransposeFloat(
  const VecFloat* const v,
  const VecFloat* const w);

// Return the product of matrix 'that' by matrix 'tho'
// Number of columns of 'that' must equal number of line of 'tho'
MatFloat* _MatFloatGetProdMatFloat(const MatFloat* const that, const MatFloat* tho);

// Return the addition of matrix 'that' with matrix 'tho'
// 'that' and 'tho' must have same dimensions
#if BUILDMODE != 0
static inline
#endif
MatFloat* _MatFloatGetAdd(MatFloat* const that, MatFloat* tho);

// Add matrix 'that' with matrix 'tho' and store the result in 'that'
// 'that' and 'tho' must have same dimensions
#if BUILDMODE != 0
static inline
#endif
void _MatFloatAdd(MatFloat* const that, MatFloat* tho);

// Multiply the matrix 'that' by 'a'
#if BUILDMODE != 0
static inline
#endif
void _MatFloatScale(MatFloat* const that, const float a);

// Return true if 'that' is equal to 'tho', false else
bool _MatFloatIsEqual(MatFloat* const that, MatFloat* tho);

// Calculate the Eigen values and vectors of the MatFloat 'that'
// Return a set of VecFloat. The first VecFloat of the set contains
// the Eigen values, with values sorted from biggest to
// smallest (in absolute value). The following VecFloat are the
// respectiev Eigen vectors
// 'that' must be a 2D square matrix
```

```
// Return the identity if the QR decompostion fails
// http://madrury.github.io/jekyll/update/statistics/2017/10/04/qr-algorithm.html
// TODO: should be improved with the Hessenberg QR method
// https://www.math.kth.se/na/SF2524/matber15/qrmethod.pdf
GSetVecFloat _MatFloatGetEigenValues(const MatFloat* const that);

// Calculate the QR decomposition of the MatFloat 'that' using the
// Householder algorithm
// Return {NULL, NULL} if the MatFloat couldn't be decomposed
// http://www.seas.ucla.edu/~vandenbe/133A/lectures/qr.pdf
QRDecomp _MatFloatGetQR(const MatFloat* const that);

// Calculate the transposed of the MatFloat 'that'
MatFloat* _MatFloatGetTranspose(const MatFloat* const that);

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

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

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

// Should be 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(const float mean, const float sigma);
Gauss GaussCreateStatic(const float mean, const 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
static inline
#endif
float GaussGet(const Gauss* const that, const float x);

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

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

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

// Convert radians to degrees
static inline float ConvRad2Deg(const float rad) {
  return rad / PBMATH_TWOPI_DIV_360;
}

// Convert degrees to radians
static inline float ConvDeg2Rad(const 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(const MatFloat* const m, const VecFloat* const 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(const SysLinEq* const that);

// 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
static inline
#endif
VecFloat* SysLinEqSolve(const SysLinEq* const that);

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

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

// -------------- Ratio

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

// Linear system of equalities
typedef struct Ratio {

  // Components
  long _base;
  unsigned int _numerator;
  unsigned int _denominator;

} Ratio;

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

// Create a new static Ratio
Ratio RatioCreateStatic(long b, unsigned int n, unsigned int d);

// Convert the float 'v' into the nearest Ratio using the Farey's algorithm
// given the precision 'prec'
Ratio RatioFromFloatPrec(float v, float prec);
#define RatioFromFloat(v) RatioFromFloatPrec((v), PBMATH_EPSILON)

// Convert the Ratio 'that' into a float
float RatioToFloat(const Ratio* that);

// Reduce the fractional part of the Ratio 'that' and update the base such as
// numerator < denominator
void RatioReduce(Ratio* that);

// Get the base of the Ratio 'that'
#if BUILDMODE != 0
static inline
#endif
long RatioGetBase(const Ratio* that);

// Get the numerator of the Ratio 'that'
#if BUILDMODE != 0
static inline
#endif
unsigned int RatioGetNumerator(const Ratio* that);
```

```
// Get the denominator of the Ratio 'that'
#if BUILDMODE != 0
static inline
#endif
unsigned int RatioGetDenominator(const Ratio* that);

// Set the base of the Ratio 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void RatioSetBase(Ratio* that, long v);

// Set the numerator of the Ratio 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void RatioSetNumerator(Ratio* that, unsigned int v);

// Set the denominator of the Ratio 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void RatioSetDenominator(Ratio* that, unsigned int v);

// Print the Ratio on 'stream' as a+b/c
void RatioPrint(const Ratio* that, FILE* stream);
#define RatioPrintln(R, S) do{RatioPrint(R,S);fprintf(S,"\n");}while(0)

// -------------- LeastSquareLinReg

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

// Linear system of equalities
typedef struct LeastSquareLinReg {

  // Component
  const MatFloat* X;

  // Matrix for computation
  MatFloat* Xp;

  // Bias of the last computed solution
  float bias;

} LeastSquareLinReg;

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

// Create a new static LeastSquareLinReg
LeastSquareLinReg LeastSquareLinRegCreateStatic(MatFloat* X);

// Free the static LeastSquareLinReg 'that'
void LeastSquareLinRegFreeStatic(LeastSquareLinReg* that);

// Compute the solution of the LeastSquareLinReg 'that' for 'Y'
VecFloat* LSLRSolve(LeastSquareLinReg* that, const VecFloat* Y);

// Set the component of the LeastSquareLinReg 'that' to 'X'
#if BUILDMODE != 0
static inline
#endif
```

```
void LSLRSetComp(LeastSquareLinReg* that, const MatFloat* X);

// Get the bias of the last computed solution of the LeastSquareLinReg 'that'
#if BUILDMODE != 0
static inline
#endif
float LSLRGetBias(const LeastSquareLinReg* that);

// Return true if the LeastSquareLinReg 'that' is solvable
#if BUILDMODE != 0
static inline
#endif
bool LSLRIsSolvable(const LeastSquareLinReg* that);

// -------------- Quaternion
// cf http://news.povray.org/povray.binaries.scene-files/message/%3CXns940C86DC9B1D4None%40204.213.191.226%3E/
// ================= Data structure ===================

// Quaternion to perform rotation in 3D
typedef struct Quaternion {

  // Components
  VecFloat4D val;

} Quaternion;

// Create a new static Quaternion
Quaternion QuaternionCreateStatic(void);

// Free the static Quaternion 'that'
void QuaternionFreeStatic(Quaternion* that);

// Create a new static Quaternion from the rotation matrix 'rotMat'
Quaternion QuaternionCreateFromRotMat(MatFloat* rotMat);

// Create a new static Quaternion corresponding to the rotation around
// 'axis' (must be normalized) by 'theta' (in radians)
Quaternion QuaternionCreateFromRotAxis(VecFloat* axis, float theta);

// Convert the Quaternion 'that' to a rotation matrix
MatFloat* QuaternionToRotMat(Quaternion* that);

// Return the quaternion equivalent to the rotation of 'that' followed by
// the rotation of 'tho'
Quaternion QuaternionGetComposition(Quaternion* that, Quaternion* tho);

// Return the quaternion equivalent to the rotation necessary to convert
// 'that' into 'tho'
// tho = QuaternionGetComposition(QuaternionGetDifference(that, tho), that)
Quaternion QuaternionGetDifference(Quaternion* that, Quaternion* tho);

// Return the inverse quaternion of the quaternion 'that'
Quaternion QuaternionGetInverse(Quaternion* that);

// Return true if the two quaternions are equals, false else
bool QuaternionIsEqual(Quaternion* that, Quaternion* tho);

// Print the Quaternion 'that' on 'stream'
void QuaternionPrint(Quaternion* that, FILE* stream);
#define QuaternionPrintln(Q, S) do {QuaternionPrint(Q, S); fprintf(S, "\n");} while(0)

// Rotate the vector 'v' by the quaternion 'that'
```

```
void QuaternionApply(Quaternion* that, VecFloat* v);

// Normalise the quaternion
void QuaternionNormalise(Quaternion* that);

// Get the rotation axis of the quaternion 'that'
VecFloat3D QuaternionGetRotAxis(Quaternion* that);

// Get the rotation angle (in radians) of the quaternion 'that'
float QuaternionGetRotAngle(Quaternion* that);

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

// -------------- 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
static inline
#endif
int powi(const int base, const int exp);

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

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

// Compute the 'iElem'-th element of the 'base'-ary version of the
// Thue-Morse sequence
// 'iElem' >= 0
// 'base' >= 2
long ThueMorseSeqGetNthElem(long iElem, long base);

// Compute the area of a triangle knowing its 3 sides length 'a', 'b', 'c'
// using the Hero's formula
double GetAreaTriangleHero(
  const double a,
  const double b,
  const double c);

// Return the Fibonacci sequence up to the 'n'-th element in a dynamically
// allocated array of unsigned long
unsigned long* GetFibonacciSeq(unsigned int n);

// Return the Fibonacci grid lattice for the 'n'-th Fibonacci number in a
// dynamically allocated array of pairs of float in [0,1]
// Stores the nb of points in 'nbPoints'
float* GetFibonacciGridLattice(
    unsigned int n,
  unsigned long* nbPoints);
```

```
// Return the Fibonacci polar lattice for the 'n'-th Fibonacci number in a
// dynamically allocated array of pairs of float in [-1,1]
// Stores the nb of points in 'nbPoints'
float* GetFibonacciPolarLattice(
    unsigned int n,
  unsigned long* nbPoints);

// Return the greatest common divisor using the Stein's algorithm
// https://en.wikipedia.org/wiki/Binary_GCD_algorithm
unsigned int GetGCD(unsigned int u, unsigned int v);

// Get the approximated inverse square root of a number using the Quake
// algorithm
// cf https://en.wikipedia.org/wiki/Fast_inverse_square_root
float GetFastInverseSquareRoot(float number);

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

#define VecClone(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatClone, \
  VecShort*: _VecShortClone, \
  VecLong*: _VecLongClone, \
  const VecFloat*: _VecFloatClone, \
  const VecShort*: _VecShortClone, \
  const VecLong*: _VecLongClone, \
  default: PBErrInvalidPolymorphism)(Vec)

#define VecEncodeAsJSON(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatEncodeAsJSON, \
  VecShort*: _VecShortEncodeAsJSON, \
  VecLong*: _VecLongEncodeAsJSON, \
  const VecFloat*: _VecFloatEncodeAsJSON, \
  const VecShort*: _VecShortEncodeAsJSON, \
  const VecLong*: _VecLongEncodeAsJSON, \
  default: PBErrInvalidPolymorphism)(Vec)

#define VecDecodeAsJSON(VecRef, Json) _Generic(VecRef, \
  VecFloat**: _VecFloatDecodeAsJSON, \
  VecShort**: _VecShortDecodeAsJSON, \
  VecLong**: _VecLongDecodeAsJSON, \
  default: PBErrInvalidPolymorphism)(VecRef, Json)

#define VecLoad(VecRef, Stream) _Generic(VecRef, \
  VecFloat**: _VecFloatLoad, \
  VecShort**: _VecShortLoad, \
  VecLong**: _VecLongLoad, \
  default: PBErrInvalidPolymorphism)(VecRef, Stream)

#define VecSave(Vec, Stream, Compact) _Generic(Vec, \
  VecFloat*: _VecFloatSave, \
  VecFloat2D*: _VecFloatSave, \
  VecFloat3D*: _VecFloatSave, \
  VecShort*: _VecShortSave, \
  VecShort2D*: _VecShortSave, \
  VecShort3D*: _VecShortSave, \
  VecShort4D*: _VecShortSave, \
  VecLong*: _VecLongSave, \
  VecLong2D*: _VecLongSave, \
  VecLong3D*: _VecLongSave, \
  VecLong4D*: _VecLongSave, \
  const VecFloat*: _VecFloatSave, \
```

```
  const VecFloat2D*: _VecFloatSave, \
  const VecFloat3D*: _VecFloatSave, \
  const VecShort*: _VecShortSave, \
  const VecShort2D*: _VecShortSave, \
  const VecShort3D*: _VecShortSave, \
  const VecShort4D*: _VecShortSave, \
  const VecLong*: _VecLongSave, \
  const VecLong2D*: _VecLongSave, \
  const VecLong3D*: _VecLongSave, \
  const VecLong4D*: _VecLongSave, \
  default: PBErrInvalidPolymorphism)( \
    _Generic(Vec,  \
      VecFloat2D*: (const VecFloat*)(Vec), \
      VecFloat3D*: (const VecFloat*)(Vec), \
      VecShort2D*: (const VecShort*)(Vec), \
      VecShort3D*: (const VecShort*)(Vec), \
      VecShort4D*: (const VecShort*)(Vec), \
      VecLong2D*: (const VecLong*)(Vec), \
      VecLong3D*: (const VecLong*)(Vec), \
      VecLong4D*: (const VecLong*)(Vec), \
      const VecFloat2D*: (const VecFloat*)(Vec), \
      const VecFloat3D*: (const VecFloat*)(Vec), \
      const VecShort2D*: (const VecShort*)(Vec), \
      const VecShort3D*: (const VecShort*)(Vec), \
      const VecShort4D*: (const VecShort*)(Vec), \
      const VecLong2D*: (const VecLong*)(Vec), \
      const VecLong3D*: (const VecLong*)(Vec), \
      const VecLong4D*: (const VecLong*)(Vec), \
      default: Vec),  \
    Stream, Compact)

#define VecFree(VecRef) _Generic(VecRef, \
  VecFloat**: _VecFloatFree, \
  VecShort**: _VecShortFree, \
  VecLong**: _VecLongFree, \
  default: PBErrInvalidPolymorphism)(VecRef)

#define VecPrint(Vec, Stream) _Generic(Vec, \
  VecFloat*: _VecFloatPrintDef, \
  VecFloat2D*: _VecFloatPrintDef, \
  VecFloat3D*: _VecFloatPrintDef, \
  VecFloat4D*: _VecFloatPrintDef, \
  VecShort*: _VecShortPrint, \
  VecShort2D*: _VecShortPrint, \
  VecShort3D*: _VecShortPrint, \
  VecShort4D*: _VecShortPrint, \
  VecLong*: _VecLongPrint, \
  VecLong2D*: _VecLongPrint, \
  VecLong3D*: _VecLongPrint, \
  VecLong4D*: _VecLongPrint, \
  const VecFloat*: _VecFloatPrintDef, \
  const VecFloat2D*: _VecFloatPrintDef, \
  const VecFloat3D*: _VecFloatPrintDef, \
  const VecFloat4D*: _VecFloatPrintDef, \
  const VecShort*: _VecShortPrint, \
  const VecShort2D*: _VecShortPrint, \
  const VecShort3D*: _VecShortPrint, \
  const VecShort4D*: _VecShortPrint, \
  const VecLong*: _VecLongPrint, \
  const VecLong2D*: _VecLongPrint, \
  const VecLong3D*: _VecLongPrint, \
  const VecLong4D*: _VecLongPrint, \
```

```
  default: PBErrInvalidPolymorphism)( \
    _Generic(Vec,  \
      VecFloat2D*: (const VecFloat*)(Vec), \
      VecFloat3D*: (const VecFloat*)(Vec), \
      VecFloat4D*: (const VecFloat*)(Vec), \
      VecShort2D*: (const VecShort*)(Vec), \
      VecShort3D*: (const VecShort*)(Vec), \
      VecShort4D*: (const VecShort*)(Vec), \
      VecLong2D*: (const VecLong*)(Vec), \
      VecLong3D*: (const VecLong*)(Vec), \
      VecLong4D*: (const VecLong*)(Vec), \
      const VecFloat2D*: (const VecFloat*)(Vec), \
      const VecFloat3D*: (const VecFloat*)(Vec), \
      const VecFloat4D*: (const VecFloat*)(Vec), \
      const VecShort2D*: (const VecShort*)(Vec), \
      const VecShort3D*: (const VecShort*)(Vec), \
      const VecShort4D*: (const VecShort*)(Vec), \
      const VecLong2D*: (const VecLong*)(Vec), \
      const VecLong3D*: (const VecLong*)(Vec), \
      const VecLong4D*: (const VecLong*)(Vec), \
      default: Vec),  \
    Stream)

#define VecPrintln(V, S) do {VecPrint(V, S); fprintf(S, "\n");} while(0)

#define VecGet(Vec, Index) _Generic(Vec, \
  VecFloat*: _VecFloatGet, \
  VecFloat2D*: _VecFloatGet2D, \
  VecFloat3D*: _VecFloatGet3D, \
  VecFloat4D*: _VecFloatGet4D, \
  VecShort*: _VecShortGet, \
  VecShort2D*: _VecShortGet2D, \
  VecShort3D*: _VecShortGet3D, \
  VecShort4D*: _VecShortGet4D, \
  VecLong*: _VecLongGet, \
  VecLong2D*: _VecLongGet2D, \
  VecLong3D*: _VecLongGet3D, \
  VecLong4D*: _VecLongGet4D, \
  const VecFloat*: _VecFloatGet, \
  const VecFloat2D*: _VecFloatGet2D, \
  const VecFloat3D*: _VecFloatGet3D, \
  const VecFloat4D*: _VecFloatGet4D, \
  const VecShort*: _VecShortGet, \
  const VecShort2D*: _VecShortGet2D, \
  const VecShort3D*: _VecShortGet3D, \
  const VecShort4D*: _VecShortGet4D, \
  const VecLong*: _VecLongGet, \
  const VecLong2D*: _VecLongGet2D, \
  const VecLong3D*: _VecLongGet3D, \
  const VecLong4D*: _VecLongGet4D, \
  default: PBErrInvalidPolymorphism)(Vec, Index)

#define VecSet(Vec, Index, Val) _Generic(Vec, \
  VecFloat*: _VecFloatSet, \
  VecFloat2D*: _VecFloatSet2D, \
  VecFloat3D*: _VecFloatSet3D, \
  VecFloat4D*: _VecFloatSet4D, \
  VecShort*: _VecShortSet, \
  VecShort2D*: _VecShortSet2D, \
  VecShort3D*: _VecShortSet3D, \
  VecShort4D*: _VecShortSet4D, \
  VecLong*: _VecLongSet, \
```

```
  VecLong2D*: _VecLongSet2D, \
  VecLong3D*: _VecLongSet3D, \
  VecLong4D*: _VecLongSet4D, \
  default: PBErrInvalidPolymorphism)(Vec, Index, Val)

#define VecSetAdd(Vec, Index, Val) _Generic(Vec, \
  VecFloat*: _VecFloatSetAdd, \
  VecFloat2D*: _VecFloatSetAdd2D, \
  VecFloat3D*: _VecFloatSetAdd3D, \
  VecShort*: _VecShortSetAdd, \
  VecShort2D*: _VecShortSetAdd2D, \
  VecShort3D*: _VecShortSetAdd3D, \
  VecShort4D*: _VecShortSetAdd4D, \
  VecLong*: _VecLongSetAdd, \
  VecLong2D*: _VecLongSetAdd2D, \
  VecLong3D*: _VecLongSetAdd3D, \
  VecLong4D*: _VecLongSetAdd4D, \
  default: PBErrInvalidPolymorphism)(Vec, Index, Val)

#define VecSetNull(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatSetNull, \
  VecFloat2D*: _VecFloatSetNull, \
  VecFloat3D*: _VecFloatSetNull, \
  VecShort*: _VecShortSetNull, \
  VecShort2D*: _VecShortSetNull, \
  VecShort3D*: _VecShortSetNull, \
  VecShort4D*: _VecShortSetNull, \
  VecLong*: _VecLongSetNull, \
  VecLong2D*: _VecLongSetNull, \
  VecLong3D*: _VecLongSetNull, \
  VecLong4D*: _VecLongSetNull, \
  default: PBErrInvalidPolymorphism)( \
    _Generic(Vec,  \
      VecFloat2D*: (VecFloat*)(Vec), \
      VecFloat3D*: (VecFloat*)(Vec), \
      VecShort2D*: (VecShort*)(Vec), \
      VecShort3D*: (VecShort*)(Vec), \
      VecShort4D*: (VecShort*)(Vec), \
      VecLong2D*: (VecLong*)(Vec), \
      VecLong3D*: (VecLong*)(Vec), \
      VecLong4D*: (VecLong*)(Vec), \
      default: Vec))

#define VecSetAll(Vec, Val) _Generic(Vec, \
  VecFloat*: _VecFloatSetAll, \
  VecFloat2D*: _VecFloatSetAll, \
  VecFloat3D*: _VecFloatSetAll, \
  VecShort*: _VecShortSetAll, \
  VecShort2D*: _VecShortSetAll, \
  VecShort3D*: _VecShortSetAll, \
  VecShort4D*: _VecShortSetAll, \
  VecLong*: _VecLongSetAll, \
  VecLong2D*: _VecLongSetAll, \
  VecLong3D*: _VecLongSetAll, \
  VecLong4D*: _VecLongSetAll, \
  default: PBErrInvalidPolymorphism)( \
    _Generic(Vec,  \
      VecFloat2D*: (VecFloat*)(Vec), \
      VecFloat3D*: (VecFloat*)(Vec), \
      VecShort2D*: (VecShort*)(Vec), \
      VecShort3D*: (VecShort*)(Vec), \
      VecShort4D*: (VecShort*)(Vec), \
```

```
      VecLong2D*: (VecLong*)(Vec), \
      VecLong3D*: (VecLong*)(Vec), \
      VecLong4D*: (VecLong*)(Vec), \
      default: Vec), Val)

#define VecCopy(VecDest, VecSrc) _Generic(VecDest, \
  VecFloat*: _Generic(VecSrc, \
    VecFloat*: _VecFloatCopy, \
    VecFloat2D*: _VecFloatCopy, \
    VecFloat3D*: _VecFloatCopy, \
    const VecFloat*: _VecFloatCopy, \
    const VecFloat2D*: _VecFloatCopy, \
    const VecFloat3D*: _VecFloatCopy, \
    default: PBErrInvalidPolymorphism), \
  VecFloat2D*: _Generic(VecSrc, \
    VecFloat*: _VecFloatCopy, \
    VecFloat2D*: _VecFloatCopy, \
    const VecFloat*: _VecFloatCopy, \
    const VecFloat2D*: _VecFloatCopy, \
    default: PBErrInvalidPolymorphism), \
  VecFloat3D*: _Generic(VecSrc, \
    VecFloat*: _VecFloatCopy, \
    VecFloat3D*: _VecFloatCopy, \
    const VecFloat*: _VecFloatCopy, \
    const VecFloat3D*: _VecFloatCopy, \
    default: PBErrInvalidPolymorphism), \
  VecShort*: _Generic(VecSrc, \
    VecShort*: _VecShortCopy, \
    VecShort2D*: _VecShortCopy, \
    VecShort3D*: _VecShortCopy, \
    VecShort4D*: _VecShortCopy, \
    const VecShort*: _VecShortCopy, \
    const VecShort2D*: _VecShortCopy, \
    const VecShort3D*: _VecShortCopy, \
    const VecShort4D*: _VecShortCopy, \
    default: PBErrInvalidPolymorphism), \
  VecShort2D*: _Generic(VecSrc, \
    VecShort*: _VecShortCopy, \
    VecShort2D*: _VecShortCopy, \
    const VecShort*: _VecShortCopy, \
    const VecShort2D*: _VecShortCopy, \
    default: PBErrInvalidPolymorphism), \
  VecShort3D*: _Generic(VecSrc, \
    VecShort*: _VecShortCopy, \
    VecShort3D*: _VecShortCopy, \
    const VecShort*: _VecShortCopy, \
    const VecShort3D*: _VecShortCopy, \
    default: PBErrInvalidPolymorphism), \
  VecShort4D*: _Generic(VecSrc, \
    VecShort*: _VecShortCopy, \
    VecShort4D*: _VecShortCopy, \
    const VecShort*: _VecShortCopy, \
    const VecShort4D*: _VecShortCopy, \
    default: PBErrInvalidPolymorphism), \
  VecLong*: _Generic(VecSrc, \
    VecLong*: _VecLongCopy, \
    VecLong2D*: _VecLongCopy, \
    VecLong3D*: _VecLongCopy, \
    VecLong4D*: _VecLongCopy, \
    const VecLong*: _VecLongCopy, \
    const VecLong2D*: _VecLongCopy, \
    const VecLong3D*: _VecLongCopy, \
```

```
    const VecLong4D*: _VecLongCopy, \
    default: PBErrInvalidPolymorphism), \
  VecLong2D*: _Generic(VecSrc, \
    VecLong*: _VecLongCopy, \
    VecLong2D*: _VecLongCopy, \
    const VecLong*: _VecLongCopy, \
    const VecLong2D*: _VecLongCopy, \
    default: PBErrInvalidPolymorphism), \
  VecLong3D*: _Generic(VecSrc, \
    VecLong*: _VecLongCopy, \
    VecLong3D*: _VecLongCopy, \
    const VecLong*: _VecLongCopy, \
    const VecLong3D*: _VecLongCopy, \
    default: PBErrInvalidPolymorphism), \
  VecLong4D*: _Generic(VecSrc, \
    VecLong*: _VecLongCopy, \
    VecLong4D*: _VecLongCopy, \
    const VecLong*: _VecLongCopy, \
    const VecLong4D*: _VecLongCopy, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)( \
    _Generic(VecDest,  \
      VecFloat2D*: (VecFloat*)(VecDest), \
      VecFloat3D*: (VecFloat*)(VecDest), \
      VecShort2D*: (VecShort*)(VecDest), \
      VecShort3D*: (VecShort*)(VecDest), \
      VecShort4D*: (VecShort*)(VecDest), \
      VecLong2D*: (VecLong*)(VecDest), \
      VecLong3D*: (VecLong*)(VecDest), \
      VecLong4D*: (VecLong*)(VecDest), \
      default: VecDest),  \
    _Generic(VecSrc,  \
      VecFloat2D*: (const VecFloat*)(VecSrc), \
      VecFloat3D*: (const VecFloat*)(VecSrc), \
      VecShort2D*: (const VecShort*)(VecSrc), \
      VecShort3D*: (const VecShort*)(VecSrc), \
      VecShort4D*: (const VecShort*)(VecSrc), \
      VecLong2D*: (const VecLong*)(VecSrc), \
      VecLong3D*: (const VecLong*)(VecSrc), \
      VecLong4D*: (const VecLong*)(VecSrc), \
      const VecFloat2D*: (const VecFloat*)(VecSrc), \
      const VecFloat3D*: (const VecFloat*)(VecSrc), \
      const VecShort2D*: (const VecShort*)(VecSrc), \
      const VecShort3D*: (const VecShort*)(VecSrc), \
      const VecShort4D*: (const VecShort*)(VecSrc), \
      const VecLong2D*: (const VecLong*)(VecSrc), \
      const VecLong3D*: (const VecLong*)(VecSrc), \
      const VecLong4D*: (const VecLong*)(VecSrc), \
      default: VecSrc))

#define VecGetDim(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatGetDim, \
  VecFloat2D*: _VecFloatGetDim, \
  VecFloat3D*: _VecFloatGetDim, \
  VecShort*: _VecShortGetDim, \
  VecShort2D*: _VecShortGetDim, \
  VecShort3D*: _VecShortGetDim, \
  VecShort4D*: _VecShortGetDim, \
  VecLong*: _VecLongGetDim, \
  VecLong2D*: _VecLongGetDim, \
  VecLong3D*: _VecLongGetDim, \
  VecLong4D*: _VecLongGetDim, \
```

```
  const VecFloat*: _VecFloatGetDim, \
  const VecFloat2D*: _VecFloatGetDim, \
  const VecFloat3D*: _VecFloatGetDim, \
  const VecShort*: _VecShortGetDim, \
  const VecShort2D*: _VecShortGetDim, \
  const VecShort3D*: _VecShortGetDim, \
  const VecShort4D*: _VecShortGetDim, \
  const VecLong*: _VecLongGetDim, \
  const VecLong2D*: _VecLongGetDim, \
  const VecLong3D*: _VecLongGetDim, \
  const VecLong4D*: _VecLongGetDim, \
  default: PBErrInvalidPolymorphism)( \
    _Generic(Vec, \
      VecFloat*: (const VecFloat*)(Vec), \
      VecFloat2D*: (const VecFloat*)(Vec), \
      VecFloat3D*: (const VecFloat*)(Vec), \
      VecShort*: (const VecShort*)(Vec), \
      VecShort2D*: (const VecShort*)(Vec), \
      VecShort3D*: (const VecShort*)(Vec), \
      VecShort4D*: (const VecShort*)(Vec), \
      VecLong*: (const VecLong*)(Vec), \
      VecLong2D*: (const VecLong*)(Vec), \
      VecLong3D*: (const VecLong*)(Vec), \
      VecLong4D*: (const VecLong*)(Vec), \
      const VecFloat2D*: (const VecFloat*)(Vec), \
      const VecFloat3D*: (const VecFloat*)(Vec), \
      const VecShort2D*: (const VecShort*)(Vec), \
      const VecShort3D*: (const VecShort*)(Vec), \
      const VecShort4D*: (const VecShort*)(Vec), \
      const VecLong2D*: (const VecLong*)(Vec), \
      const VecLong3D*: (const VecLong*)(Vec), \
      const VecLong4D*: (const VecLong*)(Vec), \
      default: Vec))

#define VecGetNewDim(Vec, Dim) _Generic(Vec, \
  VecFloat*: _VecFloatGetNewDim, \
  const VecFloat*: _VecFloatGetNewDim, \
  VecLong*: _VecLongGetNewDim, \
  const VecLong*: _VecLongGetNewDim, \
  default: PBErrInvalidPolymorphism)( \
    _Generic(Vec, \
      VecFloat*: Vec, \
      const VecFloat*: Vec, \
      VecLong*: Vec, \
      const VecLong*: Vec, \
      default: Vec), Dim)

#define VecNorm(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatNorm, \
  VecFloat2D*: _VecFloatNorm2D, \
  VecFloat3D*: _VecFloatNorm3D, \
  VecFloat4D*: _VecFloatNorm4D, \
  const VecFloat*: _VecFloatNorm, \
  const VecFloat2D*: _VecFloatNorm2D, \
  const VecFloat3D*: _VecFloatNorm3D, \
  const VecFloat4D*: _VecFloatNorm4D, \
  default: PBErrInvalidPolymorphism)(Vec)

#define VecNormalise(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatNormalise, \
  VecFloat2D*: _VecFloatNormalise2D, \
  VecFloat3D*: _VecFloatNormalise3D, \
```

```
  VecFloat4D*: _VecFloatNormalise4D, \
  default: PBErrInvalidPolymorphism)(Vec)

#define VecDist(VecA, VecB) _Generic(VecA, \
  VecFloat*: _Generic(VecB, \
    VecFloat*: _VecFloatDist, \
    const VecFloat*: _VecFloatDist, \
    default: PBErrInvalidPolymorphism), \
  VecFloat2D*: _Generic(VecB, \
    VecFloat2D*: _VecFloatDist2D, \
    const VecFloat2D*: _VecFloatDist2D, \
    default: PBErrInvalidPolymorphism), \
  VecFloat3D*: _Generic(VecB, \
    VecFloat3D*: _VecFloatDist3D, \
    const VecFloat3D*: _VecFloatDist3D, \
    default: PBErrInvalidPolymorphism), \
  VecShort*: _Generic(VecB, \
    VecShort*: _VecShortHamiltonDist,\
    const VecShort*: _VecShortHamiltonDist,\
    default: PBErrInvalidPolymorphism), \
  VecShort2D*: _Generic(VecB, \
    VecShort2D*: _VecShortHamiltonDist2D,\
    const VecShort2D*: _VecShortHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  VecShort3D*: _Generic(VecB, \
    VecShort3D*: _VecShortHamiltonDist3D,\
    const VecShort3D*: _VecShortHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  VecShort4D*: _Generic(VecB, \
    VecShort4D*: _VecShortHamiltonDist4D,\
    const VecShort4D*: _VecShortHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  VecLong*: _Generic(VecB, \
    VecLong*: _VecLongHamiltonDist,\
    const VecLong*: _VecLongHamiltonDist,\
    default: PBErrInvalidPolymorphism), \
  VecLong2D*: _Generic(VecB, \
    VecLong2D*: _VecLongHamiltonDist2D,\
    const VecLong2D*: _VecLongHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  VecLong3D*: _Generic(VecB, \
    VecLong3D*: _VecLongHamiltonDist3D,\
    const VecLong3D*: _VecLongHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  VecLong4D*: _Generic(VecB, \
    VecLong4D*: _VecLongHamiltonDist4D,\
    const VecLong4D*: _VecLongHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  const VecFloat*: _Generic(VecB, \
    VecFloat*: _VecFloatDist, \
    const VecFloat*: _VecFloatDist, \
    default: PBErrInvalidPolymorphism), \
  const VecFloat2D*: _Generic(VecB, \
    VecFloat2D*: _VecFloatDist2D, \
    const VecFloat2D*: _VecFloatDist2D, \
    default: PBErrInvalidPolymorphism), \
  const VecFloat3D*: _Generic(VecB, \
    VecFloat3D*: _VecFloatDist3D, \
    const VecFloat3D*: _VecFloatDist3D, \
    default: PBErrInvalidPolymorphism), \
  const VecShort*: _Generic(VecB, \
    VecShort*: _VecShortHamiltonDist,\
```

```
    const VecShort*: _VecShortHamiltonDist,\
    default: PBErrInvalidPolymorphism), \
  const VecShort2D*: _Generic(VecB, \
    VecShort2D*: _VecShortHamiltonDist2D,\
    const VecShort2D*: _VecShortHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  const VecShort3D*: _Generic(VecB, \
    VecShort3D*: _VecShortHamiltonDist3D,\
    const VecShort3D*: _VecShortHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  const VecShort4D*: _Generic(VecB, \
    VecShort4D*: _VecShortHamiltonDist4D,\
    const VecShort4D*: _VecShortHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  const VecLong*: _Generic(VecB, \
    VecLong*: _VecLongHamiltonDist,\
    const VecLong*: _VecLongHamiltonDist,\
    default: PBErrInvalidPolymorphism), \
  const VecLong2D*: _Generic(VecB, \
    VecLong2D*: _VecLongHamiltonDist2D,\
    const VecLong2D*: _VecLongHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  const VecLong3D*: _Generic(VecB, \
    VecLong3D*: _VecLongHamiltonDist3D,\
    const VecLong3D*: _VecLongHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  const VecLong4D*: _Generic(VecB, \
    VecLong4D*: _VecLongHamiltonDist4D,\
    const VecLong4D*: _VecLongHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(VecA, VecB)

#define VecHamiltonDist(VecA, VecB) _Generic(VecA, \
  VecFloat*: _Generic(VecB, \
    VecFloat*: _VecFloatHamiltonDist, \
    const VecFloat*: _VecFloatHamiltonDist, \
    default: PBErrInvalidPolymorphism), \
  VecFloat2D*: _Generic(VecB, \
    VecFloat2D*: _VecFloatHamiltonDist2D, \
    const VecFloat2D*: _VecFloatHamiltonDist2D, \
    default: PBErrInvalidPolymorphism), \
  VecFloat3D*: _Generic(VecB, \
    VecFloat3D*: _VecFloatHamiltonDist3D, \
    const VecFloat3D*: _VecFloatHamiltonDist3D, \
    default: PBErrInvalidPolymorphism), \
  VecShort*: _Generic(VecB, \
    VecShort*: _VecShortHamiltonDist,\
    const VecShort*: _VecShortHamiltonDist,\
    default: PBErrInvalidPolymorphism), \
  VecShort2D*: _Generic(VecB, \
    VecShort2D*: _VecShortHamiltonDist2D,\
    const VecShort2D*: _VecShortHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  VecShort3D*: _Generic(VecB, \
    VecShort3D*: _VecShortHamiltonDist3D,\
    const VecShort3D*: _VecShortHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  VecShort4D*: _Generic(VecB, \
    VecShort4D*: _VecShortHamiltonDist4D,\
    const VecShort4D*: _VecShortHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  VecLong*: _Generic(VecB, \
```

```
    VecLong*: _VecLongHamiltonDist,\
    const VecLong*: _VecLongHamiltonDist,\
    default: PBErrInvalidPolymorphism), \
  VecLong2D*: _Generic(VecB, \
    VecLong2D*: _VecLongHamiltonDist2D,\
    const VecLong2D*: _VecLongHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  VecLong3D*: _Generic(VecB, \
    VecLong3D*: _VecLongHamiltonDist3D,\
    const VecLong3D*: _VecLongHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  VecLong4D*: _Generic(VecB, \
    VecLong4D*: _VecLongHamiltonDist4D,\
    const VecLong4D*: _VecLongHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  const VecFloat*: _Generic(VecB, \
    VecFloat*: _VecFloatHamiltonDist, \
    const VecFloat*: _VecFloatHamiltonDist, \
    default: PBErrInvalidPolymorphism), \
  const VecFloat2D*: _Generic(VecB, \
    VecFloat2D*: _VecFloatHamiltonDist2D, \
    const VecFloat2D*: _VecFloatHamiltonDist2D, \
    default: PBErrInvalidPolymorphism), \
  const VecFloat3D*: _Generic(VecB, \
    VecFloat3D*: _VecFloatHamiltonDist3D, \
    const VecFloat3D*: _VecFloatHamiltonDist3D, \
    default: PBErrInvalidPolymorphism), \
  const VecShort*: _Generic(VecB, \
    VecShort*: _VecShortHamiltonDist,\
    const VecShort*: _VecShortHamiltonDist,\
    default: PBErrInvalidPolymorphism), \
  const VecShort2D*: _Generic(VecB, \
    VecShort2D*: _VecShortHamiltonDist2D,\
    const VecShort2D*: _VecShortHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  const VecShort3D*: _Generic(VecB, \
    VecShort3D*: _VecShortHamiltonDist3D,\
    const VecShort3D*: _VecShortHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  const VecShort4D*: _Generic(VecB, \
    VecShort4D*: _VecShortHamiltonDist4D,\
    const VecShort4D*: _VecShortHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  const VecLong*: _Generic(VecB, \
    VecLong*: _VecLongHamiltonDist,\
    const VecLong*: _VecLongHamiltonDist,\
    default: PBErrInvalidPolymorphism), \
  const VecLong2D*: _Generic(VecB, \
    VecLong2D*: _VecLongHamiltonDist2D,\
    const VecLong2D*: _VecLongHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  const VecLong3D*: _Generic(VecB, \
    VecLong3D*: _VecLongHamiltonDist3D,\
    const VecLong3D*: _VecLongHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  const VecLong4D*: _Generic(VecB, \
    VecLong4D*: _VecLongHamiltonDist4D,\
    const VecLong4D*: _VecLongHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(VecA, VecB)

