

NeuraMorph

P. Baillehache

August 27, 2020

Contents

1	Definitions	1
2	Interface	1
3	Code	4
3.1	neuramorph.c	4
3.2	neuramorph-inline.c	14
4	Makefile	18
5	Unit tests	19
6	Unit tests output	27

Introduction

NeuraMorph is a C library providing structures and functions to implement a neural network.

It uses the PBErr, PBMath, GSet library.

1 Definitions

2 Interface

```
// ===== NEURAMORPH.H =====
```

```

#ifndef NEURAMORPH_H
#define NEURAMORPH_H

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

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "pbmath.h"
#include "gset.h"
#include "gdataset.h"

// ---- NeuraMorphUnit

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

typedef struct NeuraMorphUnit {

    // Input indices in parent NeuraMorph
    VecLong* iInputs;

    // Output indices in parent NeuraMorph
    VecLong* iOutputs;

    // Lowest and highest values for filtering inputs
    VecFloat* lowFilters;
    VecFloat* highFilters;

    // Lowest and highest values of outputs
    VecFloat* lowOutputs;
    VecFloat* highOutputs;

    // Vector to memorize the output values
    VecFloat* outputs;

    // Transfer function coefficients
    // Seen as (nb output) triangular matrices of size (nb input + 1)
    VecFloat** coeffs;

    // Working variables to avoid reallocation of memory at each Evaluate()
    bool* activeInputs;
    VecFloat* scaledInputs;

} NeuraMorphUnit;

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

// Create a new NeuraMorphUnit between the input 'iInputs' and the
// outputs 'iOutputs'
NeuraMorphUnit* NeuraMorphUnitCreate(
    const VecLong* iInputs,
    const VecLong* iOutputs);

// Free the memory used by the NeuraMorphUnit 'that'
void NeuraMorphUnitFree(NeuraMorphUnit** that);

// Get the input indices of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline

```

```

#endif
const VecLong* NMUnitIInputs(const NeuraMorphUnit* that);

// Get the output indices of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecLong* NMUnitIOutputs(const NeuraMorphUnit* that);

// Get the output values of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* NMUnitOutputs(const NeuraMorphUnit* that);

// Calculate the outputs for the 'inputs' with the NeuraMorphUnit 'that'
// Update 'that->outputs'
void NMUnitEvaluate(
    NeuraMorphUnit* that,
    const VecFloat* inputs);

// ----- NeuraMorph

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

typedef struct NeuraMorph {

    // Number of inputs and outputs
    long nbInput;
    long nbOutput;

    // Inputs and outputs values
    VecFloat* inputs;
    VecFloat* outputs;

    // Internal values
    VecFloat* hiddens;

    // GSet of NeuraMorphUnit
    GSet units;

} NeuraMorph;

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

// Create a new NeuraMorph with 'nbInput' inputs and 'nbOutput' outputs
NeuraMorph* NeuraMorphCreate(
    long nbInput,
    long nbOutput);

// Free the memory used by the NeuraMorph 'that'
void NeuraMorphFree(NeuraMorph** that);

// Get the number of input values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbInput(NeuraMorph* that);

// Get the number of output values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline

```

```

#endif
long NMGetNbOutput(NeuraMorph* that);

// Get the input values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* NMInputs(NeuraMorph* that);

// Get the output values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* NMOutputs(const NeuraMorph* that);

// Get the number of hidden values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbHidden(NeuraMorph* that);

// Set the number of hidden values of the NeuraMorph 'that' to 'nb'
#if BUILDMODE != 0
static inline
#endif
void NMSetNbHidden(
    NeuraMorph* that,
    long nb);

// Add one NeuraMorphUnit with input and output indices 'iInputs'
// and 'iOutputs' to the NeuraMorph 'that'
// Return the created NeuraMorphUnit
NeuraMorphUnit* NMAddUnit(
    NeuraMorph* that,
    const VecLong* iInputs,
    const VecLong* iOutputs);

// Remove the NeuraMorphUnit 'unit' from the NeuraMorph 'that'
// The NeuraMorphUnit is not freed
void NMRemoveUnit(
    NeuraMorph* that,
    NeuraMorphUnit* unit);

// ===== static inliner =====

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

#endif

