# PBMath

P. Baillehache

November 22, 2017

## Contents

**1 Definitions** **2**
1.1 Vectors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.1.1 Distance between two vectors . . . . . . . . . . . . . . . 2
1.1.2 Angle between two vectors . . . . . . . . . . . . . . . . . 2

**2 Interface** **3**

**3 Code** **13**

**4 Makefile** **37**

**5 Usage** **38**

## Introduction

PBMath is C library providing mathematical structures and functions.

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

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

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

The `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 linear equation systems.

# 1 Definitions

## 1.1 Vectors

### 1.1.1 Distance between two vectors

For `VecShort`:

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

For `VecFloat`:

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

### 1.1.2 Angle between two vectors

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

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

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

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

$$\left\{ \begin{array}{l} W_x = M_a V_x + M_b V_y \\ W_y = M_c V_x + M_d V_y \end{array} \right. \tag{4}$$

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

# 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>

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

#define PBMATH_EPSILON 0.0000001
#define PBMATH_TWOPI 6.28319
#define PBMATH_PI 3.14159
#define PBMATH_HALFPI 1.57080
#define PBMATH_QUARTERPI 0.78540
#define PBMATH_SQRTTWO 1.41421
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))
#define EQUALF(a,b) (fabs(a-b)<PBMATH_EPSILON)
#define SHORT(a) ((short)(round(a)))

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

void VecTypeUnsupported(void*t, ...);
#define VecClone(V) _Generic((V), \
  VecFloat*: VecFloatClone, \
```

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

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

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

void LinSysTypeUnsupported(void*t, ...);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// Return the dot product of 'that' and 'tho'
// Return NAN if arguments are invalid
short VecShortDotProd(VecShort *that, VecShort *tho);

// Set all values of the vector 'that' to 0
// Do nothing if arguments are invalid
void VecSetNull(VecShort *that);

// 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 upper limit for each value is given by 'bound'
// Return false if arguments are invalid or
// all values of 'that' have reach there upper limit (in which case
// 'that''s values are all set back to 0
// Return true else
bool VecStep(VecShort *that, VecShort *bound);

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

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

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

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

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

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

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

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

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

// Print the VecFloat on 'stream' with 'prec' digit precision
// Do nothing if arguments are invalid
void VecFloatPrint(VecFloat *that, FILE *stream, int prec);
void VecFloatPrintDef(VecFloat *that, FILE *stream);

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

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

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

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

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

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

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

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

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

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

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

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

// Rotate CCW 'that' by 'theta' radians and store the result in 'that'
// Do nothing if arguments are invalid
void VecFloatRot2D(VecFloat *that, float theta);

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

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

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

// Return the conversion of VecFloat 'that' to a VecShort using round()
// Return null if arguments are invalid or couldn't create the result
VecShort* VecFloatToShort(VecFloat *that);

// Return the conversion of VecShort 'that' to a VecFloat
// Return null if arguments are invalid or couldn't create the result
VecFloat* VecShortToFloat(VecShort *that);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// Create a new EqLinSys with matrix 'm' and vector 'v'
// The dimension of 'v' must be equal to the number of column of 'm'
// 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 EqLinSys
EqLinSys* EqLinSysCreate(MatFloat *m, VecFloat *v);

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

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

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

// Set the matrix of the EqLinSys to a clone of 'm'
// Do nothing if arguments are invalid
void EqLinSysSetM(EqLinSys *that, MatFloat *m);

// Set the vector of the EqLinSys to a clone of 'v'
// Do nothing if arguments are invalid
void EqLinSysSetV(EqLinSys *that, VecFloat *v);

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

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

// Return x^y when x and y are int
// to avoid numerical imprecision from (pow(double,double)
// From https://stackoverflow.com/questions/29787310/
// does-pow-work-for-int-data-type-in-c
```

```
int powi(int base, int exp);

#endif
```

# 3 Code

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

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

#include "pbmath.h"

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

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

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

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

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

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

// Clone the VecShort
// Return NULL if we couldn't clone the VecShort
VecShort* VecShortClone(VecShort *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Create a clone
  VecShort *clone = VecShortCreate(that->_dim);
```

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

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

// Save the VecShort to the stream
// Return 0 upon success, or:
// 1: invalid arguments
// 2: fprintf error
int VecShortSave(VecShort *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // Save the dimension
  int ret = fprintf(stream, "%d ", that->_dim);
  // If we coudln't fprintf
```

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

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

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

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

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

```
    return;
  // Set the value
  that->_val[i] = v;
}

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

// Return the Hamiltonian distance between the VecShort 'that' and 'tho'
// Return -1 if arguments are invalid
// If dimensions are different, missing ones are considered to
// be equal to 0
short VecShortHamiltonDist(VecShort *that, VecShort *tho) {
  // Check argument
  if (that == NULL || tho == NULL)
    return -1;
  // Declare a variable to calculate the distance
  short ret = 0;
  for (int iDim = that->_dim; iDim--;) {
    short v = VecGet(that, iDim) - VecGet(tho, iDim);
    if (v < 0)
      v *= -1;
    ret += v;
  }
  // Return the distance
  return ret;
}


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

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

```
}

// Return the dot product of 'that' and 'tho'
// Return 0 if arguments are invalid
short VecShortDotProd(VecShort *that, VecShort *tho) {
  // Check argument
  if (that == NULL || tho == NULL)
    return 0;
  // Declare a variable ot memorise the result
  short res = 0;
  // For each component
  for (int iDim = that->_dim; iDim--;)
    // Calculate the product
    res += VecGet(that, iDim) * VecGet(tho, iDim);
  // Return the result
  return res;
}