#define VecPixelDist(VecA, VecB) _Generic(VecA, \
```

```
VecFloat*: _Generic(VecB, \
  VecFloat*: _VecFloatPixelDist, \
  const VecFloat*: _VecFloatPixelDist, \
  default: PBErrInvalidPolymorphism), \
VecFloat2D*: _Generic(VecB, \
  VecFloat2D*: _VecFloatPixelDist2D, \
  const VecFloat2D*: _VecFloatPixelDist2D, \
  default: PBErrInvalidPolymorphism), \
VecFloat3D*: _Generic(VecB, \
  VecFloat3D*: _VecFloatPixelDist3D, \
  const VecFloat3D*: _VecFloatPixelDist3D, \
  default: PBErrInvalidPolymorphism), \
VecShort*: _Generic(VecB, \
  VecShort*: _VecShortHamiltonDist,\
  const VecShort*: _VecShortHamiltonDist,\
  default: PBErrInvalidPolymorphism), \
VecShort2D*: _Generic(VecB, \
  VecShort2D*: _VecShortHamiltonDist2D,\
  const VecShort2D*: _VecShortHamiltonDist2D,\
  default: PBErrInvalidPolymorphism), \
VecShort3D*: _Generic(VecB, \
  VecShort3D*: _VecShortHamiltonDist3D,\
  const VecShort3D*: _VecShortHamiltonDist3D,\
  default: PBErrInvalidPolymorphism), \
VecShort4D*: _Generic(VecB, \
  VecShort4D*: _VecShortHamiltonDist4D,\
  const VecShort4D*: _VecShortHamiltonDist4D,\
  default: PBErrInvalidPolymorphism), \
VecLong*: _Generic(VecB, \
  VecLong*: _VecLongHamiltonDist,\
  const VecLong*: _VecLongHamiltonDist,\
  default: PBErrInvalidPolymorphism), \
VecLong2D*: _Generic(VecB, \
  VecLong2D*: _VecLongHamiltonDist2D,\
  const VecLong2D*: _VecLongHamiltonDist2D,\
  default: PBErrInvalidPolymorphism), \
VecLong3D*: _Generic(VecB, \
  VecLong3D*: _VecLongHamiltonDist3D,\
  const VecLong3D*: _VecLongHamiltonDist3D,\
  default: PBErrInvalidPolymorphism), \
VecLong4D*: _Generic(VecB, \
  VecLong4D*: _VecLongHamiltonDist4D,\
  const VecLong4D*: _VecLongHamiltonDist4D,\
  default: PBErrInvalidPolymorphism), \
const VecFloat*: _Generic(VecB, \
  VecFloat*: _VecFloatPixelDist, \
  const VecFloat*: _VecFloatPixelDist, \
  default: PBErrInvalidPolymorphism), \
const VecFloat2D*: _Generic(VecB, \
  VecFloat2D*: _VecFloatPixelDist2D, \
  const VecFloat2D*: _VecFloatPixelDist2D, \
  default: PBErrInvalidPolymorphism), \
const VecFloat3D*: _Generic(VecB, \
  VecFloat3D*: _VecFloatPixelDist3D, \
  const VecFloat3D*: _VecFloatPixelDist3D, \
  default: PBErrInvalidPolymorphism), \
const VecShort*: _Generic(VecB, \
  VecShort*: _VecShortHamiltonDist,\
  const VecShort*: _VecShortHamiltonDist,\
  default: PBErrInvalidPolymorphism), \
const VecShort2D*: _Generic(VecB, \
  VecShort2D*: _VecShortHamiltonDist2D,\
```

```
    const VecShort2D*: _VecShortHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  const VecShort3D*: _Generic(VecB, \
    VecShort3D*: _VecShortHamiltonDist3D,\
    const VecShort3D*: _VecShortHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  const VecShort4D*: _Generic(VecB, \
    VecShort4D*: _VecShortHamiltonDist4D,\
    const VecShort4D*: _VecShortHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  const VecLong*: _Generic(VecB, \
    VecLong*: _VecLongHamiltonDist,\
    const VecLong*: _VecLongHamiltonDist,\
    default: PBErrInvalidPolymorphism), \
  const VecLong2D*: _Generic(VecB, \
    VecLong2D*: _VecLongHamiltonDist2D,\
    const VecLong2D*: _VecLongHamiltonDist2D,\
    default: PBErrInvalidPolymorphism), \
  const VecLong3D*: _Generic(VecB, \
    VecLong3D*: _VecLongHamiltonDist3D,\
    const VecLong3D*: _VecLongHamiltonDist3D,\
    default: PBErrInvalidPolymorphism), \
  const VecLong4D*: _Generic(VecB, \
    VecLong4D*: _VecLongHamiltonDist4D,\
    const VecLong4D*: _VecLongHamiltonDist4D,\
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(VecA, VecB)

#define VecIsEqual(VecA, VecB) _Generic(VecA, \
  VecFloat*: _Generic(VecB, \
    VecFloat*: _VecFloatIsEqual, \
    VecFloat2D*: _VecFloatIsEqual, \
    VecFloat3D*: _VecFloatIsEqual, \
    VecFloat4D*: _VecFloatIsEqual, \
    const VecFloat*: _VecFloatIsEqual, \
    const VecFloat2D*: _VecFloatIsEqual, \
    const VecFloat3D*: _VecFloatIsEqual, \
    const VecFloat4D*: _VecFloatIsEqual, \
    default: PBErrInvalidPolymorphism), \
  VecFloat2D*: _Generic(VecB, \
    VecFloat*: _VecFloatIsEqual, \
    VecFloat2D*: _VecFloatIsEqual, \
    const VecFloat*: _VecFloatIsEqual, \
    const VecFloat2D*: _VecFloatIsEqual, \
    default: PBErrInvalidPolymorphism), \
  VecFloat3D*: _Generic(VecB, \
    VecFloat*: _VecFloatIsEqual, \
    VecFloat3D*: _VecFloatIsEqual, \
    const VecFloat*: _VecFloatIsEqual, \
    const VecFloat3D*: _VecFloatIsEqual, \
    default: PBErrInvalidPolymorphism), \
  VecFloat4D*: _Generic(VecB, \
    VecFloat*: _VecFloatIsEqual, \
    VecFloat4D*: _VecFloatIsEqual, \
    const VecFloat*: _VecFloatIsEqual, \
    const VecFloat4D*: _VecFloatIsEqual, \
    default: PBErrInvalidPolymorphism), \
  VecShort*: _Generic(VecB, \
    VecShort*: _VecShortIsEqual,\
    VecShort2D*: _VecShortIsEqual,\
    VecShort3D*: _VecShortIsEqual,\
    VecShort4D*: _VecShortIsEqual,\
```

```
  const VecShort*: _VecShortIsEqual,\
  const VecShort2D*: _VecShortIsEqual,\
  const VecShort3D*: _VecShortIsEqual,\
  const VecShort4D*: _VecShortIsEqual,\
  default: PBErrInvalidPolymorphism), \
VecShort2D*: _Generic(VecB, \
  VecShort*: _VecShortIsEqual,\
  VecShort2D*: _VecShortIsEqual,\
  const VecShort*: _VecShortIsEqual,\
  const VecShort2D*: _VecShortIsEqual,\
  default: PBErrInvalidPolymorphism), \
VecShort3D*: _Generic(VecB, \
  VecShort*: _VecShortIsEqual,\
  VecShort3D*: _VecShortIsEqual,\
  const VecShort*: _VecShortIsEqual,\
  const VecShort3D*: _VecShortIsEqual,\
  default: PBErrInvalidPolymorphism), \
VecShort4D*: _Generic(VecB, \
  VecShort*: _VecShortIsEqual,\
  VecShort4D*: _VecShortIsEqual,\
  const VecShort*: _VecShortIsEqual,\
  const VecShort4D*: _VecShortIsEqual,\
  default: PBErrInvalidPolymorphism), \
VecLong*: _Generic(VecB, \
  VecLong*: _VecLongIsEqual,\
  VecLong2D*: _VecLongIsEqual,\
  VecLong3D*: _VecLongIsEqual,\
  VecLong4D*: _VecLongIsEqual,\
  const VecLong*: _VecLongIsEqual,\
  const VecLong2D*: _VecLongIsEqual,\
  const VecLong3D*: _VecLongIsEqual,\
  const VecLong4D*: _VecLongIsEqual,\
  default: PBErrInvalidPolymorphism), \
VecLong2D*: _Generic(VecB, \
  VecLong*: _VecLongIsEqual,\
  VecLong2D*: _VecLongIsEqual,\
  const VecLong*: _VecLongIsEqual,\
  const VecLong2D*: _VecLongIsEqual,\
  default: PBErrInvalidPolymorphism), \
VecLong3D*: _Generic(VecB, \
  VecLong*: _VecLongIsEqual,\
  VecLong3D*: _VecLongIsEqual,\
  const VecLong*: _VecLongIsEqual,\
  const VecLong3D*: _VecLongIsEqual,\
  default: PBErrInvalidPolymorphism), \
VecLong4D*: _Generic(VecB, \
  VecLong*: _VecLongIsEqual,\
  VecLong4D*: _VecLongIsEqual,\
  const VecLong*: _VecLongIsEqual,\
  const VecLong4D*: _VecLongIsEqual,\
  default: PBErrInvalidPolymorphism), \
const VecFloat*: _Generic(VecB, \
  VecFloat*: _VecFloatIsEqual, \
  VecFloat2D*: _VecFloatIsEqual, \
  VecFloat3D*: _VecFloatIsEqual, \
  VecFloat4D*: _VecFloatIsEqual, \
  const VecFloat*: _VecFloatIsEqual, \
  const VecFloat2D*: _VecFloatIsEqual, \
  const VecFloat3D*: _VecFloatIsEqual, \
  const VecFloat4D*: _VecFloatIsEqual, \
  default: PBErrInvalidPolymorphism), \
const VecFloat2D*: _Generic(VecB, \
```

```
  VecFloat*: _VecFloatIsEqual, \
  VecFloat2D*: _VecFloatIsEqual, \
  const VecFloat*: _VecFloatIsEqual, \
  const VecFloat2D*: _VecFloatIsEqual, \
  default: PBErrInvalidPolymorphism), \
const VecFloat3D*: _Generic(VecB, \
  VecFloat*: _VecFloatIsEqual, \
  VecFloat3D*: _VecFloatIsEqual, \
  const VecFloat*: _VecFloatIsEqual, \
  const VecFloat3D*: _VecFloatIsEqual, \
  default: PBErrInvalidPolymorphism), \
const VecFloat4D*: _Generic(VecB, \
  VecFloat*: _VecFloatIsEqual, \
  VecFloat4D*: _VecFloatIsEqual, \
  const VecFloat*: _VecFloatIsEqual, \
  const VecFloat4D*: _VecFloatIsEqual, \
  default: PBErrInvalidPolymorphism), \
const VecShort*: _Generic(VecB, \
  VecShort*: _VecShortIsEqual,\
  VecShort2D*: _VecShortIsEqual,\
  VecShort3D*: _VecShortIsEqual,\
  VecShort4D*: _VecShortIsEqual,\
  const VecShort*: _VecShortIsEqual,\
  const VecShort2D*: _VecShortIsEqual,\
  const VecShort3D*: _VecShortIsEqual,\
  const VecShort4D*: _VecShortIsEqual,\
  default: PBErrInvalidPolymorphism), \
const VecShort2D*: _Generic(VecB, \
  VecShort*: _VecShortIsEqual,\
  VecShort2D*: _VecShortIsEqual,\
  const VecShort*: _VecShortIsEqual,\
  const VecShort2D*: _VecShortIsEqual,\
  default: PBErrInvalidPolymorphism), \
const VecShort3D*: _Generic(VecB, \
  VecShort*: _VecShortIsEqual,\
  VecShort3D*: _VecShortIsEqual,\
  const VecShort*: _VecShortIsEqual,\
  const VecShort3D*: _VecShortIsEqual,\
  default: PBErrInvalidPolymorphism), \
const VecShort4D*: _Generic(VecB, \
  VecShort*: _VecShortIsEqual,\
  VecShort4D*: _VecShortIsEqual,\
  const VecShort*: _VecShortIsEqual,\
  const VecShort4D*: _VecShortIsEqual,\
  default: PBErrInvalidPolymorphism), \
const VecLong*: _Generic(VecB, \
  VecLong*: _VecLongIsEqual,\
  VecLong2D*: _VecLongIsEqual,\
  VecLong3D*: _VecLongIsEqual,\
  VecLong4D*: _VecLongIsEqual,\
  const VecLong*: _VecLongIsEqual,\
  const VecLong2D*: _VecLongIsEqual,\
  const VecLong3D*: _VecLongIsEqual,\
  const VecLong4D*: _VecLongIsEqual,\
  default: PBErrInvalidPolymorphism), \
const VecLong2D*: _Generic(VecB, \
  VecLong*: _VecLongIsEqual,\
  VecLong2D*: _VecLongIsEqual,\
  const VecLong*: _VecLongIsEqual,\
  const VecLong2D*: _VecLongIsEqual,\
  default: PBErrInvalidPolymorphism), \
const VecLong3D*: _Generic(VecB, \
```

```
    VecLong*: _VecLongIsEqual,\
    VecLong3D*: _VecLongIsEqual,\
    const VecLong*: _VecLongIsEqual,\
    const VecLong3D*: _VecLongIsEqual,\
    default: PBErrInvalidPolymorphism), \
  const VecLong4D*: _Generic(VecB, \
    VecLong*: _VecLongIsEqual,\
    VecLong4D*: _VecLongIsEqual,\
    const VecLong*: _VecLongIsEqual,\
    const VecLong4D*: _VecLongIsEqual,\
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)( \
    _Generic(VecA,  \
      VecFloat2D*: (const VecFloat*)(VecA), \
      VecFloat3D*: (const VecFloat*)(VecA), \
      VecFloat4D*: (const VecFloat*)(VecA), \
      VecShort2D*: (const VecShort*)(VecA), \
      VecShort3D*: (const VecShort*)(VecA), \
      VecShort4D*: (const VecShort*)(VecA), \
      VecLong2D*: (const VecLong*)(VecA), \
      VecLong3D*: (const VecLong*)(VecA), \
      VecLong4D*: (const VecLong*)(VecA), \
      const VecFloat2D*: (const VecFloat*)(VecA), \
      const VecFloat3D*: (const VecFloat*)(VecA), \
      const VecFloat4D*: (const VecFloat*)(VecA), \
      const VecShort2D*: (const VecShort*)(VecA), \
      const VecShort3D*: (const VecShort*)(VecA), \
      const VecShort4D*: (const VecShort*)(VecA), \
      const VecLong2D*: (const VecLong*)(VecA), \
      const VecLong3D*: (const VecLong*)(VecA), \
      const VecLong4D*: (const VecLong*)(VecA), \
      default: VecA),  \
    _Generic(VecB,  \
      VecFloat2D*: (const VecFloat*)(VecB), \
      VecFloat3D*: (const VecFloat*)(VecB), \
      VecFloat4D*: (const VecFloat*)(VecB), \
      VecShort2D*: (const VecShort*)(VecB), \
      VecShort3D*: (const VecShort*)(VecB), \
      VecShort4D*: (const VecShort*)(VecB), \
      VecLong2D*: (const VecLong*)(VecB), \
      VecLong3D*: (const VecLong*)(VecB), \
      VecLong4D*: (const VecLong*)(VecB), \
      const VecFloat2D*: (const VecFloat*)(VecB), \
      const VecFloat3D*: (const VecFloat*)(VecB), \
      const VecFloat4D*: (const VecFloat*)(VecB), \
      const VecShort2D*: (const VecShort*)(VecB), \
      const VecShort3D*: (const VecShort*)(VecB), \
      const VecShort4D*: (const VecShort*)(VecB), \
      const VecLong2D*: (const VecLong*)(VecB), \
      const VecLong3D*: (const VecLong*)(VecB), \
      const VecLong4D*: (const VecLong*)(VecB), \
      default: VecB))

#define VecOp(VecA, CoeffA, VecB, CoeffB) _Generic(VecA, \
  VecFloat*: _Generic(VecB, \
    VecFloat*: _VecFloatOp, \
    const VecFloat*: _VecFloatOp, \
    default: PBErrInvalidPolymorphism), \
  VecFloat2D*: _Generic(VecB, \
    VecFloat2D*: _VecFloatOp2D, \
    const VecFloat2D*: _VecFloatOp2D, \
    default: PBErrInvalidPolymorphism), \
```

```
  VecFloat3D*: _Generic(VecB, \
    VecFloat3D*: _VecFloatOp3D, \
    const VecFloat3D*: _VecFloatOp3D, \
    default: PBErrInvalidPolymorphism), \
  VecFloat4D*: _Generic(VecB, \
    VecFloat4D*: _VecFloatOp4D, \
    const VecFloat4D*: _VecFloatOp4D, \
    default: PBErrInvalidPolymorphism), \
  VecShort*: _Generic(VecB, \
    VecShort*: _VecShortOp, \
    const VecShort*: _VecShortOp, \
    default: PBErrInvalidPolymorphism), \
  VecShort2D*: _Generic(VecB, \
    VecShort2D*: _VecShortOp2D, \
    const VecShort2D*: _VecShortOp2D, \
    default: PBErrInvalidPolymorphism), \
  VecShort3D*: _Generic(VecB, \
    VecShort3D*: _VecShortOp3D, \
    const VecShort3D*: _VecShortOp3D, \
    default: PBErrInvalidPolymorphism), \
  VecShort4D*: _Generic(VecB, \
    VecShort4D*: _VecShortOp4D, \
    const VecShort4D*: _VecShortOp4D, \
    default: PBErrInvalidPolymorphism), \
  VecLong*: _Generic(VecB, \
    VecLong*: _VecLongOp, \
    const VecLong*: _VecLongOp, \
    default: PBErrInvalidPolymorphism), \
  VecLong2D*: _Generic(VecB, \
    VecLong2D*: _VecLongOp2D, \
    const VecLong2D*: _VecLongOp2D, \
    default: PBErrInvalidPolymorphism), \
  VecLong3D*: _Generic(VecB, \
    VecLong3D*: _VecLongOp3D, \
    const VecLong3D*: _VecLongOp3D, \
    default: PBErrInvalidPolymorphism), \
  VecLong4D*: _Generic(VecB, \
    VecLong4D*: _VecLongOp4D, \
    const VecLong4D*: _VecLongOp4D, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(VecA, CoeffA, VecB, CoeffB)

#define VecGetOp(VecA, CoeffA, VecB, CoeffB) _Generic(VecA, \
  VecFloat*: _Generic(VecB, \
    VecFloat*: _VecFloatGetOp, \
    const VecFloat*: _VecFloatGetOp, \
    default: PBErrInvalidPolymorphism), \
  VecFloat2D*: _Generic(VecB, \
    VecFloat2D*: _VecFloatGetOp2D, \
    const VecFloat2D*: _VecFloatGetOp2D, \
    default: PBErrInvalidPolymorphism), \
  VecFloat3D*: _Generic(VecB, \
    VecFloat3D*: _VecFloatGetOp3D, \
    const VecFloat3D*: _VecFloatGetOp3D, \
    default: PBErrInvalidPolymorphism), \
  VecShort*: _Generic(VecB, \
    VecShort*: _VecShortGetOp, \
    const VecShort*: _VecShortGetOp, \
    default: PBErrInvalidPolymorphism), \
  VecShort2D*: _Generic(VecB, \
    VecShort2D*: _VecShortGetOp2D, \
    const VecShort2D*: _VecShortGetOp2D, \
```

```
  default: PBErrInvalidPolymorphism), \
VecShort3D*: _Generic(VecB, \
  VecShort3D*: _VecShortGetOp3D, \
  const VecShort3D*: _VecShortGetOp3D, \
  default: PBErrInvalidPolymorphism), \
VecShort4D*: _Generic(VecB, \
  VecShort4D*: _VecShortGetOp4D, \
  const VecShort4D*: _VecShortGetOp4D, \
  default: PBErrInvalidPolymorphism), \
VecLong*: _Generic(VecB, \
  VecLong*: _VecLongGetOp, \
  const VecLong*: _VecLongGetOp, \
  default: PBErrInvalidPolymorphism), \
VecLong2D*: _Generic(VecB, \
  VecLong2D*: _VecLongGetOp2D, \
  const VecLong2D*: _VecLongGetOp2D, \
  default: PBErrInvalidPolymorphism), \
VecLong3D*: _Generic(VecB, \
  VecLong3D*: _VecLongGetOp3D, \
  const VecLong3D*: _VecLongGetOp3D, \
  default: PBErrInvalidPolymorphism), \
VecLong4D*: _Generic(VecB, \
  VecLong4D*: _VecLongGetOp4D, \
  const VecLong4D*: _VecLongGetOp4D, \
  default: PBErrInvalidPolymorphism), \
const VecFloat*: _Generic(VecB, \
  VecFloat*: _VecFloatGetOp, \
  const VecFloat*: _VecFloatGetOp, \
  default: PBErrInvalidPolymorphism), \
const VecFloat2D*: _Generic(VecB, \
  VecFloat2D*: _VecFloatGetOp2D, \
  const VecFloat2D*: _VecFloatGetOp2D, \
  default: PBErrInvalidPolymorphism), \
const VecFloat3D*: _Generic(VecB, \
  VecFloat3D*: _VecFloatGetOp3D, \
  const VecFloat3D*: _VecFloatGetOp3D, \
  default: PBErrInvalidPolymorphism), \
const VecShort*: _Generic(VecB, \
  VecShort*: _VecShortGetOp, \
  const VecShort*: _VecShortGetOp, \
  default: PBErrInvalidPolymorphism), \
const VecShort2D*: _Generic(VecB, \
  VecShort2D*: _VecShortGetOp2D, \
  const VecShort2D*: _VecShortGetOp2D, \
  default: PBErrInvalidPolymorphism), \
const VecShort3D*: _Generic(VecB, \
  VecShort3D*: _VecShortGetOp3D, \
  const VecShort3D*: _VecShortGetOp3D, \
  default: PBErrInvalidPolymorphism), \
const VecShort4D*: _Generic(VecB, \
  VecShort4D*: _VecShortGetOp4D, \
  const VecShort4D*: _VecShortGetOp4D, \
  default: PBErrInvalidPolymorphism), \
const VecLong*: _Generic(VecB, \
  VecLong*: _VecLongGetOp, \
  const VecLong*: _VecLongGetOp, \
  default: PBErrInvalidPolymorphism), \
const VecLong2D*: _Generic(VecB, \
  VecLong2D*: _VecLongGetOp2D, \
  const VecLong2D*: _VecLongGetOp2D, \
  default: PBErrInvalidPolymorphism), \
const VecLong3D*: _Generic(VecB, \
```

```
    VecLong3D*: _VecLongGetOp3D, \
    const VecLong3D*: _VecLongGetOp3D, \
    default: PBErrInvalidPolymorphism), \
  const VecLong4D*: _Generic(VecB, \
    VecLong4D*: _VecLongGetOp4D, \
    const VecLong4D*: _VecLongGetOp4D, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(VecA, CoeffA, VecB, CoeffB)

#define VecHadamardProd(VecA, VecB) _Generic(VecA, \
  VecFloat*: _Generic(VecB, \
    VecFloat*: _VecFloatHadamardProd, \
    const VecFloat*: _VecFloatHadamardProd, \
    default: PBErrInvalidPolymorphism), \
  VecFloat2D*: _Generic(VecB, \
    VecFloat2D*: _VecFloatHadamardProd2D, \
    const VecFloat2D*: _VecFloatHadamardProd2D, \
    default: PBErrInvalidPolymorphism), \
  VecFloat3D*: _Generic(VecB, \
    VecFloat3D*: _VecFloatHadamardProd3D, \
    const VecFloat3D*: _VecFloatHadamardProd3D, \
    default: PBErrInvalidPolymorphism), \
  VecShort*: _Generic(VecB, \
    VecShort*: _VecShortHadamardProd, \
    const VecShort*: _VecShortHadamardProd, \
    default: PBErrInvalidPolymorphism), \
  VecShort2D*: _Generic(VecB, \
    VecShort2D*: _VecShortHadamardProd2D, \
    const VecShort2D*: _VecShortHadamardProd2D, \
    default: PBErrInvalidPolymorphism), \
  VecShort3D*: _Generic(VecB, \
    VecShort3D*: _VecShortHadamardProd3D, \
    const VecShort3D*: _VecShortHadamardProd3D, \
    default: PBErrInvalidPolymorphism), \
  VecShort4D*: _Generic(VecB, \
    VecShort4D*: _VecShortHadamardProd4D, \
    const VecShort4D*: _VecShortHadamardProd4D, \
    default: PBErrInvalidPolymorphism), \
  VecLong*: _Generic(VecB, \
    VecLong*: _VecLongHadamardProd, \
    const VecLong*: _VecLongHadamardProd, \
    default: PBErrInvalidPolymorphism), \
  VecLong2D*: _Generic(VecB, \
    VecLong2D*: _VecLongHadamardProd2D, \
    const VecLong2D*: _VecLongHadamardProd2D, \
    default: PBErrInvalidPolymorphism), \
  VecLong3D*: _Generic(VecB, \
    VecLong3D*: _VecLongHadamardProd3D, \
    const VecLong3D*: _VecLongHadamardProd3D, \
    default: PBErrInvalidPolymorphism), \
  VecLong4D*: _Generic(VecB, \
    VecLong4D*: _VecLongHadamardProd4D, \
    const VecLong4D*: _VecLongHadamardProd4D, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(VecA, VecB)

#define VecGetHadamardProd(VecA, VecB) _Generic(VecA, \
  VecFloat*: _Generic(VecB, \
    VecFloat*: _VecFloatGetHadamardProd, \
    const VecFloat*: _VecFloatGetHadamardProd, \
    default: PBErrInvalidPolymorphism), \
  VecFloat2D*: _Generic(VecB, \
```

```
  VecFloat2D*: _VecFloatGetHadamardProd2D, \
  const VecFloat2D*: _VecFloatGetHadamardProd2D, \
  default: PBErrInvalidPolymorphism), \
VecFloat3D*: _Generic(VecB, \
  VecFloat3D*: _VecFloatGetHadamardProd3D, \
  const VecFloat3D*: _VecFloatGetHadamardProd3D, \
  default: PBErrInvalidPolymorphism), \
VecShort*: _Generic(VecB, \
  VecShort*: _VecShortGetHadamardProd, \
  const VecShort*: _VecShortGetHadamardProd, \
  default: PBErrInvalidPolymorphism), \
VecShort2D*: _Generic(VecB, \
  VecShort2D*: _VecShortGetHadamardProd2D, \
  const VecShort2D*: _VecShortGetHadamardProd2D, \
  default: PBErrInvalidPolymorphism), \
VecShort3D*: _Generic(VecB, \
  VecShort3D*: _VecShortGetHadamardProd3D, \
  const VecShort3D*: _VecShortGetHadamardProd3D, \
  default: PBErrInvalidPolymorphism), \
VecShort4D*: _Generic(VecB, \
  VecShort4D*: _VecShortGetHadamardProd4D, \
  const VecShort4D*: _VecShortGetHadamardProd4D, \
  default: PBErrInvalidPolymorphism), \
VecLong*: _Generic(VecB, \
  VecLong*: _VecLongGetHadamardProd, \
  const VecLong*: _VecLongGetHadamardProd, \
  default: PBErrInvalidPolymorphism), \
VecLong2D*: _Generic(VecB, \
  VecLong2D*: _VecLongGetHadamardProd2D, \
  const VecLong2D*: _VecLongGetHadamardProd2D, \
  default: PBErrInvalidPolymorphism), \
VecLong3D*: _Generic(VecB, \
  VecLong3D*: _VecLongGetHadamardProd3D, \
  const VecLong3D*: _VecLongGetHadamardProd3D, \
  default: PBErrInvalidPolymorphism), \
VecLong4D*: _Generic(VecB, \
  VecLong4D*: _VecLongGetHadamardProd4D, \
  const VecLong4D*: _VecLongGetHadamardProd4D, \
  default: PBErrInvalidPolymorphism), \
const VecFloat*: _Generic(VecB, \
  VecFloat*: _VecFloatGetHadamardProd, \
  const VecFloat*: _VecFloatGetHadamardProd, \
  default: PBErrInvalidPolymorphism), \
const VecFloat2D*: _Generic(VecB, \
  VecFloat2D*: _VecFloatGetHadamardProd2D, \
  const VecFloat2D*: _VecFloatGetHadamardProd2D, \
  default: PBErrInvalidPolymorphism), \
const VecFloat3D*: _Generic(VecB, \
  VecFloat3D*: _VecFloatGetHadamardProd3D, \
  const VecFloat3D*: _VecFloatGetHadamardProd3D, \
  default: PBErrInvalidPolymorphism), \
const VecShort*: _Generic(VecB, \
  VecShort*: _VecShortGetHadamardProd, \
  const VecShort*: _VecShortGetHadamardProd, \
  default: PBErrInvalidPolymorphism), \
const VecShort2D*: _Generic(VecB, \
  VecShort2D*: _VecShortGetHadamardProd2D, \
  const VecShort2D*: _VecShortGetHadamardProd2D, \
  default: PBErrInvalidPolymorphism), \
const VecShort3D*: _Generic(VecB, \
  VecShort3D*: _VecShortGetHadamardProd3D, \
  const VecShort3D*: _VecShortGetHadamardProd3D, \
```

```
    default: PBErrInvalidPolymorphism), \
  const VecShort4D*: _Generic(VecB, \
    VecShort4D*: _VecShortGetHadamardProd4D, \
    const VecShort4D*: _VecShortGetHadamardProd4D, \
    default: PBErrInvalidPolymorphism), \
  const VecLong*: _Generic(VecB, \
    VecLong*: _VecLongGetHadamardProd, \
    const VecLong*: _VecLongGetHadamardProd, \
    default: PBErrInvalidPolymorphism), \
  const VecLong2D*: _Generic(VecB, \
    VecLong2D*: _VecLongGetHadamardProd2D, \
    const VecLong2D*: _VecLongGetHadamardProd2D, \
    default: PBErrInvalidPolymorphism), \
  const VecLong3D*: _Generic(VecB, \
    VecLong3D*: _VecLongGetHadamardProd3D, \
    const VecLong3D*: _VecLongGetHadamardProd3D, \
    default: PBErrInvalidPolymorphism), \
  const VecLong4D*: _Generic(VecB, \
    VecLong4D*: _VecLongGetHadamardProd4D, \
    const VecLong4D*: _VecLongGetHadamardProd4D, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(VecA, VecB)

#define VecScale(Vec, Scale) _Generic(Vec, \
  VecFloat*: _VecFloatScale, \
  VecFloat2D*: _VecFloatScale2D, \
  VecFloat3D*: _VecFloatScale3D, \
  VecFloat4D*: _VecFloatScale4D, \
  default: PBErrInvalidPolymorphism)(Vec, Scale)

#define VecGetScale(Vec, Scale) _Generic(Vec, \
  VecFloat*: _VecFloatGetScale, \
  const VecFloat*: _VecFloatGetScale, \
  VecFloat2D*: _VecFloatGetScale2D, \
  const VecFloat2D*: _VecFloatGetScale2D, \
  VecFloat3D*: _VecFloatGetScale3D, \
  const VecFloat3D*: _VecFloatGetScale3D, \
  default: PBErrInvalidPolymorphism)(Vec, Scale)

#define VecRot(Vec, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatRot2D, \
  VecFloat2D*: _VecFloatRot2D, \
  default: PBErrInvalidPolymorphism)((VecFloat2D*)(Vec), Theta)

#define VecGetRot(Vec, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatGetRot2D, \
  const VecFloat*: _VecFloatGetRot2D, \
  VecFloat2D*: _VecFloatGetRot2D, \
  const VecFloat2D*: _VecFloatGetRot2D, \
  default: PBErrInvalidPolymorphism)((const VecFloat2D*)(Vec), Theta)

#define VecRotAxis(Vec, Axis, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatRotAxis, \
  VecFloat3D*: _VecFloatRotAxis, \
  default: PBErrInvalidPolymorphism)((VecFloat3D*)(Vec), \
    (VecFloat3D*)(Axis), Theta)

#define VecGetRotAxis(Vec, Axis, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatGetRotAxis, \
  const VecFloat*: _VecFloatGetRotAxis, \
  VecFloat3D*: _VecFloatGetRotAxis, \
  const VecFloat3D*: _VecFloatGetRotAxis, \
```

```
  default: PBErrInvalidPolymorphism)((const VecFloat3D*)(Vec), \
    (const VecFloat3D*)(Axis), Theta)

#define VecRotX(Vec, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatRotX, \
  VecFloat3D*: _VecFloatRotX, \
  default: PBErrInvalidPolymorphism)((VecFloat3D*)(Vec), Theta)

#define VecGetRotX(Vec, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatGetRotX, \
  const VecFloat*: _VecFloatGetRotX, \
  VecFloat3D*: _VecFloatGetRotX, \
  const VecFloat3D*: _VecFloatGetRotX, \
  default: PBErrInvalidPolymorphism)((const VecFloat3D*)(Vec), Theta)

#define VecRotY(Vec, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatRotY, \
  VecFloat3D*: _VecFloatRotY, \
  default: PBErrInvalidPolymorphism)((VecFloat3D*)(Vec), Theta)

#define VecGetRotY(Vec, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatGetRotY, \
  const VecFloat*: _VecFloatGetRotY, \
  VecFloat3D*: _VecFloatGetRotY, \
  const VecFloat3D*: _VecFloatGetRotY, \
  default: PBErrInvalidPolymorphism)((const VecFloat3D*)(Vec), Theta)

#define VecRotZ(Vec, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatRotZ, \
  VecFloat3D*: _VecFloatRotZ, \
  default: PBErrInvalidPolymorphism)((VecFloat3D*)(Vec), Theta)

#define VecGetRotZ(Vec, Theta) _Generic(Vec, \
  VecFloat*: _VecFloatGetRotZ, \
  const VecFloat*: _VecFloatGetRotZ, \
  VecFloat3D*: _VecFloatGetRotZ, \
  const VecFloat3D*: _VecFloatGetRotZ, \
  default: PBErrInvalidPolymorphism)((const VecFloat3D*)(Vec), Theta)

#define VecDotProd(VecA, VecB) _Generic(VecA, \
  VecShort*: _VecShortDotProd,\
  const VecShort*: _VecShortDotProd,\
  VecShort2D*: _VecShortDotProd2D,\
  const VecShort2D*: _VecShortDotProd2D,\
  VecShort3D*: _VecShortDotProd3D,\
  const VecShort3D*: _VecShortDotProd3D,\
  VecShort4D*: _VecShortDotProd4D,\
  const VecShort4D*: _VecShortDotProd4D,\
  VecLong*: _VecLongDotProd,\
  const VecLong*: _VecLongDotProd,\
  VecLong2D*: _VecLongDotProd2D,\
  const VecLong2D*: _VecLongDotProd2D,\
  VecLong3D*: _VecLongDotProd3D,\
  const VecLong3D*: _VecLongDotProd3D,\
  VecLong4D*: _VecLongDotProd4D,\
  const VecLong4D*: _VecLongDotProd4D,\
  VecFloat*: _VecFloatDotProd, \
  const VecFloat*: _VecFloatDotProd, \
  VecFloat2D*: _VecFloatDotProd2D, \
  const VecFloat2D*: _VecFloatDotProd2D, \
  VecFloat3D*: _VecFloatDotProd3D, \
  const VecFloat3D*: _VecFloatDotProd3D, \
```

```
  default: PBErrInvalidPolymorphism) (VecA, VecB) \

#define VecCrossProd(VecA, VecB) _Generic(VecA, \
  VecFloat*: _VecFloatGetCrossProd, \
  const VecFloat*: _VecFloatGetCrossProd, \
  VecFloat3D*: _VecFloatGetCrossProd3D, \
  const VecFloat3D*: _VecFloatGetCrossProd3D, \
  default: PBErrInvalidPolymorphism) (VecA, VecB) \

#define VecAngleTo(VecFrom, VecTo) _Generic(VecFrom, \
  VecFloat*: _VecFloatAngleTo2D, \
  const VecFloat*: _VecFloatAngleTo2D, \
  VecFloat2D*: _VecFloatAngleTo2D, \
  const VecFloat2D*: _VecFloatAngleTo2D, \
  default: PBErrInvalidPolymorphism)((const VecFloat2D*)(VecFrom), \
    (const VecFloat2D*)(VecTo))

#define VecStep(Vec, VecBound) _Generic(Vec, \
  VecShort*: _VecShortStep, \
  VecShort2D*: _VecShortStep, \
  VecShort3D*: _VecShortStep, \
  VecShort4D*: _VecShortStep, \
  VecLong*: _VecLongStep, \
  VecLong2D*: _VecLongStep, \
  VecLong3D*: _VecLongStep, \
  VecLong4D*: _VecLongStep, \
  default: PBErrInvalidPolymorphism)(_Generic(Vec, \
    VecShort*: (VecShort*)(Vec), \
    VecShort2D*: (VecShort*)(Vec), \
    VecShort3D*: (VecShort*)(Vec), \
    VecShort4D*: (VecShort*)(Vec), \
    VecLong*: (VecLong*)(Vec), \
    VecLong2D*: (VecLong*)(Vec), \
    VecLong3D*: (VecLong*)(Vec), \
    VecLong4D*: (VecLong*)(Vec)), _Generic(VecBound, \
    VecShort*: (const VecShort*)(VecBound), \
    VecShort2D*: (const VecShort*)(VecBound), \
    VecShort3D*: (const VecShort*)(VecBound), \
    VecShort4D*: (const VecShort*)(VecBound), \
    const VecShort*: (const VecShort*)(VecBound), \
    const VecShort2D*: (const VecShort*)(VecBound), \
    const VecShort3D*: (const VecShort*)(VecBound), \
    const VecShort4D*: (const VecShort*)(VecBound), \
    VecLong*: (const VecLong*)(VecBound), \
    VecLong2D*: (const VecLong*)(VecBound), \
    VecLong3D*: (const VecLong*)(VecBound), \
    VecLong4D*: (const VecLong*)(VecBound), \
    const VecLong*: (const VecLong*)(VecBound), \
    const VecLong2D*: (const VecLong*)(VecBound), \
    const VecLong3D*: (const VecLong*)(VecBound), \
    const VecLong4D*: (const VecLong*)(VecBound)))

#define VecPStep(Vec, VecBound) _Generic(Vec, \
  VecShort*: _VecShortPStep, \
  VecShort2D*: _VecShortPStep, \
  VecShort3D*: _VecShortPStep, \
  VecShort4D*: _VecShortPStep, \
  VecLong*: _VecLongPStep, \
  VecLong2D*: _VecLongPStep, \
  VecLong3D*: _VecLongPStep, \
  VecLong4D*: _VecLongPStep, \
  default: PBErrInvalidPolymorphism)(_Generic(Vec, \
```

```
    VecShort*: (VecShort*)(Vec), \
    VecShort2D*: (VecShort*)(Vec), \
    VecShort3D*: (VecShort*)(Vec), \
    VecShort4D*: (VecShort*)(Vec), \
    VecLong*: (VecLong*)(Vec), \
    VecLong2D*: (VecLong*)(Vec), \
    VecLong3D*: (VecLong*)(Vec), \
    VecLong4D*: (VecLong*)(Vec)), _Generic(VecBound, \
    VecShort*: (const VecShort*)(VecBound), \
    VecShort2D*: (const VecShort*)(VecBound), \
    VecShort3D*: (const VecShort*)(VecBound), \
    VecShort4D*: (const VecShort*)(VecBound), \
    const VecShort*: (const VecShort*)(VecBound), \
    const VecShort2D*: (const VecShort*)(VecBound), \
    const VecShort3D*: (const VecShort*)(VecBound), \
    const VecShort4D*: (const VecShort*)(VecBound), \
    VecLong*: (const VecLong*)(VecBound), \
    VecLong2D*: (const VecLong*)(VecBound), \
    VecLong3D*: (const VecLong*)(VecBound), \
    VecLong4D*: (const VecLong*)(VecBound), \
    const VecLong*: (const VecLong*)(VecBound), \
    const VecLong2D*: (const VecLong*)(VecBound), \
    const VecLong3D*: (const VecLong*)(VecBound), \
    const VecLong4D*: (const VecLong*)(VecBound)))

