PBMath

P. Baillehache

September 28, 2017

Contents

Introduction

PBMath is C library providing mathematical structures and functions.

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

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

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

1 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_PI 3.14159

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

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

void VecTypeUnsupported(void*t, ...);
#define VecClone(V) _Generic((V), \
  VecFloat*: VecFloatClone, \
  default: VecTypeUnsupported)(V)
#define VecLoad(V, S) _Generic((V), \
  VecFloat**: VecFloatLoad, \
  default: VecTypeUnsupported)(V, S)
#define VecSave(V, S) _Generic((V), \
  VecFloat*: VecFloatSave, \
  default: VecTypeUnsupported)(V, S)
#define VecFree(V) _Generic((V), \
  VecFloat**: VecFloatFree, \
  default: VecTypeUnsupported)(V)
#define VecPrint(V, S) _Generic((V), \
  VecFloat*: VecFloatPrintDef, \
  default: VecTypeUnsupported)(V, S)
#define VecGet(V, I) _Generic((V), \
  VecFloat*: VecFloatGet, \
  default: VecTypeUnsupported)(V, I)
#define VecSet(V, I, VAL) _Generic((V), \
  VecFloat*: VecFloatSet, \
  default: VecTypeUnsupported)(V, I, VAL)
#define VecCopy(V, W) _Generic((V), \
  VecFloat*: VecFloatCopy, \
  default: VecTypeUnsupported)(V, W)
#define VecDim(V) _Generic((V), \
  VecFloat*: VecFloatDim, \
  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, \
  default: VecTypeUnsupported)(V, W)
#define VecIsEqual(V, W) _Generic((V), \
  VecFloat*: VecFloatIsEqual, \
  default: VecTypeUnsupported)(V, W)
#define VecOp(V, A, W, B) _Generic((V), \
  VecFloat*: VecFloatOp, \
  default: VecTypeUnsupported)(V, A, W, B)
```

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

// ================= 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 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);

// -------------- 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);

#endif
```

2 Code

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

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

#include "pbmath.h"

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

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

// -------------- 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) {
    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 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;
}

// -------------- 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);
}
```

3 Makefile

```
OPTIONS_DEBUG=-ggdb -g3 -Wall
OPTIONS_RELEASE=-O3
OPTIONS=$(OPTIONS_DEBUG)

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) -c main.c

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

4 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));
  // -------------- 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 1;
  }
  // 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");
  // Save the vector
  FILE *f = fopen("./vec.txt", "w");
  if (f == NULL) {
    fprintf(stderr, "fopen failed\n");
    return 2;
  }
  int ret = VecSave(v, f);
  if (ret != 0) {
    fprintf(stderr, "VecSave failed (%d)\n", ret);
    return 3;
  }
  fclose(f);
  // Load the vector
  f = fopen("./vec.txt", "r");
  if (f == NULL) {
    fprintf(stderr, "fopen failed\n");
    return 4;
  }
  VecFloat *w = NULL;
  ret = VecLoad(&w, f);
  if (ret != 0) {
    fprintf(stderr, "VecLoad failed (%d)\n", ret);
    return 5;
  }
  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));
// 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 6;
}
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 7;
}
VecSet(v, 0, 1.0);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
VecRot2D(v, 0.5 * PBMATH_PI);
fprintf(stdout, "v: ");
VecPrint(v, stdout);
fprintf(stdout, "\n");
VecFree(&x);
x = VecGetRot2D(v, 0.5 * PBMATH_PI);
if (v == NULL) {
  fprintf(stderr, "VecGetRot2D failed\n");
  return 8;
}
fprintf(stdout, "x: ");
VecPrint(x, stdout);
fprintf(stdout, "\n");
// Free memory
VecFree(&x);
VecFree(&w);
```

```
  VecFree(&v);
  // -------------- 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 9;
  }
  // 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));

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

Output:

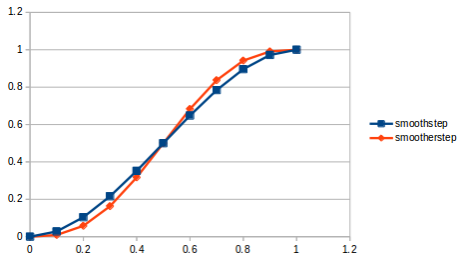
```
-------- VecFloat
v: <0.000,0.000,0.000>
v: <0.000,1.000,0.000>
w: 3 0.000000 1.000000 0.000000
w: <2.000,1.000,3.000>
v: <2.000,1.000,3.000>
Norm of v: 3.742
Normalized v: <0.535,0.267,0.802>
Distance between v and w: 2.742
v != w
v = v
x: <6.000,3.000,9.000>
v: <2.000,1.000,3.000>
dot prod v.x: 42.000
v: <1.000,0.000>
v: <0.000,1.000>
x: <-1.000,0.000>
-------- 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):
1.072 1.214
1.298 0.467
1.431 0.964
1.946 0.701
1.362 0.550
0.904 1.405
1.213 1.039
1.414 0.501
0.778 2.277
1.252 0.534
-------- 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
```

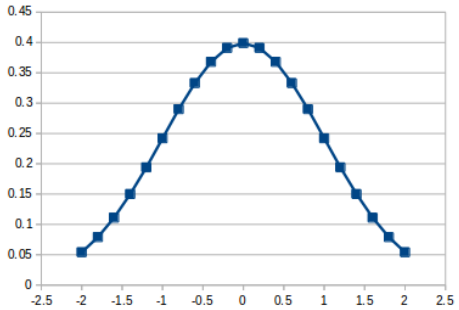
vec.txt:

```
3 0.000000 1.000000 0.000000
```

smoother functions:



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



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