// Set all values of the vector 'that' to 0
// Do nothing if arguments are invalid
void VecSetNull(VecShort *that) {
  // Check arguments
  if (that == NULL)
    return;
  // Set values
  for (int iDim = that->_dim; iDim--;)
    that->_val[iDim] = 0;
}

// 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 upper limit for each value is given by 'bound'
// Return false if arguments are invalid or
// all values of 'that' have reach there upper limit (in which case
// 'that''s values are all set back to 0
// Return true else
bool VecStep(VecShort *that, VecShort *bound) {
  // Check arguments
  if (that == NULL || bound == NULL || that->_dim != bound->_dim)
    return false;
  // Declare a variable for the returned flag
  bool ret = true;
  // Declare a variable to memorise the dimension currently increasing
  int iDim = that->_dim - 1;
  // Increment
  bool flag = true;
  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;
}
```

```
// ------------- VecFloat

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

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

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

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

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

```
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // If 'that' is already allocated
  if (*that != NULL) {
    // Free memory
    VecFloatFree(that);
  }
  // Read the number of dimension
  int dim;
  int ret = fscanf(stream, "%d", &dim);
  // If we coudln't fscanf
  if (ret == EOF)
    return 4;
  if (dim <= 0)
    return 3;
  // Allocate memory
  *that = VecFloatCreate(dim);
  // If we coudln't allocate memory
  if (*that == NULL) {
    return 2;
  }
  // Read the values
  for (int i = 0; i < dim; ++i) {
    ret = fscanf(stream, "%f", (*that)->_val + i);
    // If we coudln't fscanf
    if (ret == EOF)
      return 4;
  }
  // Return success code
  return 0;
}

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

// Free the memory used by a VecFloat
```

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

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

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

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

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

```
  return that->_dim;
}

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

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

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

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

// Return the Hamiltonian distance between the VecFloat 'that' and 'tho'
// Return NaN if arguments are invalid
// If dimensions are different, missing ones are considered to
// be equal to 0.0
```

```
float VecFloatHamiltonDist(VecFloat *that, VecFloat *tho) {
  // Check argument
  if (that == NULL || tho == NULL)
    return NAN;
  // Declare a variable to calculate the distance
  float ret = 0.0;
  for (int iDim = that->_dim; iDim--;)
    ret += fabs(VecGet(that, iDim) - VecGet(tho, iDim));
  // Return the distance
  return ret;
}

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

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

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

```
  }
}

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

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

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

```
  return res;
}

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

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

```
    theta *= -1.0;
  // Free memory
  VecFree(&v);
  VecFree(&w);
  // Return the result
  return theta;
}

// Return the conversion of VecFloat 'that' to a VecShort using round()
// Return null if arguments are invalid or couldn't create the result
VecShort* VecFloatToShort(VecFloat *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Create the result
  VecShort *res = VecShortCreate(that->_dim);
  if (res != NULL) {
    for (int iDim = that->_dim; iDim--;)
      VecSet(res, iDim, SHORT(VecGet(that, iDim)));
  }
  // Return the result
  return res;
}

// Return the conversion of VecShort 'that' to a VecFloat
// Return null if arguments are invalid or couldn't create the result
VecFloat* VecShortToFloat(VecShort *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Create the result
  VecFloat *res = VecFloatCreate(that->_dim);
  if (res != NULL) {
    for (int iDim = that->_dim; iDim--;)
      VecSet(res, iDim, (float)VecGet(that, iDim));
  }
  // Return the result
  return res;
}

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

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

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

// Create a new MatFloat of dimension 'dim' (nbcol, nbline)
// Values are initalized to 0.0, 'dim' must be a VecShort of dimension 2
// Return NULL if we couldn't create the MatFloat
MatFloat* MatFloatCreate(VecShort *dim) {
  // Check argument
  if (dim == NULL || VecDim(dim) != 2)
    return NULL;
  // Allocate memory
  MatFloat *that = (MatFloat*)malloc(sizeof(MatFloat));
  //If we could allocate memory
  if (that != NULL) {
    // Set the dimension
    that->_dim = VecClone(dim);
    if (that->_dim == NULL) {
      // Free memory
      free(that);
```

```
      // Stop here
      return NULL;
    }
    // Allocate memory for values
    int d = VecGet(dim, 0) * VecGet(dim, 1);
    that->_val = (float*)malloc(sizeof(float) * d);
    // If we couldn't allocate memory
    if (that->_val == NULL) {
      // Free memory
      free(that);
      // Stop here
      return NULL;
    }
    // Set the default values
    for (int i = d; i--;)
      that->_val[i] = 0.0;
  }
  // Return the new MatFloat
  return that;
}

// Set the MatFloat to the identity matrix
// The matrix must be a square matrix
// Do nothing if arguments are invalid
void MatFloatSetIdentity(MatFloat *that) {
  // Check argument
  if (that == NULL || VecGet(that->_dim, 0) != VecGet(that->_dim, 1))
    return;
  // Set the values
  VecShort *i = VecShortCreate(2);
  if (i != NULL) {
    for (VecSet(i, 0, 0); VecGet(i, 0) < VecGet(that->_dim, 0);
      VecSet(i, 0, VecGet(i, 0) + 1)) {
      for (VecSet(i, 1, 0); VecGet(i, 1) < VecGet(that->_dim, 1);
        VecSet(i, 1, VecGet(i, 1) + 1)) {
        if (VecGet(i, 0) == VecGet(i, 1))
          MatSet(that, i, 1.0);
        else
          MatSet(that, i, 0.0);
      }
    }
  }
  VecFree(&i);
}