#define VecShiftStep(Vec, VecFrom, VecTo) _Generic(Vec, \
  VecShort*: _VecShortShiftStep, \
  VecShort2D*: _VecShortShiftStep, \
  VecShort3D*: _VecShortShiftStep, \
  VecShort4D*: _VecShortShiftStep, \
  VecLong*: _VecLongShiftStep, \
  VecLong2D*: _VecLongShiftStep, \
  VecLong3D*: _VecLongShiftStep, \
  VecLong4D*: _VecLongShiftStep, \
  default: PBErrInvalidPolymorphism)(_Generic(Vec, \
    VecShort*: (VecShort*)(Vec), \
    VecShort2D*: (VecShort*)(Vec), \
    VecShort3D*: (VecShort*)(Vec), \
    VecShort4D*: (VecShort*)(Vec), \
    VecLong*: (VecLong*)(Vec), \
    VecLong2D*: (VecLong*)(Vec), \
    VecLong3D*: (VecLong*)(Vec), \
    VecLong4D*: (VecLong*)(Vec)), _Generic(VecFrom, \
    VecShort*: (const VecShort*)(VecFrom), \
    VecShort2D*: (const VecShort*)(VecFrom), \
    VecShort3D*: (const VecShort*)(VecFrom), \
    VecShort4D*: (const VecShort*)(VecFrom), \
    const VecShort*: (const VecShort*)(VecFrom), \
    const VecShort2D*: (const VecShort*)(VecFrom), \
    const VecShort3D*: (const VecShort*)(VecFrom), \
    const VecShort4D*: (const VecShort*)(VecFrom), \
    VecLong*: (const VecLong*)(VecFrom), \
    VecLong2D*: (const VecLong*)(VecFrom), \
    VecLong3D*: (const VecLong*)(VecFrom), \
    VecLong4D*: (const VecLong*)(VecFrom), \
    const VecLong*: (const VecLong*)(VecFrom), \
    const VecLong2D*: (const VecLong*)(VecFrom), \
    const VecLong3D*: (const VecLong*)(VecFrom), \
    const VecLong4D*: (const VecLong*)(VecFrom)), _Generic(VecTo, \
    VecShort*: (const VecShort*)(VecTo), \
    VecShort2D*: (const VecShort*)(VecTo), \
    VecShort3D*: (const VecShort*)(VecTo), \
```

```
    VecShort4D*: (const VecShort*)(VecTo), \
    const VecShort*: (const VecShort*)(VecTo), \
    const VecShort2D*: (const VecShort*)(VecTo), \
    const VecShort3D*: (const VecShort*)(VecTo), \
    const VecShort4D*: (const VecShort*)(VecTo), \
    VecLong*: (const VecLong*)(VecTo), \
    VecLong2D*: (const VecLong*)(VecTo), \
    VecLong3D*: (const VecLong*)(VecTo), \
    VecLong4D*: (const VecLong*)(VecTo), \
    const VecLong*: (const VecLong*)(VecTo), \
    const VecLong2D*: (const VecLong*)(VecTo), \
    const VecLong3D*: (const VecLong*)(VecTo), \
    const VecLong4D*: (const VecLong*)(VecTo)))

#define VecPStepDelta(Vec, VecBound, VecDelta) _Generic(Vec, \
  VecShort*: _VecShortPStepDelta, \
  VecShort2D*: _VecShortPStepDelta, \
  VecShort3D*: _VecShortPStepDelta, \
  VecShort4D*: _VecShortPStepDelta, \
  VecLong*: _VecLongPStepDelta, \
  VecLong2D*: _VecLongPStepDelta, \
  VecLong3D*: _VecLongPStepDelta, \
  VecLong4D*: _VecLongPStepDelta, \
  default: PBErrInvalidPolymorphism)(_Generic(Vec, \
    VecShort*: (VecShort*)(Vec), \
    VecShort2D*: (VecShort*)(Vec), \
    VecShort3D*: (VecShort*)(Vec), \
    VecShort4D*: (VecShort*)(Vec), \
    VecLong*: (VecLong*)(Vec), \
    VecLong2D*: (VecLong*)(Vec), \
    VecLong3D*: (VecLong*)(Vec), \
    VecLong4D*: (VecLong*)(Vec)), _Generic(VecBound, \
    VecShort*: (const VecShort*)(VecBound), \
    VecShort2D*: (const VecShort*)(VecBound), \
    VecShort3D*: (const VecShort*)(VecBound), \
    VecShort4D*: (const VecShort*)(VecBound), \
    const VecShort*: (const VecShort*)(VecBound), \
    const VecShort2D*: (const VecShort*)(VecBound), \
    const VecShort3D*: (const VecShort*)(VecBound), \
    const VecShort4D*: (const VecShort*)(VecBound), \
    VecLong*: (const VecLong*)(VecBound), \
    VecLong2D*: (const VecLong*)(VecBound), \
    VecLong3D*: (const VecLong*)(VecBound), \
    VecLong4D*: (const VecLong*)(VecBound), \
    const VecLong*: (const VecLong*)(VecBound), \
    const VecLong2D*: (const VecLong*)(VecBound), \
    const VecLong3D*: (const VecLong*)(VecBound), \
    const VecLong4D*: (const VecLong*)(VecBound)), _Generic(VecDelta, \
    VecShort*: (const VecShort*)(VecDelta), \
    VecShort2D*: (const VecShort*)(VecDelta), \
    VecShort3D*: (const VecShort*)(VecDelta), \
    VecShort4D*: (const VecShort*)(VecDelta), \
    const VecShort*: (const VecShort*)(VecDelta), \
    const VecShort2D*: (const VecShort*)(VecDelta), \
    const VecShort3D*: (const VecShort*)(VecDelta), \
    const VecShort4D*: (const VecShort*)(VecDelta), \
    VecLong*: (const VecLong*)(VecDelta), \
    VecLong2D*: (const VecLong*)(VecDelta), \
    VecLong3D*: (const VecLong*)(VecDelta), \
    VecLong4D*: (const VecLong*)(VecDelta), \
    const VecLong*: (const VecLong*)(VecDelta), \
    const VecLong2D*: (const VecLong*)(VecDelta), \
```

```
    const VecLong3D*: (const VecLong*)(VecDelta), \
    const VecLong4D*: (const VecLong*)(VecDelta)))

#define VecGetMaxVal(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatGetMaxVal, \
  const VecFloat*: _VecFloatGetMaxVal, \
  VecFloat2D*: _VecFloatGetMaxVal, \
  const VecFloat2D*: _VecFloatGetMaxVal, \
  VecFloat3D*: _VecFloatGetMaxVal, \
  const VecFloat3D*: _VecFloatGetMaxVal, \
  VecShort*: _VecShortGetMaxVal, \
  const VecShort*: _VecShortGetMaxVal, \
  VecShort2D*: _VecShortGetMaxVal, \
  const VecShort2D*: _VecShortGetMaxVal, \
  VecShort3D*: _VecShortGetMaxVal, \
  const VecShort3D*: _VecShortGetMaxVal, \
  VecShort4D*: _VecShortGetMaxVal, \
  const VecShort4D*: _VecShortGetMaxVal, \
  VecLong*: _VecLongGetMaxVal, \
  const VecLong*: _VecLongGetMaxVal, \
  VecLong2D*: _VecLongGetMaxVal, \
  const VecLong2D*: _VecLongGetMaxVal, \
  VecLong3D*: _VecLongGetMaxVal, \
  const VecLong3D*: _VecLongGetMaxVal, \
  VecLong4D*: _VecLongGetMaxVal, \
  const VecLong4D*: _VecLongGetMaxVal, \
  default: PBErrInvalidPolymorphism) (_Generic(Vec, \
    VecFloat2D*: (const VecFloat*)(Vec), \
    const VecFloat2D*: (const VecFloat*)(Vec), \
    VecFloat3D*: (const VecFloat*)(Vec), \
    const VecFloat3D*: (const VecFloat*)(Vec), \
    VecShort2D*: (const VecShort*)(Vec), \
    const VecShort2D*: (const VecShort*)(Vec), \
    VecShort3D*: (const VecShort*)(Vec), \
    const VecShort3D*: (const VecShort*)(Vec), \
    VecShort4D*: (const VecShort*)(Vec), \
    const VecShort4D*: (const VecShort*)(Vec), \
    VecLong2D*: (const VecLong*)(Vec), \
    const VecLong2D*: (const VecLong*)(Vec), \
    VecLong3D*: (const VecLong*)(Vec), \
    const VecLong3D*: (const VecLong*)(Vec), \
    VecLong4D*: (const VecLong*)(Vec), \
    const VecLong4D*: (const VecLong*)(Vec), \
    default: Vec))

#define VecGetMinVal(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatGetMinVal, \
  const VecFloat*: _VecFloatGetMinVal, \
  VecFloat2D*: _VecFloatGetMinVal, \
  const VecFloat2D*: _VecFloatGetMinVal, \
  VecFloat3D*: _VecFloatGetMinVal, \
  const VecFloat3D*: _VecFloatGetMinVal, \
  VecShort*: _VecShortGetMinVal, \
  const VecShort*: _VecShortGetMinVal, \
  VecShort2D*: _VecShortGetMinVal, \
  const VecShort2D*: _VecShortGetMinVal, \
  VecShort3D*: _VecShortGetMinVal, \
  const VecShort3D*: _VecShortGetMinVal, \
  VecShort4D*: _VecShortGetMinVal, \
  const VecShort4D*: _VecShortGetMinVal, \
  VecLong*: _VecLongGetMinVal, \
  const VecLong*: _VecLongGetMinVal, \
```

```
  VecLong2D*: _VecLongGetMinVal, \
  const VecLong2D*: _VecLongGetMinVal, \
  VecLong3D*: _VecLongGetMinVal, \
  const VecLong3D*: _VecLongGetMinVal, \
  VecLong4D*: _VecLongGetMinVal, \
  const VecLong4D*: _VecLongGetMinVal, \
  default: PBErrInvalidPolymorphism) (_Generic(Vec, \
    VecFloat2D*: (const VecFloat*)(Vec), \
    const VecFloat2D*: (const VecFloat*)(Vec), \
    VecFloat3D*: (const VecFloat*)(Vec), \
    const VecFloat3D*: (const VecFloat*)(Vec), \
    VecShort2D*: (const VecShort*)(Vec), \
    const VecShort2D*: (const VecShort*)(Vec), \
    VecShort3D*: (const VecShort*)(Vec), \
    const VecShort3D*: (const VecShort*)(Vec), \
    VecShort4D*: (const VecShort*)(Vec), \
    const VecShort4D*: (const VecShort*)(Vec), \
    VecLong2D*: (const VecLong*)(Vec), \
    const VecLong2D*: (const VecLong*)(Vec), \
    VecLong3D*: (const VecLong*)(Vec), \
    const VecLong3D*: (const VecLong*)(Vec), \
    VecLong4D*: (const VecLong*)(Vec), \
    const VecLong4D*: (const VecLong*)(Vec), \
    default: Vec))

#define VecGetMaxValAbs(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatGetMaxValAbs, \
  const VecFloat*: _VecFloatGetMaxValAbs, \
  VecFloat2D*: _VecFloatGetMaxValAbs, \
  const VecFloat2D*: _VecFloatGetMaxValAbs, \
  VecFloat3D*: _VecFloatGetMaxValAbs, \
  const VecFloat3D*: _VecFloatGetMaxValAbs, \
  VecShort*: _VecShortGetMaxValAbs, \
  const VecShort*: _VecShortGetMaxValAbs, \
  VecShort2D*: _VecShortGetMaxValAbs, \
  const VecShort2D*: _VecShortGetMaxValAbs, \
  VecShort3D*: _VecShortGetMaxValAbs, \
  const VecShort3D*: _VecShortGetMaxValAbs, \
  VecShort4D*: _VecShortGetMaxValAbs, \
  const VecShort4D*: _VecShortGetMaxValAbs, \
  VecLong*: _VecLongGetMaxValAbs, \
  const VecLong*: _VecLongGetMaxValAbs, \
  VecLong2D*: _VecLongGetMaxValAbs, \
  const VecLong2D*: _VecLongGetMaxValAbs, \
  VecLong3D*: _VecLongGetMaxValAbs, \
  const VecLong3D*: _VecLongGetMaxValAbs, \
  VecLong4D*: _VecLongGetMaxValAbs, \
  const VecLong4D*: _VecLongGetMaxValAbs, \
  default: PBErrInvalidPolymorphism) (_Generic(Vec, \
    VecFloat2D*: (const VecFloat*)(Vec), \
    const VecFloat2D*: (const VecFloat*)(Vec), \
    VecFloat3D*: (const VecFloat*)(Vec), \
    const VecFloat3D*: (const VecFloat*)(Vec), \
    VecShort2D*: (const VecShort*)(Vec), \
    const VecShort2D*: (const VecShort*)(Vec), \
    VecShort3D*: (const VecShort*)(Vec), \
    const VecShort3D*: (const VecShort*)(Vec), \
    VecShort4D*: (const VecShort*)(Vec), \
    const VecShort4D*: (const VecShort*)(Vec), \
    VecLong2D*: (const VecLong*)(Vec), \
    const VecLong2D*: (const VecLong*)(Vec), \
    VecLong3D*: (const VecLong*)(Vec), \
```

```
    const VecLong3D*: (const VecLong*)(Vec), \
    VecLong4D*: (const VecLong*)(Vec), \
    const VecLong4D*: (const VecLong*)(Vec), \
    default: Vec))

#define VecGetMinValAbs(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatGetMinValAbs, \
  const VecFloat*: _VecFloatGetMinValAbs, \
  VecFloat2D*: _VecFloatGetMinValAbs, \
  const VecFloat2D*: _VecFloatGetMinValAbs, \
  VecFloat3D*: _VecFloatGetMinValAbs, \
  const VecFloat3D*: _VecFloatGetMinValAbs, \
  VecShort*: _VecShortGetMinValAbs, \
  const VecShort*: _VecShortGetMinValAbs, \
  VecShort2D*: _VecShortGetMinValAbs, \
  const VecShort2D*: _VecShortGetMinValAbs, \
  VecShort3D*: _VecShortGetMinValAbs, \
  const VecShort3D*: _VecShortGetMinValAbs, \
  VecShort4D*: _VecShortGetMinValAbs, \
  const VecShort4D*: _VecShortGetMinValAbs, \
  VecLong*: _VecLongGetMinValAbs, \
  const VecLong*: _VecLongGetMinValAbs, \
  VecLong2D*: _VecLongGetMinValAbs, \
  const VecLong2D*: _VecLongGetMinValAbs, \
  VecLong3D*: _VecLongGetMinValAbs, \
  const VecLong3D*: _VecLongGetMinValAbs, \
  VecLong4D*: _VecLongGetMinValAbs, \
  const VecLong4D*: _VecLongGetMinValAbs, \
  default: PBErrInvalidPolymorphism) (_Generic(Vec, \
    VecFloat2D*: (const VecFloat*)(Vec), \
    const VecFloat2D*: (const VecFloat*)(Vec), \
    VecFloat3D*: (const VecFloat*)(Vec), \
    const VecFloat3D*: (const VecFloat*)(Vec), \
    VecShort2D*: (const VecShort*)(Vec), \
    const VecShort2D*: (const VecShort*)(Vec), \
    VecShort3D*: (const VecShort*)(Vec), \
    const VecShort3D*: (const VecShort*)(Vec), \
    VecShort4D*: (const VecShort*)(Vec), \
    const VecShort4D*: (const VecShort*)(Vec), \
    VecLong2D*: (const VecLong*)(Vec), \
    const VecLong2D*: (const VecLong*)(Vec), \
    VecLong3D*: (const VecLong*)(Vec), \
    const VecLong3D*: (const VecLong*)(Vec), \
    VecLong4D*: (const VecLong*)(Vec), \
    const VecLong4D*: (const VecLong*)(Vec), \
    default: Vec))

#define VecStepDelta(Vec, VecBound, Delta) _Generic(Vec, \
  VecFloat*: _VecFloatStepDelta, \
  VecFloat2D*: _VecFloatStepDelta, \
  VecFloat3D*: _VecFloatStepDelta, \
  VecShort*: _VecShortStepDelta, \
  VecShort2D*: _VecShortStepDelta, \
  VecShort3D*: _VecShortStepDelta, \
  VecShort4D*: _VecShortStepDelta, \
  VecLong*: _VecLongStepDelta, \
  VecLong2D*: _VecLongStepDelta, \
  VecLong3D*: _VecLongStepDelta, \
  VecLong4D*: _VecLongStepDelta, \
  default: PBErrInvalidPolymorphism)(_Generic(Vec, \
    VecFloat*: Vec, \
    VecFloat2D*: (VecFloat*)(Vec), \
```

```
    VecFloat3D*: (VecFloat*)(Vec), \
    VecShort*: Vec, \
    VecShort2D*: (VecShort*)(Vec), \
    VecShort3D*: (VecShort*)(Vec), \
    VecShort4D*: (VecShort*)(Vec), \
    VecLong*: Vec, \
    VecLong2D*: (VecLong*)(Vec), \
    VecLong3D*: (VecLong*)(Vec), \
    VecLong4D*: (VecLong*)(Vec)), _Generic(Vec, \
    VecFloat*: VecBound, \
    VecFloat2D*: (VecFloat*)(VecBound), \
    VecFloat3D*: (VecFloat*)(VecBound), \
    VecShort*: VecBound, \
    VecShort2D*: (VecShort*)(VecBound), \
    VecShort3D*: (VecShort*)(VecBound), \
    VecShort4D*: (VecShort*)(VecBound), \
    VecLong*: VecBound, \
    VecLong2D*: (VecLong*)(VecBound), \
    VecLong3D*: (VecLong*)(VecBound), \
    VecLong4D*: (VecLong*)(VecBound)), _Generic(Vec, \
    VecFloat*: Delta, \
    VecFloat2D*: (VecFloat*)(Delta), \
    VecFloat3D*: (VecFloat*)(Delta), \
    VecShort*: Delta, \
    VecShort2D*: (VecShort*)(Delta), \
    VecShort3D*: (VecShort*)(Delta), \
    VecShort4D*: (VecShort*)(Delta), \
    VecLong*: Delta, \
    VecLong2D*: (VecLong*)(Delta), \
    VecLong3D*: (VecLong*)(Delta), \
    VecLong4D*: (VecLong*)(Delta)))

#define VecShiftStepDelta(Vec, VecFrom, VecTo, Delta) _Generic(Vec, \
  VecFloat*: _VecFloatShiftStepDelta, \
  VecFloat2D*: _VecFloatShiftStepDelta, \
  VecFloat3D*: _VecFloatShiftStepDelta, \
  default: PBErrInvalidPolymorphism)((VecFloat*)(Vec), \
    (VecFloat*)(VecFrom), (VecFloat*)(VecTo), (VecFloat*)(Delta))

#define VecGetIMaxVal(Vec) _Generic(Vec, \
  VecFloat*: _VecFloatGetIMaxVal, \
  const VecFloat*: _VecFloatGetIMaxVal, \
  VecFloat2D*: _VecFloatGetIMaxVal, \
  const VecFloat2D*: _VecFloatGetIMaxVal, \
  VecFloat3D*: _VecFloatGetIMaxVal, \
  const VecFloat3D*: _VecFloatGetIMaxVal, \
  VecShort*: _VecShortGetIMaxVal, \
  const VecShort*: _VecShortGetIMaxVal, \
  VecShort2D*: _VecShortGetIMaxVal, \
  const VecShort2D*: _VecShortGetIMaxVal, \
  VecShort3D*: _VecShortGetIMaxVal, \
  const VecShort3D*: _VecShortGetIMaxVal, \
  VecShort4D*: _VecShortGetIMaxVal, \
  const VecShort4D*: _VecShortGetIMaxVal, \
  VecLong*: _VecLongGetIMaxVal, \
  const VecLong*: _VecLongGetIMaxVal, \
  VecLong2D*: _VecLongGetIMaxVal, \
  const VecLong2D*: _VecLongGetIMaxVal, \
  VecLong3D*: _VecLongGetIMaxVal, \
  const VecLong3D*: _VecLongGetIMaxVal, \
  VecLong4D*: _VecLongGetIMaxVal, \
  const VecLong4D*: _VecLongGetIMaxVal, \
```

```
  default: PBErrInvalidPolymorphism) (_Generic(Vec, \
    VecFloat2D*: (const VecFloat*)(Vec), \
    const VecFloat2D*: (const VecFloat*)(Vec), \
    VecFloat3D*: (const VecFloat*)(Vec), \
    const VecFloat3D*: (const VecFloat*)(Vec), \
    VecShort2D*: (const VecShort*)(Vec), \
    const VecShort2D*: (const VecShort*)(Vec), \
    VecShort3D*: (const VecShort*)(Vec), \
    const VecShort3D*: (const VecShort*)(Vec), \
    VecShort4D*: (const VecShort*)(Vec), \
    const VecShort4D*: (const VecShort*)(Vec), \
    VecLong2D*: (const VecLong*)(Vec), \
    const VecLong2D*: (const VecLong*)(Vec), \
    VecLong3D*: (const VecLong*)(Vec), \
    const VecLong3D*: (const VecLong*)(Vec), \
    VecLong4D*: (const VecLong*)(Vec), \
    const VecLong4D*: (const VecLong*)(Vec), \
    default: Vec))

#define MatClone(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatClone, \
  const MatFloat*: _MatFloatClone, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatEncodeAsJSON(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatEncodeAsJSON, \
  const MatFloat*: _MatFloatEncodeAsJSON, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatDecodeAsJSON(MatRef, Json) _Generic(MatRef, \
  MatFloat**: _MatFloatDecodeAsJSON, \
  default: PBErrInvalidPolymorphism)(MatRef, Json)

#define MatLoad(MatRef, Stream) _Generic(MatRef, \
  MatFloat**: _MatFloatLoad, \
  default: PBErrInvalidPolymorphism)(MatRef, Stream)

#define MatSave(Mat, Stream, Compact) _Generic(Mat, \
  MatFloat*: _MatFloatSave, \
  const MatFloat*: _MatFloatSave, \
  default: PBErrInvalidPolymorphism)(Mat, Stream, Compact)

#define MatFree(MatRef) _Generic(MatRef, \
  MatFloat**: _MatFloatFree, \
  default: PBErrInvalidPolymorphism)(MatRef)

#define MatPrintln(Mat, Stream) _Generic(Mat, \
  MatFloat*: _MatFloatPrintlnDef, \
  const MatFloat*: _MatFloatPrintlnDef, \
  default: PBErrInvalidPolymorphism)(Mat, Stream)

#define MatGet(Mat, VecIndex) _Generic(Mat, \
  MatFloat*: _MatFloatGet, \
  const MatFloat*: _MatFloatGet, \
  default: PBErrInvalidPolymorphism)(Mat, VecIndex)

#define MatSet(Mat, VecIndex, Val) _Generic(Mat, \
  MatFloat*: _MatFloatSet, \
  default: PBErrInvalidPolymorphism)(Mat, VecIndex, Val)

#define MatCopy(MatDest, MatSrc) _Generic(MatDest, \
  MatFloat*: _Generic (MatSrc, \
```

```
    MatFloat*: _MatFloatCopy, \
    const MatFloat*: _MatFloatCopy, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(MatDest, MatSrc)

#define MatDim(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatDim, \
  const MatFloat*: _MatFloatDim, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatGetDim(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatGetDim, \
  const MatFloat*: _MatFloatGetDim, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatGetEigenValues(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatGetEigenValues, \
  const MatFloat*: _MatFloatGetEigenValues, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatGetQR(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatGetQR, \
  const MatFloat*: _MatFloatGetQR, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatGetInv(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatGetInv, \
  const MatFloat*: _MatFloatGetInv, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatGetTranspose(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatGetTranspose, \
  const MatFloat*: _MatFloatGetTranspose, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatGetNbRow(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatGetNbRow, \
  const MatFloat*: _MatFloatGetNbRow, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatGetNbCol(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatGetNbCol, \
  const MatFloat*: _MatFloatGetNbCol, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatGetProdMat(MatA, MatB) _Generic(MatA, \
  MatFloat*: _Generic(MatB, \
    MatFloat*: _MatFloatGetProdMatFloat, \
    const MatFloat*: _MatFloatGetProdMatFloat, \
    default: PBErrInvalidPolymorphism), \
  const MatFloat*: _Generic(MatB, \
    MatFloat*: _MatFloatGetProdMatFloat, \
    const MatFloat*: _MatFloatGetProdMatFloat, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(MatA, MatB)

#define MatGetProdVec(Mat, Vec) _Generic(Mat, \
  MatFloat*: _Generic(Vec, \
    VecFloat*: _MatFloatGetProdVecFloat, \
    const VecFloat*: _MatFloatGetProdVecFloat, \
    VecFloat2D*: _MatFloatGetProdVecFloat, \
    const VecFloat2D*: _MatFloatGetProdVecFloat, \
```

```
    VecFloat3D*: _MatFloatGetProdVecFloat, \
    const VecFloat3D*: _MatFloatGetProdVecFloat, \
    default: PBErrInvalidPolymorphism), \
  const MatFloat*: _Generic(Vec, \
    VecFloat*: _MatFloatGetProdVecFloat, \
    const VecFloat*: _MatFloatGetProdVecFloat, \
    VecFloat2D*: _MatFloatGetProdVecFloat, \
    const VecFloat2D*: _MatFloatGetProdVecFloat, \
    VecFloat3D*: _MatFloatGetProdVecFloat, \
    const VecFloat3D*: _MatFloatGetProdVecFloat, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(Mat, (VecFloat*)(Vec))

#define MatGetProdVecVecTranspose(VecA, VecB) _Generic(VecA, \
  VecFloat*: _Generic(VecB, \
    VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    default: PBErrInvalidPolymorphism), \
  const VecFloat*: _Generic(VecB, \
    VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    default: PBErrInvalidPolymorphism), \
  VecFloat2D*: _Generic(VecB, \
    VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    default: PBErrInvalidPolymorphism), \
  const VecFloat2D*: _Generic(VecB, \
    VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    default: PBErrInvalidPolymorphism), \
  VecFloat3D*: _Generic(VecB, \
    VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    default: PBErrInvalidPolymorphism), \
  const VecFloat3D*: _Generic(VecB, \
    VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat2D*: _MatFloatGetProdVecVecTransposeFloat, \
    VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    const VecFloat3D*: _MatFloatGetProdVecVecTransposeFloat, \
    default: PBErrInvalidPolymorphism), \
```

```
  default: PBErrInvalidPolymorphism)((VecFloat*)(VecA), \
    (VecFloat*)(VecB))

#define MatAdd(MatA, MatB) _Generic(MatA, \
  MatFloat*: _Generic(MatB, \
    MatFloat*: _MatFloatAdd, \
    const MatFloat*: _MatFloatAdd, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(MatA, MatB)

#define MatScale(MatA, A) _Generic(MatA, \
  MatFloat*: _MatFloatScale, \
  const MatFloat*: _MatFloatScale, \
  default: PBErrInvalidPolymorphism)(MatA, A)

#define MatGetAdd(MatA, MatB) _Generic(MatA, \
  MatFloat*: _Generic(MatB, \
    MatFloat*: _MatFloatGetAdd, \
    const MatFloat*: _MatFloatGetAdd, \
    default: PBErrInvalidPolymorphism), \
  const MatFloat*: _Generic(MatB, \
    MatFloat*: _MatFloatGetAdd, \
    const MatFloat*: _MatFloatGetAdd, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(MatA, MatB)

#define MatSetIdentity(Mat) _Generic(Mat, \
  MatFloat*: _MatFloatSetIdentity, \
  default: PBErrInvalidPolymorphism)(Mat)

#define MatIsEqual(MatA, MatB) _Generic(MatA, \
  MatFloat*: _Generic(MatB, \
    MatFloat*: _MatFloatIsEqual, \
    const MatFloat*: _MatFloatIsEqual, \
    default: PBErrInvalidPolymorphism), \
  const MatFloat*: _Generic(MatB, \
    MatFloat*: _MatFloatIsEqual, \
    const MatFloat*: _MatFloatIsEqual, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)(MatA, MatB)

#define SysLinEqCreate(Mat, Vec) _Generic(Vec, \
  VecFloat*: _SLECreate, \
  const VecFloat*: _SLECreate, \
  VecFloat2D*: _SLECreate, \
  const VecFloat2D*: _SLECreate, \
  VecFloat3D*: _SLECreate, \
  const VecFloat3D*: _SLECreate, \
  default: PBErrInvalidPolymorphism)(Mat, (VecFloat*)(Vec))

#define SysLinEqSetV(Sys, Vec) _Generic(Vec, \
  VecFloat*: _SLESetV, \
  const VecFloat*: _SLESetV, \
  VecFloat2D*: _SLESetV, \
  const VecFloat2D*: _SLESetV, \
  VecFloat3D*: _SLESetV, \
  const VecFloat3D*: _SLESetV, \
  default: PBErrInvalidPolymorphism)(Sys, (VecFloat*)(Vec))

// ================ static 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 initalized to 0.0
VecShort* VecShortCreate(const long dim) {
#if BUILDMODE == 0
  if (dim <= 0) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "invalid 'dim' (%ld)", dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Allocate memory
  VecShort* that = PBErrMalloc(PBMathErr,
    offsetof(VecShort, _val) + sizeof(short) * dim);
  // Set the default values
  that->_dim = dim;
  for (long 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(const VecShort* const 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;
}

// Function which return the JSON encoding of 'that'
JSONNode* _VecShortEncodeAsJSON(const VecShort* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the dimension
  sprintf(val, "%ld", VecGetDim(that));
  JSONAddProp(json, "_dim", val);
  // Encode the values
  JSONArrayVal setVal = JSONArrayValCreateStatic();
  for (long i = 0; i < VecGetDim(that); ++i) {
    sprintf(val, "%d", VecGet(that, i));
    JSONArrayValAdd(&setVal, val);
  }
  JSONAddProp(json, "_val", &setVal);
  // Free memory
  JSONArrayValFlush(&setVal);
  // Return the created JSON
  return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool _VecShortDecodeAsJSON(VecShort** that, const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (json == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'json' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // If 'that' is already allocated
  if (*that != NULL)
    // Free memory
    _VecShortFree(that);
  // Get the dimension from the JSON
  JSONNode* prop = JSONProperty(json, "_dim");
  if (prop == NULL) {
    return false;
  }
  long dim = atol(JSONLblVal(prop));
  // If data are invalid
  if (dim < 1)
    return false;
  // Allocate memory
  *that = VecShortCreate(dim);
```

```
  // Get the values
  prop = JSONProperty(json, "_val");
  if (prop == NULL) {
    return false;
  }
  for (long i = 0; i < dim; ++i) {
    long val = atol(JSONLabel(JSONValue(prop, i)));
    VecSet(*that, i, val);
  }
  // Return the success code
  return true;
}

// 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* const 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
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (!JSONLoad(json, stream)) {
    return false;
  }
  // Decode the data from the JSON
  if (!VecDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return the success code
  return true;
}

// Save the VecShort to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true in case of success, else false
bool _VecShortSave(const VecShort* const that,
  FILE* const stream, const bool compact) {
#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
  // Get the JSON encoding
  JSONNode* json = VecEncodeAsJSON(that);
  // Save the JSON
  if (!JSONSave(json, stream, compact)) {
    return false;
  }
  // Free memory
  JSONFree(&json);
  // 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(const VecShort* const that,
  FILE* const 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
  // Print the values
  fprintf(stream, "[");
  for (long 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* const that, const VecShort* const bound) {
#if 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 (%ld==%ld)",
      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
  long 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* const that, const VecShort* const bound) {
#if 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 (%ld==%ld)",
      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
  long 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;
}

// Step the values of the vector incrementally by 1
// in the following order (for example) :
// (0,0,0)->(0,0,1)->(0,0,2)->(0,1,0)->(0,1,1)->...
// The lower limit for each value is given by 'from' (val[i] >= from[i])
// The upper limit for each value is given by 'to' (val[i] < to[i])
// 'that' must be initialised to 'from' before the first call of this
// function
// Return false if all values of 'that' have reached their upper limit
// (in which case 'that''s values are all set back to from)
// Return true else
bool _VecShortShiftStep(VecShort* const that,
  const VecShort* const from, const VecShort* const to) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (from == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'from' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != from->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'from' dimensions don't match (%ld==%ld)",
      that->_dim, from->_dim);
    PBErrCatch(PBMathErr);
  }
  if (to == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'to' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != to->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'to' dimensions don't match (%ld==%ld)",
      that->_dim, to->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
```

```
  // Declare a variable for the returned flag
  bool ret = true;
  // Declare a variable to memorise the dimension currently increasing
  long iDim = that->_dim - 1;
  // Declare a flag for the loop condition
  bool flag = true;
  // Increment
  do {
    ++(that->_val[iDim]);
    if (that->_val[iDim] >= to->_val[iDim]) {
      that->_val[iDim] = from->_val[iDim];
      --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 delta 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 after 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 _VecShortStepDelta(VecShort* const that,
  const VecShort* const bound, const VecShort* const delta) {
#if 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 (delta == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'delta' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != bound->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "dimensions don't match (%ld==%ld)",
      that->_dim, bound->_dim);
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != delta->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "dimensions don't match (%ld==%ld)",
      that->_dim, delta->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable for the returned flag
  bool ret = true;
```

```
  // Declare a variable to memorise the dimension currently increasing
  long iDim = that->_dim - 1;
  // Declare a flag for the loop condition
  bool flag = true;
  // Increment
  do {
    that->_val[iDim] += delta->_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 delta 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 after 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 _VecShortPStepDelta(VecShort* const that,
  const VecShort* const bound, const VecShort* const delta) {
#if 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 (delta == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'delta' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != bound->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "dimensions don't match (%ld==%ld)",
      that->_dim, bound->_dim);
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != delta->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "dimensions don't match (%ld==%ld)",
      that->_dim, delta->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable for the returned flag
  bool ret = true;
  // Declare a variable to memorise the dimension currently increasing
  long iDim = 0;
```

```
  // Declare a flag for the loop condition
  bool flag = true;
  // Increment
  do {
    that->_val[iDim] += delta->_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;
}

// -------------- VecLong

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

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

// Clone the VecLong
// Return NULL if we couldn't clone the VecLong
VecLong* _VecLongClone(const VecLong* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create a clone
  VecLong* clone = VecLongCreate(that->_dim);
  // Copy the values
  memcpy(clone, that, sizeof(VecLong) + sizeof(long) * that->_dim);
  // Return the clone
  return clone;
}

// Function which return the JSON encoding of 'that'
```

```
JSONNode* _VecLongEncodeAsJSON(const VecLong* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the dimension
  sprintf(val, "%ld", VecGetDim(that));
  JSONAddProp(json, "_dim", val);
  // Encode the values
  JSONArrayVal setVal = JSONArrayValCreateStatic();
  for (long i = 0; i < VecGetDim(that); ++i) {
    sprintf(val, "%ld", VecGet(that, i));
    JSONArrayValAdd(&setVal, val);
  }
  JSONAddProp(json, "_val", &setVal);
  // Free memory
  JSONArrayValFlush(&setVal);
  // Return the created JSON
  return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool _VecLongDecodeAsJSON(VecLong** that, const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (json == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'json' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // If 'that' is already allocated
  if (*that != NULL)
    // Free memory
    _VecLongFree(that);
  // Get the dimension from the JSON
  JSONNode* prop = JSONProperty(json, "_dim");
  if (prop == NULL) {
    return false;
  }
  long dim = atol(JSONLblVal(prop));
  // If data are invalid
  if (dim < 1)
    return false;
  // Allocate memory
  *that = VecLongCreate(dim);
  // Get the values
  prop = JSONProperty(json, "_val");
  if (prop == NULL) {
    return false;
  }
```

```
  for (long i = 0; i < dim; ++i) {
    long val = atol(JSONLabel(JSONValue(prop, i)));
    VecSet(*that, i, val);
  }
  // Return the success code
  return true;
}

// Load the VecLong from the stream
// If the VecLong is already allocated, it is freed before loading
// Return true in case of success, else false
bool _VecLongLoad(VecLong** that, FILE* const 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
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (!JSONLoad(json, stream)) {
    return false;
  }
  // Decode the data from the JSON
  if (!VecDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return the success code
  return true;
}

// Save the VecLong to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true in case of success, else false
bool _VecLongSave(const VecLong* const that,
  FILE* const stream, const bool compact) {
#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
  // Get the JSON encoding
  JSONNode* json = VecEncodeAsJSON(that);
  // Save the JSON
  if (!JSONSave(json, stream, compact)) {
```

```
    return false;
  }
  // Free memory
  JSONFree(&json);
  // Return success code
  return true;
}

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

// Print the VecLong on 'stream' with 'prec' digit precision
void _VecLongPrint(const VecLong* const that,
  FILE* const 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
  // Print the values
  fprintf(stream, "[");
  for (long i = 0; i < that->_dim; ++i) {
    fprintf(stream, "%ld", 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 _VecLongStep(VecLong* const that, const VecLong* const bound) {
#if 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 (%ld==%ld)",
      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
  long 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 _VecLongPStep(VecLong* const that, const VecLong* const bound) {
#if 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 (%ld==%ld)",
      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
  long 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;
}

// Step the values of the vector incrementally by 1
// in the following order (for example) :
// (0,0,0)->(0,0,1)->(0,0,2)->(0,1,0)->(0,1,1)->...
// The lower limit for each value is given by 'from' (val[i] >= from[i])
// The upper limit for each value is given by 'to' (val[i] < to[i])
// 'that' must be initialised to 'from' before the first call of this
// function
// Return false if all values of 'that' have reached their upper limit
// (in which case 'that''s values are all set back to from)
// Return true else
bool _VecLongShiftStep(VecLong* const that,
  const VecLong* const from, const VecLong* const to) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (from == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'from' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != from->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'from' dimensions don't match (%ld==%ld)",
      that->_dim, from->_dim);
    PBErrCatch(PBMathErr);
  }
  if (to == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'to' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != to->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'to' dimensions don't match (%ld==%ld)",
      that->_dim, to->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable for the returned flag
  bool ret = true;
  // Declare a variable to memorise the dimension currently increasing
  long iDim = that->_dim - 1;
  // Declare a flag for the loop condition
```

```
  bool flag = true;
  // Increment
  do {
    ++(that->_val[iDim]);
    if (that->_val[iDim] >= to->_val[iDim]) {
      that->_val[iDim] = from->_val[iDim];
      --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 delta 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 after 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 _VecLongStepDelta(VecLong* const that,
  const VecLong* const bound, const VecLong* const delta) {
#if 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 (delta == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'delta' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != bound->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "dimensions don't match (%ld==%ld)",
      that->_dim, bound->_dim);
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != delta->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "dimensions don't match (%ld==%ld)",
      that->_dim, delta->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable for the returned flag
  bool ret = true;
  // Declare a variable to memorise the dimension currently increasing
  long iDim = that->_dim - 1;
  // Declare a flag for the loop condition
  bool flag = true;
  // Increment
```

```
  do {
    that->_val[iDim] += delta->_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 delta 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 after 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 _VecLongPStepDelta(VecLong* const that,
  const VecLong* const bound, const VecLong* const delta) {
#if 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 (delta == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'delta' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != bound->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "dimensions don't match (%ld==%ld)",
      that->_dim, bound->_dim);
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != delta->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "dimensions don't match (%ld==%ld)",
      that->_dim, delta->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable for the returned flag
  bool ret = true;
  // Declare a variable to memorise the dimension currently increasing
  long iDim = 0;
  // Declare a flag for the loop condition
  bool flag = true;
  // Increment
  do {
    that->_val[iDim] += delta->_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;
}