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

```
  return clone;
}

// Copy the values of 'w' in 'that' (must have same dimensions)
// Do nothing if arguments are invalid
void MatFloatCopy(MatFloat *that, MatFloat *w) {
  // Check argument
  if (that == NULL || w == NULL)
    return;
  // Copy the matrix values
  int d = VecGet(that->_dim, 0) * VecGet(that->_dim, 1);
  for (int i = d; i--;)
    that->_val[i] = w->_val[i];
}

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

// Save the MatFloat to the stream
```

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

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

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

```
      fprintf(stream, ",");
    }
    if (i < VecGet(that->_dim, 1) - 1)
      fprintf(stream, "\n");
  }
  fprintf(stream, "]");
}
void MatFloatPrintDef(MatFloat *that, FILE *stream) {
  MatFloatPrint(that, stream, 3);
}

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

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

// Return a VecShort of dimension 2 containing the dimension of
// the MatFloat
// Return NULL if arguments are invalid
VecShort* MatFloatDim(MatFloat *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Return the dimension
  return VecClone(that->_dim);
}

// Return the inverse matrix of 'that'
// The matrix must be a square matrix
// Return null if arguments are invalids
MatFloat* MatFloatInv(MatFloat *that) {
  // Check arguments
  if (that == NULL || VecGet(that->_dim, 0) != VecGet(that->_dim, 1))
    return NULL;
  // Allocate memory for the pivot
  VecShort *pivot = VecShortCreate(2);
  if (pivot == NULL)
    return NULL;
  // Allocate memory for the result
  MatFloat *res = MatFloatCreate(that->_dim);
```

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

```
  }
  // Divide the values by the pivot
  float p = MatGet(copy, pivot);
  VecSet(pivot, 0, 0);
  while (VecGet(pivot, 0) < VecGet(that->_dim, 0)) {
    MatSet(copy, pivot, MatGet(copy, pivot) / p);
    MatSet(res, pivot, MatGet(res, pivot) / p);
    VecSet(pivot, 0, VecGet(pivot, 0) + 1);
  }
  // Reposition the pivot
  VecSet(pivot, 0, VecGet(pivot, 1));
  // Nullify the values below the pivot
  VecSet(pivot, 0, 0);
  VecSet(index, 1, VecGet(pivot, 1) + 1);
  while (VecGet(index, 1) < VecGet(that->_dim, 1)) {
    VecSet(index, 0, VecGet(pivot, 1));
    p = MatGet(copy, index);
    VecSet(index, 0, 0);
    while (VecGet(index, 0) < VecGet(that->_dim, 0)) {
      MatSet(copy, index,
        MatGet(copy, index) - MatGet(copy, pivot) * p);
      MatSet(res, index,
        MatGet(res, index) - MatGet(res, pivot) * p);
      VecSet(pivot, 0, VecGet(pivot, 0) + 1);
      VecSet(index, 0, VecGet(index, 0) + 1);
    }
    VecSet(pivot, 0, 0);
    VecSet(index, 0, 0);
    VecSet(index, 1, VecGet(index, 1) + 1);
  }
  // Reposition the pivot
  VecSet(pivot, 0, VecGet(pivot, 1));
}
// Now the matrix is triangular, move back through the pivots
// to make it diagonal
for (; VecGet(pivot, 0) >= 0;
  VecSet(pivot, 0, VecGet(pivot, 0) - 1),
  VecSet(pivot, 1, VecGet(pivot, 1) - 1)) {
  // Nullify the values above the pivot by substracting the line
  // of the pivot
  VecSet(pivot, 0, 0);
  VecSet(index, 1, VecGet(pivot, 1) - 1);
  while (VecGet(index, 1) >= 0) {
    VecSet(index, 0, VecGet(pivot, 1));
    float p = MatGet(copy, index);
    VecSet(index, 0, 0);
    while (VecGet(index, 0) < VecGet(that->_dim, 0)) {
      MatSet(copy, index,
        MatGet(copy, index) - MatGet(copy, pivot) * p);
      MatSet(res, index,
        MatGet(res, index) - MatGet(res, pivot) * p);
      VecSet(pivot, 0, VecGet(pivot, 0) + 1);
      VecSet(index, 0, VecGet(index, 0) + 1);
    }
    VecSet(pivot, 0, 0);
    VecSet(index, 0, 0);
    VecSet(index, 1, VecGet(index, 1) - 1);
  }
  // Reposition the pivot
  VecSet(pivot, 0, VecGet(pivot, 1));
}
// Free memory
```

```
      MatFree(&copy);
      VecFree(&index);
    }
  }
  // Free memory
  VecShortFree(&pivot);
  // Return the result
  return res;
}

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

// Return the product of matrix 'that' by matrix 'tho'
// Number of columns of 'that' must equal number of line of 'tho'
// Return null if arguments are invalids
MatFloat* MatFloatProdMatFloat(MatFloat *that, MatFloat *tho) {
  // Check arguments
  if (that == NULL || tho == NULL ||
    VecGet(that->_dim, 0) != VecGet(tho->_dim, 1))
    return NULL;
  // Declare 3 variables to memorize the index in the matrix
  VecShort *i = VecShortCreate(2);
  if (i == NULL)
    return NULL;
  VecShort *j = VecShortCreate(2);
  if (j == NULL) {
    VecFree(&i);
    return NULL;
  }
  VecShort *k = VecShortCreate(2);
  if (k == NULL) {
    VecFree(&i);
    VecFree(&j);
    return NULL;
```

```
  }
  // Allocate memory for the solution
  VecSet(i, 0, VecGet(tho->_dim, 0));
  VecSet(i, 1, VecGet(that->_dim, 1));
  MatFloat *ret = MatFloatCreate(i);
  // If we could allocate memory
  if (ret != NULL) {
    for (VecSet(i, 0, 0); VecGet(i, 0) < VecGet(tho->_dim, 0);
      VecSet(i, 0, VecGet(i, 0) + 1)) {
      for (VecSet(i, 1, 0); VecGet(i, 1) < VecGet(that->_dim, 1);
        VecSet(i, 1, VecGet(i, 1) + 1)) {
        for (VecSet(j, 0, 0), VecSet(j, 1, VecGet(i, 1)),
          VecSet(k, 0, VecGet(i, 0)), VecSet(k, 1, 0);
          VecGet(j, 0) < VecGet(that->_dim, 0);
          VecSet(j, 0, VecGet(j, 0) + 1),
          VecSet(k, 1, VecGet(k, 1) + 1)) {
          MatSet(ret, i, MatGet(ret, i) +
            MatGet(that, j) * MatGet(tho, k));
        }
      }
    }
  }
  // Free memory
  VecFree(&i);
  VecFree(&j);
  VecFree(&k);
  // Return the result
  return ret;
}

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

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

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

// Create a new Gauss of mean 'mean' and sigma 'sigma'
// Return NULL if we couldn't create the Gauss
Gauss* GaussCreate(float mean, float sigma) {
  // Allocate memory
  Gauss *that = (Gauss*)malloc(sizeof(Gauss));
  // If we could allocate memory
  if (that != NULL) {
    // Set properties
    that->_mean = mean;
    that->_sigma = sigma;
  }
  // REturn the new Gauss
  return that;
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// Create a new EqLinSys with matrix 'm' and vector 'v'
// The dimension of 'v' must be equal to the number of column of 'm'
// 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 EqLinSys
EqLinSys* EqLinSysCreate(MatFloat *m, VecFloat *v) {
  // Check arguments
  if (m == NULL || VecGet(m->_dim, 0) != VecGet(m->_dim, 1))
    return NULL;
  if (v != NULL && VecGet(m->_dim, 0) != VecDim(v))
    return NULL;
  // Allocate memory
  EqLinSys *that = (EqLinSys*)malloc(sizeof(EqLinSys));
  // If we could allocate memory
  if (that != NULL) {
    that->_M = MatClone(m);
    that->_Minv = MatInv(that->_M);
    if (v != NULL)
      that->_V = VecClone(v);
    else
      that->_V = VecFloatCreate(VecGet(m->_dim, 0));
    if (that->_M == NULL || that->_V == NULL || that->_Minv == NULL)
      EqLinSysFree(&that);
  }
  // Return the new EqLinSys
  return that;
}

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