// Return a new VecLong as a copy of the VecLong 'that' with
// dimension changed to 'dim'
// if it is extended, the values of new components are 0
// If it is shrinked, values are discarded from the end of the vector
VecLong* _VecLongGetNewDim(const VecLong* const that, const long dim) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (dim <= 0) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'dim' is invalid match (%ld>0)", dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // If the new dimension is the same as the current one
  if (dim == VecGetDim(that)) {
    // Return the clone of the vector
    return VecClone(that);
  // Else, the new dimension is actually different
  } else {
    // Declare the returned vector
    VecLong* ret = VecLongCreate(dim);
    // Copy the components
    for (long iAxis = MIN(VecGetDim(that), dim); iAxis--;)
      VecSet(ret, iAxis, VecGet(that, iAxis));
    // Return the new vector
    return ret;
  }
}

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

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

// Create a new Vec of dimension 'dim'
// Values are initalized to 0.0
VecFloat* VecFloatCreate(const long dim) {
#if BUILDMODE == 0
  if (dim <= 0) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "invalid 'dim' (%ld)", dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Allocate memory
  VecFloat* that = PBErrMalloc(PBMathErr,
```

```
    offsetof(VecFloat, _val) + sizeof(float) * dim);
  // Set the default values
  that->_dim = dim;
  for (long i = dim; i--;)
    that->_val[i] = 0.0;
  // Return the new VecFloat
  return that;
}

// Clone the VecFloat
VecFloat* _VecFloatClone(const VecFloat* const that) {
#if 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;
}

// Function which return the JSON encoding of 'that'
JSONNode* _VecFloatEncodeAsJSON(const VecFloat* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the dimension
  sprintf(val, "%ld", VecGetDim(that));
  JSONAddProp(json, "_dim", val);
  // Encode the values
  JSONArrayVal setVal = JSONArrayValCreateStatic();
  for (long i = 0; i < VecGetDim(that); ++i) {
    sprintf(val, "%f", VecGet(that, i));
    JSONArrayValAdd(&setVal, val);
  }
  JSONAddProp(json, "_val", &setVal);
  // Free memory
  JSONArrayValFlush(&setVal);
  // Return the created JSON
  return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool _VecFloatDecodeAsJSON(VecFloat** that, const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
```

```
  }
  if (json == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'json' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // If 'that' is already allocated
  if (*that != NULL)
    // Free memory
    _VecFloatFree(that);
  // Get the dimension from the JSON
  JSONNode* prop = JSONProperty(json, "_dim");
  if (prop == NULL) {
    return false;
  }
  long dim = atol(JSONLblVal(prop));
  // If data are invalid
  if (dim < 1)
    return false;
  // Allocate memory
  *that = VecFloatCreate(dim);
  // Get the values
  prop = JSONProperty(json, "_val");
  if (prop == NULL) {
    return false;
  }
  for (long i = 0; i < dim; ++i) {
    float val = atof(JSONLabel(JSONValue(prop, i)));
    VecSet(*that, i, val);
  }
  // Return the success code
  return true;
}

// Load the VecFloat from the stream
// If the VecFloat is already allocated, it is freed before loading
bool _VecFloatLoad(VecFloat** that, FILE* const 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
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (!JSONLoad(json, stream)) {
    return false;
  }
  // Decode the data from the JSON
  if (!VecDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
```

```
  // Return the success code
  return true;
}

// Save the VecFloat to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true in case of success, else false
bool _VecFloatSave(const VecFloat* const that,
  FILE* const stream, const bool compact) {
#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
  // Get the JSON encoding
  JSONNode* json = VecEncodeAsJSON(that);
  // Save the JSON
  if (!JSONSave(json, stream, compact)) {
    return false;
  }
  // Free memory
  JSONFree(&json);
  // 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(const VecFloat* const that, FILE* const stream,
  const 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 (long 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(const VecFloat2D* const that,
  const VecFloat2D* const 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 (!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 = PBMATH_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;
}

// Return a VecFloat3D equal to 'that' rotated right-hand by 'theta'
// radians around 'axis'
// 'axis' must be normalized
// https://en.wikipedia.org/wiki/Rotation_matrix
VecFloat3D _VecFloatGetRotAxis(const VecFloat3D* const that,
  const VecFloat3D* const axis, const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (axis == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'axis' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld=3)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(axis) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'axis' 's dimension is invalid (%ld=3)",
      VecGetDim(axis));
    PBErrCatch(PBMathErr);
  }
  if (ISEQUALF(VecNorm(axis), 1.0) == false) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'axis' is not normalized");
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare variable for optimisation
  float cosTheta = cos(theta);
  float sinTheta = sin(theta);
  // Create the rotation matrix
  VecShort2D d = VecShortCreateStatic2D();
  VecSet(&d, 0, 3); VecSet(&d, 1, 3);
  MatFloat* rot = MatFloatCreate(&d);
  VecSet(&d, 0, 0); VecSet(&d, 1, 0);
  float v = cosTheta + fastpow(VecGet(axis, 0), 2) * (1.0 - cosTheta);
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 0);
  v = VecGet(axis, 0) * VecGet(axis, 1) * (1.0 - cosTheta) -
    VecGet(axis, 2) * sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 0);
```

```
  v = VecGet(axis, 0) * VecGet(axis, 2) * (1.0 - cosTheta) +
    VecGet(axis, 1) * sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 0); VecSet(&d, 1, 1);
  v = VecGet(axis, 0) * VecGet(axis, 1) * (1.0 - cosTheta) +
    VecGet(axis, 2) * sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 1);
  v = cosTheta + fastpow(VecGet(axis, 1), 2) * (1.0 - cosTheta);
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 1);
  v = VecGet(axis, 1) * VecGet(axis, 2) * (1.0 - cosTheta) -
    VecGet(axis, 0) * sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 0); VecSet(&d, 1, 2);
  v = VecGet(axis, 0) * VecGet(axis, 2) * (1.0 - cosTheta) -
    VecGet(axis, 1) * sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 2);
  v = VecGet(axis, 1) * VecGet(axis, 2) * (1.0 - cosTheta) +
    VecGet(axis, 0) * sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 2);
  v = cosTheta + fastpow(VecGet(axis, 2), 2) * (1.0 - cosTheta);
  MatSet(rot, &d, v);
  // Calculate the result vector
  VecFloat* w = MatGetProdVec(rot, that);
  VecFloat3D res = VecFloatCreateStatic3D();
  VecCopy(&res, w);
  // Free memory
  VecFree(&w);
  MatFree(&rot);
  // Return the result
  return res;
}

// Return a VecFloat3D equal to 'that' rotated right-hand by 'theta'
// radians around X
VecFloat3D _VecFloatGetRotX(const VecFloat3D* const that,
  const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld=3)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare variable for optimisation
  float cosTheta = cos(theta);
  float sinTheta = sin(theta);
  // Create the rotation matrix
  VecShort2D d = VecShortCreateStatic2D();
  VecSet(&d, 0, 3); VecSet(&d, 1, 3);
  MatFloat* rot = MatFloatCreate(&d);
  VecSet(&d, 0, 0); VecSet(&d, 1, 0);
  float v = 1.0;
```

```
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 0);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 0);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 0); VecSet(&d, 1, 1);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 1);
  v = cosTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 1);
  v = -sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 0); VecSet(&d, 1, 2);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 2);
  v = sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 2);
  v = cosTheta;
  MatSet(rot, &d, v);
  // Calculate the result vector
  VecFloat* w = MatGetProdVec(rot, that);
  VecFloat3D res = VecFloatCreateStatic3D();
  VecCopy(&res, w);
  // Free memory
  VecFree(&w);
  MatFree(&rot);
  // Return the result
  return res;
}

// Return a VecFloat3D equal to 'that' rotated right-hand by 'theta'
// radians around Y
VecFloat3D _VecFloatGetRotY(const VecFloat3D* const that,
  const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld=3)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare variable for optimisation
  float cosTheta = cos(theta);
  float sinTheta = sin(theta);
  // Create the rotation matrix
  VecShort2D d = VecShortCreateStatic2D();
  VecSet(&d, 0, 3); VecSet(&d, 1, 3);
  MatFloat* rot = MatFloatCreate(&d);
  VecSet(&d, 0, 0); VecSet(&d, 1, 0);
  float v = cosTheta;
```

```
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 0);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 0);
  v = sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 0); VecSet(&d, 1, 1);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 1);
  v = 1.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 1);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 0); VecSet(&d, 1, 2);
  v = -sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 2);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 2);
  v = cosTheta;
  MatSet(rot, &d, v);
  // Calculate the result vector
  VecFloat* w = MatGetProdVec(rot, that);
  VecFloat3D res = VecFloatCreateStatic3D();
  VecCopy(&res, w);
  // Free memory
  VecFree(&w);
  MatFree(&rot);
  // Return the result
  return res;
}

// Return a VecFloat3D equal to 'that' rotated right-hand by 'theta'
// radians around Z
VecFloat3D _VecFloatGetRotZ(const VecFloat3D* const that,
  const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld=3)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare variable for optimisation
  float cosTheta = cos(theta);
  float sinTheta = sin(theta);
  // Create the rotation matrix
  VecShort2D d = VecShortCreateStatic2D();
  VecSet(&d, 0, 3); VecSet(&d, 1, 3);
  MatFloat* rot = MatFloatCreate(&d);
  VecSet(&d, 0, 0); VecSet(&d, 1, 0);
  float v = cosTheta;
```

```
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 0);
  v = -sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 0);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 0); VecSet(&d, 1, 1);
  v = sinTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 1);
  v = cosTheta;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 1);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 0); VecSet(&d, 1, 2);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 1); VecSet(&d, 1, 2);
  v = 0.0;
  MatSet(rot, &d, v);
  VecSet(&d, 0, 2); VecSet(&d, 1, 2);
  v = 1.0;
  MatSet(rot, &d, v);
  // Calculate the result vector
  VecFloat* w = MatGetProdVec(rot, that);
  VecFloat3D res = VecFloatCreateStatic3D();
  VecCopy(&res, w);
  // Free memory
  VecFree(&w);
  MatFree(&rot);
  // Return the result
  return res;
}

// Step the values of the vector incrementally by delta 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 after 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 _VecFloatStepDelta(VecFloat* const that,
  const VecFloat* const bound, const VecFloat* const delta) {
#if 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 (delta == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'delta' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != bound->_dim) {
```

```
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg,
      "'bound' 's dimensions don't match (%ld==%ld)",
      that->_dim, bound->_dim);
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != delta->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg,
      "'delta' 's dimensions don't match (%ld==%ld)",
      that->_dim, delta->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable for the returned flag
  bool ret = true;
  // Declare a variable to memorise the dimension currently increasing
  long iDim = that->_dim - 1;
  // Declare a flag for the loop condition
  bool flag = true;
  // Increment
  do {
    that->_val[iDim] += delta->_val[iDim];
    if (that->_val[iDim] > bound->_val[iDim] + PBMATH_EPSILON) {
      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 delta
// in the following order (for example) :
// (0.,0.,0.)->(0.,0.,1.)->(0.,0.,2.)->(0.,1.,0.)->(0.,1.,1.)->...
// The lower limit for each value is given by 'from' (val[i] >= from[i])
// The upper limit for each value is given by 'to' (val[i] <= to[i])
// 'that' must be initialised to 'from' before the first call of this
// function
// Return false after all values of 'that' have reached their upper
// limit (in which case 'that''s values are all set back to from)
// Return true else
bool _VecFloatShiftStepDelta(VecFloat* const that,
  const VecFloat* const from, const VecFloat* const to,
  const VecFloat* const delta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (from == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'from' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != from->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
```

```
    sprintf(PBMathErr->_msg, "'from' dimensions don't match (%ld==%ld)",
      that->_dim, from->_dim);
    PBErrCatch(PBMathErr);
  }
  if (to == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'to' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != to->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'to' dimensions don't match (%ld==%ld)",
      that->_dim, to->_dim);
    PBErrCatch(PBMathErr);
  }
  if (delta == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'delta' is null");
    PBErrCatch(PBMathErr);
  }
  if (that->_dim != delta->_dim) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'delta' dimensions don't match (%ld==%ld)",
      that->_dim, delta->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable for the returned flag
  bool ret = true;
  // Declare a variable to memorise the dimension currently increasing
  long iDim = that->_dim - 1;
  // Declare a flag for the loop condition
  bool flag = true;
  // Increment
  do {
    that->_val[iDim] += delta->_val[iDim];
    if (that->_val[iDim] > to->_val[iDim] + PBMATH_EPSILON) {
      that->_val[iDim] = from->_val[iDim];
      --iDim;
    } else {
      flag = false;
    }
  } while (iDim >= 0 && flag == true);
  if (iDim == -1)
    ret = false;
  // Return the flag
  return ret;
}

// Return a new VecFloat as a copy of the VecFloat 'that' with
// dimension changed to 'dim'
// if it is extended, the values of new components are 0.0
// If it is shrinked, values are discarded from the end of the vector
VecFloat* _VecFloatGetNewDim(const VecFloat* const that, const long dim) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (dim <= 0) {
    PBMathErr->_type = PBErrTypeInvalidArg;
```

```
    sprintf(PBMathErr->_msg, "'dim' is invalid match (%ld>0)", dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // If the new dimension is the same as the current one
  if (dim == VecGetDim(that)) {
    // Return the clone of the vector
    return VecClone(that);
  // Else, the new dimension is actually different
  } else {
    // Declare the returned vector
    VecFloat* ret = VecFloatCreate(dim);
    // Copy the components
    for (long iAxis = MIN(VecGetDim(that), dim); iAxis--;)
      VecSet(ret, iAxis, VecGet(that, iAxis));
    // Return the new vector
    return ret;
  }
}

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

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

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

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

// Clone the MatFloat
MatFloat* _MatFloatClone(const MatFloat* const 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
  long d = VecGet(&(that->_dim), 0) * VecGet(&(that->_dim), 1);
```

```
  for (long i = d; i--;)
    clone->_val[i] = that->_val[i];
  // Return the clone
  return clone;
}

// Function which return the JSON encoding of 'that'
JSONNode* _MatFloatEncodeAsJSON(MatFloat* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the dimensions
  sprintf(val, "%d", VecGet(&(that->_dim), 0));
  JSONAddProp(json, "_nbRow", val);
  sprintf(val, "%d", VecGet(&(that->_dim), 1));
  JSONAddProp(json, "_nbCol", val);
  // Encode the values
  JSONArrayVal setVal = JSONArrayValCreateStatic();
  VecShort2D index = VecShortCreateStatic2D();
  do {
    sprintf(val, "%f", MatGet(that, &index));
    JSONArrayValAdd(&setVal, val);
  } while (VecStep(&index, &(that->_dim)));
  JSONAddProp(json, "_val", &setVal);
  // Free memory
  JSONArrayValFlush(&setVal);
  // Return the created JSON
  return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool _MatFloatDecodeAsJSON(MatFloat** that, JSONNode* json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (json == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'json' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // If 'that' is already allocated
  if (*that != NULL)
    // Free memory
    _MatFloatFree(that);
  // Get the dimensions from the JSON
  JSONNode* prop = JSONProperty(json, "_nbRow");
  if (prop == NULL) {
    return false;
  }
  VecShort2D dim = VecShortCreateStatic2D();
```

```
  VecSet(&dim, 0, atoi(JSONLblVal(prop)));
  prop = JSONProperty(json, "_nbCol");
  if (prop == NULL) {
    return false;
  }
  VecSet(&dim, 1, atoi(JSONLblVal(prop)));
  // If data are invalid
  if (VecGet(&dim, 0) < 1 || VecGet(&dim, 1) < 1)
    return false;
  // Allocate memory
  *that = MatFloatCreate(&dim);
  // Get the values
  prop = JSONProperty(json, "_val");
  if (prop == NULL) {
    return false;
  }
  VecShort2D index = VecShortCreateStatic2D();
  int i = 0;
  do {
    MatSet(*that, &index, atof(JSONLabel(JSONValue(prop, i))));
    ++i;
  } while (VecStep(&index, &dim));
  // Return the success code
  return true;
}

// 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
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (!JSONLoad(json, stream)) {
    return false;
  }
  // Decode the data from the JSON
  if (!MatDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return the success code
  return true;
}

// Save the MatFloat to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true upon success, else false
```

```
bool _MatFloatSave(MatFloat* const that, FILE* stream, bool compact) {
#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
  // Get the JSON encoding
  JSONNode* json = MatEncodeAsJSON(that);
  // Save the JSON
  if (!JSONSave(json, stream, compact)) {
    return false;
  }
  // Free memory
  JSONFree(&json);
  // 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* const 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, "[");
  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
// Return NULL if the matrix is not invertible, or in some case when
// the matrix's diagonal contains null values and the matrix's size
// is greater than 3
MatFloat* _MatFloatGetInv(const MatFloat* const that) {
#if 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);
  }
#endif
  // Allocate memory for the result
  MatFloat* res = NULL;
  // If the matrix is of dimension 1x1
  if (VecGet(&(that->_dim), 0) == 1) {
    if (fabs(that->_val[0]) > PBMATH_EPSILON) {
      // Allocate memory for the result
      res = MatFloatCreate(&(that->_dim));
      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];
    if (!ISEQUALF(det, 0.0)) {
      // Allocate memory for the result
      res = MatFloatCreate(&(that->_dim));
      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]);
    if (!ISEQUALF(det, 0.0)) {
```

```
    // Allocate memory for the result
    res = MatFloatCreate(&(that->_dim));
    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;
  }
} else {
  // Clone the matrix to be inverted
  res = MatClone(that);
  // Farooq Hamid algorithm (modified to handle some matrix with null
  // values on the diagonal)
  // https://www.researchgate.net/publication/
  //   220337322_An_Efficient_and_Simple_Algorithm_for_Matrix_Inversion
  //float det = 1.0;
  short size = VecGet(&(that->_dim), 0);
  float* mat = res->_val;
  bool flagHasChanged = true;
  short nbRemaining = size;
  bool* hasPivotChanged = PBErrMalloc(PBMathErr, size * sizeof(bool));
  for(short p = 0; p < size; ++p) {
    hasPivotChanged[p] = false;
  }
  while (flagHasChanged == true && nbRemaining > 0) {
    flagHasChanged = false;
    for(short p = 0; p < size; ++p) {
      float pivot = mat[p * size + p];
      if (fabs(pivot) > FLT_MIN && !(hasPivotChanged[p])) {
        flagHasChanged = true;
        --nbRemaining;
        hasPivotChanged[p] = true;
        //det *= pivot;
        for (short i = 0; i < size; ++i) {
          mat[i * size + p] = -1.0 * mat[i * size + p] / pivot;
        }
        for (short i = 0; i < size; ++i) {
          if (i != p) {
            for (short j = 0; j < size; ++j) {
              if (j != p) {
                mat[i * size + j] =
                  mat[i * size + j] + mat[p * size + j] * mat[i * size + p];
              }
            }
          }
        }
        for (short j = 0; j < size; ++j) {
          mat[p * size + j] =  mat[p * size + j] / pivot;
        }
```

```
          mat[p * size + p] = 1.0 / pivot;
        }
      }
    }
    free(hasPivotChanged);
    if (nbRemaining > 0) {
      MatFree(&res);
      return NULL;
    }
  }
  // Return the result
  return res;
}

// Return the product of matrix 'that' and vector 'v'
// Number of colum of 'that' must equal dimension of 'v'
VecFloat* _MatFloatGetProdVecFloat(
  const MatFloat* const that, const 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) != VecGetDim(v)) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg,
      "the matrix and vector have incompatible dimensions (%d==%ld)",
      VecGet(&(that->_dim), 0), VecGetDim(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); VecSetAdd(&i, 0, 1))
      for (VecSet(&i, 1, 0); VecGet(&i, 1) < VecGet(&(that->_dim), 1); VecSetAdd(&i, 1, 1))
        VecSetAdd(ret, VecGet(&i, 1),
          VecGet(v, VecGet(&i, 0)) * MatGet(that, &i));
  // Return the result
  return ret;
}

// Return the product of vector 'v' and transpose of vector 'w'
MatFloat* _MatFloatGetProdVecVecTransposeFloat(
  const VecFloat* const v,
  const VecFloat* const w) {
#if BUILDMODE == 0
  if (v == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'v' is null");
    PBErrCatch(PBMathErr);
  }
  if (w == NULL) {
```

```
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'w' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable to memorize the position in the matrix
  VecShort2D pos = VecShortCreateStatic2D();
  // Allocate memory for the solution
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, VecGetDim(w));
  VecSet(&dim, 1, VecGetDim(v));
  MatFloat* ret = MatFloatCreate(&dim);
  // Calculate the result
  do {
    MatSet(ret, &pos,
      VecGet(v, VecGet(&pos, 1)) * VecGet(w, VecGet(&pos, 0)));
  } while(VecStep(&pos, &dim));
  // 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* _MatFloatGetProdMatFloat(const MatFloat* const that, const 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 (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); VecSetAdd(&i, 0, 1))
    for (VecSet(&i, 1, 0); VecGet(&i, 1) < VecGet(&(that->_dim), 1); VecSetAdd(&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);
        VecSetAdd(&j, 0, 1),
        VecSetAdd(&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* const 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);
  }
#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;
}

// Calculate the Eigen values and vectors of the MatFloat 'that'
// Return a set of VecFloat. The first VecFloat of the set contains
// the Eigen values, with values sorted from biggest to
// smallest (in absolute value). The following VecFloat are the
// respectiev Eigen vectors
// 'that' must be a 2D square matrix
// Return the identity if the QR decompostion fails
// http://madrury.github.io/jekyll/update/statistics/2017/10/04/qr-algorithm.html
// TODO: should be improved with the Hessenberg QR method
// https://www.math.kth.se/na/SF2524/matber15/qrmethod.pdf
GSetVecFloat _MatFloatGetEigenValues(const MatFloat* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGet(MatDim(that), 0) != VecGet(MatDim(that), 1)) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' is not squared");
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare the result set
  GSetVecFloat set = GSetVecFloatCreateStatic();
  // Clone the original matrix
  MatFloat* A = MatClone(that);
  // Create a matrix to compute the Eigen vectors
  MatFloat* Q = MatFloatCreate(MatDim(that));
  MatSetIdentity(Q);
  // Apply the QR algorithm
  VecShort2D pos = VecShortCreateStatic2D();
  float err = 0.0;
  do {
    QRDecomp QR = MatGetQR(A);
```

```
    if (QR._Q != NULL) {
      MatFloat* RQ = MatGetProdMat(QR._R, QR._Q);
      MatFree(&A);
      A = RQ;
      MatFloat* M = MatGetProdMat(Q, QR._Q);
      MatFree(&Q);
      Q = M;
      float newErr = 0.0;
      do {
        if (VecGet(&pos, 0) != VecGet(&pos, 1))
          newErr = MAX(newErr, fabs(MatGet(A, &pos)));
      } while (VecStep(&pos, MatDim(A)));
      if (!ISEQUALF(newErr, err))
        err = newErr;
      else
        err = 0.0;
      QRDecompFreeStatic(&QR);
    } else {
      MatSetIdentity(A);
      MatSetIdentity(Q);
      err = 0.0;
    }
  } while (err > PBMATH_EPSILON);
  // Extract the results
  VecFloat* values = VecFloatCreate(MatGetNbCol(that));
  GSetPush(&set, values);
  for (int i = 0; i < MatGetNbCol(that); ++i) {
    VecSet(&pos, 0, i);
    VecSet(&pos, 1, i);
    VecSet(values, i, MatGet(A, &pos));
    GSetAppend(&set, VecFloatCreate(MatGetNbCol(that)));
  }
  VecSetNull(&pos);
  do {
    VecSet(GSetGet(&set, 1 + VecGet(&pos, 0)), VecGet(&pos, 1),
      MatGet(Q, &pos));
  } while (VecStep(&pos, MatDim(Q)));
  // Free memory
  MatFree(&A);
  MatFree(&Q);
  // Return the result
  return set;
}

// Calculate the QR decomposition of the MatFloat 'that' using the
// Householder algorithm
// Return {NULL, NULL} if the MatFloat couldn't be decomposed
// http://www.seas.ucla.edu/~vandenbe/133A/lectures/qr.pdf
QRDecomp _MatFloatGetQR(const MatFloat* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (MatGetNbCol(that) > MatGetNbRow(that)) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg,
      "'that' must have at least as many rows as columns (%d<=%d)",
      MatGetNbCol(that), MatGetNbRow(that));
    PBErrCatch(PBMathErr);
  }
```

```
#endif

  // Allocate memory for the final R matrix
  VecShort2D dimR = VecShortCreateStatic2D();
  VecSet(&dimR, 0, MatGetNbCol(that));
  VecSet(&dimR, 1, MatGetNbCol(that));
  MatFloat* R = MatFloatCreate(&dimR);

  // Allocate memory for the final Q matrix
  MatFloat* Q = MatFloatCreate(MatDim(that));

  // Allocate memory for the QQ~ matrix
  VecShort2D dimQQtilde = VecShortCreateStatic2D();
  VecSet(&dimQQtilde, 0, MatGetNbRow(that));
  VecSet(&dimQQtilde, 1, MatGetNbRow(that));
  MatFloat* QQtilde = MatFloatCreate(&dimQQtilde);
  MatSetIdentity(QQtilde);

  // Create a clone of that to be overwritten during computation
  MatFloat* A = MatClone(that);

  // Declare two vectors to access value in the arrays
  VecShort2D pos = VecShortCreateStatic2D();
  VecShort2D shiftPos = VecShortCreateStatic2D();

  // Householder algorithm
  for (short k = 0; k < MatGetNbCol(that); ++k) {
    // Calculate w
    VecFloat* w = VecFloatCreate(MatGetNbRow(that) - k);
    VecSet(&pos, 0, k);
    for (short i = 0; i < VecGetDim(w); ++i) {
      VecSet(&pos, 1, k + i);
      VecSet(w, i, MatGet(A, &pos));
    }
    if(fabs(VecNorm(w))<0.0000000001) {
      MatFree(&R);
      MatFree(&Q);
      MatFree(&QQtilde);
      MatFree(&A);
      VecFree(&w);
      return (QRDecomp){._Q = NULL, ._R = NULL};
    }
    float sign = (VecGet(w, 0) >= 0.0 ? 1.0 : -1.0);
    VecSet(w, 0, VecGet(w, 0) + sign * VecNorm(w));

    // Calculate v = w / ||w||
    VecFloat* v = VecClone(w);
    VecNormalise(v);

    //                          I 0
    // Calculate the reflector 0 H where H = I - 2vv^t
    VecShort2D dimH = VecShortCreateStatic2D();
    VecSet(&dimH, 0, VecGetDim(v));
    VecSet(&dimH, 1, VecGetDim(v));
    MatFloat* H = MatFloatCreate(&dimH);
    MatSetIdentity(H);
    MatFloat* vvt = MatGetProdVecVecTranspose(v, v);
    MatScale(vvt, -2.0);
    MatAdd(H, vvt);
    MatFloat* reflector = MatFloatCreate(&dimQQtilde);
    MatSetIdentity(reflector);
    VecSetNull(&pos);
```

```
    do {
      VecSet(&shiftPos, 0, VecGet(&pos, 0) + k);
      VecSet(&shiftPos, 1, VecGet(&pos, 1) + k);
      MatSet(reflector, &shiftPos, MatGet(H, &pos));
    } while (VecStep(&pos, &dimH));

    // Update A := reflector . A
    MatFloat* M = MatGetProdMat(reflector, A);
    MatFree(&A);
    A = M;

    // Update QQtilde := QQtilde.reflector
    M = MatGetProdMat(QQtilde, reflector);
    MatFree(&QQtilde);
    QQtilde = M;

    // Free memory
    MatFree(&reflector);
    MatFree(&H);
    MatFree(&vvt);
    VecFree(&v);
    VecFree(&w);
  }

  // Extract R from the final A
  VecSetNull(&pos);
  do {
    MatSet(R, &pos, MatGet(A, &pos));
  } while (VecStep(&pos, &dimR));

  // Extract Q from the final QQtilde
  VecSetNull(&pos);
  do {
    MatSet(Q, &pos, MatGet(QQtilde, &pos));
  } while (VecStep(&pos, MatDim(that)));

  // Create the result QR decomposition
  QRDecomp qr = {._Q = Q, ._R = R};

  // Free memory
  MatFree(&A);
  MatFree(&QQtilde);

  // Return the decomposition
  return qr;
}

// Calculate the transposed of the MatFloat 'that'
MatFloat* _MatFloatGetTranspose(const MatFloat* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Allocate memory for the result matrix
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, VecGet(MatDim(that), 1));
  VecSet(&dim, 1, VecGet(MatDim(that), 0));
  MatFloat* res = MatFloatCreate(&dim);
  // Calculate the transposed matrix
```

```
  VecShort2D pos = VecShortCreateStatic2D();
  VecShort2D posB = VecShortCreateStatic2D();
  do {
    VecSet(&posB, 0, VecGet(&pos, 1));
    VecSet(&posB, 1, VecGet(&pos, 0));
    MatSet(res, &pos, MatGet(that, &posB));
  } while (VecStep(&pos, &dim));
  // Return the transposed matrix
  return res;
}


// -------------- 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(const float mean, const 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(const float mean, const 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(const MatFloat* const m, const VecFloat* const v) {
#if BUILDMODE == 0
  if (m == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'m' is null");
    PBErrCatch(PBMathErr);
```

```
  }
  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) != VecGetDim(v)) {
      PBMathErr->_type = PBErrTypeInvalidArg;
      sprintf(PBMathErr->_msg,
        "the matrix and vector have incompatible dimensions (%d==%ld)",
        VecGet(&(m->_dim), 0), VecGetDim(v));
      PBErrCatch(PBMathErr);
    }
  }
#endif
  // Allocate memory
  SysLinEq* that = PBErrMalloc(PBMathErr, sizeof(SysLinEq));
  that->_M = MatClone(m);
  that->_Minv = MatGetInv(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) {
#if 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(const SysLinEq* const that) {
#if 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;
}

// -------------- Ratio

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

// Create a new static Ratio
Ratio RatioCreateStatic(long b, unsigned int n, unsigned int d) {
#if BUILDMODE == 0
  if (d == 0) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'d' is invalid (%u > 0)", d);
    PBErrCatch(PBMathErr);
  }
#endif

  // Create the Ratio
  Ratio ratio = {
    ._base = b,
    ._numerator = n,
    ._denominator = d
  };

  // Return the Ratio
  return ratio;

}

// Convert the float 'v' into the nearest Ratio using the Farey's algorithm
// given the precision 'prec'
Ratio RatioFromFloatPrec(float v, float prec) {

  // Create the two bounding Ratio
  Ratio ratioLow = RatioCreateStatic(0, 0, 1);
  Ratio ratioHigh = RatioCreateStatic(0, 1, 1);

  // Create the result ratio
  Ratio ratio = RatioCreateStatic(floor(v), 0, 1);

  // Get the decimals of 'v'
  float dec = v - RatioGetBase(&ratio);
```

```
  // Loop until the bounding Ratio reachs the requested precision
  Ratio mediant;
  while(
    RatioToFloat(&ratioHigh) - RatioToFloat(&ratioLow) > prec &&
    fabs(RatioToFloat(&ratioHigh) - dec) > prec &&
    fabs(RatioToFloat(&ratioLow) - dec) > prec) {

    mediant = RatioCreateStatic(0,
      RatioGetNumerator(&ratioLow) + RatioGetNumerator(&ratioHigh),
      RatioGetDenominator(&ratioLow) + RatioGetDenominator(&ratioHigh));

    if (RatioToFloat(&mediant) > dec) {

      ratioHigh = mediant;

    } else {

      ratioLow = mediant;

    }

  }

  // Update the fractional part of the result
  if (fabs(RatioToFloat(&ratioHigh) - dec) <= prec) {

    RatioSetNumerator(&ratio, RatioGetNumerator(&ratioHigh));
    RatioSetDenominator(&ratio, RatioGetDenominator(&ratioHigh));

  } else if (fabs(RatioToFloat(&ratioLow) - dec) <= prec) {

    RatioSetNumerator(&ratio, RatioGetNumerator(&ratioLow));
    RatioSetDenominator(&ratio, RatioGetDenominator(&ratioLow));

  } else {

    RatioSetNumerator(&ratio, RatioGetNumerator(&mediant));
    RatioSetDenominator(&ratio, RatioGetDenominator(&mediant));

  }

  // Reduce the result
  RatioReduce(&ratio);

  // Return the result
  return ratio;

}

// Convert the Ratio 'that' into a float
float RatioToFloat(const Ratio* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Return the Ratio converted to float
  return (float)RatioGetBase(that) +
    (float)RatioGetNumerator(that) / (float)RatioGetDenominator(that);
}
```

```
// Reduce the fractional part of the Ratio 'that' and update the base such as
// numerator < denominator
void RatioReduce(Ratio* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif

  // If the numerator is greater than the denominator
  if (RatioGetNumerator(that) >= RatioGetDenominator(that)) {

    // Update the component to keep the fractional part less than 1.0
    unsigned int delta = RatioGetNumerator(that) / RatioGetDenominator(that);
    RatioSetBase(that, RatioGetBase(that) + delta);
    RatioSetNumerator(that,
      RatioGetNumerator(that) - delta * RatioGetDenominator(that));

  }

  // Get the GCD of the numerator and denominator
  unsigned int div = GetGCD(
    RatioGetNumerator(that), RatioGetDenominator(that));
  // Divide the numerator and denominator by the gcd
  RatioSetNumerator(that, RatioGetNumerator(that) / div);
  RatioSetDenominator(that, RatioGetDenominator(that) / div);

}

// Print the Ratio on 'stream' as a+b/c
void RatioPrint(const Ratio* 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

  fprintf(stream, "%ld+%u/%u",
    RatioGetBase(that), RatioGetNumerator(that), RatioGetDenominator(that));

}

// -------------- Usefull basic functions

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

// Compute the 'iElem'-th element of the 'base'-ary version of the
// Thue-Morse sequence
// 'iElem' >= 0
// 'base' >= 2
long ThueMorseSeqGetNthElem(long iElem, long base) {
#if BUILDMODE == 0
```

```
  if (iElem < 0) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'iElem' is invalid (%ld>=0)", iElem);
    PBErrCatch(PBMathErr);
  }
  if (base < 2) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'base' is invalid (%ld>=2)", base);
    PBErrCatch(PBMathErr);
  }
#endif
  if (base > iElem) {
    return iElem;
  } else {
    ldiv_t d = ldiv(iElem, base);
    return (ThueMorseSeqGetNthElem(d.quot, base) + d.rem) % base;
  }
}

// Return a set of two vectors containing the bounds of the vectors in
// the GSet 'that'
// The set must have at least one element
// The returned set is ordered as follow: (boundMin, boundMax)
GSetVecFloat _GSetVecFloatGetBounds(const GSetVecFloat* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (GSetNbElem(that) < 1) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' is empty");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the set containing the bounds
  GSetVecFloat bounds = GSetVecFloatCreateStatic();
  // Create the two bounds vector, initialised with the first vector of
  // the set
  VecFloat* boundMin = _VecFloatClone(GSetGet(that, 0));
  VecFloat* boundMax = _VecFloatClone(GSetGet(that, 0));
  GSetAppend(&bounds, boundMin);
  GSetAppend(&bounds, boundMax);
  // Get the dimension of the vectors, supposes they are all with same
  // dimension
  long dim = _VecFloatGetDim(boundMin);
  // Loop on the vectors of the set, expect the first one
  GSetIterForward iter = GSetIterForwardCreateStatic(that);
  while (GSetIterStep(&iter)) {
    VecFloat* v = GSetIterGet(&iter);
    // Loop on dimension
    for (int iDim = dim; iDim--;) {
      // Update bounds
      if (_VecFloatGet(boundMin, iDim) > _VecFloatGet(v, iDim))
        _VecFloatSet(boundMin, iDim, _VecFloatGet(v, iDim));
      if (_VecFloatGet(boundMax, iDim) < _VecFloatGet(v, iDim))
        _VecFloatSet(boundMax, iDim, _VecFloatGet(v, iDim));
    }
  }
  // Return the set containing the bounds
  return bounds;
```

```
}

// Compute the area of a triangle knowing its 3 sides length 'a', 'b', 'c'
// using the Hero's formula
double GetAreaTriangleHero(
  const double a,
  const double b,
  const double c) {

  double s = 0.5 * (a + b + c);
  double area = sqrt(s * (s - a) * (s - b) * (s - c));
  return area;

}

// Return the Fibonacci sequence up to the 'n'-th element in a dynamically
// allocated array of unsigned long
unsigned long* GetFibonacciSeq(unsigned int n) {

  if (n == 0) {

    return NULL;

  } else {

    unsigned long* seq =
      PBErrMalloc(
        PBMathErr,
        sizeof(unsigned long) * n);
    for (
      unsigned int i = 0;
      i < n && i < 2;
      ++i) {

      seq[i] = 1L;

    }
    for (
      unsigned int i = 2;
      i < n;
      ++i) {

      seq[i] = seq[i - 1] + seq[i - 2];

    }

    return seq;

  }

}

// Return the Fibonacci grid lattice for the 'n'-th Fibonacci number in a
// dynamically allocated array of pairs of float in [0,1]
// Stores the nb of points in 'nbPoints'
float* GetFibonacciGridLattice(
    unsigned int n,
  unsigned long* nbPoints) {

#if BUILDMODE == 0

  if (nbPoints == NULL) {
```

```
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'nbPoints' is null");
    PBErrCatch(PBMathErr);

  }

#endif

  if (n == 0) {

    *nbPoints = 0;
    return NULL;

  } else {

    // Get the Fibonacci sequence
    unsigned long* seq = GetFibonacciSeq(n);

    // Update the number of points
    *nbPoints = seq[n - 1];

    // Allocate memory for the result
    float* lattice =
      PBErrMalloc(
        PBMathErr,
        sizeof(float) * 2L * (*nbPoints));

    // Generate the lattice points
    for (
      unsigned long iPoint = 0;
      iPoint < *nbPoints;
      ++iPoint) {

      lattice[iPoint * 2L] =
        fmodf(
          (float)iPoint / (float)seq[n > 1 ? n - 1 : 0],
          1.0);
      lattice[iPoint * 2L + 1L] =
        fmodf(
          (float)iPoint * (float)seq[n > 2 ? n - 2 : 0] /
            (float)seq[n > 1 ? n - 1 : 0],
          1.0);

    }

    // Free memory
    free(seq);

    // Return the lattice
    return lattice;

  }

}

// Return the Fibonacci polar lattice for the 'n'-th Fibonacci number in a
// dynamically allocated array of pairs of float in [-1,1]
// Stores the nb of points in 'nbPoints'
float* GetFibonacciPolarLattice(
    unsigned int n,
  unsigned long* nbPoints) {
```

```
#if BUILDMODE == 0

  if (nbPoints == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'nbPoints' is null");
    PBErrCatch(PBMathErr);

  }

#endif

  // Get the grid lattice
  float* lattice =
    GetFibonacciGridLattice(
      n,
      nbPoints);

  // Convert each points to polar coordinates
  for (
    unsigned long iPoint = *nbPoints;
    iPoint--;) {

    lattice[iPoint * 2L] = sqrt(lattice[iPoint * 2L]);
    lattice[iPoint * 2L + 1L] = 2.0 * PBMATH_PI * lattice[iPoint * 2L + 1L];

  }

  // Return the lattice
  return lattice;

}

// Return the greatest common divisor using the Stein's algorithm
// https://en.wikipedia.org/wiki/Binary_GCD_algorithm
unsigned int GetGCD(unsigned int u, unsigned int v) {

    unsigned int shift = 0;
    if (u == 0)
      return v;
    if (v == 0)
      return u;
    while (((u | v) & 1) == 0) {
        ++shift;
        u >>= 1;
        v >>= 1;
    }
    while ((u & 1) == 0)
        u >>= 1;
    do {
        while ((v & 1) == 0)
            v >>= 1;
        if (u > v) {
            unsigned int t = v;
            v = u;
            u = t;
        }
        v -= u;
    } while (v != 0);
    return u << shift;
```

```
}

// Get the approximated inverse square root of a number using the Quake
// algorithm
// cf https://en.wikipedia.org/wiki/Fast_inverse_square_root
float GetFastInverseSquareRoot(float number) {

  const float x2 = number * 0.5F;
  const float threehalfs = 1.5F;

  union {
    float f;
    uint32_t i;
  } conv  = { .f = number };

  conv.i  = 0x5f3759df - ( conv.i >> 1 );
  conv.f  *= threehalfs - ( x2 * conv.f * conv.f );
  return conv.f;

}
// -------------- LeastSquareLinReg

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

// Create a new static LeastSquareLinReg
LeastSquareLinReg LeastSquareLinRegCreateStatic(MatFloat* X) {

#if BUILDMODE == 0

  if (X == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'X' is null");
    PBErrCatch(PBMathErr);

  }

#endif

  // Declare the new LeastSquareLin
  LeastSquareLinReg that;

  // Set the properties
  that.Xp = NULL;
  LSLRSetComp(
    &that,
    X);
  that.bias = 0.0;

  // Return the new LeastSquareLin
  return that;

}

// Free the static LeastSquareLinReg 'that'
void LeastSquareLinRegFreeStatic(LeastSquareLinReg* that) {

  if (that == NULL) {

    return;

  }
```

```
  // Free memory
  MatFree(&(that->Xp));

}

// Compute the solution of the LeastSquareLinReg 'that' for 'Y'
VecFloat* LSLRSolve(LeastSquareLinReg* that, const VecFloat* Y) {

#if BUILDMODE == 0

  if (that == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);

  }

  if (that->X == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that->X' is null");
    PBErrCatch(PBMathErr);

  }

  if (that->Xp == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that->Xp' is null");
    PBErrCatch(PBMathErr);

  }

  if (Y == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'Y' is null");
    PBErrCatch(PBMathErr);

  }

#endif

  // Get the result
  VecFloat* beta =
    MatGetProdVec(
      that->Xp,
      Y);

  // Calculate the bias
  VecFloat* Ybeta =
    MatGetProdVec(
      that->X,
      beta);
  VecOp(
    Ybeta,
    1.0,
    Y,
    -1.0);
  that->bias = VecNorm(Ybeta);
```

```
  VecFree(&Ybeta);

  // Return the result
  return beta;

}

// -------------- Quaternion

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

// Create a new static Quaternion
Quaternion QuaternionCreateStatic(void) {

  // Declare the new Quaternion
  Quaternion that;

  // Initialise the properties
  that.val = VecFloatCreateStatic4D();
  VecSet(&(that.val), 3, 1.0);

  // Return the Quaternion
  return that;

}

// Free the static Quaternion 'that'
void QuaternionFreeStatic(Quaternion* that) {

  if (that == NULL) {

    return;

  }

  // Nothing to do

}

// Create a new static Quaternion from the rotation matrix 'rotMat'
Quaternion QuaternionCreateFromRotMat(MatFloat* rotMat) {
#if BUILDMODE == 0

  if (rotMat == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);

  }

  if (VecGet(MatDim(rotMat), 0) != 3 ||
    VecGet(MatDim(rotMat), 1) != 3) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'rotMat' is not a 3x3 matrix (%d,%d)",
      VecGet(MatDim(rotMat), 0), VecGet(MatDim(rotMat), 1));
    PBErrCatch(PBMathErr);

  }

#endif
```

```
// Create the new quaternion
Quaternion that = QuaternionCreateStatic();

// Calculate the components of the quaternion
float sumDiag = 1.0;
VecShort2D pos = VecShortCreateStatic2D();
float diagVal[3];
for (int i = 3; i--;) {

  VecSet(&pos, 0, i);
  VecSet(&pos, 1, i);
  diagVal[i] = MatGet(rotMat, &pos);
  sumDiag += diagVal[i];

}

if (sumDiag > 0.0) {

  float s = sqrt(sumDiag) * 2.0;

  VecSet(&pos, 0, 1);
  VecSet(&pos, 1, 2);
  float v = MatGet(rotMat, &pos);
  VecSet(&pos, 0, 2);
  VecSet(&pos, 1, 1);
  v -= MatGet(rotMat, &pos);
  v /= s;
  VecSet(&(that.val), 0, v);

  VecSet(&pos, 0, 2);
  VecSet(&pos, 1, 0);
  v = MatGet(rotMat, &pos);
  VecSet(&pos, 0, 0);
  VecSet(&pos, 1, 2);
  v -= MatGet(rotMat, &pos);
  v /= s;
  VecSet(&(that.val), 1, v);

  VecSet(&pos, 0, 0);
  VecSet(&pos, 1, 1);
  v = MatGet(rotMat, &pos);
  VecSet(&pos, 0, 1);
  VecSet(&pos, 1, 0);
  v -= MatGet(rotMat, &pos);
  v /= s;
  VecSet(&(that.val), 2, v);

  VecSet(&(that.val), 3, 0.25 * s);

} else {

  if (diagVal[0] > diagVal[1] && diagVal[0] > diagVal[2]) {

    float s = sqrt(1.0 + diagVal[0] - diagVal[1] - diagVal[2]) * 2.0;

    VecSet(&(that.val), 0, 0.25 * s);

    VecSet(&pos, 0, 0);
    VecSet(&pos, 1, 1);
    float v = MatGet(rotMat, &pos);
    VecSet(&pos, 0, 1);
```

```
    VecSet(&pos, 1, 0);
    v += MatGet(rotMat, &pos);
    v /= s;
    VecSet(&(that.val), 1, v);

    VecSet(&pos, 0, 2);
    VecSet(&pos, 1, 0);
    v = MatGet(rotMat, &pos);
    VecSet(&pos, 0, 0);
    VecSet(&pos, 1, 2);
    v += MatGet(rotMat, &pos);
    v /= s;
    VecSet(&(that.val), 2, v);

    VecSet(&pos, 0, 1);
    VecSet(&pos, 1, 2);
    v = MatGet(rotMat, &pos);
    VecSet(&pos, 0, 2);
    VecSet(&pos, 1, 1);
    v -= MatGet(rotMat, &pos);
    v /= s;
    VecSet(&(that.val), 3, v);

  } else if (diagVal[1] > diagVal[2]) {

    float s = sqrt(1.0 - diagVal[0] + diagVal[1] - diagVal[2]) * 2.0;

    VecSet(&pos, 0, 0);
    VecSet(&pos, 1, 1);
    float v = MatGet(rotMat, &pos);
    VecSet(&pos, 0, 1);
    VecSet(&pos, 1, 0);
    v += MatGet(rotMat, &pos);
    v /= s;
    VecSet(&(that.val), 0, v);

    VecSet(&(that.val), 1, 0.25 * s);

    VecSet(&pos, 0, 1);
    VecSet(&pos, 1, 2);
    v = MatGet(rotMat, &pos);
    VecSet(&pos, 0, 2);
    VecSet(&pos, 1, 1);
    v += MatGet(rotMat, &pos);
    v /= s;
    VecSet(&(that.val), 2, v);

    VecSet(&pos, 0, 2);
    VecSet(&pos, 1, 0);
    v = MatGet(rotMat, &pos);
    VecSet(&pos, 0, 0);
    VecSet(&pos, 1, 2);
    v -= MatGet(rotMat, &pos);
    v /= s;
    VecSet(&(that.val), 3, v);

  } else {

    float s = sqrt(1.0 - diagVal[0] - diagVal[1] + diagVal[2]) * 2.0;

    VecSet(&pos, 0, 2);
    VecSet(&pos, 1, 0);
```

```
      float v = MatGet(rotMat, &pos);
      VecSet(&pos, 0, 0);
      VecSet(&pos, 1, 2);
      v += MatGet(rotMat, &pos);
      v /= s;
      VecSet(&(that.val), 0, v);

      VecSet(&pos, 0, 1);
      VecSet(&pos, 1, 2);
      v = MatGet(rotMat, &pos);
      VecSet(&pos, 0, 2);
      VecSet(&pos, 1, 1);
      v += MatGet(rotMat, &pos);
      v /= s;
      VecSet(&(that.val), 2, v);

      VecSet(&(that.val), 1, 0.25 * s);

      VecSet(&pos, 0, 0);
      VecSet(&pos, 1, 1);
      v = MatGet(rotMat, &pos);
      VecSet(&pos, 0, 1);
      VecSet(&pos, 1, 0);
      v -= MatGet(rotMat, &pos);
      v /= s;
      VecSet(&(that.val), 3, v);

    }

  }

  // Return the quaternion
  return that;

}

// Create a new static Quaternion corresponding to the rotation around
// 'axis' (must be normalized) by 'theta' (in radians)
Quaternion QuaternionCreateFromRotAxis(VecFloat* axis, float theta) {

#if BUILDMODE == 0

  if (axis == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'axis' is null");
    PBErrCatch(PBMathErr);

  }

  if (VecGetDim(axis) != 3) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'axis' must be of dimension 3 (was %ld)",
      VecGetDim(axis));
    PBErrCatch(PBMathErr);

  }

#endif

  // Create the new quaternion
```

```
  Quaternion that = QuaternionCreateStatic();

  // Set the components of the quaternion
  VecSet(&(that.val), 0, VecGet(axis, 0) * sin(theta / 2.0));
  VecSet(&(that.val), 1, VecGet(axis, 1) * sin(theta / 2.0));
  VecSet(&(that.val), 2, VecGet(axis, 2) * sin(theta / 2.0));
  VecSet(&(that.val), 3, cos(theta / 2.0));

  // Return the quaternion
  return that;

}

// Convert the Quaternion 'that' to a rotation matrix
MatFloat* QuaternionToRotMat(Quaternion* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);

  }

#endif

  // Create the rotation matrix
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3);
  VecSet(&dim, 1, 3);
  MatFloat* rotMat = MatFloatCreate(&dim);

  // Set the components of the matrix

  float x2 = VecGet(&(that->val), 0) * 2.0;
  float y2 = VecGet(&(that->val), 1) * 2.0;
  float z2 = VecGet(&(that->val), 2) * 2.0;
  float xx = VecGet(&(that->val), 0) * x2;
  float xy = VecGet(&(that->val), 0) * y2;
  float xz = VecGet(&(that->val), 0) * z2;
  float yy = VecGet(&(that->val), 1) * y2;
  float yz = VecGet(&(that->val), 1) * z2;
  float zz = VecGet(&(that->val), 2) * z2;
  float wx = VecGet(&(that->val), 3) * x2;
  float wy = VecGet(&(that->val), 3) * y2;
  float wz = VecGet(&(that->val), 3) * z2;
  VecShort2D pos = VecShortCreateStatic2D();
  VecSet(&pos, 0, 0);
  VecSet(&pos, 1, 0);
  MatSet(rotMat, &pos, 1.0 - (yy + zz));
  VecSet(&pos, 0, 0);
  VecSet(&pos, 1, 1);
  MatSet(rotMat, &pos, xy + wz);
  VecSet(&pos, 0, 0);
  VecSet(&pos, 1, 2);
  MatSet(rotMat, &pos, xz - wy);
  VecSet(&pos, 0, 1);
  VecSet(&pos, 1, 0);
  MatSet(rotMat, &pos, xy - wz);
  VecSet(&pos, 0, 1);
```

```
  VecSet(&pos, 1, 1);
  MatSet(rotMat, &pos, 1.0 - (xx + zz));
  VecSet(&pos, 0, 1);
  VecSet(&pos, 1, 2);
  MatSet(rotMat, &pos, yz + wx);
  VecSet(&pos, 0, 2);
  VecSet(&pos, 1, 0);
  MatSet(rotMat, &pos, xz + wy);
  VecSet(&pos, 0, 2);
  VecSet(&pos, 1, 1);
  MatSet(rotMat, &pos, yz - wx);
  VecSet(&pos, 0, 2);
  VecSet(&pos, 1, 2);
  MatSet(rotMat, &pos, 1.0 - (xx + yy));

  // Return the rotation matrix
  return rotMat;

}

// Return the quaternion equivalent to the rotation of 'that' followed by
// the rotation of 'tho'
Quaternion QuaternionGetComposition(Quaternion* that, Quaternion* 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

  // Create the result quaternion
  Quaternion quat = QuaternionCreateStatic();

  // Calculate the addition of rotation
  VecSet(&(quat.val), 0,
    VecGet(&(that->val), 0) * VecGet(&(tho->val), 3) +
    VecGet(&(that->val), 3) * VecGet(&(tho->val), 0) +
    VecGet(&(that->val), 1) * VecGet(&(tho->val), 2) -
    VecGet(&(that->val), 2) * VecGet(&(tho->val), 1));
  VecSet(&(quat.val), 1,
    VecGet(&(that->val), 1) * VecGet(&(tho->val), 3) +
    VecGet(&(that->val), 3) * VecGet(&(tho->val), 1) +
    VecGet(&(that->val), 2) * VecGet(&(tho->val), 0) -
    VecGet(&(that->val), 0) * VecGet(&(tho->val), 2));
  VecSet(&(quat.val), 2,
    VecGet(&(that->val), 2) * VecGet(&(tho->val), 3) +
    VecGet(&(that->val), 3) * VecGet(&(tho->val), 2) +
    VecGet(&(that->val), 0) * VecGet(&(tho->val), 1) -
```

```
    VecGet(&(that->val), 1) * VecGet(&(tho->val), 0));
  VecSet(&(quat.val), 3,
    VecGet(&(that->val), 3) * VecGet(&(tho->val), 3) -
    VecGet(&(that->val), 0) * VecGet(&(tho->val), 0) -
    VecGet(&(that->val), 1) * VecGet(&(tho->val), 1) -
    VecGet(&(that->val), 2) * VecGet(&(tho->val), 2));

  // Return the result
  return quat;

}

// Return the quaternion equivalent to the rotation necessary to convert
// 'that' into 'tho'
// tho = QuaternionGetComposition(QuaternionGetDifference(that, tho), that)
Quaternion QuaternionGetDifference(Quaternion* that, Quaternion* 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

  // Calculate the difference of rotation
  Quaternion inv = QuaternionGetInverse(that);
  Quaternion quat = QuaternionGetComposition(tho, &inv);
  if (VecGet(&(quat.val), 3) < 0.0) {

    VecScale(&(quat.val), -1.0);

  }

  // Return the result
  return quat;

}

// Return the inverse quaternion of the quaternion 'that'
Quaternion QuaternionGetInverse(Quaternion* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
```

```
  }

#endif

  // Calculate the inverse
  Quaternion quat = *that;
  for (int i = 3; i--;) {

    VecSet(&(quat.val), i, -1.0 * VecGet(&(quat.val), i));

  }

  // Return the result
  return quat;


}

// Return true if the two quaternions are equals, false else
bool QuaternionIsEqual(Quaternion* that, Quaternion* 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 VecIsEqual(&(that->val), &(tho->val));

}

// Print the Quaternion 'that' on 'stream'
void QuaternionPrint(Quaternion* that, FILE* stream) {

#if BUILDMODE == 0

  if (that == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);

  }

#endif

  VecPrint(&(that->val), stream);
```

```
}

// Rotate the vector 'v' by the quaternion 'that'
void QuaternionApply(Quaternion* 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 (VecGetDim(v) != 3) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'v' must be of dimension 3 (was %ld)",
      VecGetDim(v));
    PBErrCatch(PBMathErr);

  }

#endif

  // Calculate the result
  Quaternion p = QuaternionCreateStatic();
  VecSet(&(p.val), 0, VecGet(v, 0));
  VecSet(&(p.val), 1, VecGet(v, 1));
  VecSet(&(p.val), 2, VecGet(v, 2));
  VecSet(&(p.val), 3, 0.0);
  Quaternion inv = QuaternionGetInverse(that);
  Quaternion q = QuaternionGetComposition(that, &p);
  Quaternion r = QuaternionGetComposition(&q, &inv);
  VecSet(v, 0, VecGet(&(r.val), 0));
  VecSet(v, 1, VecGet(&(r.val), 1));
  VecSet(v, 2, VecGet(&(r.val), 2));

}

// Normalise the quaternion
void QuaternionNormalise(Quaternion* that) {
#if BUILDMODE == 0

  if (that == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);

  }

#endif
```

```
  // Normalise the quaternion
  VecNormalise(&(that->val));

}

// Get the rotation axis of the quaternion 'that'
VecFloat3D QuaternionGetRotAxis(Quaternion* that) {
#if BUILDMODE == 0

  if (that == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);

  }

#endif

  // Create the result vector
  VecFloat3D res = VecFloatCreateStatic3D();

  // Calucate the rotation axis
  float sa = sqrt(1.0 - VecGet(&(that->val), 3) * VecGet(&(that->val), 3));
  VecSet(&res, 0, VecGet(&(that->val), 0) / sa);
  VecSet(&res, 1, VecGet(&(that->val), 1) / sa);
  VecSet(&res, 2, VecGet(&(that->val), 2) / sa);
  VecNormalise(&res);

  // Return the result
  return res;

}

// Get the rotation angle (in radians) of the quaternion 'that'
float QuaternionGetRotAngle(Quaternion* that) {
#if BUILDMODE == 0

  if (that == NULL) {

    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);

  }

#endif

  return acos(VecGet(&(that->val), 3)) * 2.0;

}
```

3.2 pbmath-inline.c

```
// ============ PBMATH_static inline.C ================

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

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

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

// Return the i-th value of the VecShort
#if BUILDMODE != 0
static inline
#endif
short _VecShortGet(const VecShort* const that, const long 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<=%ld<%ld)", i,
      that->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  return ((short*)(((void*)that) + sizeof(long)))[i];
}
#if BUILDMODE != 0
static inline
#endif
short _VecShortGet2D(const VecShort2D* const that, const long 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<=%ld<2)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  return that->_val[i];
}
#if BUILDMODE != 0
```

```
static inline
#endif
short _VecShortGet3D(const VecShort3D* const that, const long 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<=%ld<3)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  return that->_val[i];
}
#if BUILDMODE != 0
static inline
#endif
short _VecShortGet4D(const VecShort4D* const that, const long 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<=%ld<4)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  return that->_val[i];
}

// Set the i-th value of the VecShort to v
#if BUILDMODE != 0
static inline
#endif
void _VecShortSet(VecShort* const that, const long i, const 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<=%ld<%ld)", i,
      that->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  ((short*)(((void*)that) + sizeof(long)))[i] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortSet2D(VecShort2D* const that, const long i, const 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<=%ld<2)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortSet3D(VecShort3D* const that, const long i, const 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<=%ld<3)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortSet4D(VecShort4D* const that, const long i, const 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<=%ld<4)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] = v;
}