```
  *that = NULL;
}

// Clone the EqLinSys 'that'
// Return NULL if we couldn't clone the EqLinSys
EqLinSys* EqLinSysClone(EqLinSys * that) {
  // Check arguments
  if (that == NULL)
    return NULL;
  // Declare a variable for the result
  EqLinSys *ret = (EqLinSys*)malloc(sizeof(EqLinSys));
  // If we could allocate memory
  if (ret != NULL) {
    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)
      EqLinSysFree(&ret);
  }
  // Return the new EqLinSys
  return ret;
}

// Solve the EqLinSys _M.x = _V
// Return the solution vector, or null if there is no solution or the
// arguments are invalid
VecFloat* EqLinSysSolve(EqLinSys *that) {
  // Check the argument
  if (that == NULL)
    return NULL;
  // Declare a variable to memorize the solution
  VecFloat *ret = NULL;
  // Calculate the solution
  ret = MatProd(that->_Minv, that->_V);
  // Return the solution vector
  return ret;
}

// Set the matrix of the EqLinSys to a copy of 'm'
// 'm' must have same dimensions has the current matrix
// Do nothing if arguments are invalid
void EqLinSysSetM(EqLinSys *that, MatFloat *m) {
  // Check the arguments
  if (that == NULL || m == NULL ||
    VecIsEqual(m->_dim, that->_M->_dim) == false)
    return;
  // Update the matrix values
  MatCopy(that->_M, m);
  // Update the inverse matrix
  MatFloat *inv = MatInv(that->_M);
  if (inv != NULL) {
    MatCopy(that->_Minv, inv);
    MatFree(&inv);
  }
}

// Set the vector of the EqLinSys to a copy of 'v'
// 'v' must have same dimensions has the current vector
// Do nothing if arguments are invalid
void EqLinSysSetV(EqLinSys *that, VecFloat *v) {
  // Check the arguments
  if (that == NULL || v == NULL || v->_dim != that->_V->_dim)
```