// Set the i-th value of the VecShort to v plus its current value
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetAdd(VecShort* const that, const long i, const 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<=%ld<%ld)", i,
      that->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  ((short*)(((void*)that) + sizeof(long)))[i] += v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetAdd2D(VecShort2D* const that, const long i, const 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<=%ld<2)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] += v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetAdd3D(VecShort3D* const that, const long i, const 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<=%ld<3)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] += v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortSetAdd4D(VecShort4D* const that, const long i, const 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<=%ld<4)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] += v;
}
```

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

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

// Return the dimension of the VecShort
#if BUILDMODE != 0
static inline
#endif
long _VecShortGetDim(const VecShort* const 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
static inline
#endif
short _VecShortHamiltonDist(const VecShort* const that, const VecShort* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable to calculate the distance
  short ret = 0;
  for (long iDim = VecGetDim(that); iDim--;)
    ret += abs(VecGet(that, iDim) - VecGet(tho, iDim));
  // Return the distance
  return ret;
}
#if BUILDMODE != 0
static inline
#endif
short _VecShortHamiltonDist2D(const VecShort2D* const that, const VecShort2D* const 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
static inline
#endif
short _VecShortHamiltonDist3D(const VecShort3D* const that, const VecShort3D* const 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
static inline
#endif
short _VecShortHamiltonDist4D(const VecShort4D* const that, const VecShort4D* const 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
static inline
#endif
bool _VecShortIsEqual(const VecShort* const that,
  const VecShort* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  return
    (memcmp(that->_val, tho->_val, sizeof(short) * that->_dim) == 0);
}

// Copy the values of 'tho' in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecShortCopy(VecShort* const that, const VecShort* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Copy the values
  memcpy(that->_val, tho->_val, sizeof(short) * that->_dim);
}

// Return the dot product of 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
short _VecShortDotProd(const VecShort* const that,
  const VecShort* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable ot memorise the result
  short res = 0;
  // For each component
  for (long iDim = that->_dim; iDim--;)
    // Calculate the product
    res += VecGet(that, iDim) * VecGet(tho, iDim);
  // Return the result
  return res;
}
#if BUILDMODE != 0
static inline
#endif
short _VecShortDotProd2D(const VecShort2D* const that,
  const VecShort2D* const 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);
}
#if BUILDMODE != 0
static inline
#endif
short _VecShortDotProd3D(const VecShort3D* const that,
  const VecShort3D* const 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);
}
#if BUILDMODE != 0
static inline
#endif
short _VecShortDotProd4D(const VecShort4D* const that,
  const VecShort4D* const 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);
}

// Calculate (that * a + tho * b) and store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecShortOp(VecShort* const that, const short a,
  const VecShort* const tho, const short 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  for (long iDim = that->_dim; iDim--;)
    VecSet(that, iDim,
      a * VecGet(that, iDim) + b * VecGet(tho, iDim));
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortOp2D(VecShort2D* const that, const short a,
  const VecShort2D* const tho, const short 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
static inline
#endif
void _VecShortOp3D(VecShort3D* const that, const short a,
  const VecShort3D* const tho, const short 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));
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortOp4D(VecShort4D* const that, const short a,
  const VecShort4D* const tho, const short 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));
  VecSet(that, 3, a * VecGet(that, 3) + b * VecGet(tho, 3));
}

// Return a VecShort equal to (that * a + tho * b)
#if BUILDMODE != 0
static inline
#endif
VecShort* _VecShortGetOp(const VecShort* const that, const short a,
  const VecShort* const tho, const short 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  VecShort* res = VecShortCreate(that->_dim);
  for (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim,
      a * VecGet(that, iDim) + b * VecGet(tho, iDim));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecShort2D _VecShortGetOp2D(const VecShort2D* const that, const short a,
  const VecShort2D* const tho, const short 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
  VecShort2D res = VecShortCreateStatic2D();
```

```
  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
static inline
#endif
VecShort3D _VecShortGetOp3D(const VecShort3D* const that, const short a,
  const VecShort3D* const tho, const short 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
  VecShort3D res = VecShortCreateStatic3D();
  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;
}
#if BUILDMODE != 0
static inline
#endif
VecShort4D _VecShortGetOp4D(const VecShort4D* const that, const short a,
  const VecShort4D* const tho, const short 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
  VecShort4D res = VecShortCreateStatic4D();
  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));
  VecSet(&res, 3, a * VecGet(that, 3) + b * VecGet(tho, 3));
  return res;
}

// Calculate the Hadamard product of that by tho and store the
// result in 'that'
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
void _VecShortHadamardProd(VecShort* const that,
  const VecShort* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  for (long iDim = that->_dim; iDim--;)
    VecSet(that, iDim, VecGet(that, iDim) * VecGet(tho, iDim));
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortHadamardProd2D(VecShort2D* const that,
  const VecShort2D* const 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
  VecSet(that, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(that, 1, VecGet(that, 1) * VecGet(tho, 1));
}
#if BUILDMODE != 0
static inline
#endif
void _VecShortHadamardProd3D(VecShort3D* const that,
  const VecShort3D* const 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
  VecSet(that, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(that, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(that, 2, VecGet(that, 2) * VecGet(tho, 2));
}
#if BUILDMODE != 0
static inline
```

```
#endif
void _VecShortHadamardProd4D(VecShort4D* const that,
  const VecShort4D* const 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
  VecSet(that, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(that, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(that, 2, VecGet(that, 2) * VecGet(tho, 2));
  VecSet(that, 3, VecGet(that, 3) * VecGet(tho, 3));
}

// Return a VecShort equal to the hadamard product of 'that' and 'tho'
// Return NULL if arguments are invalid
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
VecShort* _VecShortGetHadamardProd(const VecShort* const that,
  const VecShort* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  VecShort* res = VecShortCreate(that->_dim);
  for (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim, VecGet(that, iDim) * VecGet(tho, iDim));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecShort2D _VecShortGetHadamardProd2D(const VecShort2D* const that,
  const VecShort2D* const 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
  VecShort2D res = VecShortCreateStatic2D();
  VecSet(&res, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(&res, 1, VecGet(that, 1) * VecGet(tho, 1));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecShort3D _VecShortGetHadamardProd3D(const VecShort3D* const that,
  const VecShort3D* const 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
  VecShort3D res = VecShortCreateStatic3D();
  VecSet(&res, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(&res, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(&res, 2, VecGet(that, 2) * VecGet(tho, 2));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecShort4D _VecShortGetHadamardProd4D(const VecShort4D* const that,
  const VecShort4D* const 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
  VecShort4D res = VecShortCreateStatic4D();
  VecSet(&res, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(&res, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(&res, 2, VecGet(that, 2) * VecGet(tho, 2));
  VecSet(&res, 3, VecGet(that, 3) * VecGet(tho, 3));
  return res;
}

// Get the max value in components of the vector 'that'
#if BUILDMODE != 0
```

```
static inline
#endif
short _VecShortGetMaxVal(const VecShort* const 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 result
  short max = VecGet(that, 0);
  // Search for the maximum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    max = MAX(max, VecGet(that, i));
  // Return the result
  return max;
}

// Get the index of the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecShortGetIMaxVal(const VecShort* const 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 result
  long iMax = 0;
  // Declare a variable to memorize the max value
  short max = VecGet(that, iMax);
  // Search for the maximum value
  for (long i = VecGetDim(that); i-- && i != 0;) {
    if (max < VecGet(that, i)) {
      max = VecGet(that, i);
      iMax = i;
    }
  }
  // Return the result
  return iMax;
}

// Get the min value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
short _VecShortGetMinVal(const VecShort* const 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 result
  short min = VecGet(that, 0);
  // Search for the minimum value
  for (long i = VecGetDim(that); i-- && i != 0;)
```

```
    min = MIN(min, VecGet(that, i));
  // Return the result
  return min;
}

// Get the max value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
short _VecShortGetMaxValAbs(const VecShort* const 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 result
  short max = abs(VecGet(that, 0));
  // Search for the maximum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    max = (abs(max) > abs(VecGet(that, i)) ? max : VecGet(that, i));
  // Return the result
  return max;
}

// Get the min value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
short _VecShortGetMinValAbs(const VecShort* const 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 result
  short min = abs(VecGet(that, 0));
  // Search for the minimum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    min = (abs(min) < abs(VecGet(that, i)) ? min : VecGet(that, i));
  // Return the result
  return min;
}

// -------------- VecLong

// Static constructors for VecLong
#if BUILDMODE != 0
static inline
#endif
VecLong2D VecLongCreateStatic2D() {
  VecLong2D v = {._val = {0, 0}, ._dim = 2};
  return v;
}
#if BUILDMODE != 0
static inline
#endif
```

```
VecLong3D VecLongCreateStatic3D() {
  VecLong3D v = {._val = {0, 0, 0}, ._dim = 3};
  return v;
}
#if BUILDMODE != 0
static inline
#endif
VecLong4D VecLongCreateStatic4D() {
  VecLong4D v = {._val = {0, 0, 0, 0}, ._dim = 4};
  return v;
}

// Return the i-th value of the VecLong
#if BUILDMODE != 0
static inline
#endif
long _VecLongGet(const VecLong* const that, const long 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<=%ld<%ld)", i,
      that->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  return ((long*)(((void*)that) + sizeof(long)))[i];
}
#if BUILDMODE != 0
static inline
#endif
long _VecLongGet2D(const VecLong2D* const that, const long 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<=%ld<2)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  return that->_val[i];
}
#if BUILDMODE != 0
static inline
#endif
long _VecLongGet3D(const VecLong3D* const that, const long 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<=%ld<3)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  return that->_val[i];
}
#if BUILDMODE != 0
static inline
#endif
long _VecLongGet4D(const VecLong4D* const that, const long 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<=%ld<4)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  return that->_val[i];
}

// Set the i-th value of the VecLong to v
#if BUILDMODE != 0
static inline
#endif
void _VecLongSet(VecLong* const that, const long i, const long 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<=%ld<%ld)", i,
      that->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  ((long*)(((void*)that) + sizeof(long)))[i] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongSet2D(VecLong2D* const that, const long i, const long 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<=%ld<2)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] = v;
```

```
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongSet3D(VecLong3D* const that, const long i, const long 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<=%ld<3)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongSet4D(VecLong4D* const that, const long i, const long 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<=%ld<4)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] = v;
}

// Set the i-th value of the VecLong to v plus its current value
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetAdd(VecLong* const that, const long i, const long 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<=%ld<%ld)", i,
      that->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  ((long*)(((void*)that) + sizeof(long)))[i] += v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetAdd2D(VecLong2D* const that, const long i, const long 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<=%ld<2)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] += v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetAdd3D(VecLong3D* const that, const long i, const long 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<=%ld<3)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] += v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetAdd4D(VecLong4D* const that, const long i, const long 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<=%ld<4)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  that->_val[i] += v;
}

// Set all values of the vector 'that' to 0
#if BUILDMODE != 0
static inline
#endif
void _VecLongSetNull(VecLong* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
```

```
#endif
  // Set values
  for (long iDim = that->_dim; iDim--;)
    that->_val[iDim] = 0;
}

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

// Return the dimension of the VecLong
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetDim(const VecLong* const 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 VecLong 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
long _VecLongHamiltonDist(const VecLong* const that, const VecLong* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable to calculate the distance
  long ret = 0;
```

```
  for (long iDim = VecGetDim(that); iDim--;)
    ret += labs(VecGet(that, iDim) - VecGet(tho, iDim));
  // Return the distance
  return ret;
}
#if BUILDMODE != 0
static inline
#endif
long _VecLongHamiltonDist2D(const VecLong2D* const that, const VecLong2D* const 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 labs(VecGet(that, 0) - VecGet(tho, 0)) +
    labs(VecGet(that, 1) - VecGet(tho, 1));
}
#if BUILDMODE != 0
static inline
#endif
long _VecLongHamiltonDist3D(const VecLong3D* const that, const VecLong3D* const 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 labs(VecGet(that, 0) - VecGet(tho, 0)) +
    labs(VecGet(that, 1) - VecGet(tho, 1)) +
    labs(VecGet(that, 2) - VecGet(tho, 2));
}
#if BUILDMODE != 0
static inline
#endif
long _VecLongHamiltonDist4D(const VecLong4D* const that, const VecLong4D* const 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 labs(VecGet(that, 0) - VecGet(tho, 0)) +
    labs(VecGet(that, 1) - VecGet(tho, 1)) +
    labs(VecGet(that, 2) - VecGet(tho, 2)) +
    labs(VecGet(that, 3) - VecGet(tho, 3));
}

// Return true if the VecLong 'that' is equal to 'tho', else false
#if BUILDMODE != 0
static inline
#endif
bool _VecLongIsEqual(const VecLong* const that,
  const VecLong* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  return
    (memcmp(that->_val, tho->_val, sizeof(long) * that->_dim) == 0);
}

// Copy the values of 'tho' in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecLongCopy(VecLong* const that, const VecLong* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Copy the values
  memcpy(that->_val, tho->_val, sizeof(long) * that->_dim);
}
```

```
// Return the dot product of 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
long _VecLongDotProd(const VecLong* const that,
  const VecLong* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable ot memorise the result
  long res = 0;
  // For each component
  for (long iDim = that->_dim; iDim--;)
    // Calculate the product
    res += VecGet(that, iDim) * VecGet(tho, iDim);
  // Return the result
  return res;
}
#if BUILDMODE != 0
static inline
#endif
long _VecLongDotProd2D(const VecLong2D* const that,
  const VecLong2D* const 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);
}
#if BUILDMODE != 0
static inline
#endif
long _VecLongDotProd3D(const VecLong3D* const that,
  const VecLong3D* const 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);
}
#if BUILDMODE != 0
static inline
#endif
long _VecLongDotProd4D(const VecLong4D* const that,
  const VecLong4D* const 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);
}

// Calculate (that * a + tho * b) and store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecLongOp(VecLong* const that, const long a,
  const VecLong* const tho, const long 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  for (long iDim = that->_dim; iDim--;)
    VecSet(that, iDim,
      a * VecGet(that, iDim) + b * VecGet(tho, iDim));
```

```
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongOp2D(VecLong2D* const that, const long a,
  const VecLong2D* const tho, const long 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
static inline
#endif
void _VecLongOp3D(VecLong3D* const that, const long a,
  const VecLong3D* const tho, const long 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));
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongOp4D(VecLong4D* const that, const long a,
  const VecLong4D* const tho, const long 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));
```

```
  VecSet(that, 3, a * VecGet(that, 3) + b * VecGet(tho, 3));
}

// Return a VecLong equal to (that * a + tho * b)
#if BUILDMODE != 0
static inline
#endif
VecLong* _VecLongGetOp(const VecLong* const that, const long a,
  const VecLong* const tho, const long 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  VecLong* res = VecLongCreate(that->_dim);
  for (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim,
      a * VecGet(that, iDim) + b * VecGet(tho, iDim));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecLong2D _VecLongGetOp2D(const VecLong2D* const that, const long a,
  const VecLong2D* const tho, const long 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
  VecLong2D res = VecLongCreateStatic2D();
  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
static inline
#endif
VecLong3D _VecLongGetOp3D(const VecLong3D* const that, const long a,
  const VecLong3D* const tho, const long 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
  VecLong3D res = VecLongCreateStatic3D();
  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;
}
#if BUILDMODE != 0
static inline
#endif
VecLong4D _VecLongGetOp4D(const VecLong4D* const that, const long a,
  const VecLong4D* const tho, const long 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
  VecLong4D res = VecLongCreateStatic4D();
  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));
  VecSet(&res, 3, a * VecGet(that, 3) + b * VecGet(tho, 3));
  return res;
}

// Calculate the Hadamard product of that by tho and store the
// result in 'that'
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
void _VecLongHadamardProd(VecLong* const that,
  const VecLong* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  for (long iDim = that->_dim; iDim--;)
    VecSet(that, iDim, VecGet(that, iDim) * VecGet(tho, iDim));
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongHadamardProd2D(VecLong2D* const that,
  const VecLong2D* const 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
  VecSet(that, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(that, 1, VecGet(that, 1) * VecGet(tho, 1));
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongHadamardProd3D(VecLong3D* const that,
  const VecLong3D* const 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
  VecSet(that, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(that, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(that, 2, VecGet(that, 2) * VecGet(tho, 2));
}
#if BUILDMODE != 0
static inline
#endif
void _VecLongHadamardProd4D(VecLong4D* const that,
  const VecLong4D* const 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
  VecSet(that, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(that, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(that, 2, VecGet(that, 2) * VecGet(tho, 2));
  VecSet(that, 3, VecGet(that, 3) * VecGet(tho, 3));
}

// Return a VecLong equal to the hadamard product of 'that' and 'tho'
// Return NULL if arguments are invalid
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
VecLong* _VecLongGetHadamardProd(const VecLong* const that,
  const VecLong* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  VecLong* res = VecLongCreate(that->_dim);
  for (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim, VecGet(that, iDim) * VecGet(tho, iDim));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecLong2D _VecLongGetHadamardProd2D(const VecLong2D* const that,
  const VecLong2D* const 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
  VecLong2D res = VecLongCreateStatic2D();
  VecSet(&res, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(&res, 1, VecGet(that, 1) * VecGet(tho, 1));
  return res;
```

```
}
#if BUILDMODE != 0
static inline
#endif
VecLong3D _VecLongGetHadamardProd3D(const VecLong3D* const that,
  const VecLong3D* const 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
  VecLong3D res = VecLongCreateStatic3D();
  VecSet(&res, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(&res, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(&res, 2, VecGet(that, 2) * VecGet(tho, 2));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecLong4D _VecLongGetHadamardProd4D(const VecLong4D* const that,
  const VecLong4D* const 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
  VecLong4D res = VecLongCreateStatic4D();
  VecSet(&res, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(&res, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(&res, 2, VecGet(that, 2) * VecGet(tho, 2));
  VecSet(&res, 3, VecGet(that, 3) * VecGet(tho, 3));
  return res;
}

// Get the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetMaxVal(const VecLong* const 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 result
```

```
  long max = VecGet(that, 0);
  // Search for the maximum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    max = MAX(max, VecGet(that, i));
  // Return the result
  return max;
}

// Get the index of the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetIMaxVal(const VecLong* const 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 result
  long iMax = 0;
  // Declare a variable to memorize the max value
  long max = VecGet(that, iMax);
  // Search for the maximum value
  for (long i = VecGetDim(that); i-- && i != 0;) {
    if (max < VecGet(that, i)) {
      max = VecGet(that, i);
      iMax = i;
    }
  }
  // Return the result
  return iMax;
}

// Get the min value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetMinVal(const VecLong* const 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 result
  long min = VecGet(that, 0);
  // Search for the minimum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    min = MIN(min, VecGet(that, i));
  // Return the result
  return min;
}

// Get the max value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetMaxValAbs(const VecLong* const 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 result
  long max = labs(VecGet(that, 0));
  // Search for the maximum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    max = (labs(max) > labs(VecGet(that, i)) ? max : VecGet(that, i));
  // Return the result
  return max;
}

// Get the min value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecLongGetMinValAbs(const VecLong* const 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 result
  long min = labs(VecGet(that, 0));
  // Search for the minimum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    min = (labs(min) < labs(VecGet(that, i)) ? min : VecGet(that, i));
  // Return the result
  return min;
}

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

// Static constructors for VecFloat
#if BUILDMODE != 0
static inline
#endif
VecFloat2D VecFloatCreateStatic2D() {
  VecFloat2D v = {._val = {0.0, 0.0}, ._dim = 2};
  return v;
}
#if BUILDMODE != 0
static inline
#endif
VecFloat3D VecFloatCreateStatic3D() {
  VecFloat3D v = {._val = {0.0, 0.0, 0.0}, ._dim = 3};
  return v;
}
#if BUILDMODE != 0
static inline
#endif
VecFloat4D VecFloatCreateStatic4D() {
  VecFloat4D v = {._val = {0.0, 0.0, 0.0, 0.0}, ._dim = 4};
  return v;
}
```

```
// Return the i-th value of the VecFloat
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGet(const VecFloat* const that, const long 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<=%ld<%ld)", i, that->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Return the value
  return that->_val[i];
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGet2D(const VecFloat2D* const that, const long 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<=%ld<2)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  // Return the value
  return that->_val[i];
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGet3D(const VecFloat3D* const that, const long 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<=%ld<3)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  // Return the value
  return that->_val[i];
}
#if BUILDMODE != 0
static inline
```

```
#endif
float _VecFloatGet4D(const VecFloat4D* const that, const long 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<=%ld<4)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  // Return the value
  return that->_val[i];
}

// Set the i-th value of the VecFloat to v
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSet(VecFloat* const that, const long i, const 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<=%ld<%ld)", i, that->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Set the value
  that->_val[i] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSet2D(VecFloat2D* const that, const long i, const 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<=%ld<2)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  // Set the value
  that->_val[i] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSet3D(VecFloat3D* const that, const long i, const 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<=%ld<3)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  // Set the value
  that->_val[i] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSet4D(VecFloat4D* const that, const long i, const float 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<=%ld<4)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  // Set the value
  that->_val[i] = v;
}

// Set the i-th value of the VecFloat to v plus its current value
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAdd(VecFloat* const that, const long i, const 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<=%ld<%ld)", i, that->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Set the value
  that->_val[i] += v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAdd2D(VecFloat2D* const that, const long i,
  const 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<=%ld<2)", i);
    PBErrCatch(PBMathErr);
  }
#endif
  // Set the value
  that->_val[i] += v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAdd3D(VecFloat3D* const that, const long i,
  const 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<=%ld<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
static inline
#endif
void _VecFloatSetNull(VecFloat* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Set values
  for (long iDim = that->_dim; iDim--;)
    that->_val[iDim] = 0.0;
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetNull2D(VecFloat2D* const 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
static inline
#endif
void _VecFloatSetNull3D(VecFloat3D* const 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;
}

// Set all values of the vector 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAll(VecFloat* const that, float v) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Set values
  for (long iDim = that->_dim; iDim--;)
    that->_val[iDim] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAll2D(VecFloat2D* const that, float v) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Set values
  that->_val[0] = v;
  that->_val[1] = v;
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatSetAll3D(VecFloat3D* const that, float v) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
```

```
#endif
  // Set values
  that->_val[0] = v;
  that->_val[1] = v;
  that->_val[2] = v;
}

// Return the dimension of the VecFloat
#if BUILDMODE != 0
static inline
#endif
long _VecFloatGetDim(const VecFloat* const 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
static inline
#endif
void _VecFloatCopy(VecFloat* const that, const VecFloat* const 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 (%ld==%ld)",
      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
static inline
#endif
float _VecFloatNorm(const VecFloat* const 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 (long iDim = that->_dim; iDim--;)
    ret += fsquare(VecGet(that, iDim));
  ret = sqrt(ret);
  // Return the result
  return ret;
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatNorm2D(const VecFloat2D* const 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
static inline
#endif
float _VecFloatNorm3D(const VecFloat3D* const 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)));
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatNorm4D(const VecFloat4D* const 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)) + fsquare(VecGet(that, 3)));
}

// Normalise the VecFloat
#if BUILDMODE != 0
static inline
#endif
void _VecFloatNormalise(VecFloat* const 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 (long iDim = that->_dim; iDim--;)
    VecSet(that, iDim, VecGet(that, iDim) / norm);
}

#if BUILDMODE != 0
static inline
#endif
void _VecFloatNormalise2D(VecFloat2D* const 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
static inline
#endif
void _VecFloatNormalise3D(VecFloat3D* const 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);
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatNormalise4D(VecFloat4D* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Normalise
  float norm = _VecFloatNorm4D(that);
  VecSet(that, 0, VecGet(that, 0) / norm);
  VecSet(that, 1, VecGet(that, 1) / norm);
  VecSet(that, 2, VecGet(that, 2) / norm);
  VecSet(that, 3, VecGet(that, 3) / norm);
}

// Return the distance between the VecFloat 'that' and 'tho'
```

```
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDist(const VecFloat* const that,
  const VecFloat* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable to calculate the distance
  float ret = 0.0;
  for (long iDim = that->_dim; iDim--;)
    ret += fsquare(VecGet(that, iDim) - VecGet(tho, iDim));
  ret = sqrt(ret);
  // Return the distance
  return ret;
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDist2D(const VecFloat2D* const that,
  const VecFloat2D* const 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)));
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDist3D(const VecFloat3D* const that,
  const VecFloat3D* const 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)) +
    fsquare(VecGet(that, 2) - VecGet(tho, 2)));
}

// Return the Hamiltonian distance between the VecFloat 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatHamiltonDist(const VecFloat* const that,
  const VecFloat* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable to calculate the distance
  float ret = 0.0;
  for (long iDim = that->_dim; iDim--;)
    ret += fabs(VecGet(that, iDim) - VecGet(tho, iDim));
  // Return the distance
  return ret;
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatHamiltonDist2D(const VecFloat2D* const that,
  const VecFloat2D* const 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
static inline
#endif
float _VecFloatHamiltonDist3D(const VecFloat3D* const that,
  const VecFloat3D* const 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
static inline
#endif
float _VecFloatPixelDist(const VecFloat* const that,
  const VecFloat* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable to calculate the distance
  float ret = 0.0;
  for (long iDim = that->_dim; iDim--;)
    ret += fabs(floor(VecGet(that, iDim)) - floor(VecGet(tho, iDim)));
  // Return the distance
  return ret;
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatPixelDist2D(const VecFloat2D* const that,
  const VecFloat2D* const 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(floor(VecGet(that, 0)) - floor(VecGet(tho, 0))) +
    fabs(floor(VecGet(that, 1)) - floor(VecGet(tho, 1)));
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatPixelDist3D(const VecFloat3D* const that,
  const VecFloat3D* const 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(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
static inline
#endif
bool _VecFloatIsEqual(const VecFloat* const that,
  const VecFloat* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
```

```
#endif
  // For each component
  for (long 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;
}

// Calculate (that * a + tho * b) and store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatOp(VecFloat* const that, const float a,
  const VecFloat* const tho, const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  for (long iDim = that->_dim; iDim--;)
    VecSet(that, iDim,
      a * VecGet(that, iDim) + b * VecGet(tho, iDim));
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatOp2D(VecFloat2D* const that, const float a,
  const VecFloat2D* const tho, const 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
static inline
#endif
```

```
void _VecFloatOp3D(VecFloat3D* const that, const float a,
  const VecFloat3D* const tho, const 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));
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatOp4D(VecFloat4D* const that, const float a,
  const VecFloat4D* const tho, const 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));
  VecSet(that, 3, a * VecGet(that, 3) + b * VecGet(tho, 3));
}

// Return a VecFloat equal to (that * a + tho * b)
#if BUILDMODE != 0
static inline
#endif
VecFloat* _VecFloatGetOp(const VecFloat* const that, const float a,
  const VecFloat* const tho, const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
```

```
    PBErrCatch(PBMathErr);
  }
#endif
  VecFloat* res = VecFloatCreate(that->_dim);
  for (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim,
      a * VecGet(that, iDim) + b * VecGet(tho, iDim));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecFloat2D _VecFloatGetOp2D(const VecFloat2D* const that, const float a,
  const VecFloat2D* const tho, const 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
static inline
#endif
VecFloat3D _VecFloatGetOp3D(const VecFloat3D* const that, const float a,
  const VecFloat3D* const tho, const 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 the Hadamard product of that by tho and store the
// result in 'that'
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
void _VecFloatHadamardProd(VecFloat* const that,
```

```
  const VecFloat* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  for (long iDim = that->_dim; iDim--;)
    VecSet(that, iDim, VecGet(that, iDim) * VecGet(tho, iDim));
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatHadamardProd2D(VecFloat2D* const that,
  const VecFloat2D* const 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
  VecSet(that, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(that, 1, VecGet(that, 1) * VecGet(tho, 1));
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatHadamardProd3D(VecFloat3D* const that,
  const VecFloat3D* const 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
  VecSet(that, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(that, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(that, 2, VecGet(that, 2) * VecGet(tho, 2));
```

```
}

// Return a VecFloat equal to the hadamard product of 'that' and 'tho'
// Return NULL if arguments are invalid
// 'tho' and 'that' must be of same dimension
#if BUILDMODE != 0
static inline
#endif
VecFloat* _VecFloatGetHadamardProd(const VecFloat* const that,
  const VecFloat* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  VecFloat* res = VecFloatCreate(that->_dim);
  for (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim, VecGet(that, iDim) * VecGet(tho, iDim));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecFloat2D _VecFloatGetHadamardProd2D(const VecFloat2D* const that,
  const VecFloat2D* const 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
  VecFloat2D res = VecFloatCreateStatic2D();
  VecSet(&res, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(&res, 1, VecGet(that, 1) * VecGet(tho, 1));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecFloat3D _VecFloatGetHadamardProd3D(const VecFloat3D* const that,
  const VecFloat3D* const 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
  VecFloat3D res = VecFloatCreateStatic3D();
  VecSet(&res, 0, VecGet(that, 0) * VecGet(tho, 0));
  VecSet(&res, 1, VecGet(that, 1) * VecGet(tho, 1));
  VecSet(&res, 2, VecGet(that, 2) * VecGet(tho, 2));
  return res;
}

// Calculate (that * a) and store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatScale(VecFloat* const that, const float a) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  for (long iDim = that->_dim; iDim--;)
    VecSet(that, iDim, a * VecGet(that, iDim));
}
#if BUILDMODE != 0
static inline
#endif
void _VecFloatScale2D(VecFloat2D* const that, const 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
static inline
#endif
void _VecFloatScale3D(VecFloat3D* const that, const 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));
}
#if BUILDMODE != 0
```

```
static inline
#endif
void _VecFloatScale4D(VecFloat4D* const that, const 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));
  VecSet(that, 3, a * VecGet(that, 3));
}

// Return a VecFloat equal to (that * a)
#if BUILDMODE != 0
static inline
#endif
VecFloat* _VecFloatGetScale(const VecFloat* const that, const 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 (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim, a * VecGet(that, iDim));
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecFloat2D _VecFloatGetScale2D(const VecFloat2D* const that,
  const 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
static inline
#endif
VecFloat3D _VecFloatGetScale3D(const VecFloat3D* const that,
  const 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
static inline
#endif
void _VecFloatRot2D(VecFloat2D* const that, const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 2) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld==2)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
#endif
  *that = _VecFloatGetRot2D(that, theta);
}

// Return a VecFloat2D equal to 'that' rotated CCW by 'theta' radians
#if BUILDMODE != 0
static inline
#endif
VecFloat2D _VecFloatGetRot2D(const VecFloat2D* const that, const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 2) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld==2)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable to memorize the result
  VecFloat2D res = VecFloatCreateStatic2D();
  // Declare variable for optimization
  float cosTheta = cos(theta);
  float sinTheta = sin(theta);
  // Calculate the rotation
  VecSet(&res, 0,
    cosTheta * VecGet(that, 0) - sinTheta * VecGet(that, 1));
  VecSet(&res, 1,
    sinTheta * VecGet(that, 0) + cosTheta * VecGet(that, 1));
  // Return the result
  return res;
}

// Return the dot product of 'that' and 'tho'
#if BUILDMODE != 0
```

```
static inline
#endif
float _VecFloatDotProd(const VecFloat* const that,
  const VecFloat* const 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 (%ld==%ld)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable to memorize the result
  float res = 0.0;
  // Calculate
  for (long iDim = that->_dim; iDim--;)
    res += that->_val[iDim] * tho->_val[iDim];
  // Return the result
  return res;
}
#if BUILDMODE != 0
static inline
#endif
float _VecFloatDotProd2D(const VecFloat2D* const that,
  const VecFloat2D* const 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
static inline
#endif
float _VecFloatDotProd3D(const VecFloat3D* const that,
  const VecFloat3D* const 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 cross product of 'that' and 'tho'
#if BUILDMODE != 0
static inline
#endif
VecFloat* _VecFloatGetCrossProd(const VecFloat* const that,
  const VecFloat* const 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 || tho->_dim != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "invalid dimensions (%ld==%ld==3)",
      that->_dim, tho->_dim);
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable to memorize the result
  VecFloat* res = VecFloatCreate(3);
  // Calculate
  VecSet(res, 0,
    VecGet(that, 1) * VecGet(tho, 2) -
    VecGet(that, 2) * VecGet(tho, 1));
  VecSet(res, 1, -1.0 *
    (VecGet(that, 0) * VecGet(tho, 2) -
    VecGet(that, 2) * VecGet(tho, 0)));
  VecSet(res, 2,
    VecGet(that, 0) * VecGet(tho, 1) -
    VecGet(that, 1) * VecGet(tho, 0));
  // Return the result
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecFloat3D _VecFloatGetCrossProd3D(const VecFloat3D* const that,
  const VecFloat3D* const 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
  // Declare a variable to memorize the result
  VecFloat3D res = VecFloatCreateStatic3D();
  // Calculate
  VecSet(&res, 0,
    VecGet(that, 1) * VecGet(tho, 2) -
    VecGet(that, 2) * VecGet(tho, 1));
  VecSet(&res, 1, -1.0 *
    (VecGet(that, 0) * VecGet(tho, 2) -
    VecGet(that, 2) * VecGet(tho, 0)));
  VecSet(&res, 2,
    VecGet(that, 0) * VecGet(tho, 1) -
    VecGet(that, 1) * VecGet(tho, 0));
  // Return the result
  return res;
}

// Return the conversion of VecFloat 'that' to a VecShort using round()
#if BUILDMODE != 0
static inline
#endif
VecShort* VecFloatToShort(const VecFloat* const 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 (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim, SHORT(VecGet(that, iDim)));
  // Return the result
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecShort2D VecFloatToShort2D(const VecFloat2D* const 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
static inline
#endif
VecShort3D VecFloatToShort3D(const VecFloat3D* const 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
static inline
#endif
VecFloat* VecShortToFloat(const VecShort* const 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 (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim, (float)VecGet(that, iDim));
  // Return the result
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecFloat2D VecShortToFloat2D(const VecShort2D* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(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
static inline
#endif
VecFloat3D VecShortToFloat3D(const VecShort3D* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // 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;
}

// Return the conversion of VecLong 'that' to a VecFloat
#if BUILDMODE != 0
static inline
#endif
VecFloat* VecLongToFloat(const VecLong* const 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 (long iDim = that->_dim; iDim--;)
    VecSet(res, iDim, (float)VecGet(that, iDim));
  // Return the result
  return res;
}
#if BUILDMODE != 0
static inline
#endif
VecFloat2D VecLongToFloat2D(const VecLong2D* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(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
static inline
#endif
VecFloat3D VecLongToFloat3D(const VecLong3D* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // 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;
}

// Get the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGetMaxVal(const VecFloat* const 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 result
  float max = VecGet(that, 0);
  // Search for the maximum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    max = MAX(max, VecGet(that, i));
  // Return the result
  return max;
}

// Get the index of the max value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
long _VecFloatGetIMaxVal(const VecFloat* const 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 result
  long iMax = 0;
  // Declare a variable to memorize the max value
  float max = VecGet(that, iMax);
  // Search for the maximum value
  for (long i = VecGetDim(that); i-- && i != 0;) {
    if(max < VecGet(that, i)) {
      max = VecGet(that, i);
      iMax = i;
    }
  }
  // Return the result
  return iMax;
}

// Get the min value in components of the vector 'that'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGetMinVal(const VecFloat* const 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 result
  float min = VecGet(that, 0);
  // Search for the minimum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    min = MIN(min, VecGet(that, i));
  // Return the result
  return min;
}

// Get the max value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGetMaxValAbs(const VecFloat* const 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 result
  float max = fabs(VecGet(that, 0));
  // Search for the maximum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    max = (fabs(max) > fabs(VecGet(that, i)) ? max : VecGet(that, i));
  // Return the result
  return max;
}

// Get the min value (in absolute value) in components of the
// vector 'that'
#if BUILDMODE != 0
static inline
#endif
float _VecFloatGetMinValAbs(const VecFloat* const 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 result
  float min = fabs(VecGet(that, 0));
  // Search for the minimum value
  for (long i = VecGetDim(that); i-- && i != 0;)
    min = (fabs(min) < fabs(VecGet(that, i)) ? min : VecGet(that, i));
  // Return the result
  return min;
}

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

// Return the addition of matrix 'that' with matrix 'tho'
// 'that' and 'tho' must have same dimensions
#if BUILDMODE != 0
static inline
#endif
MatFloat* _MatFloatGetAdd(MatFloat* const 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(MatDim(that), MatDim(tho)) == false) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg,
      "'that' and 'tho' have different dimensions");
    PBErrCatch(PBMathErr);
  }
#endif
  // Declare a variable for the result
  MatFloat* res = MatFloatCreate(MatDim(that));
  // Add each values
  VecShort2D i = VecShortCreateStatic2D();
  do {
    MatSet(res, &i, MatGet(that, &i) + MatGet(tho, &i));
  } while (VecStep(&i, MatDim(that)));
  // Return the result
  return res;
}

// Add matrix 'that' with matrix 'tho' and store the result in 'that'
// 'that' and 'tho' must have same dimensions
#if BUILDMODE != 0
static inline
#endif
void _MatFloatAdd(MatFloat* const 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(MatDim(that), MatDim(tho)) == false) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg,
      "'that' and 'tho' have different dimensions");
    PBErrCatch(PBMathErr);
  }
#endif
  // Add each values
  VecShort2D i = VecShortCreateStatic2D();
  do {
    MatSet(that, &i, MatGet(that, &i) + MatGet(tho, &i));
  } while (VecStep(&i, MatDim(that)));
}

// Multiply the matrix 'that' by 'a'
#if BUILDMODE != 0
static inline
#endif
void _MatFloatScale(MatFloat* const that, const float a) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Multiply each values
  VecShort2D i = VecShortCreateStatic2D();
  do {
    MatSet(that, &i, MatGet(that, &i) * a);
  } while (VecStep(&i, MatDim(that)));
}

// Copy the values of 'w' in 'that' (must have same dimensions)
#if BUILDMODE != 0
static inline
#endif
void _MatFloatCopy(MatFloat* const that, const MatFloat* const 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
static inline
#endif
float _MatFloatGet(const MatFloat* const 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
static inline
#endif
void _MatFloatSet(MatFloat* const 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 the dimension of the MatFloat
#if BUILDMODE != 0
static inline
#endif
const VecShort2D* _MatFloatDim(const MatFloat* const 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 a VecShort2D containing the dimension of the MatFloat
#if BUILDMODE != 0
static inline
#endif
VecShort2D _MatFloatGetDim(const MatFloat* const 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 number of rows of the MatFloat 'that'
#if BUILDMODE != 0
static inline
#endif
short _MatFloatGetNbRow(const MatFloat* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
```

```
#endif
  // Return the nb of rows
  return VecGet(&(that->_dim), 1);
}

// Return the number of columns of the MatFloat 'that'
#if BUILDMODE != 0
static inline
#endif
short _MatFloatGetNbCol(const MatFloat* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Return the nb of cols
  return VecGet(&(that->_dim), 0);

}

// Return the value of the Gauss 'that' at 'x'
#if BUILDMODE != 0
static inline
#endif
float GaussGet(const Gauss* const that, const 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
static inline
#endif
float GaussRnd(Gauss* const 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
static inline
#endif
float SmoothStep(const 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
static inline
#endif
float SmootherStep(const 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
static inline
#endif
VecFloat* SysLinEqSolve(const SysLinEq* const 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 = MatGetProdVec(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
static inline
#endif
void SysLinEqSetM(SysLinEq* const that, const MatFloat* const 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 = MatGetInv(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
static inline
#endif
void _SLESetV(SysLinEq* const that, const VecFloat* const 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 (VecGetDim(v) != VecGetDim(that->_V)) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'v' has invalid dimension (%ld==%ld)",
```

```
      VecGetDim(v), VecGetDim(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
static inline
#endif
int powi(const int base, const int exp) {
  // Declare a variable to memorize the result and init to 1
  int res = 1;
  // Loop on exponent
  int e = exp;
  int b = base;
  while (e) {
    // Do some magic trick
    if (e & 1)
      res *= b;
    e /= 2;
    b *= b;
  }
  // Return the result
  return res;
}

// Rotate right-hand 'that' by 'theta' radians around 'axis' and
// store the result in 'that'
// 'axis' must be normalized
// https://en.wikipedia.org/wiki/Rotation_matrix
#if BUILDMODE != 0
static inline
#endif
void _VecFloatRotAxis(VecFloat3D* const that,
  const VecFloat3D* const axis, const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (axis == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'axis' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld==3)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(axis) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'axis' 's dimension is invalid (%ld==3)",
      VecGetDim(axis));
```

```
    PBErrCatch(PBMathErr);
  }
  if (ISEQUALF(VecNorm(axis), 1.0) == false) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'axis' is not normalized");
    PBErrCatch(PBMathErr);
  }
#endif
  VecFloat3D v = _VecFloatGetRotAxis(that, axis, theta);
  VecCopy(that, &v);
}

// Rotate right-hand 'that' by 'theta' radians around X and
// store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatRotX(VecFloat3D* const that, const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld==3)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
#endif
  VecFloat3D v = _VecFloatGetRotX(that, theta);
  VecCopy(that, &v);
}

// Rotate right-hand 'that' by 'theta' radians around Y and
// store the result in 'that'
#if BUILDMODE != 0
static inline
#endif
void _VecFloatRotY(VecFloat3D* const that, const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld==3)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
#endif
  VecFloat3D v = _VecFloatGetRotY(that, theta);
  VecCopy(that, &v);
}

// Rotate right-hand 'that' by 'theta' radians around Z and
// store the result in 'that'
#if BUILDMODE != 0
static inline
```

```
#endif
void _VecFloatRotZ(VecFloat3D* const that, const float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (VecGetDim(that) != 3) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'that' 's dimension is invalid (%ld==3)",
      VecGetDim(that));
    PBErrCatch(PBMathErr);
  }
#endif
  VecFloat3D v = _VecFloatGetRotZ(that, theta);
  VecCopy(that, &v);
}

// Free memory used by the QRDecomp 'that'
#if BUILDMODE != 0
static inline
#endif
void QRDecompFreeStatic(QRDecomp* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  MatFree(&(that->_Q));
  MatFree(&(that->_R));
}

// Get the base of the Ratio 'that'
#if BUILDMODE != 0
static inline
#endif
long RatioGetBase(const Ratio* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  return that->_base;
}

// Get the numerator of the Ratio 'that'
#if BUILDMODE != 0
static inline
#endif
unsigned int RatioGetNumerator(const Ratio* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
```

```
  return that->_numerator;
}

// Get the denominator of the Ratio 'that'
#if BUILDMODE != 0
static inline
#endif
unsigned int  RatioGetDenominator(const Ratio* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  return that->_denominator;
}

// Set the base of the Ratio 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void RatioSetBase(Ratio* that, long v) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  that->_base = v;
}

// Set the numerator of the Ratio 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void RatioSetNumerator(Ratio* that, unsigned int v) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  that->_numerator = v;
}

// Set the denominator of the Ratio 'that' to 'v'
#if BUILDMODE != 0
static inline
#endif
void RatioSetDenominator(Ratio* that, unsigned int v) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (v == 0) {
    PBMathErr->_type = PBErrTypeInvalidArg;
    sprintf(PBMathErr->_msg, "'v' is invalid (%u > 0)", v);
```

```
    PBErrCatch(PBMathErr);
  }
#endif
  that->_denominator = v;
}

// -------------- LeastSquareLinReg

// Set the component of the LeastSquareLinReg 'that' to 'X'
#if BUILDMODE != 0
static inline
#endif
void LSLRSetComp(LeastSquareLinReg* that, const MatFloat* X) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (X == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'X' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  that->X = X;
  if (that->Xp != NULL) {
    MatFree(&(that->Xp));
  }
  MatFloat* transp = MatGetTranspose(that->X);
  MatFloat* A = MatGetProdMat(transp, that->X);
  MatFloat* inv = MatGetInv(A);
  if (inv != NULL) {
    that->Xp = MatGetProdMat(inv, transp);
  }
  MatFree(&transp);
  MatFree(&A);
  MatFree(&inv);
}

// Get the bias of the last computed solution of the LeastSquareLinReg 'that'
#if BUILDMODE != 0
static inline
#endif
float LSLRGetBias(const LeastSquareLinReg* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  return that->bias;
}

// Return true if the LeastSquareLinReg 'that' is solvable
#if BUILDMODE != 0
static inline
#endif
bool LSLRIsSolvable(const LeastSquareLinReg* that) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  return (that->Xp != NULL);
}
```

4 Makefile

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

all: pbmake_wget main

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

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

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

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

5 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "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();
  VecPrintln(v, stdout);
  VecPrintln(&v2, stdout);
  VecPrintln(&v3, stdout);
  VecPrintln(&v4, stdout);
  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 (long i = 5; i--;) VecSet(v, i, i + 1);
  for (long i = 2; i--;) VecSet(&v2, i, i + 1);
  for (long i = 3; i--;) VecSet(&v3, i, i + 1);
  for (long 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);
  }
  bool compact = false;
  if (!VecSave(v, f, compact)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortSave NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecSave(&v2, f, compact)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortSave NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecSave(&v3, f, compact)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortSave NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecSave(&v4, f, compact)) {
    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 = ret;
}

void UnitTestVecShortGetSetDim() {
  VecShort* v = VecShortCreate(5);
  VecShort2D v2 = VecShortCreateStatic2D();
  VecShort3D v3 = VecShortCreateStatic3D();
  VecShort4D v4 = VecShortCreateStatic4D();
  if (VecGetDim(v) != 5) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortGetDim 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);
  }
for (int i = 5; i--;) VecSetAdd(v, i, i + 1);
for (int i = 2; i--;) VecSetAdd(&v2, i, i + 1);
for (int i = 3; i--;) VecSetAdd(&v3, i, i + 1);
for (int i = 4; i--;) VecSetAdd(&v4, i, i + 1);
for (int i = 5; i--;)
  if (VecGet(v, i) != 2 * (i + 1)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortSetAdd NOK1");
    PBErrCatch(PBMathErr);
  }
for (int i = 2; i--;)
  if (VecGet(&v2, i) != 2 * (i + 1)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortSetAdd NOK2");
    PBErrCatch(PBMathErr);
  }
for (int i = 3; i--;)
  if (VecGet(&v3, i) != 2 * (i + 1)) {
```

```
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortSetAdd NOK3");
    PBErrCatch(PBMathErr);
  }
for (int i = 4; i--;)
  if (VecGet(&v4, i) != 2 * (i + 1)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortSetAdd NOK4");
    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, "_VecShortNull NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 2; i--;)
  if (VecGet(&v2, i) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortNull NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 3; i--;)
  if (VecGet(&v3, i) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortNull NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 4; i--;)
  if (VecGet(&v4, i) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortNull NOK");
    PBErrCatch(PBMathErr);
  }
VecSetAll(v, 1);
VecSetAll(&v2, 1);
VecSetAll(&v3, 1);
VecSetAll(&v4, 1);
for (int i = 5; i--;)
  if (VecGet(v, i) != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortAll NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 2; i--;)
  if (VecGet(&v2, i) != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortAll NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 3; i--;)
  if (VecGet(&v3, i) != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortAll NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 4; i--;)
  if (VecGet(&v4, i) != 1) {
```

```
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortAll NOK");
      PBErrCatch(PBMathErr);
    }
  VecFree(&v);
  printf("_VecShortGetSetDim OK\n");
}

void UnitTestVecShortStep() {
  VecShort* v = VecShortCreate(5);
  VecShort2D v2 = VecShortCreateStatic2D();
  VecShort3D v3 = VecShortCreateStatic3D();
  VecShort4D v4 = VecShortCreateStatic4D();
  VecShort* bv = VecShortCreate(5);
  VecShort2D bv2 = VecShortCreateStatic2D();
  VecShort3D bv3 = VecShortCreateStatic3D();
  VecShort4D bv4 = VecShortCreateStatic4D();
  short b[5] = {2, 3, 4, 5, 6};
  for (int i = 5; i--;) VecSet(bv, i, b[i]);
  for (int i = 2; i--;) VecSet(&bv2, i, b[i]);
  for (int i = 3; i--;) VecSet(&bv3, i, b[i]);
  for (int i = 4; i--;) VecSet(&bv4, i, b[i]);
  int acheck[2 * 3 * 4 * 5 * 6];
  for (int i = 0; i < 2 * 3 * 4 * 5 * 6; ++i)
    acheck[i] = i;
  int iCheck = 0;
  do {
    int a = VecGet(v, 0);
    for (int i = 1; i < VecGetDim(v); ++i)
      a = a * b[i] + VecGet(v, i);
    if (a != acheck[iCheck]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortStep NOK");
      PBErrCatch(PBMathErr);
    }
    ++iCheck;
  } while (VecStep(v, bv));
  iCheck = 0;
  do {
    int a = VecGet(&v2, 0);
    for (int i = 1; i < 2; ++i)
      a = a * b[i] + VecGet(&v2, i);
    if (a != acheck[iCheck]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortStep NOK");
      PBErrCatch(PBMathErr);
    }
    ++iCheck;
  } while (VecStep(&v2, &bv2));
  iCheck = 0;
  do {
    int a = VecGet(&v3, 0);
    for (int i = 1; i < 3; ++i)
      a = a * b[i] + VecGet(&v3, i);
    if (a != acheck[iCheck]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortStep NOK");
      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, VecGetDim(v) - 1);
  for (int i = VecGetDim(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);
VecShort2D w = VecShortCreateStatic2D();
VecShort2D wDelta = VecShortCreateStatic2D();
```

```
  VecShort2D wBound = VecShortCreateStatic2D();
  VecSet(&wDelta, 0, 2);
  VecSet(&wDelta, 1, 3);
  VecSet(&wBound, 0, 4);
  VecSet(&wBound, 1, 6);
  int checkDelta[8] = {0, 0, 0, 3, 2, 0, 2, 3};
  iCheck = 0;
  do {
    if (VecGet(&w, 0) != checkDelta[iCheck * 2] ||
      VecGet(&w, 1) != checkDelta[iCheck * 2 + 1]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortStepDelta NOK");
      PBErrCatch(PBMathErr);
    }
    ++iCheck;
  } while (VecStepDelta(&w, &wBound, &wDelta));
  int checkDeltaB[8] = {0, 0, 2, 0, 0, 3, 2, 3};
  VecSetNull(&w);
  iCheck = 0;
  do {
    if (VecGet(&w, 0) != checkDeltaB[iCheck * 2] ||
      VecGet(&w, 1) != checkDeltaB[iCheck * 2 + 1]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortStepDelta NOK");
      PBErrCatch(PBMathErr);
    }
    ++iCheck;
  } while (VecPStepDelta(&w, &wBound, &wDelta));

  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)(VecGetDim(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)(VecGetDim(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 > 5.0 * 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() * (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);
  VecPrintln(w, stdout);
  VecPrintln(&w2, stdout);
  VecPrintln(&w3, stdout);
  VecFree(&v);
  VecFree(&w);
  printf("UnitTestVecShortToFloat OK\n");
}

void UnitTestVecLongToFloat() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  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 = VecLongToFloat(v);
  VecFloat2D w2 = VecLongToFloat2D(&v2);
  VecFloat3D w3 = VecLongToFloat3D(&v3);
  VecPrintln(w, stdout);
  VecPrintln(&w2, stdout);
  VecPrintln(&w3, stdout);
  VecFree(&v);
  VecFree(&w);
  printf("UnitTestVecLongToFloat OK\n");
}
```

```
void UnitTestVecShortOp() {
  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();
  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);
  short a[2] = {-1, 2};
  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);
  VecShort* u = VecGetOp(v, a[0], w, a[1]);
  VecShort2D u2 = VecGetOp(&v2, a[0], &w2, a[1]);
  VecShort3D u3 = VecGetOp(&v3, a[0], &w3, a[1]);
  VecShort4D u4 = VecGetOp(&v4, a[0], &w4, a[1]);
  short checku[5] = {8,5,2,-1,-4};
  short checku2[2] = {2,-1};
  short checku3[3] = {4,1,-2};
  short checku4[4] = {6,3,0,-3};
  for (int i = 5; i--;)
    if (!ISEQUALF(VecGet(u, i), checku[i])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortGetOp NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 2; i--;)
    if (!ISEQUALF(VecGet(&u2, i), checku2[i])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortGetOp NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 3; i--;)
    if (!ISEQUALF(VecGet(&u3, i), checku3[i])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortGetOp NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 4; i--;)
    if (!ISEQUALF(VecGet(&u4, i), checku4[i])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecShortGetOp NOK");
      PBErrCatch(PBMathErr);
    }
  VecOp(v, a[0], w, a[1]);
  VecOp(&v2, a[0], &w2, a[1]);
  VecOp(&v3, a[0], &w3, a[1]);
  VecOp(&v4, a[0], &w4, a[1]);
  if (!VecIsEqual(v, u)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortOp NOK");
    PBErrCatch(PBMathErr);
```

```
  }
  if (!VecIsEqual(&v2, &u2)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortOp NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecIsEqual(&v3, &u3)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortOp NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecIsEqual(&v4, &u4)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecShortOp NOK");
    PBErrCatch(PBMathErr);
  }
  VecFree(&v);
  VecFree(&w);
  VecFree(&u);
  printf("UnitTestVecShortOp OK\n");
}

void UnitTestVecShortShiftStep() {
  VecShort3D v = VecShortCreateStatic3D();
  VecShort3D from = VecShortCreateStatic3D();
  VecShort3D to = VecShortCreateStatic3D();
  VecSet(&from, 0, 0);
  VecSet(&from, 1, 1);
  VecSet(&from, 2, 2);
  VecSet(&to, 0, 3);
  VecSet(&to, 1, 4);
  VecSet(&to, 2, 5);
  VecCopy(&v, &from);
  short check[81] = {
    0, 1, 2, 0, 1, 3, 0, 1, 4,
    0, 2, 2, 0, 2, 3, 0, 2, 4,
    0, 3, 2, 0, 3, 3, 0, 3, 4,
    1, 1, 2, 1, 1, 3, 1, 1, 4,
    1, 2, 2, 1, 2, 3, 1, 2, 4,
    1, 3, 2, 1, 3, 3, 1, 3, 4,
    2, 1, 2, 2, 1, 3, 2, 1, 4,
    2, 2, 2, 2, 2, 3, 2, 2, 4,
    2, 3, 2, 2, 3, 3, 2, 3, 4
    };
  int iCheck = 0;
  do {
    for (int i = 0; i < 3; ++i) {
      if (ISEQUALF(check[iCheck], VecGet(&v, i)) == false) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShiftStep NOK");
        PBErrCatch(PBMathErr);
      }
      ++iCheck;
    }
  } while(VecShiftStep(&v, &from, &to));
  printf("UnitTestVecShortShiftStep OK\n");
}

void UnitTestVecShortGetMinMax() {
  VecShort3D v = VecShortCreateStatic3D();
  VecSet(&v, 0, 2); VecSet(&v, 1, 4); VecSet(&v, 2, 3);
  short val = VecGetMaxVal(&v);
```

```
  if (val != 4) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMaxVal NOK");
    PBErrCatch(PBMathErr);
  }
  if (VecGetIMaxVal(&v) != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetIMaxVal NOK");
    PBErrCatch(PBMathErr);
  }
  VecSet(&v, 0, 2); VecSet(&v, 1, 1); VecSet(&v, 2, 3);
  val = VecGetMinVal(&v);
  if (val != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMinVal NOK");
    PBErrCatch(PBMathErr);
  }
  VecSet(&v, 0, 2); VecSet(&v, 1, -4); VecSet(&v, 2, 3);
  val = VecGetMaxValAbs(&v);
  if (val != -4) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMaxValAbs NOK");
    PBErrCatch(PBMathErr);
  }
  VecSet(&v, 0, -2); VecSet(&v, 1, 1); VecSet(&v, 2, 3);
  val = VecGetMinValAbs(&v);
  if (val != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMinValAbs NOK");
    PBErrCatch(PBMathErr);
  }
  printf("UnitTestVecShortGetMinMax OK\n");
}

void UnitTestVecShortHadamardProd() {
  VecShort* u = VecShortCreate(3);
  for (int i = 3; i--;)
    VecSet(u, i, i + 2);
  VecShort* uprod = VecGetHadamardProd(u, u);
  VecHadamardProd(u, u);
  short checku[3] = {4, 9, 16};
  for (int i = 3; i--;)
    if (ISEQUALF(VecGet(uprod, i), checku[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(uprod, u) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  VecFree(&uprod);
  VecFree(&u);
  VecShort2D v = VecShortCreateStatic2D();
  for (int i = 2; i--;)
    VecSet(&v, i, i + 2);
  VecShort2D vprod = VecGetHadamardProd(&v, &v);
  VecHadamardProd(&v, &v);
  short checkv[2] = {4, 9};
  for (int i = 2; i--;)
    if (ISEQUALF(VecGet(&vprod, i), checkv[i]) == false) {
```

```
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(&vprod, &v) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  VecShort3D w = VecShortCreateStatic3D();
  for (int i = 3; i--;)
    VecSet(&w, i, i + 2);
  VecShort3D wprod = VecGetHadamardProd(&w, &w);
  VecHadamardProd(&w, &w);
  short checkw[3] = {4, 9, 16};
  for (int i = 3; i--;)
    if (ISEQUALF(VecGet(&wprod, i), checkw[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(&wprod, &w) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  VecShort4D x = VecShortCreateStatic4D();
  for (int i = 4; i--;)
    VecSet(&x, i, i + 2);
  VecShort4D xprod = VecGetHadamardProd(&x, &x);
  VecHadamardProd(&x, &x);
  short checkx[4] = {4, 9, 16, 25};
  for (int i = 4; i--;)
    if (ISEQUALF(VecGet(&xprod, i), checkx[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(&xprod, &x) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  printf("UnitTestVecShortHadamardProd OK\n");
}

void UnitTestVecShort() {
  UnitTestVecShortCreateFree();
  UnitTestVecShortClone();
  UnitTestVecShortLoadSave();
  UnitTestVecShortGetSetDim();
  UnitTestVecShortStep();
  UnitTestVecShortHamiltonDist();
  UnitTestVecShortIsEqual();
  UnitTestVecShortDotProd();
  UnitTestVecShortCopy();
  UnitTestSpeedVecShort();
  UnitTestVecShortToFloat();
  UnitTestVecLongToFloat();
  UnitTestVecShortOp();
  UnitTestVecShortShiftStep();
  UnitTestVecShortGetMinMax();
```

```
  UnitTestVecShortHadamardProd();
  printf("UnitTestVecShort OK\n");
}

void UnitTestVecLongCreateFree() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  VecPrintln(v, stdout);
  VecPrintln(&v2, stdout);
  VecPrintln(&v3, stdout);
  VecPrintln(&v4, stdout);
  VecFree(&v);
  if (v != NULL) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecLong is not null after VecFree");
    PBErrCatch(PBMathErr);
  }
  printf("VecLongCreateFree OK\n");
}

void UnitTestVecLongClone() {
  VecLong* v = VecLongCreate(5);
  for (int i = 5; i--;) VecSet(v, i, i + 1);
  VecLong* w = VecClone(v);
  if (memcmp(v, w, sizeof(VecLong) + sizeof(long) * 5) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongClone NOK");
    PBErrCatch(PBMathErr);
  }
  VecFree(&v);
  VecFree(&w);
  printf("_VecLongClone OK\n");
}

void UnitTestVecLongLoadSave() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  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("./UnitTestVecLongLoadSave.txt", "w");
  if (f == NULL) {
    PBMathErr->_type = PBErrTypeOther;
    sprintf(PBMathErr->_msg,
      "Can't open ./UnitTestVecLongLoadSave.txt for writing");
    PBErrCatch(PBMathErr);
  }
  bool compact = false;
  if (!VecSave(v, f, compact)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongSave NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecSave(&v2, f, compact)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongSave NOK");
    PBErrCatch(PBMathErr);
```