```
    return;
  // Update the vector values
  VecCopy(that->_V, v);
}

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

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

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

# 4 Makefile

```
OPTIONS_DEBUG=-ggdb -g3 -Wall
OPTIONS_RELEASE=-O3
OPTIONS=$(OPTIONS_RELEASE)
INCPATH=/home/bayashi/Coding/Include
LIBPATH=/home/bayashi/Coding/Include

all : main

main: main.o pbmath.o Makefile
gcc $(OPTIONS) main.o pbmath.o -o main -lm

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

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

clean :
rm -rf *.o main

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

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

# 5 Usage

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

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

  // ------------- VecShort
  fprintf(stdout, "-------- VecShort\n");
  // Create a vector of dimension 3
  VecShort *a = VecShortCreate(3);
  // If we couldn't create the vector
  if (a == NULL) {
    fprintf(stderr, "VecCreate failed\n");
    return 1;
  }
  // Print the vector
  fprintf(stdout, "a: ");
  VecPrint(a, stdout);
  fprintf(stdout, "\n");
  // Set the 2nd value to 1
  VecSet(a, 1, 1);
  // Print the vector
  fprintf(stdout, "a: ");
  VecPrint(a, stdout);
  fprintf(stdout, "\n");
  // Clone the vector
  VecShort *cloneShort = VecClone(a);
  if (cloneShort == NULL) {
    fprintf(stderr, "VecClone failed\n");
    return 2;
  }
  // Print the vector
  fprintf(stdout, "cloneShort: ");
  VecPrint(cloneShort, stdout);
  fprintf(stdout, "\n");
  VecFree(&cloneShort);
  // Save the vector
  FILE *f = fopen("./vecshort.txt", "w");
  if (f == NULL) {
    fprintf(stderr, "fopen failed\n");
    return 3;
  }
  int ret = VecSave(a, f);
  if (ret != 0) {
    fprintf(stderr, "VecSave failed (%d)\n", ret);
    return 4;
  }
  fclose(f);
  // Load the vector
  f = fopen("./vecshort.txt", "r");
  if (f == NULL) {
    fprintf(stderr, "fopen failed\n");
    return 5;
  }
  VecShort *b = NULL;
```

```
ret = VecLoad(&b, f);
if (ret != 0) {
  fprintf(stderr, "VecLoad failed (%d)\n", ret);
  return 6;
}
fclose(f);
// Get the dimension and values of the loaded vector
fprintf(stdout, "b: %d ", VecDim(b));
for (int i = 0; i < VecDim(b); ++i)
  fprintf(stdout, "%d ", VecGet(b, i));
fprintf(stdout, "\n");
// Change the values of the loaded vector and print it
VecSet(b, 0, 2);
VecSet(b, 2, 3);
fprintf(stdout, "b: ");
VecPrint(b, stdout);
fprintf(stdout, "\n");
// VecProd
short prod = VecDotProd(a, b);
fprintf(stdout,"VecProd(a,b): %d\n",prod);
// Copy the loaded vector into the first one and print the first one
VecCopy(a, b);
fprintf(stdout, "a: ");
VecPrint(a, stdout);
fprintf(stdout, "\n");
// Reset a
fprintf(stdout,"Reset a:");
VecSetNull(a);
VecPrint(a, stdout);
fprintf(stdout, "\n");
// step a up to b
fprintf(stdout,"Step a up to b:\n");
do {
  VecPrint(a, stdout);
  fprintf(stdout, "\n");
} while(VecStep(a, b));
// Free memory
VecFree(&a);
VecFree(&b);

// -------------- VecFloat
fprintf(stdout, "-------- VecFloat\n");
// Create a vector of dimension 3
VecFloat *v = VecFloatCreate(3);
// If we couldn't create the vector
if (v == NULL) {
  fprintf(stderr, "VecCreate failed\n");
  return 7;
}
// Print the vector
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
// Set the 2nd value to 1.0
VecSet(v, 1, 1.0);
// Print the vector
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
// Clone the vector
VecFloat *cloneFloat = VecClone(v);
if (cloneFloat == NULL) {
```

```
  fprintf(stderr, "VecClone failed\n");
  return 8;
}
// Print the vector
fprintf(stdout, "cloneFloat: ");
VecPrint(cloneFloat, stdout);
fprintf(stdout, "\n");
VecFree(&cloneFloat);
// Save the vector
f = fopen("./vecfloat.txt", "w");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 9;
}
ret = VecSave(v, f);
if (ret != 0) {
  fprintf(stderr, "VecSave failed (%d)\n", ret);
  return 10;
}
fclose(f);
// Load the vector
f = fopen("./vecfloat.txt", "r");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 11;
}
VecFloat *w = NULL;
ret = VecLoad(&w, f);
if (ret != 0) {
  fprintf(stderr, "VecLoad failed (%d)\n", ret);
  return 12;
}
fclose(f);
// Get the dimension and values of the loaded vector
fprintf(stdout, "w: %d ", VecDim(w));
for (int i = 0; i < VecDim(w); ++i)
  fprintf(stdout, "%f ", VecGet(w, i));
fprintf(stdout, "\n");
// Change the values of the loaded vector and print it
VecSet(w, 0, 2.0);
VecSet(w, 2, 3.0);
fprintf(stdout, "w: ");
VecPrint(w, stdout);
fprintf(stdout, "\n");
// Copy the loaded vector into the first one and print the first one
VecCopy(v, w);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
// Get the norm
float norm = VecNorm(v);
fprintf(stdout, "Norm of v: %.3f\n", norm);
// Normalise