```
}
if (!VecSave(&v3, f, compact)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongSave NOK");
  PBErrCatch(PBMathErr);
}
if (!VecSave(&v4, f, compact)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongSave NOK");
  PBErrCatch(PBMathErr);
}
fclose(f);
VecLong* w = VecLongCreate(2);
f = fopen("./UnitTestVecLongLoadSave.txt", "r");
if (f == NULL) {
  PBMathErr->_type = PBErrTypeOther;
  sprintf(PBMathErr->_msg,
    "Can't open ./UnitTestVecLongLoadSave.txt for reading");
  PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongLoad NOK");
  PBErrCatch(PBMathErr);
}
if (memcmp(v, w, sizeof(VecLong) + sizeof(long) * 5) != 0) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongLoadSave NOK");
  PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongLoad NOK");
  PBErrCatch(PBMathErr);
}
if (memcmp(&v2, w, sizeof(VecLong) + sizeof(long) * 2) != 0) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongLoadSave NOK");
  PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongLoad NOK");
  PBErrCatch(PBMathErr);
}
if (memcmp(&v3, w, sizeof(VecLong) + sizeof(long) * 3) != 0) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongLoadSave NOK");
  PBErrCatch(PBMathErr);
}
if (!VecLoad(&w, f)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongLoad NOK");
  PBErrCatch(PBMathErr);
}
if (memcmp(&v4, w, sizeof(VecLong) + sizeof(long) * 4) != 0) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongLoadSave NOK");
  PBErrCatch(PBMathErr);
}
fclose(f);
VecFree(&v);
```

```
  VecFree(&w);
  int ret = system("cat ./UnitTestVecLongLoadSave.txt");
  printf("_VecLongLoadSave OK\n");
  ret = ret;
}

void UnitTestVecLongGetSetDim() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  if (VecGetDim(v) != 5) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongGetDim 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, "_VecLongSet NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 2; i--;)
    if (v2._val[i] != i + 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongSet NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 3; i--;)
    if (v3._val[i] != i + 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongSet NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 4; i--;)
    if (v4._val[i] != i + 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongSet NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 5; i--;)
    if (VecGet(v, i) != i + 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongGet NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 2; i--;)
    if (VecGet(&v2, i) != i + 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongGet NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 3; i--;)
    if (VecGet(&v3, i) != i + 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongGet NOK");
      PBErrCatch(PBMathErr);
    }
```

```
for (int i = 4; i--;)
  if (VecGet(&v4, i) != i + 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongGet NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 5; i--;) VecSetAdd(v, i, i + 1);
for (int i = 2; i--;) VecSetAdd(&v2, i, i + 1);
for (int i = 3; i--;) VecSetAdd(&v3, i, i + 1);
for (int i = 4; i--;) VecSetAdd(&v4, i, i + 1);
for (int i = 5; i--;)
  if (VecGet(v, i) != 2 * (i + 1)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongSetAdd NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 2; i--;)
  if (VecGet(&v2, i) != 2 * (i + 1)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongSetAdd NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 3; i--;)
  if (VecGet(&v3, i) != 2 * (i + 1)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongSetAdd NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 4; i--;)
  if (VecGet(&v4, i) != 2 * (i + 1)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongSetAdd 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, "_VecLongGet NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 2; i--;)
  if (VecGet(&v2, i) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongGet NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 3; i--;)
  if (VecGet(&v3, i) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongGet NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 4; i--;)
  if (VecGet(&v4, i) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongGet NOK");
    PBErrCatch(PBMathErr);
  }
```

```
  VecSetAll(v, 1);
  VecSetAll(&v2, 1);
  VecSetAll(&v3, 1);
  VecSetAll(&v4, 1);
  for (int i = 5; i--;)
    if (VecGet(v, i) != 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongAll NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 2; i--;)
    if (VecGet(&v2, i) != 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongAll NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 3; i--;)
    if (VecGet(&v3, i) != 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongAll NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 4; i--;)
    if (VecGet(&v4, i) != 1) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongAll NOK");
      PBErrCatch(PBMathErr);
    }
  VecFree(&v);
  printf("_VecLongGetSetDim OK\n");
}

void UnitTestVecLongStep() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  VecLong* bv = VecLongCreate(5);
  VecLong2D bv2 = VecLongCreateStatic2D();
  VecLong3D bv3 = VecLongCreateStatic3D();
  VecLong4D bv4 = VecLongCreateStatic4D();
  long b[5] = {2, 3, 4, 5, 6};
  for (int i = 5; i--;) VecSet(bv, i, b[i]);
  for (int i = 2; i--;) VecSet(&bv2, i, b[i]);
  for (int i = 3; i--;) VecSet(&bv3, i, b[i]);
  for (int i = 4; i--;) VecSet(&bv4, i, b[i]);
  int acheck[2 * 3 * 4 * 5 * 6];
  for (int i = 0; i < 2 * 3 * 4 * 5 * 6; ++i)
    acheck[i] = i;
  int iCheck = 0;
  do {
    int a = VecGet(v, 0);
    for (int i = 1; i < VecGetDim(v); ++i)
      a = a * b[i] + VecGet(v, i);
    if (a != acheck[iCheck]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongStep NOK");
      PBErrCatch(PBMathErr);
    }
    ++iCheck;
  } while (VecStep(v, bv));
  iCheck = 0;
```

```
do {
  int a = VecGet(&v2, 0);
  for (int i = 1; i < 2; ++i)
    a = a * b[i] + VecGet(&v2, i);
  if (a != acheck[iCheck]) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongStep NOK");
    PBErrCatch(PBMathErr);
  }
  ++iCheck;
} while (VecStep(&v2, &bv2));
iCheck = 0;
do {
  int a = VecGet(&v3, 0);
  for (int i = 1; i < 3; ++i)
    a = a * b[i] + VecGet(&v3, i);
  if (a != acheck[iCheck]) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongStep NOK");
    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, "_VecLongStep NOK");
    PBErrCatch(PBMathErr);
  }
  ++iCheck;
} while (VecStep(&v4, &bv4));
iCheck = 0;
do {
  int a = VecGet(v, VecGetDim(v) - 1);
  for (int i = VecGetDim(v) - 2; i >= 0; --i)
    a = a * b[i] + VecGet(v, i);
  if (a != acheck[iCheck]) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongPStep 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, "_VecLongPStep 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, "_VecLongPStep 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, "_VecLongPStep NOK");
      PBErrCatch(PBMathErr);
    }
    ++iCheck;
  } while (VecPStep(&v4, &bv4));
  VecFree(&v);
  VecFree(&bv);
  VecLong2D w = VecLongCreateStatic2D();
  VecLong2D wDelta = VecLongCreateStatic2D();
  VecLong2D wBound = VecLongCreateStatic2D();
  VecSet(&wDelta, 0, 2);
  VecSet(&wDelta, 1, 3);
  VecSet(&wBound, 0, 4);
  VecSet(&wBound, 1, 6);
  int checkDelta[8] = {0, 0, 0, 3, 2, 0, 2, 3};
  iCheck = 0;
  do {
    if (VecGet(&w, 0) != checkDelta[iCheck * 2] ||
      VecGet(&w, 1) != checkDelta[iCheck * 2 + 1]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongStepDelta NOK");
      PBErrCatch(PBMathErr);
    }
    ++iCheck;
  } while (VecStepDelta(&w, &wBound, &wDelta));
  int checkDeltaB[8] = {0, 0, 2, 0, 0, 3, 2, 3};
  VecSetNull(&w);
  iCheck = 0;
  do {
    if (VecGet(&w, 0) != checkDeltaB[iCheck * 2] ||
      VecGet(&w, 1) != checkDeltaB[iCheck * 2 + 1]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecLongStepDelta NOK");
      PBErrCatch(PBMathErr);
    }
    ++iCheck;
  } while (VecPStepDelta(&w, &wBound, &wDelta));

  printf("UnitTestVecLongStep OK\n");
}

void UnitTestVecLongHamiltonDist() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  VecLong* w = VecLongCreate(5);
```

```
  VecLong2D w2 = VecLongCreateStatic2D();
  VecLong3D w3 = VecLongCreateStatic3D();
  VecLong4D w4 = VecLongCreateStatic4D();
  long 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);
  long dist = VecHamiltonDist(v, w);
  if (dist != 13) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongHamiltonDist NOK");
    PBErrCatch(PBMathErr);
  }
  dist = VecHamiltonDist(&v2, &w2);
  if (dist != 2) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongHamiltonDist NOK");
    PBErrCatch(PBMathErr);
  }
  dist = VecHamiltonDist(&v3, &w3);
  if (dist != 5) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongHamiltonDist NOK");
    PBErrCatch(PBMathErr);
  }
  dist = VecHamiltonDist(&v4, &w4);
  if (dist != 8) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongHamiltonDist NOK");
    PBErrCatch(PBMathErr);
  }
  VecFree(&v);
  VecFree(&w);
  printf("UnitTestVecLongHamiltonDist OK\n");
}

void UnitTestVecLongIsEqual() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  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);
  VecLong* w = VecLongCreate(5);
  VecLong2D w2 = VecLongCreateStatic2D();
  VecLong3D w3 = VecLongCreateStatic3D();
  VecLong4D w4 = VecLongCreateStatic4D();
  if (VecIsEqual(v, w)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongIsEqual NOK");
    PBErrCatch(PBMathErr);
  }
  if (VecIsEqual(&v2, &w2)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongIsEqual NOK");
```

```
    PBErrCatch(PBMathErr);
  }
  if (VecIsEqual(&v3, &w3)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongIsEqual NOK");
    PBErrCatch(PBMathErr);
  }
  if (VecIsEqual(&v4, &w4)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongIsEqual 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, "_VecLongIsEqual NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecIsEqual(&v2, &w2)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongIsEqual NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecIsEqual(&v3, &w3)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongIsEqual NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecIsEqual(&v4, &w4)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongIsEqual NOK");
    PBErrCatch(PBMathErr);
  }
  VecFree(&v);
  VecFree(&w);
  printf("UnitTestVecLongIsEqual OK\n");
}

void UnitTestVecLongCopy() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  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);
  VecLong* w = VecLongCreate(5);
  VecLong2D w2 = VecLongCreateStatic2D();
  VecLong3D w3 = VecLongCreateStatic3D();
  VecLong4D w4 = VecLongCreateStatic4D();
  VecCopy(w, v);
  VecCopy(&w2, &v2);
  VecCopy(&w3, &v3);
  VecCopy(&w4, &v4);
  if (!VecIsEqual(v, w)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongCopy NOK");
    PBErrCatch(PBMathErr);
```

```
  }
  if (!VecIsEqual(&v2, &w2)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongCopy NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecIsEqual(&v3, &w3)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongCopy NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecIsEqual(&v4, &w4)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongCopy NOK");
    PBErrCatch(PBMathErr);
  }
  VecFree(&v);
  VecFree(&w);
  printf("UnitTestVecLongCopy OK\n");
}

void UnitTestVecLongDotProd() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  VecLong* w = VecLongCreate(5);
  VecLong2D w2 = VecLongCreateStatic2D();
  VecLong3D w3 = VecLongCreateStatic3D();
  VecLong4D w4 = VecLongCreateStatic4D();
  long 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);
  long prod = VecDotProd(v, w);
  if (prod != -10) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongDotProd NOK");
    PBErrCatch(PBMathErr);
  }
  prod = VecDotProd(&v2, &w2);
  if (prod != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongDotProd NOK");
    PBErrCatch(PBMathErr);
  }
  prod = VecDotProd(&v3, &w3);
  if (prod != -2) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongDotProd NOK");
    PBErrCatch(PBMathErr);
  }
  prod = VecDotProd(&v4, &w4);
  if (prod != -6) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongDotProd NOK");
    PBErrCatch(PBMathErr);
```

```
  }
  VecFree(&v);
  VecFree(&w);
  printf("UnitTestVecLongDotProd OK\n");
}

void UnitTestSpeedVecLong() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  int nbTest = 100000;

  srandom(RANDOMSEED);
  int i = nbTest;
  clock_t clockBefore = clock();
  for (; i--;) {
    int j = INT(rnd() * ((float)(VecGetDim(v) - 1) - PBMATH_EPSILON));
    long val = 1;
    VecSet(v, j, val);
    long 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();
  long* array = malloc(sizeof(long) * 5);
  for (; i--;) {
    int j = INT(rnd() * ((float)(VecGetDim(v) - 1) - PBMATH_EPSILON));
    long val = 1;
    array[j] = val;
    long valb = array[j];
    valb = valb;
  }
  clockAfter = clock();
  double timeRef = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
  printf("VecLong: %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, "UnitTestSpeedVecLong NOK");
    PBErrCatch(PBMathErr);
  }

  srandom(RANDOMSEED);
  i = nbTest;
  clockBefore = clock();
  for (; i--;) {
    int j = INT(rnd() * (1.0 - PBMATH_EPSILON));
    long val = 1;
    VecSet(&v2, j, val);
    long valb = VecGet(&v2, j);
    valb = valb;
  }
  clockAfter = clock();
```

```
  timeV = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
  srandom(RANDOMSEED);
  i = nbTest;
  clockBefore = clock();
  long array2[2];
  for (; i--;) {
    int j = INT(rnd() * (1.0 - PBMATH_EPSILON));
    long val = 1;
    array2[j] = val;
    long valb = array2[j];
    valb = valb;
  }
  clockAfter = clock();
  timeRef = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
  printf("VecLong2D: %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, "UnitTestSpeedVecLong NOK");
    PBErrCatch(PBMathErr);
  }

  srandom(RANDOMSEED);
  i = nbTest;
  clockBefore = clock();
  for (; i--;) {
    int j = INT(rnd() * (2.0 - PBMATH_EPSILON));
    long val = 1;
    VecSet(&v3, j, val);
    long valb = VecGet(&v3, j);
    valb = valb;
  }
  clockAfter = clock();
  timeV = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
  srandom(RANDOMSEED);
  i = nbTest;
  clockBefore = clock();
  long array3[3];
  for (; i--;) {
    int j = INT(rnd() * (2.0 - PBMATH_EPSILON));
    long val = 1;
    array3[j] = val;
    long valb = array3[j];
    valb = valb;
  }
  clockAfter = clock();
  timeRef = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
  printf("VecLong3D: %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, "UnitTestSpeedVecLong NOK");
```

```
    PBErrCatch(PBMathErr);
  }

  srandom(RANDOMSEED);
  i = nbTest;
  clockBefore = clock();
  for (; i--;) {
    int j = INT(rnd() * (3.0 - PBMATH_EPSILON));
    long val = 1;
    VecSet(&v4, j, val);
    long valb = VecGet(&v4, j);
    valb = valb;
  }
  clockAfter = clock();
  timeV = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
  srandom(RANDOMSEED);
  i = nbTest;
  clockBefore = clock();
  long array4[4];
  for (; i--;) {
    int j = INT(rnd() * (3.0 - PBMATH_EPSILON));
    long val = 1;
    array4[j] = val;
    long valb = array4[j];
    valb = valb;
  }
  clockAfter = clock();
  timeRef = ((double)(clockAfter - clockBefore)) /
    CLOCKS_PER_SEC * 1000.0;
  printf("VecLong4D: %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, "UnitTestSpeedVecLong NOK");
    PBErrCatch(PBMathErr);
  }

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

void UnitTestVecLongOp() {
  VecLong* v = VecLongCreate(5);
  VecLong2D v2 = VecLongCreateStatic2D();
  VecLong3D v3 = VecLongCreateStatic3D();
  VecLong4D v4 = VecLongCreateStatic4D();
  VecLong* w = VecLongCreate(5);
  VecLong2D w2 = VecLongCreateStatic2D();
  VecLong3D w3 = VecLongCreateStatic3D();
  VecLong4D w4 = VecLongCreateStatic4D();
  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);
  long a[2] = {-1, 2};
  long 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);
VecLong* u = VecGetOp(v, a[0], w, a[1]);
VecLong2D u2 = VecGetOp(&v2, a[0], &w2, a[1]);
VecLong3D u3 = VecGetOp(&v3, a[0], &w3, a[1]);
VecLong4D u4 = VecGetOp(&v4, a[0], &w4, a[1]);
long checku[5] = {8,5,2,-1,-4};
long checku2[2] = {2,-1};
long checku3[3] = {4,1,-2};
long checku4[4] = {6,3,0,-3};
for (int i = 5; i--;)
  if (!ISEQUALF(VecGet(u, i), checku[i])) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongGetOp NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 2; i--;)
  if (!ISEQUALF(VecGet(&u2, i), checku2[i])) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongGetOp NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 3; i--;)
  if (!ISEQUALF(VecGet(&u3, i), checku3[i])) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongGetOp NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 4; i--;)
  if (!ISEQUALF(VecGet(&u4, i), checku4[i])) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecLongGetOp NOK");
    PBErrCatch(PBMathErr);
  }
VecOp(v, a[0], w, a[1]);
VecOp(&v2, a[0], &w2, a[1]);
VecOp(&v3, a[0], &w3, a[1]);
VecOp(&v4, a[0], &w4, a[1]);
if (!VecIsEqual(v, u)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongOp NOK");
  PBErrCatch(PBMathErr);
}
if (!VecIsEqual(&v2, &u2)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongOp NOK");
  PBErrCatch(PBMathErr);
}
if (!VecIsEqual(&v3, &u3)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongOp NOK");
  PBErrCatch(PBMathErr);
}
if (!VecIsEqual(&v4, &u4)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecLongOp NOK");
  PBErrCatch(PBMathErr);
}
```

```
  VecFree(&v);
  VecFree(&w);
  VecFree(&u);
  printf("UnitTestVecLongOp OK\n");
}

void UnitTestVecLongShiftStep() {
  VecLong3D v = VecLongCreateStatic3D();
  VecLong3D from = VecLongCreateStatic3D();
  VecLong3D to = VecLongCreateStatic3D();
  VecSet(&from, 0, 0);
  VecSet(&from, 1, 1);
  VecSet(&from, 2, 2);
  VecSet(&to, 0, 3);
  VecSet(&to, 1, 4);
  VecSet(&to, 2, 5);
  VecCopy(&v, &from);
  long check[81] = {
    0, 1, 2, 0, 1, 3, 0, 1, 4,
    0, 2, 2, 0, 2, 3, 0, 2, 4,
    0, 3, 2, 0, 3, 3, 0, 3, 4,
    1, 1, 2, 1, 1, 3, 1, 1, 4,
    1, 2, 2, 1, 2, 3, 1, 2, 4,
    1, 3, 2, 1, 3, 3, 1, 3, 4,
    2, 1, 2, 2, 1, 3, 2, 1, 4,
    2, 2, 2, 2, 2, 3, 2, 2, 4,
    2, 3, 2, 2, 3, 3, 2, 3, 4
    };
  int iCheck = 0;
  do {
    for (int i = 0; i < 3; ++i) {
      if (ISEQUALF(check[iCheck], VecGet(&v, i)) == false) {
        PBMathErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBMathErr->_msg, "VecShiftStep NOK");
        PBErrCatch(PBMathErr);
      }
      ++iCheck;
    }
  } while(VecShiftStep(&v, &from, &to));
  printf("UnitTestVecLongShiftStep OK\n");
}

void UnitTestVecLongGetMinMax() {
  VecLong3D v = VecLongCreateStatic3D();
  VecSet(&v, 0, 2); VecSet(&v, 1, 4); VecSet(&v, 2, 3);
  long val = VecGetMaxVal(&v);
  if (val != 4) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMaxVal NOK");
    PBErrCatch(PBMathErr);
  }
  if (VecGetIMaxVal(&v) != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetIMaxVal NOK");
    PBErrCatch(PBMathErr);
  }
  VecSet(&v, 0, 2); VecSet(&v, 1, 1); VecSet(&v, 2, 3);
  val = VecGetMinVal(&v);
  if (val != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMinVal NOK");
    PBErrCatch(PBMathErr);
```

```
  }
  VecSet(&v, 0, 2); VecSet(&v, 1, -4); VecSet(&v, 2, 3);
  val = VecGetMaxValAbs(&v);
  if (val != -4) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMaxValAbs NOK");
    PBErrCatch(PBMathErr);
  }
  VecSet(&v, 0, -2); VecSet(&v, 1, 1); VecSet(&v, 2, 3);
  val = VecGetMinValAbs(&v);
  if (val != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMinValAbs NOK");
    PBErrCatch(PBMathErr);
  }
  printf("UnitTestVecLongGetMinMax OK\n");
}

void UnitTestVecLongHadamardProd() {
  VecLong* u = VecLongCreate(3);
  for (int i = 3; i--;)
    VecSet(u, i, i + 2);
  VecLong* uprod = VecGetHadamardProd(u, u);
  VecHadamardProd(u, u);
  long checku[3] = {4, 9, 16};
  for (int i = 3; i--;)
    if (ISEQUALF(VecGet(uprod, i), checku[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(uprod, u) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  VecFree(&uprod);
  VecFree(&u);
  VecLong2D v = VecLongCreateStatic2D();
  for (int i = 2; i--;)
    VecSet(&v, i, i + 2);
  VecLong2D vprod = VecGetHadamardProd(&v, &v);
  VecHadamardProd(&v, &v);
  long checkv[2] = {4, 9};
  for (int i = 2; i--;)
    if (ISEQUALF(VecGet(&vprod, i), checkv[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(&vprod, &v) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  VecLong3D w = VecLongCreateStatic3D();
  for (int i = 3; i--;)
    VecSet(&w, i, i + 2);
  VecLong3D wprod = VecGetHadamardProd(&w, &w);
  VecHadamardProd(&w, &w);
  long checkw[3] = {4, 9, 16};
  for (int i = 3; i--;)
```

```
    if (ISEQUALF(VecGet(&wprod, i), checkw[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(&wprod, &w) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  VecLong4D x = VecLongCreateStatic4D();
  for (int i = 4; i--;)
    VecSet(&x, i, i + 2);
  VecLong4D xprod = VecGetHadamardProd(&x, &x);
  VecHadamardProd(&x, &x);
  long checkx[4] = {4, 9, 16, 25};
  for (int i = 4; i--;)
    if (ISEQUALF(VecGet(&xprod, i), checkx[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(&xprod, &x) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  printf("UnitTestVecLongHadamardProd OK\n");
}

void UnitTestVecLongGetNewDim() {
  VecLong* v = VecLongCreate(3);
  for (int i = 3; i--;)
    VecSet(v, i, i);
  VecLong* u = VecGetNewDim(v, 2);
  if (VecGetDim(u) != 2 ||
    VecGet(u, 0) != 0 ||
    VecGet(u, 1) != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetNewDim NOK 1");
    PBErrCatch(PBMathErr);
  }
  VecLong* w = VecGetNewDim(v, 4);
  if (VecGetDim(w) != 4 ||
    VecGet(w, 0) != 0 ||
    VecGet(w, 1) != 1 ||
    VecGet(w, 2) != 2 ||
    VecGet(w, 3) != 0) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetNewDim NOK 2");
    PBErrCatch(PBMathErr);
  }
  VecFree(&v);
  VecFree(&u);
  VecFree(&w);
  printf("UnitTestVecLongGetNewDim OK\n");
}

void UnitTestVecLong() {
  UnitTestVecLongCreateFree();
  UnitTestVecLongClone();
  UnitTestVecLongLoadSave();
```

```
  UnitTestVecLongGetSetDim();
  UnitTestVecLongStep();
  UnitTestVecLongHamiltonDist();
  UnitTestVecLongIsEqual();
  UnitTestVecLongDotProd();
  UnitTestVecLongCopy();
  UnitTestSpeedVecLong();
  UnitTestVecLongOp();
  UnitTestVecLongShiftStep();
  UnitTestVecLongGetMinMax();
  UnitTestVecLongHadamardProd();
  UnitTestVecLongGetNewDim();
  printf("UnitTestVecLong OK\n");
}

void UnitTestVecFloatCreateFree() {
  VecFloat* v = VecFloatCreate(5);
  VecFloat2D v2 = VecFloatCreateStatic2D();
  VecFloat3D v3 = VecFloatCreateStatic3D();
  VecPrintln(v, stdout);
  VecPrintln(&v2, stdout);
  VecPrintln(&v3, stdout);
  _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);
  }
  bool compact = false;
  if (!VecSave(v, f, compact)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
```

```
    sprintf(PBMathErr->_msg, "_VecFloatSave NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecSave(&v2, f, compact)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatSave NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecSave(&v3, f, compact)) {
    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 = ret;
}

void UnitTestVecFloatGetSetDim() {
  VecFloat* v = VecFloatCreate(5);
```

```
VecFloat2D v2 = VecFloatCreateStatic2D();
VecFloat3D v3 = VecFloatCreateStatic3D();
if (VecGetDim(v) != 5) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "_VecFloatGetDim 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);
  }
for (int i = 5; i--;) VecSetAdd(v, i, (float)(i + 1));
for (int i = 2; i--;) VecSetAdd(&v2, i, (float)(i + 1));
for (int i = 3; i--;) VecSetAdd(&v3, i, (float)(i + 1));
for (int i = 5; i--;)
  if (!ISEQUALF(VecGet(v, i), 2.0 * (float)(i + 1))) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatSetAdd NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 2; i--;)
  if (!ISEQUALF(VecGet(&v2, i), 2.0 * (float)(i + 1))) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatSetAdd NOK");
    PBErrCatch(PBMathErr);
  }
for (int i = 3; i--;)
```

```
    if (!ISEQUALF(VecGet(&v3, i), 2.0 * (float)(i + 1))) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecFloatSetAdd 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, "_VecFloatSetNull NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 2; i--;)
    if (!ISEQUALF(VecGet(&v2, i), 0.0)) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecFloatSetNull NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 3; i--;)
    if (!ISEQUALF(VecGet(&v3, i), 0.0)) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecFloatSetNull NOK");
      PBErrCatch(PBMathErr);
    }
  VecSetAll(v, 1.0);
  VecSetAll(&v2, 1.0);
  VecSetAll(&v3, 1.0);
  for (int i = 5; i--;)
    if (!ISEQUALF(VecGet(v, i), 1.0)) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecFloatSetAll NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 2; i--;)
    if (!ISEQUALF(VecGet(&v2, i), 1.0)) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecFloatSetAll NOK");
      PBErrCatch(PBMathErr);
    }
  for (int i = 3; i--;)
    if (!ISEQUALF(VecGet(&v3, i), 1.0)) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecFloatSetAll 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");
    PBErrCatch(PBMathErr);
  }
  if (!ISEQUALF(VecDist(&v3, &w3), 3.840573)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatDist NOK");
    PBErrCatch(PBMathErr);
  }
  if (!ISEQUALF(VecHamiltonDist(v, w), 13.5)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatHamiltonDist NOK");
    PBErrCatch(PBMathErr);
  }
  if (!ISEQUALF(VecHamiltonDist(&v2, &w2), 3.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatHamiltonDist NOK");
    PBErrCatch(PBMathErr);
  }
  if (!ISEQUALF(VecHamiltonDist(&v3, &w3), 5.5)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatHamiltonDist NOK");
    PBErrCatch(PBMathErr);
  }
  if (!ISEQUALF(VecPixelDist(v, w), 13.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatPixelDist NOK");
    PBErrCatch(PBMathErr);
  }
  if (!ISEQUALF(VecPixelDist(&v2, &w2), 2.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatPixelDist NOK");
    PBErrCatch(PBMathErr);
  }
  if (!ISEQUALF(VecPixelDist(&v3, &w3), 5.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_VecFloatPixelDist NOK");
    PBErrCatch(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 = 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);
  }
  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 UnitTestVecFloatCrossProd() {
  VecFloat* v = VecFloatCreate(3);
  VecFloat3D v3 = VecFloatCreateStatic3D();
  VecFloat* w = VecFloatCreate(3);
  VecFloat3D w3 = VecFloatCreateStatic3D();
  float a[3] = {3.0, -3.0, 1.0};
  float b[3] = {4.0, 9.0, 2.0};
  float c[3] = {-15.0, -2.0, 39.0};
  for (int i = 3; i--;) VecSet(v, i, a[i]);
  for (int i = 3; i--;) VecSet(&v3, i, a[i]);
  for (int i = 3; i--;) VecSet(w, i, b[i]);
  for (int i = 3; i--;) VecSet(&w3, i, b[i]);
  VecFloat* prod = VecCrossProd(v, w);
  for (int i = 3; i--;)
    if (!ISEQUALF(VecGet(prod, i), c[i])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecFloatCrossProd NOK");
      PBErrCatch(PBMathErr);
    }
  VecFloat3D prod3 = VecCrossProd(&v3, &w3);
  for (int i = 3; i--;)
    if (!ISEQUALF(VecGet(&prod3, i), c[i])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "_VecFloatCrossProd3D NOK");
      PBErrCatch(PBMathErr);
    }
  VecFree(&v);
  VecFree(&w);
  VecFree(&prod);
  printf("UnitTestVecFloatCrossProd 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);
  VecFloatPrint(v,stdout,6);printf("\n");
  if (!ISEQUALF(VecGet(v, 0), 0.707107) ||
    !ISEQUALF(VecGet(v, 1), 0.707107)) {
    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);
  VecPrintln(w, stdout);
  VecPrintln(&w2, stdout);
  VecPrintln(&w3, stdout);
  VecFree(&v);
  VecFree(&w);
  printf("UnitTestVecFloatToShort OK\n");
}

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

  srandom(RANDOMSEED);
  int i = nbTest;
  clock_t clockBefore = clock();
  for (; i--;) {
    int j = INT(rnd() * ((float)(VecGetDim(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;
  srandom(RANDOMSEED);
  i = nbTest;
  clockBefore = clock();
  float* array = malloc(sizeof(float) * 5);
  for (; i--;) {
    int j = INT(rnd() * ((float)(VecGetDim(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) {
#if 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() * (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) {
#if 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 UnitTestVecFloatRotAxis() {
  VecFloat3D v = VecFloatCreateStatic3D();
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
  VecFloat3D axis = VecFloatCreateStatic3D();
  VecSet(&axis, 0, 1.0); VecSet(&axis, 1, 1.0); VecSet(&axis, 2, 1.0);
  VecNormalise(&axis);
  float theta = PBMATH_PI;
  VecRotAxis(&v, &axis, theta);
  if (!ISEQUALF(VecGet(&v, 0), 0.333333) ||
    !ISEQUALF(VecGet(&v, 1), 1.333333) ||
    !ISEQUALF(VecGet(&v, 2), 0.333333)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecRotAxis NOK");
    PBErrCatch(PBMathErr);
  }
  theta = PBMATH_HALFPI;
  VecRotAxis(&v, &axis, theta);
  if (!ISEQUALF(VecGet(&v, 0), 0.089316) ||
    !ISEQUALF(VecGet(&v, 1), 0.666666) ||
    !ISEQUALF(VecGet(&v, 2), 1.244017)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecRotAxis NOK");
    PBErrCatch(PBMathErr);
  }
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 1.0);
  theta = PBMATH_PI;
  VecRotX(&v, theta);
  if (!ISEQUALF(VecGet(&v, 0), 1.0) ||
    !ISEQUALF(VecGet(&v, 1), -1.0) ||
    !ISEQUALF(VecGet(&v, 2), -1.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecRotX NOK");
    PBErrCatch(PBMathErr);
  }
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 1.0);
  theta = PBMATH_PI;
  VecRotY(&v, theta);
  if (!ISEQUALF(VecGet(&v, 0), -1.0) ||
    !ISEQUALF(VecGet(&v, 1), 1.0) ||
    !ISEQUALF(VecGet(&v, 2), -1.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecRotY NOK");
    PBErrCatch(PBMathErr);
  }
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 1.0);
  theta = PBMATH_PI;
  VecRotZ(&v, theta);
  if (!ISEQUALF(VecGet(&v, 0), -1.0) ||
    !ISEQUALF(VecGet(&v, 1), -1.0) ||
    !ISEQUALF(VecGet(&v, 2), 1.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecRotZ NOK");
    PBErrCatch(PBMathErr);
```

```
  }
  printf("UnitTestVecFloatRotAxis OK\n");
}

void UnitTestVecFloatGetMinMax() {
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 2.0);
  float val = VecGetMaxVal(&v);
  if (ISEQUALF(val, 2.0) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMaxVal NOK");
    PBErrCatch(PBMathErr);
  }
  if (VecGetIMaxVal(&v) != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetIMaxVal NOK");
    PBErrCatch(PBMathErr);
  }
  val = VecGetMinVal(&v);
  if (ISEQUALF(val, 1.0) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMinVal NOK");
    PBErrCatch(PBMathErr);
  }
  VecSet(&v, 0, 1.0); VecSet(&v, 1, -2.0);
  val = VecGetMaxValAbs(&v);
  if (ISEQUALF(val, -2.0) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMaxValAbs NOK");
    PBErrCatch(PBMathErr);
  }
  val = VecGetMinValAbs(&v);
  if (ISEQUALF(val, 1.0) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetMinValAbs NOK");
    PBErrCatch(PBMathErr);
  }
  printf("UnitTestVecFloatGetMinMax OK\n");
}

void UnitTestVecFloatGetNewDim() {
  VecFloat* v = VecFloatCreate(3);
  for (int i = 3; i--;)
    VecSet(v, i, (float)i);
  VecFloat* u = VecGetNewDim(v, 2);
  if (VecGetDim(u) != 2 ||
    ISEQUALF(VecGet(u, 0), 0.0) == false ||
    ISEQUALF(VecGet(u, 1), 1.0) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetNewDim NOK");
    PBErrCatch(PBMathErr);
  }
  VecFloat* w = VecGetNewDim(v, 4);
  if (VecGetDim(w) != 4 ||
    ISEQUALF(VecGet(w, 0), 0.0) == false ||
    ISEQUALF(VecGet(w, 1), 1.0) == false ||
    ISEQUALF(VecGet(w, 2), 2.0) == false ||
    ISEQUALF(VecGet(w, 3), 0.0) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecGetNewDim NOK");
    PBErrCatch(PBMathErr);
  }
```

```
  VecFree(&v);
  VecFree(&u);
  VecFree(&w);
  printf("UnitTestVecFloatGetNewDim OK\n");
}

void UnitTestVecFloatHadamardProd() {
  VecFloat* u = VecFloatCreate(3);
  for (int i = 3; i--;)
    VecSet(u, i, (float)i + 2.0);
  VecFloat* uprod = VecGetHadamardProd(u, u);
  VecHadamardProd(u, u);
  float checku[3] = {4.0, 9.0, 16.0};
  for (int i = 3; i--;)
    if (ISEQUALF(VecGet(uprod, i), checku[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(uprod, u) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  VecFree(&uprod);
  VecFree(&u);
  VecFloat2D v = VecFloatCreateStatic2D();
  for (int i = 2; i--;)
    VecSet(&v, i, (float)i + 2.0);
  VecFloat2D vprod = VecGetHadamardProd(&v, &v);
  VecHadamardProd(&v, &v);
  float checkv[2] = {4.0, 9.0};
  for (int i = 2; i--;)
    if (ISEQUALF(VecGet(&vprod, i), checkv[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(&vprod, &v) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  VecFloat3D w = VecFloatCreateStatic3D();
  for (int i = 3; i--;)
    VecSet(&w, i, (float)i + 2.0);
  VecFloat3D wprod = VecGetHadamardProd(&w, &w);
  VecHadamardProd(&w, &w);
  float checkw[3] = {4.0, 9.0, 16.0};
  for (int i = 3; i--;)
    if (ISEQUALF(VecGet(&wprod, i), checkw[i]) == false) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "VecGetHadamardProd NOK");
      PBErrCatch(PBMathErr);
    }
  if (VecIsEqual(&wprod, &w) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "VecHadamardProd NOK");
    PBErrCatch(PBMathErr);
  }
  printf("UnitTestVecFloatHadamardProd OK\n");
}
```

```
void UnitTestVecFloat() {
  UnitTestVecFloatCreateFree();
  UnitTestVecFloatClone();
  UnitTestVecFloatLoadSave();
  UnitTestVecFloatGetSetDim();
  UnitTestVecFloatCopy();
  UnitTestVecFloatNorm();
  UnitTestVecFloatDist();
  UnitTestVecFloatIsEqual();
  UnitTestVecFloatScale();
  UnitTestVecFloatOp();
  UnitTestVecFloatDotProd();
  UnitTestVecFloatCrossProd();
  UnitTestVecFloatRotAngleTo();
  UnitTestVecFloatToShort();
  UnitTestVecFloatGetMinMax();
  UnitTestVecFloatRotAxis();
  UnitTestVecFloatGetNewDim();
  UnitTestVecFloatHadamardProd();
  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");
    PBErrCatch(PBMathErr);
  }
  if (!VecIsEqual(MatDim(mat), &dim)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "UnitTestMatFloatGetSetDim NOK");
    PBErrCatch(PBMathErr);
  }
  if (MatGetNbRow(mat) != 3) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
```

```
    sprintf(PBMathErr->_msg, "MatGetNbRow NOK");
    PBErrCatch(PBMathErr);
  }
  if (MatGetNbCol(mat) != 2) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "MatGetNbCol NOK");
    PBErrCatch(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");
        PBErrCatch(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");
      PBErrCatch(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");
    PBErrCatch(PBMathErr);
  }
  VecSetNull(&i);
  do {
    if (!ISEQUALF(MatGet(mat, &i), MatGet(clone, &i))) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "UnitTestMatFloatClone NOK");
```

```
      PBErrCatch(PBMathErr);
    }
  } while(VecStep(&i, &dim));
  if (MatIsEqual(mat, clone) == false) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "UnitTestMatFloatIsEqual NOK1");
    PBErrCatch(PBMathErr);
  }
  VecSet(&i, 0, 0); VecSet(&i, 1, 0);
  MatSet(clone, &i, -1.0);
  if (MatIsEqual(mat, clone) == true) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "UnitTestMatFloatIsEqual NOK2");
    PBErrCatch(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 = PBErrTypeOther;
    sprintf(PBMathErr->_msg,
      "Can't open ./UnitTestMatFloatLoadSave.txt for writing");
    PBErrCatch(PBMathErr);
  }
  bool compact = false;
  if (!MatSave(mat, f, compact)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_MatFloatSave NOK");
    PBErrCatch(PBMathErr);
  }
  fclose(f);
  MatFloat* clone = MatFloatCreate(&dim);
  f = fopen("./UnitTestMatFloatLoadSave.txt", "r");
  if (f == NULL) {
    PBMathErr->_type = PBErrTypeOther;
    sprintf(PBMathErr->_msg,
      "Can't open ./UnitTestMatFloatLoadSave.txt for reading");
    PBErrCatch(PBMathErr);
  }
  if (!MatLoad(&clone, f)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "_MatFloatLoad NOK");
    PBErrCatch(PBMathErr);
  }
  if (!VecIsEqual(&(mat->_dim), &(clone->_dim))) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "UnitTestMatFloatLoadSave NOK");
    PBErrCatch(PBMathErr);
```