VecNormalise(v);
fprintf(stdout, "Normalized v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
// Distance between v and w
fprintf(stdout, "Distance between v and w: %.3f\n", VecDist(v, w));
fprintf(stdout, "Hamiltonian distance between v and w: %.3f\n",
  VecHamiltonDist(v, w));
fprintf(stdout, "Pixel distance between v and w: %.3f\n",
```

```
  VecPixelDist(v, w));
// Equality
if (VecIsEqual(v, w) == true)
  fprintf(stdout, "v = w\n");
else
  fprintf(stdout, "v != w\n");
if (VecIsEqual(v, v) == true)
  fprintf(stdout, "v = v\n");
else
  fprintf(stdout, "v != v\n");
// Op
VecFloat *x = VecGetOp(v, norm, w, 2.0);
if (x == NULL) {
  fprintf(stderr, "VecGetOp failed\n");
  return 13;
}
fprintf(stdout, "x: ");
VecPrint(x, stdout);
fprintf(stdout, "\n");
VecOp(v, norm, NULL, 0.0);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
// Dot prod
fprintf(stdout, "dot prod v.x: %.3f\n", VecDotProd(v, x));
// Rotate
VecFree(&v);
v = VecFloatCreate(2);
if (v == NULL) {
  fprintf(stderr, "malloc failed\n");
  return 14;
}
VecSet(v, 0, 1.0);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
VecRot2D(v, PBMATH_QUARTERPI);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
VecFree(&x);
x = VecGetRot2D(v, PBMATH_QUARTERPI);
if (v == NULL) {
  fprintf(stderr, "VecGetRot2D failed\n");
  return 15;
}
fprintf(stdout, "x: ");
VecPrint(x, stdout);
fprintf(stdout, "\n");
// AngleTo
fprintf(stdout, "Angle between vector:\n");
float dtheta = PBMATH_PI / 6.0;
VecSet(x, 0, 1.0); VecSet(x, 1, 0.0);
for (int i = 0; i < 12; ++i) {
  VecSet(v, 0, 1.0); VecSet(v, 1, 0.0);
  for (int j = 0; j < 12; ++j) {
    VecPrint(x, stdout);
    fprintf(stdout, " ");
    VecPrint(v, stdout);
    fprintf(stdout, " %.3f\n", ConvRad2Deg(VecAngleTo2D(x, v)));
    VecRot2D(v, dtheta);
  }
```

```
  VecRot2D(x, dtheta);
}
// Free memory
VecFree(&x);
VecFree(&w);
VecFree(&v);

// ------------- MatFloat
fprintf(stdout, "-------- MatFloat\n");
// Create a matrix of dimension 3,2
VecShort *dimMat = VecShortCreate(2);
VecSet(dimMat, 0, 3);
VecSet(dimMat, 1, 2);
MatFloat *mat = MatFloatCreate(dimMat);
// If we couldn't create the matrix
if (mat == NULL) {
  fprintf(stderr, "MatCreate failed\n");
  return 16;
}
// Print the matrix
fprintf(stdout, "mat: \n");
MatPrint(mat, stdout);
fprintf(stdout, "\n");
// Set some values
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 0);
MatSet(mat, dimMat, 0.5);
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 1);
MatSet(mat, dimMat, 2.0);
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 2);
MatSet(mat, dimMat, 1.0);
VecSet(dimMat, 0, 1);
VecSet(dimMat, 1, 0);
MatSet(mat, dimMat, 2.0);
VecSet(dimMat, 0, 2);
VecSet(dimMat, 1, 1);
MatSet(mat, dimMat, 1.0);
// Print the matrix
fprintf(stdout, "mat: \n");
MatPrint(mat, stdout);
fprintf(stdout, "\n");
// Clone the matrix
MatFloat *cloneMatFloat = MatClone(mat);
if (cloneMatFloat == NULL) {
  fprintf(stderr, "MatClone failed\n");
  return 17;
}
// Print the matrix
fprintf(stdout, "cloneMatFloat:\n");
MatPrint(cloneMatFloat, stdout);
fprintf(stdout, "\n");
MatFree(&cloneMatFloat);
// Save the matrix
f = fopen("./matfloat.txt", "w");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 18;
}
ret = MatSave(mat, f);
if (ret != 0) {
```

```
  fprintf(stderr, "MatSave failed (%d)\n", ret);
  return 19;
}
fclose(f);
// Load the matrix
f = fopen("./matfloat.txt", "r");
if (f == NULL) {
  fprintf(stderr, "fopen failed\n");
  return 20;
}
MatFloat *matb = NULL;
ret = MatLoad(&matb, f);
if (ret != 0) {
  fprintf(stderr, "MatLoad failed (%d)\n", ret);
  return 21;
}
fclose(f);
// Get the dimension and values of the loaded matrix
VecShort *dimMatb = MatDim(matb);
fprintf(stdout, "dim loaded matrix: ");
VecPrint(dimMatb, stdout);
fprintf(stdout, "\n");
for (VecSet(dimMat, 1, 0); VecGet(dimMat, 1) < VecGet(dimMatb, 1);
  VecSet(dimMat, 1, VecGet(dimMat, 1) + 1)) {
  for (VecSet(dimMat, 0, 0); VecGet(dimMat, 0) < VecGet(dimMatb, 0);
    VecSet(dimMat, 0, VecGet(dimMat, 0) + 1))
    fprintf(stdout, "%f ", MatGet(matb, dimMat));
  fprintf(stdout, "\n");
}
// MatProdVec
v = VecFloatCreate(3);
if (v == NULL) {
  fprintf(stderr, "VecFloatCreate failed\n");
  return 22;
}
VecSet(v, 0, 2.0);
VecSet(v, 1, 3.0);
VecSet(v, 2, 4.0);
w = MatProd(matb, v);
if (w == NULL) {
  fprintf(stderr, "MatProd failed\n");
  return 23;
}
fprintf(stdout, "Mat prod of\n");
MatPrint(matb, stdout);
fprintf(stdout, "\nand\n");
VecPrint(v, stdout);
fprintf(stdout, "\nequals\n");
VecPrint(w, stdout);
fprintf(stdout, "\n");
VecFree(&v);
VecFree(&w);
// MatProdMat
VecSet(dimMat, 0, VecGet(dimMatb, 1));
VecSet(dimMat, 1, VecGet(dimMatb, 0));
MatFloat *matc = MatFloatCreate(dimMat);
if (matc == NULL) {
  fprintf(stderr, "MatFloatCreate failed\n");
  return 24;
}
VecSet(dimMat, 0, 0);
VecSet(dimMat, 1, 0);
```