```
  }
  VecSetNull(&i);
  do {
    if (!ISEQUALF(MatGet(mat, &i), MatGet(clone, &i))) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "UnitTestMatFloatLoadSave NOK");
      PBErrCatch(PBMathErr);
    }
  } while(VecStep(&i, &dim));
  fclose(f);
  MatFree(&mat);
  MatFree(&clone);
  int ret = system("cat ./UnitTestMatFloatLoadSave.txt");
  ret = ret;
  printf("UnitTestMatFloatLoadSave OK\n");
}

void UnitTestMatFloatTransposeScale() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 2);
  VecSet(&dim, 1, 3);
  MatFloat* mat = MatFloatCreate(&dim);
  VecShort2D i = VecShortCreateStatic2D();
  float v[6] = {3.0, 2.0, 1.0, 2.0, -2.0, 1.0};
  int j = 0;
  do {
    MatSet(mat, &i, v[j]);
    ++j;
  } while(VecStep(&i, &dim));
  MatFloat* trans = MatGetTranspose(mat);
  float w[6] = {3.0, 2.0, 2.0, -2.0, 1.0, 1.0};
  VecSet(&dim, 0, 3);
  VecSet(&dim, 1, 2);
  VecSetNull(&i);
  j = 0;
  do {
    if (!ISEQUALF(MatGet(trans, &i), w[j])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "UnitTestMatFloatTranspose NOK");
      PBErrCatch(PBMathErr);
    }
    ++j;
  } while(VecStep(&i, &dim));
  MatScale(mat, 2.0);
  j = 0;
  VecSet(&dim, 0, 2);
  VecSet(&dim, 1, 3);
  VecSetNull(&i);
  float u[6] = {6.0, 4.0, 2.0, 4.0, -4.0, 2.0};
  do {
    if (!ISEQUALF(MatGet(mat, &i), u[j])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "MatScale NOK");
      PBErrCatch(PBMathErr);
    }
    ++j;
  } while(VecStep(&i, &dim));
  MatFree(&mat);
  MatFree(&trans);
  printf("UnitTestMatFloatTransposeScale 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 = MatGetInv(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 (1)");
      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 = MatGetInv(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 (2)");
      PBErrCatch(PBMathErr);
    }
    ++j;
  } while(VecStep(&i, &dim));
  MatFree(&mat);
  MatFree(&inv);


  VecSet(&dim, 0, 4);
  VecSet(&dim, 1, 4);
  mat = MatFloatCreate(&dim);
  float vc[16] = {4, 0, 0, 0, 0, 1, 2, 0, 0, 0, 2, 0, 1, 0, 0, 1};
  VecSetNull(&i);
  j = 0;
  do {
    MatSet(mat, &i, vc[j]);
    ++j;
  } while(VecStep(&i, &dim));
```

```
inv = MatGetInv(mat);
if (inv == NULL) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (3)");
  PBErrCatch(PBMathErr);
}
float wc[16] = {0.25, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0.5, 0, -0.25, 0, 0, 1};
VecSetNull(&i);
j = 0;
do {
  if (!ISEQUALF(MatGet(inv, &i), wc[j])) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (4)");
    PBErrCatch(PBMathErr);
  }
  ++j;
} while(VecStep(&i, &dim));
MatFree(&mat);
MatFree(&inv);

mat = MatFloatCreate(&dim);
float vd[16] = {4, 0, 0, 0, 0, 0, 2, 0, 0, 1, 2, 0, 1, 0, 0, 1};
VecSetNull(&i);
j = 0;
do {
  MatSet(mat, &i, vd[j]);
  ++j;
} while(VecStep(&i, &dim));
inv = MatGetInv(mat);
if (inv == NULL) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (3)");
  PBErrCatch(PBMathErr);
}
float wd[16] = {0.25, 0, 0, 0, 0, -1, 1, 0, 0, 0.5, 0, 0, -0.25, 0, 0, 1};
VecSetNull(&i);
j = 0;
do {
  if (!ISEQUALF(MatGet(inv, &i), wd[j])) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (5)");
    PBErrCatch(PBMathErr);
  }
  ++j;
} while(VecStep(&i, &dim));
MatFree(&mat);
MatFree(&inv);

mat = MatFloatCreate(&dim);
float ve[16] = {4, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 1, 0, 0, 1};
VecSetNull(&i);
j = 0;
do {
  MatSet(mat, &i, ve[j]);
  ++j;
} while(VecStep(&i, &dim));
inv = MatGetInv(mat);
if (inv != NULL) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (6)");
  PBErrCatch(PBMathErr);
}
```

```
MatFree(&mat);

mat = MatFloatCreate(&dim);
float vf[16] = {0, 1, 2, 0, 0, 0, 2, 0, 1, 0, 0, 1, 4, 0, 0, 0};
VecSetNull(&i);
j = 0;
do {
  MatSet(mat, &i, vf[j]);
  ++j;
} while(VecStep(&i, &dim));
inv = MatGetInv(mat);
if (inv != NULL) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (7)");
  PBErrCatch(PBMathErr);
}
/* TODO
if (inv == NULL) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (7)");
  PBErrCatch(PBMathErr);
}
float wf[16] = {0, 0, 0, 0.25, 1, -1, 0, 0, 0, 0.5, 0, 0, 0, 0, 1, -0.25};
VecSetNull(&i);
j = 0;
do {
  if (!ISEQUALF(MatGet(inv, &i), wf[j])) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (8)");
    PBErrCatch(PBMathErr);
  }
  ++j;
} while(VecStep(&i, &dim));
MatFree(&inv);
*/
MatFree(&mat);

mat = MatFloatCreate(&dim);
float vg[16] = {0, 1, 2, 0, 0, 0, 2, 0, 4, 0, 0, 0, 1, 0, 0, 1};
VecSetNull(&i);
j = 0;
do {
  MatSet(mat, &i, vg[j]);
  ++j;
} while(VecStep(&i, &dim));
inv = MatGetInv(mat);
if (inv != NULL) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (9)");
  PBErrCatch(PBMathErr);
}
/* TODO
if (inv == NULL) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (9)");
  PBErrCatch(PBMathErr);
}
float wg[16] = {0, 0, 0.25, 0, 1, -1, 0, 0, 0, 0.5, 0, 0, 0, 0, -0.25, 1};
VecSetNull(&i);
j = 0;
do {
  if (!ISEQUALF(MatGet(inv, &i), wg[j])) {
```

```
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "UnitTestMatFloatInv NOK (10)");
      PBErrCatch(PBMathErr);
    }
    ++j;
  } while(VecStep(&i, &dim));
  MatFree(&inv);
  */
  MatFree(&mat);

  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 = MatGetProdVec(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 = MatGetProdMat(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 UnitTestMatFloatProdVecVecTranspose() {
  VecFloat2D v = VecFloatCreateStatic2D();
  VecFloat3D w = VecFloatCreateStatic3D();
  VecSet(&v, 0, 2.0);
  VecSet(&v, 1, 3.0);
  VecSet(&w, 0, 4.0);
  VecSet(&w, 1, 5.0);
  VecSet(&w, 2, 6.0);
  MatFloat* mat = MatGetProdVecVecTranspose(&v, &w);
  VecShort2D pos = VecShortCreateStatic2D();
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3);
  VecSet(&dim, 1, 2);
  float check[6] = {8.0, 12.0, 10.0, 15.0, 12.0, 18.0};
  int i = 0;
  do {
    if (!ISEQUALF(MatGet(mat, &pos), check[i])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "MatGetProdVecVecTranspose NOK");
      PBErrCatch(PBMathErr);
    }
    ++i;
  } while (VecStep(&pos, &dim));
  MatFree(&mat);
  printf("UnitTestMatFloatProdVecVecTranspose OK\n");
}

void UnitTestMatFloatGetEigenValues() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3);
  VecSet(&dim, 1, 3);
  MatFloat* mat = MatFloatCreate(&dim);
  VecShort2D pos = VecShortCreateStatic2D();
  float check[3][3] = {
    { 2.92, 0.86, -1.15},
    { 0.86, 6.51,  3.32},
    {-1.15, 3.32,  4.57}
  };
```

```
  do {
    MatSet(mat, &pos, check[VecGet(&pos, 1)][VecGet(&pos, 0)]);
  } while (VecStep(&pos, &dim));
  printf("mat:\n"); MatPrintln(mat, stdout);
  GSetVecFloat set = MatGetEigenValues(mat);
  printf("Eigen values: ");
  VecPrintln(GSetGet(&set, 0), stdout);
  VecFloat3D checkValues = VecFloatCreateStatic3D();
  VecSet(&checkValues, 0, 8.998802);
  VecSet(&checkValues, 1, 3.996595);
  VecSet(&checkValues, 2, 1.004607);
  if (!VecIsEqual(GSetGet(&set, 0), &checkValues)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "MatGetEigenValues NOK");
    PBErrCatch(PBMathErr);
  }
  printf("Eigen vector 1: ");
  VecPrintln(GSetGet(&set, 1), stdout);
  printf("Eigen vector 2: ");
  VecPrintln(GSetGet(&set, 2), stdout);
  printf("Eigen vector 3: ");
  VecPrintln(GSetGet(&set, 3), stdout);
  VecFloat3D checkVecA = VecFloatCreateStatic3D();
  VecSet(&checkVecA, 0, 0.000290);
  VecSet(&checkVecA, 1, -0.800102);
  VecSet(&checkVecA, 2, -0.599864);
  if (!VecIsEqual(GSetGet(&set, 1), &checkVecA)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "MatGetEigenValues NOK");
    PBErrCatch(PBMathErr);
  }
  VecFloat3D checkVecB = VecFloatCreateStatic3D();
  VecSet(&checkVecB, 0, 0.800110);
  VecSet(&checkVecB, 1, 0.360017);
  VecSet(&checkVecB, 2, -0.479806);
  if (!VecIsEqual(GSetGet(&set, 2), &checkVecB)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "MatGetEigenValues NOK");
    PBErrCatch(PBMathErr);
  }
  VecFloat3D checkVecC = VecFloatCreateStatic3D();
  VecSet(&checkVecC, 0, 0.599855);
  VecSet(&checkVecC, 1, -0.479817);
  VecSet(&checkVecC, 2, 0.640273);
  if (!VecIsEqual(GSetGet(&set, 3), &checkVecC)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "MatGetEigenValues NOK");
    PBErrCatch(PBMathErr);
  }
  do {
    VecFloat* v = GSetPop(&set);
    VecFree(&v);
  } while (GSetNbElem(&set) > 0);
  MatFree(&mat);
  printf("UnitTestMatFloatGetEigenValues 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) {
#if 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 UnitTestMatFloatGetQR() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3);
  VecSet(&dim, 1, 4);
  MatFloat* mat = MatFloatCreate(&dim);
  VecShort2D pos = VecShortCreateStatic2D();
  float val[4][3] = {
      {-1.0, -1.0, 1.0},
      { 1.0,  3.0, 3.0},
      {-1.0, -1.0, 5.0},
      { 1.0,  3.0, 7.0}
    };
  do {
    MatSet(mat, &pos, val[VecGet(&pos, 1)][VecGet(&pos, 0)]);
  } while (VecStep(&pos, &dim));
  QRDecomp qr = MatGetQR(mat);
```

```
  MatFloat* QR = MatGetProdMat(qr._Q, qr._R);

  printf("mat:\n");
  MatPrintln(mat, stdout);
  printf("Q:\n");
  MatPrintln(qr._Q, stdout);
  printf("R:\n");
  MatPrintln(qr._R, stdout);
  printf("QR:\n");
  MatPrintln(QR, stdout);

  MatFloat* Q = MatFloatCreate(&dim);
  VecSetNull(&pos);
  float checkQ[4][3] = {
      {-0.5, -0.5,  0.5},
      { 0.5, -0.5,  0.5},
      {-0.5, -0.5, -0.5},
      { 0.5, -0.5, -0.5}
    };
  do {
    MatSet(Q, &pos, checkQ[VecGet(&pos, 1)][VecGet(&pos, 0)]);
  } while (VecStep(&pos, &dim));
  VecSet(&dim, 1, 3);
  MatFloat* R = MatFloatCreate(&dim);
  VecSetNull(&pos);
  float checkR[3][3] = {
      {2.0,  4.0,  2.0},
      {0.0, -2.0, -8.0},
      {0.0,  0.0, -4.0}
    };
  do {
    MatSet(R, &pos, checkR[VecGet(&pos, 1)][VecGet(&pos, 0)]);
  } while (VecStep(&pos, &dim));
  if (!MatIsEqual(Q, qr._Q) || !MatIsEqual(R, qr._R) ||
    !MatIsEqual(QR, mat)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "MatGetQR NOK");
    PBErrCatch(PBMathErr);
  }
  MatFree(&mat);
  MatFree(&Q);
  MatFree(&R);
  MatFree(&QR);
  QRDecompFreeStatic(&qr);
  printf("UnitTestMatFloatGetQR OK\n");
}

void UnitTestMatFloat() {
  UnitTestMatFloatCreateFree();
  UnitTestMatFloatGetSetDim();
  UnitTestMatFloatCloneIsEqual();
  UnitTestMatFloatLoadSave();
  UnitTestMatFloatInv();
  UnitTestMatFloatTransposeScale();
  UnitTestMatFloatProdVecFloat();
  UnitTestMatFloatProdMatFloat();
  UnitTestMatFloatGetQR();
  UnitTestMatFloatProdVecVecTranspose();
  UnitTestMatFloatGetEigenValues();
  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 UnitTestThueMorseSeq() {
  long seq_2[16] = {0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0};
  long seq_3[27] = {0,1,2,1,2,0,2,0,1,1,2,0,2,0,1,0,
    1,2,2,0,1,0,1,2,1,2,0};
  long seq_4[64] = {0,1,2,3,1,2,3,0,2,3,0,1,3,0,1,2,
    1,2,3,0,2,3,0,1,3,0,1,2,0,1,2,3,2,3,0,1,3,0,1,2,
    0,1,2,3,1,2,3,0,3,0,1,2,0,1,2,3,1,2,3,0,2,3,0,1};
  for (long iElem = 0; iElem < 16; ++iElem) {
    long thuemorse = ThueMorseSeqGetNthElem(iElem, 2);
    if (thuemorse != seq_2[iElem]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "ThueMorseSeqGetNthElem NOK (%ld,2)",
        iElem);
      PBErrCatch(PBMathErr);
    }
  }
  for (long iElem = 0; iElem < 27; ++iElem) {
    long thuemorse = ThueMorseSeqGetNthElem(iElem, 3);
    if (thuemorse != seq_3[iElem]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "ThueMorseSeqGetNthElem NOK (%ld,3)",
        iElem);
      PBErrCatch(PBMathErr);
    }
  }
  for (long iElem = 0; iElem < 64; ++iElem) {
    long thuemorse = ThueMorseSeqGetNthElem(iElem, 4);
    if (thuemorse != seq_4[iElem]) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "ThueMorseSeqGetNthElem NOK (%ld,4)",
        iElem);
      PBErrCatch(PBMathErr);
    }
  }
  printf("UnitTestThueMorseSeq OK\n");
}

void UnitTestGetAreaTriangleHero() {
  double area = GetAreaTriangleHero(5.0, 29.0, 30.0);
  if (!ISEQUALF(area, 72.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
```

```
    sprintf(PBMathErr->_msg, "GetAreaTriangleHero NOK (%f)",
      area);
    PBErrCatch(PBMathErr);
  }

  printf("UnitTestGetAreaTriangleHero OK\n");
}

void UnitTestGetFibonacciSeq() {
  unsigned long* seq = GetFibonacciSeq(14);
  if (
    seq[0] != 1 ||
    seq[1] != 1 ||
    seq[2] != 2 ||
    seq[3] != 3 ||
    seq[4] != 5 ||
    seq[5] != 8 ||
    seq[6] != 13 ||
    seq[7] != 21 ||
    seq[8] != 34 ||
    seq[9] != 55 ||
    seq[10] != 89 ||
    seq[11] != 144 ||
    seq[12] != 233 ||
    seq[13] != 377) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "GetFibonacciSeq NOK");
    PBErrCatch(PBMathErr);
  }
  free(seq);

  printf("UnitTestGetFibonacciSeq OK\n");
}

void UnitTestGetFibonacciLattice() {

  unsigned long nbPoints = 0;
  float* latticeGrid =
    GetFibonacciGridLattice(
      5,
      &nbPoints);
  if (nbPoints != 5) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "GetFibonacciGridLattice NOK");
    PBErrCatch(PBMathErr);
  }
  float checkGrid[10] =
    {
      0.000000, 0.000000,
      0.200000, 0.600000,
      0.400000, 0.200000,
      0.600000, 0.800000,
      0.800000, 0.400000
    };
  for (int i = 0; i < 10; ++i) {
    if (!ISEQUALF(latticeGrid[i], checkGrid[i])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "GetFibonacciGridLattice NOK");
      PBErrCatch(PBMathErr);
    }
  }
  free(latticeGrid);
```

```
  nbPoints = 0;
  float* latticePolar =
    GetFibonacciPolarLattice(
      5,
      &nbPoints);
  if (nbPoints != 5) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "GetFibonacciPolarLattice NOK");
    PBErrCatch(PBMathErr);
  }
  float checkPolar[10] =
    {
      0.000000, 0.000000,
      0.447214, 3.769911,
      0.632456, 1.256637,
      0.774597, 5.026548,
      0.894427, 2.513275
    };
  for (int i = 0; i < 10; ++i) {
    if (!ISEQUALF(latticePolar[i], checkPolar[i])) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "GetFibonacciPolarLattice NOK");
      PBErrCatch(PBMathErr);
    }
  }
  free(latticePolar);

  printf("UnitTestGetFibonacciLattice OK\n");

}

void UnitTestGetGCD() {

  unsigned int gcd = GetGCD(4, 6);
  if (gcd != 2) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "GetGCD NOK");
    PBErrCatch(PBMathErr);
  }
  gcd = GetGCD(6, 4);
  if (gcd != 2) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "GetGCD NOK");
    PBErrCatch(PBMathErr);
  }
  gcd = GetGCD(10, 6);
  if (gcd != 2) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "GetGCD NOK");
    PBErrCatch(PBMathErr);
  }

  printf("UnitTestGetGCD OK\n");

}

void UnitTestFastInverseSquareRoot() {

  for (float number = 2.0; number < 100.0; number += 1.0) {
    float fsr = GetFastInverseSquareRoot(number);
    float check = 1.0 / sqrt(number);
```

```
    if (fabs(fsr - check) > 0.001) {
      PBMathErr->_type = PBErrTypeUnitTestFailed;
      sprintf(PBMathErr->_msg, "GetFastInverseSquareRoot NOK");
      PBErrCatch(PBMathErr);
    }
  }

  printf("UnitTestGetFastInverseSquareRoot OK\n");
}

void UnitTestBasicFunctions() {
  UnitTestConv();
  UnitTestPowi();
  UnitTestFastPow();
  UnitTestSpeedFastPow();
  UnitTestFSquare();
  UnitTestConv();
  UnitTestThueMorseSeq();
  UnitTestGetAreaTriangleHero();
  UnitTestGetFibonacciSeq();
  UnitTestGetFibonacciLattice();
  UnitTestGetGCD();
  UnitTestFastInverseSquareRoot();
  printf("UnitTestBasicFunctions OK\n");
}

void UnitTestRatio() {
  Ratio ratio = RatioCreateStatic(1, 2, 3);
  if (
    ratio._base != 1 ||
    ratio._numerator != 2 ||
    ratio._denominator != 3) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioCreateStatic NOK");
    PBErrCatch(PBMathErr);
  }
  if (RatioGetBase(&ratio) != 1) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioGetBase NOK");
    PBErrCatch(PBMathErr);
  }
  if (RatioGetNumerator(&ratio) != 2) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioGetNumerator NOK");
    PBErrCatch(PBMathErr);
  }
  if (RatioGetDenominator(&ratio) != 3) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioGetDenominator NOK");
    PBErrCatch(PBMathErr);
  }
  if (!ISEQUALF(RatioToFloat(&ratio), 1.666666)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioToFloat NOK");
    PBErrCatch(PBMathErr);
  }
  RatioSetBase(&ratio, 4);
  if (RatioGetBase(&ratio) != 4) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioSetBase NOK");
    PBErrCatch(PBMathErr);
```

```
  }
  RatioSetNumerator(&ratio, 5);
  if (RatioGetNumerator(&ratio) != 5) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioSetNumerator NOK");
    PBErrCatch(PBMathErr);
  }
  RatioSetDenominator(&ratio, 6);
  if (RatioGetDenominator(&ratio) != 6) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioSetDenominator NOK");
    PBErrCatch(PBMathErr);
  }
  Ratio ratiob = RatioCreateStatic(0, 10, 6);
  RatioPrint(&ratiob, stdout);
  printf(" -> ");
  RatioReduce(&ratiob);
  RatioPrintln(&ratiob, stdout);
  if (
    ratiob._base != 1 ||
    ratiob._numerator != 2 ||
    ratiob._denominator != 3) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioReduce NOK");
    PBErrCatch(PBMathErr);
  }
  Ratio ratioc = RatioFromFloat(1.666666);
  printf("1.666666=");
  RatioPrintln(&ratioc, stdout);
  if (
    ratioc._base != 1 ||
    ratioc._numerator != 2 ||
    ratioc._denominator != 3 ||
    !ISEQUALF(RatioToFloat(&ratioc), 1.666666)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioFromFloat NOK");
    PBErrCatch(PBMathErr);
  }
  ratioc = RatioFromFloat(PBMATH_PI);
  printf("PI=");
  RatioPrintln(&ratioc, stdout);
  if (
    ratioc._base != 3 ||
    ratioc._numerator != 16 ||
    ratioc._denominator != 113 ||
    !ISEQUALF(RatioToFloat(&ratioc), PBMATH_PI)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "RatioFromFloat NOK");
    PBErrCatch(PBMathErr);
  }

  printf("UnitTestRatio OK\n");
}

void UnitTestLSLR() {

  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 2);
  VecSet(&dim, 1, 3);
  MatFloat* mat = MatFloatCreate(&dim);
  VecShort2D i = VecShortCreateStatic2D();
  float v[6] = {1.0, 1.0, 1.0, 1.0, 2.0, 3.0};
```

```
  int j = 0;
  do {
    MatSet(mat, &i, v[j]);
    ++j;
  } while(VecStep(&i, &dim));
  LeastSquareLinReg lslr = LeastSquareLinRegCreateStatic(mat);
  VecFloat* Y = VecFloatCreate(3);
  VecSet(Y, 0, 3.0);
  VecSet(Y, 1, 5.0);
  VecSet(Y, 2, 7.0);
  VecFloat* beta = LSLRSolve(&lslr, Y);
  VecFloat* check = VecFloatCreate(2);
  VecSet(check, 0, 1.0);
  VecSet(check, 1, 2.0);
  if (
    VecIsEqual(beta, check) == false ||
    !ISEQUALF(LSLRGetBias(&lslr), 0.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "LSLRSolve NOK (1)");
    PBErrCatch(PBMathErr);
  }
  VecFree(&beta);
  VecSet(Y, 0, 2.75);
  VecSet(Y, 1, 5.25);
  VecSet(Y, 2, 6.75);
  beta = LSLRSolve(&lslr, Y);
  VecSet(check, 0, 0.916666);
  VecSet(check, 1, 2.0);
  if (
    VecIsEqual(beta, check) == false ||
    !ISEQUALF(LSLRGetBias(&lslr), 0.408248)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "LSLRSolve NOK (2)");
    PBErrCatch(PBMathErr);
  }
  VecFree(&beta);
  VecFree(&check);
  VecFree(&Y);
  MatFree(&mat);
  LeastSquareLinRegFreeStatic(&lslr);
  printf("UnitTestLSLR OK\n");
}

void UnitTestQuaternion() {
  Quaternion quat = QuaternionCreateStatic();
  if (
    !ISEQUALF(VecGet(&(quat.val), 0), 0.0) ||
    !ISEQUALF(VecGet(&(quat.val), 1), 0.0) ||
    !ISEQUALF(VecGet(&(quat.val), 2), 0.0) ||
    !ISEQUALF(VecGet(&(quat.val), 3), 1.0)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "QuaternionCreateStatic NOK");
    PBErrCatch(PBMathErr);
  }
  float theta = ConvDeg2Rad(10.0);
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3);
  VecSet(&dim, 1, 3);
  VecShort2D pos = VecShortCreateStatic2D();
  MatFloat* rotMatX = MatFloatCreate(&dim);
  VecSet(&pos, 0, 0);
  VecSet(&pos, 1, 0);
```

```
MatSet(rotMatX, &pos, 1.0);
VecSet(&pos, 0, 1);
VecSet(&pos, 1, 1);
MatSet(rotMatX, &pos, cos(theta));
VecSet(&pos, 0, 2);
VecSet(&pos, 1, 1);
MatSet(rotMatX, &pos, -sin(theta));
VecSet(&pos, 0, 1);
VecSet(&pos, 1, 2);
MatSet(rotMatX, &pos, sin(theta));
VecSet(&pos, 0, 2);
VecSet(&pos, 1, 2);
MatSet(rotMatX, &pos, cos(theta));
quat = QuaternionCreateFromRotMat(rotMatX);
MatFloat* rotMat = QuaternionToRotMat(&quat);
if (!MatIsEqual(rotMat, rotMatX)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "QuaternionToRotMat NOK (1)");
  PBErrCatch(PBMathErr);
}
VecFloat3D axis = VecFloatCreateStatic3D();
VecSet(&axis, 0, 1.0);
Quaternion quatFromAxis =
  QuaternionCreateFromRotAxis((VecFloat*)(&axis), theta);
if (!QuaternionIsEqual(&quat, &quatFromAxis)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "QuaternionCreateFromRotAxis NOK (1)");
  // fails due to imprecision, deactivate for now
  //PBErrCatch(PBMathErr);
}
MatFree(&rotMat);
Quaternion addQuat = quat;

MatFloat* rotMatY = MatFloatCreate(&dim);
VecSet(&pos, 0, 1);
VecSet(&pos, 1, 1);
MatSet(rotMatY, &pos, 1.0);
VecSet(&pos, 0, 0);
VecSet(&pos, 1, 0);
MatSet(rotMatY, &pos, cos(theta));
VecSet(&pos, 0, 2);
VecSet(&pos, 1, 0);
MatSet(rotMatY, &pos, sin(theta));
VecSet(&pos, 0, 0);
VecSet(&pos, 1, 2);
MatSet(rotMatY, &pos, -sin(theta));
VecSet(&pos, 0, 2);
VecSet(&pos, 1, 2);
MatSet(rotMatY, &pos, cos(theta));
quat = QuaternionCreateFromRotMat(rotMatY);
rotMat = QuaternionToRotMat(&quat);
if (!MatIsEqual(rotMat, rotMatY)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "QuaternionToRotMat NOK (2)");
  PBErrCatch(PBMathErr);
}
VecSet(&axis, 0, 0.0);
VecSet(&axis, 1, 1.0);
quatFromAxis =
  QuaternionCreateFromRotAxis((VecFloat*)(&axis), theta);
if (!QuaternionIsEqual(&quat, &quatFromAxis)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
```

```
  sprintf(PBMathErr->_msg, "QuaternionCreateFromRotAxis NOK (2)");
  PBErrCatch(PBMathErr);
}
MatFree(&rotMat);
Quaternion addQuatXY = QuaternionGetComposition(&addQuat, &quat);

MatFloat* rotMatZ = MatFloatCreate(&dim);
VecSet(&pos, 0, 2);
VecSet(&pos, 1, 2);
MatSet(rotMatZ, &pos, 1.0);
VecSet(&pos, 0, 0);
VecSet(&pos, 1, 0);
MatSet(rotMatZ, &pos, cos(theta));
VecSet(&pos, 0, 1);
VecSet(&pos, 1, 0);
MatSet(rotMatZ, &pos, -sin(theta));
VecSet(&pos, 0, 0);
VecSet(&pos, 1, 1);
MatSet(rotMatZ, &pos, sin(theta));
VecSet(&pos, 0, 1);
VecSet(&pos, 1, 1);
MatSet(rotMatZ, &pos, cos(theta));
quat = QuaternionCreateFromRotMat(rotMatZ);
rotMat = QuaternionToRotMat(&quat);
if (!MatIsEqual(rotMat, rotMatZ)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "QuaternionToRotMat NOK (3)");
  //PBErrCatch(PBMathErr);
}
VecSet(&axis, 1, 0.0);
VecSet(&axis, 2, 1.0);
quatFromAxis =
  QuaternionCreateFromRotAxis((VecFloat*)(&axis), theta);
if (!QuaternionIsEqual(&quat, &quatFromAxis)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "QuaternionCreateFromRotAxis NOK (3)");
  // fails due to imprecision, deactivate for now
  //PBErrCatch(PBMathErr);
}
MatFree(&rotMat);

Quaternion addQuatXYZ = QuaternionGetComposition(&addQuatXY, &quat);
MatFloat* sumRotMatXY = MatGetProdMat(rotMatX, rotMatY);
MatFloat* sumRotMatXYZ = MatGetProdMat(sumRotMatXY, rotMatZ);
rotMat = QuaternionToRotMat(&addQuatXYZ);
if (!MatIsEqual(rotMat, sumRotMatXYZ)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "QuaternionGetComposition NOK");
  PBErrCatch(PBMathErr);
}
MatFree(&rotMat);

Quaternion diffQuat = QuaternionGetDifference(&addQuatXY, &addQuatXYZ);
Quaternion checkDiffQuat = QuaternionGetComposition(&diffQuat, &addQuatXY);
if (!QuaternionIsEqual(&addQuatXYZ, &checkDiffQuat)) {
  PBMathErr->_type = PBErrTypeUnitTestFailed;
  sprintf(PBMathErr->_msg, "QuaternionGetDifference NOK");
  PBErrCatch(PBMathErr);
}

VecFloat3D v = VecFloatCreateStatic3D();
VecSet(&v, 0, 1.0);
```

```
  VecSet(&v, 1, 1.0);
  VecSet(&v, 2, 1.0);

  VecFloat* vRot = MatGetProdVec(sumRotMatXYZ, &v);
  QuaternionApply(&addQuatXYZ, (VecFloat*)(&v));
  if (!VecIsEqual(vRot, &v)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg, "QuaternionApply NOK");
    PBErrCatch(PBMathErr);
  }

  float phi = QuaternionGetRotAngle(&addQuatXYZ);
  VecFloat3D rotAxis = QuaternionGetRotAxis(&addQuatXYZ);
  Quaternion quatB = QuaternionCreateFromRotAxis((VecFloat*)(&rotAxis), phi);
  if (!QuaternionIsEqual(&addQuatXYZ, &quatB)) {
    PBMathErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBMathErr->_msg,
      "QuaternionGetRotAngle or QuaternionGetRotAxis NOK");
    PBErrCatch(PBMathErr);
  }

  VecFree(&vRot);
  QuaternionFreeStatic(&quat);
  MatFree(&rotMatX);
  MatFree(&rotMatY);
  MatFree(&rotMatZ);
  MatFree(&sumRotMatXY);
  MatFree(&sumRotMatXYZ);
  printf("UnitTestQuaternion OK\n");
}

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

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

6 Unit tests output

```
{
  "_dim":"5",
  "_val":["1","2","3","4","5"]
}
```

```
{
  "_dim":"2",
  "_val":["1","2"]
}
{
  "_dim":"3",
  "_val":["1","2","3"]
}
{
  "_dim":"4",
  "_val":["1","2","3","4"]
}
{
  "_dim":"5",
  "_val":["1","2","3","4","5"]
}
{
  "_dim":"2",
  "_val":["1","2"]
}
{
  "_dim":"3",
  "_val":["1","2","3"]
}
{
  "_dim":"4",
  "_val":["1","2","3","4"]
}
{
  "_dim":"5",
  "_val":["1.000000","2.000000","3.000000","4.000000","5.000000"]
}
{
  "_dim":"2",
  "_val":["1.000000","2.000000"]
}
{
  "_dim":"3",
  "_val":["1.000000","2.000000","3.000000"]
}
{
  "_nbRow":"2",
  "_nbCol":"3",
  "_val":["1.000000","2.000000","3.000000","4.000000","5.000000","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: 0.000054ms, array: 0.000014ms
VecShort2D: 0.000006ms, array: 0.000007ms
VecShort3D: 0.000006ms, array: 0.000006ms
VecShort4D: 0.000007ms, array: 0.000007ms
```

```
UnitTestSpeedVecShort OK
[1.000,2.000,3.000,4.000,5.000]
[1.000,2.000]
[1.000,2.000,3.000]
UnitTestVecShortToFloat OK
[1.000,2.000,3.000,4.000,5.000]
[1.000,2.000]
[1.000,2.000,3.000]
UnitTestVecLongToFloat OK
UnitTestVecShortOp OK
UnitTestVecShortShiftStep OK
UnitTestVecShortGetMinMax OK
UnitTestVecShortHadamardProd OK
UnitTestVecShort OK
[0,0,0,0,0]
[0,0]
[0,0,0]
[0,0,0,0]
VecLongCreateFree OK
_VecLongClone OK
_VecLongLoadSave OK
_VecLongGetSetDim OK
UnitTestVecLongStep OK
UnitTestVecLongHamiltonDist OK
UnitTestVecLongIsEqual OK
UnitTestVecLongDotProd OK
UnitTestVecLongCopy OK
VecLong: 0.000014ms, array: 0.000014ms
VecLong2D: 0.000007ms, array: 0.000006ms
VecLong3D: 0.000006ms, array: 0.000007ms
VecLong4D: 0.000006ms, array: 0.000007ms
UnitTestSpeedVecLong OK
UnitTestVecLongOp OK
UnitTestVecLongShiftStep OK
UnitTestVecLongGetMinMax OK
UnitTestVecLongHadamardProd OK
UnitTestVecLongGetNewDim OK
UnitTestVecLong 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
UnitTestVecFloatCrossProd OK
[0.707107,0.707107]
UnitTestVecFloatAngleTo OK
[1,2,3,4,5]
[1,2]
[1,2,3]
UnitTestVecFloatToShort OK
UnitTestVecFloatGetMinMax OK
UnitTestVecFloatRotAxis OK
UnitTestVecFloatGetNewDim OK
```

```
UnitTestVecFloatHadamardProd OK
VecFloat: 0.000015ms, array: 0.000015ms
VecFloat2D: 0.000006ms, array: 0.000007ms
VecFloat3D: 0.000006ms, array: 0.000006ms
UnitTestSpeedVecFloat OK
UnitTestVecFloat OK
UnitTestMatFloatCreateFree OK
UnitTestMatFloatGetSetDim OK
UnitTestMatFloatCloneIsEqual OK
UnitTestMatFloatLoadSave OK
UnitTestMatFloatInv OK
UnitTestMatFloatTransposeScale OK
UnitTestMatFloatProdVecFloat OK
UnitTestMatFloatProdMatFloat OK
mat:
[-1.000, -1.000,  1.000
  1.000,  3.000,  3.000
 -1.000, -1.000,  5.000
  1.000,  3.000,  7.000]
Q:
[-0.500, -0.500,  0.500
  0.500, -0.500,  0.500
 -0.500, -0.500, -0.500
  0.500, -0.500, -0.500]
R:
[ 2.000,  4.000,  2.000
 -0.000, -2.000, -8.000
  0.000, -0.000, -4.000]
QR:
[-1.000, -1.000,  1.000
  1.000,  3.000,  3.000
 -1.000, -1.000,  5.000
  1.000,  3.000,  7.000]
UnitTestMatFloatGetQR OK
UnitTestMatFloatProdVecVecTranspose OK
mat:
[ 2.920,  0.860, -1.150
  0.860,  6.510,  3.320
 -1.150,  3.320,  4.570]
Eigen values: [8.999,3.997,1.005]
Eigen vector 1: [0.000,-0.800,-0.600]
Eigen vector 2: [0.800,0.360,-0.480]
Eigen vector 3: [0.600,-0.480,0.640]
UnitTestMatFloatGetEigenValues OK
MatFloat: 0.000003ms, array: 0.000002ms
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: 0.000013ms, pow: 0.000057ms
speed fastpow OK
fsquare OK
UnitTestConv OK
UnitTestThueMorseSeq OK
UnitTestGetAreaTriangleHero OK
UnitTestGetFibonacciSeq OK
UnitTestGetFibonacciLattice OK
```

```
UnitTestGetGCD OK
UnitTestBasicFunctions OK
0+10/6 -> 1+2/3
1.666666=1+2/3
PI=3+16/113
UnitTestRatio OK
UnitTestLSLR OK
UnitTestAll OK
```

7 Examples

UnitTestVecShortLoadSave.txt:

```
{
  "_dim":"5",
  "_val":["1","2","3","4","5"]
}
{
  "_dim":"2",
  "_val":["1","2"]
}
{
  "_dim":"3",
  "_val":["1","2","3"]
}
{
  "_dim":"4",
  "_val":["1","2","3","4"]
}
```

UnitTestVecLongLoadSave.txt:

```
{
  "_dim":"5",
  "_val":["1","2","3","4","5"]
}
{
  "_dim":"2",
  "_val":["1","2"]
}
{
  "_dim":"3",
  "_val":["1","2","3"]
}
{
  "_dim":"4",
  "_val":["1","2","3","4"]
}
```

UnitTestVecFloatLoadSave.txt:

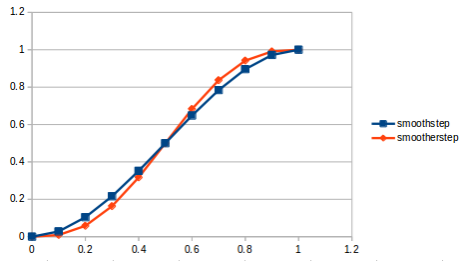
```
{
  "_dim":"5",
  "_val":["1.000000","2.000000","3.000000","4.000000","5.000000"]
}
{
```

```
  "_dim":"2",
  "_val":["1.000000","2.000000"]
}
{
  "_dim":"3",
  "_val":["1.000000","2.000000","3.000000"]
}
```

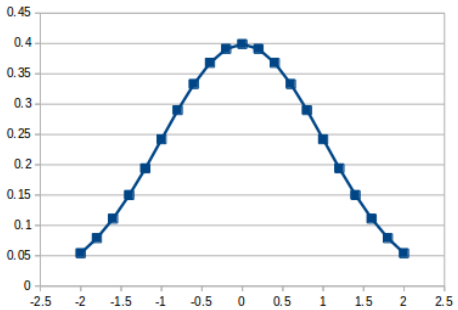
matfloat.txt:

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

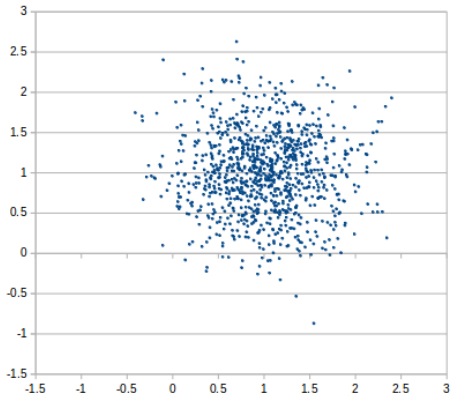
smoother functions:



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



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


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