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

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

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

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

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

```
  fprintf(stdout, "%f radians -> %f degrees\n", 90.0,
    ConvDeg2Rad(90.0));

  // -------------- Usefull basic functions
  fprintf(stdout, "-------- Usefull basic functions\n");
  fprintf(stdout, "10^2 = %d\n", powi(10, 2));

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

Output:

```
-------- VecShort
a: <0,0,0>
a: <0,1,0>
cloneShort: <0,1,0>
b: 3 0 1 0
b: <2,1,3>
VecProd(a,b): 1
a: <2,1,3>
Reset a:<0,0,0>
Step a up to b:
<0,0,0>
<0,0,1>
<0,0,2>
<1,0,0>
<1,0,1>
<1,0,2>
-------- VecFloat
v: <0.000,0.000,0.000>
v: <0.000,1.000,0.000>
cloneFloat: <0.000,1.000,0.000>
w: 3 0.000000 1.000000 0.000000
w: <2.000,1.000,3.000>
v: <2.000,1.000,3.000>
Norm of v: 3.742
Normalized v: <0.535,0.267,0.802>
Distance between v and w: 2.742
Hamiltonian distance between v and w: 4.396
Pixel distance between v and w: 6.000
v != w
v = v
x: <6.000,3.000,9.000>
v: <2.000,1.000,3.000>
dot prod v.x: 42.000
v: <1.000,0.000>
v: <0.707,0.707>
x: <-0.000,1.000>
Angle between vector:
<1.000,0.000> <1.000,0.000> 0.000
<1.000,0.000> <0.866,0.500> 30.000
<1.000,0.000> <0.500,0.866> 60.000
<1.000,0.000> <0.000,1.000> 90.000
<1.000,0.000> <-0.500,0.866> 120.000
<1.000,0.000> <-0.866,0.500> 150.000
<1.000,0.000> <-1.000,0.000> -180.000
<1.000,0.000> <-0.866,-0.500> -150.000
<1.000,0.000> <-0.500,-0.866> -120.000
<1.000,0.000> <-0.000,-1.000> -90.000
```

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

```
<-1.000,0.000> <1.000,0.000> 180.000
<-1.000,0.000> <0.866,0.500> -150.000
<-1.000,0.000> <0.500,0.866> -120.000
<-1.000,0.000> <0.000,1.000> -90.000
<-1.000,0.000> <-0.500,0.866> -60.000
<-1.000,0.000> <-0.866,0.500> -30.000
<-1.000,0.000> <-1.000,0.000> 0.000
<-1.000,0.000> <-0.866,-0.500> 30.000
<-1.000,0.000> <-0.500,-0.866> 60.000
<-1.000,0.000> <-0.000,-1.000> 90.000
<-1.000,0.000> <0.500,-0.866> 120.000
<-1.000,0.000> <0.866,-0.500> 150.000
<-0.866,-0.500> <1.000,0.000> 150.000
<-0.866,-0.500> <0.866,0.500> 180.000
<-0.866,-0.500> <0.500,0.866> -150.000
<-0.866,-0.500> <0.000,1.000> -120.000
<-0.866,-0.500> <-0.500,0.866> -90.000
<-0.866,-0.500> <-0.866,0.500> -60.000
<-0.866,-0.500> <-1.000,0.000> -30.000
<-0.866,-0.500> <-0.866,-0.500> 0.000
<-0.866,-0.500> <-0.500,-0.866> 30.000
<-0.866,-0.500> <-0.000,-1.000> 60.000
<-0.866,-0.500> <0.500,-0.866> 90.000
<-0.866,-0.500> <0.866,-0.500> 120.000
<-0.500,-0.866> <1.000,0.000> 120.000
<-0.500,-0.866> <0.866,0.500> 150.000
<-0.500,-0.866> <0.500,0.866> -180.000
<-0.500,-0.866> <0.000,1.000> -150.000
<-0.500,-0.866> <-0.500,0.866> -120.000
<-0.500,-0.866> <-0.866,0.500> -90.000
<-0.500,-0.866> <-1.000,0.000> -60.000
<-0.500,-0.866> <-0.866,-0.500> -30.000
<-0.500,-0.866> <-0.500,-0.866> 0.000
<-0.500,-0.866> <-0.000,-1.000> 30.000
<-0.500,-0.866> <0.500,-0.866> 60.000
<-0.500,-0.866> <0.866,-0.500> 90.000
<-0.000,-1.000> <1.000,0.000> 90.000
<-0.000,-1.000> <0.866,0.500> 120.000
<-0.000,-1.000> <0.500,0.866> 150.000
<-0.000,-1.000> <0.000,1.000> -180.000
<-0.000,-1.000> <-0.500,0.866> -150.000
<-0.000,-1.000> <-0.866,0.500> -120.000
<-0.000,-1.000> <-1.000,0.000> -90.000
<-0.000,-1.000> <-0.866,-0.500> -60.000
<-0.000,-1.000> <-0.500,-0.866> -30.000
<-0.000,-1.000> <-0.000,-1.000> 0.000
<-0.000,-1.000> <0.500,-0.866> 30.000
<-0.000,-1.000> <0.866,-0.500> 60.000
<0.500,-0.866> <1.000,0.000> 60.000
<0.500,-0.866> <0.866,0.500> 90.000
<0.500,-0.866> <0.500,0.866> 120.000
<0.500,-0.866> <0.000,1.000> 150.000
<0.500,-0.866> <-0.500,0.866> -179.982
<0.500,-0.866> <-0.866,0.500> -150.000
<0.500,-0.866> <-1.000,0.000> -120.000
<0.500,-0.866> <-0.866,-0.500> -90.000
<0.500,-0.866> <-0.500,-0.866> -60.000
<0.500,-0.866> <-0.000,-1.000> -30.000
<0.500,-0.866> <0.500,-0.866> 0.007
<0.500,-0.866> <0.866,-0.500> 30.000
<0.866,-0.500> <1.000,0.000> 30.000
<0.866,-0.500> <0.866,0.500> 60.000
```

```
<0.866,-0.500> <0.500,0.866> 90.000
<0.866,-0.500> <0.000,1.000> 120.000
<0.866,-0.500> <-0.500,0.866> 150.000
<0.866,-0.500> <-0.866,0.500> 180.000
<0.866,-0.500> <-1.000,0.000> -150.000
<0.866,-0.500> <-0.866,-0.500> -120.000
<0.866,-0.500> <-0.500,-0.866> -90.000
<0.866,-0.500> <-0.000,-1.000> -60.000
<0.866,-0.500> <0.500,-0.866> -30.000
<0.866,-0.500> <0.866,-0.500> 0.000
-------- MatFloat
mat:
[0.000,0.000,0.000
 0.000,0.000,0.000]
mat:
[0.500,2.000,0.000
 2.000,0.000,1.000]
cloneMatFloat:
[0.500,2.000,0.000
 2.000,0.000,1.000]
dim loaded matrix: <3,2>
0.500000 2.000000 0.000000
2.000000 0.000000 1.000000
Mat prod of
[0.500,2.000,0.000
 2.000,0.000,1.000]
and
<2.000,3.000,4.000>
equals
<7.000,8.000>
Mat prod of
[0.500,2.000,0.000
 2.000,0.000,1.000]
and
[1.000,4.000
 2.000,5.000
 3.000,6.000]
equals
[4.500,12.000
 5.000,14.000]
identity:
[1.000,0.000,0.000
 0.000,1.000,0.000
 0.000,0.000,1.000]
inverse of:
[3.000,0.000,2.000
 2.000,0.000,-2.000
 0.000,1.000,1.000]
equals
[0.200,0.200,0.000
 -0.200,0.300,1.000
 0.200,-0.300,0.000]
check of inverse:
[1.000,0.000,0.000
 0.000,1.000,0.000
 0.000,0.000,1.000]
-------- Gauss
Gauss function (mean:0.0, sigma:1.0):
-2.000 0.054
-1.800 0.079
-1.600 0.111
-1.400 0.150
```

```
-1.200 0.194
-1.000 0.242
-0.800 0.290
-0.600 0.333
-0.400 0.368
-0.200 0.391
0.000 0.399
0.200 0.391
0.400 0.368
0.600 0.333
0.800 0.290
1.000 0.242
1.200 0.194
1.400 0.150
1.600 0.111
1.800 0.079
2.000 0.054
Gauss rnd (mean:1.0, sigma:0.5):
1.246 0.887
2.061 0.587
0.644 1.590
0.710 1.316
1.099 0.722
0.876 -0.294
1.421 1.146
1.469 0.987
1.401 0.219
1.509 1.525
-------- Smoother
0.000 0.000 0.000
0.100 0.028 0.009
0.200 0.104 0.058
0.300 0.216 0.163
0.400 0.352 0.317
0.500 0.500 0.500
0.600 0.648 0.683
0.700 0.784 0.837
0.800 0.896 0.942
0.900 0.972 0.991
1.000 1.000 1.000
-------- Conversion functions
0.785400 radians -> 45.000069 degrees
90.000000 radians -> 1.570797 degrees
-------- Usefull basic functions
10^2 = 100
```

vecshort.txt:

```
3 0 1 0
```

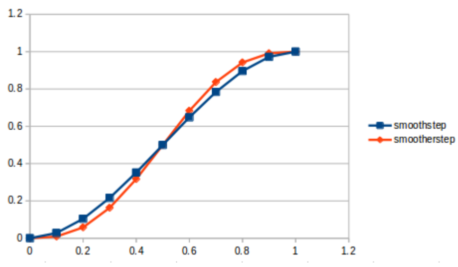
vecfloat.txt:

```
3 0.000000 1.000000 0.000000
```

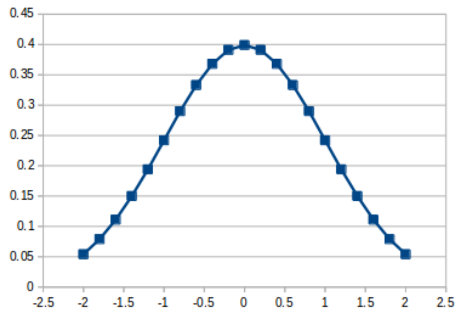
matfloat.txt:

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

smoother functions:




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



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