```

3 Code

3.1 neuramorph.c

```

// ===== NEURAMORPH.C =====

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

```

```

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

// ----- NeuraMorphUnit

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

// Return the number of coefficients of a NeuraMorphUnit having 'nbIn' inputs
long NMUnitGetNBCoeff(long nbIn);

// Get the coefficient for the pair of inputs 'iInputA', 'iInputB' in the
// NeuraMorphUnit 'that' for the output 'iOutput'
float NMUnitGetCoeff(
    const NeuraMorphUnit* that,
        long iInputA,
        long iInputB,
        long iOutput);

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

// Create a new NeuraMorphUnit between the input 'iInputs' and the
// outputs 'iOutputs'
NeuraMorphUnit* NeuraMorphUnitCreate(
    const VecLong* iInputs,
    const VecLong* iOutputs) {

#if BUILDMODE == 0

    if (iInputs == NULL) {

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

    }

    if (iOutputs == NULL) {

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

    }

#endif

    // Allocate memory for the NeuraMorphUnit
    NeuraMorphUnit* that =
        PBErrMalloc(
            NeuraMorphErr,
            sizeof(NeuraMorphUnit));

    // Get the number of inputs (including the constant) and outputs
    long nbIn = VecGetDim(iInputs) + 1;
    long nbOut = VecGetDim(iOutputs);

```

```

// Init properties
that->iInputs = VecClone(iInputs);
that->iOutputs = VecClone(iOutputs);
that->lowFilters = VecFloatCreate(nbIn);
that->highFilters = VecFloatCreate(nbIn);
that->lowOutputs = NULL;
that->highOutputs = NULL;
that->outputs = VecFloatCreate(nbOut);
that->coeffs =
    PBErrMalloc(
        NeuraMorphErr,
        sizeof(VecFloat*) * nbOut);
long nbCoeff = NMUnitGetNBCoeff(nbIn);
for (
    long iOut = nbOut;
    iOut--;
    that->coeffs[iOut] = VecFloatCreate(nbCoeff));

// 'nbIn + 1' for the constant
that->activeInputs =
    PBErrMalloc(
        NeuraMorphErr,
        sizeof(bool) * nbIn);
that->scaledInputs = VecFloatCreate(nbIn);

// Set the input value, filters and active flag for the constant
VecSet(
    that->scaledInputs,
    0,
    1.0);
that->activeInputs[0] = true;

// Return the new NeuraMorphUnit
return that;
}

// Free the memory used by the NeuraMorphUnit 'that'
void NeuraMorphUnitFree(NeuraMorphUnit** that) {

    // Check the input
    if (that == NULL || *that == NULL) {

        return;
    }

    // Free memory
    long nbOut = VecGetDim((*that)->iOutputs);
    VecFree(&((*that)->iInputs));
    VecFree(&((*that)->iOutputs));
    VecFree(&((*that)->lowFilters));
    VecFree(&((*that)->highFilters));
    if ((*that)->lowOutputs != NULL) {

        VecFree(&((*that)->lowOutputs));
    }

    if ((*that)->highOutputs != NULL) {

```

```

        VecFree(&((*that)->highOutputs));

    }

    VecFree(&((*that)->outputs));
    for (
        long iOut = nbOut;
        iOut--;
        VecFree((*that)->coeffs + iOut));
    free((*that)->coeffs);
    free((*that)->activeInputs);
    VecFree(&((*that)->scaledInputs));
    free(*that);
    *that = NULL;

}

// Return the number of coefficients of a NeuraMorphUnit having 'nbIn' inputs
long NMUnitGetNBCoeff(long nbIn) {

#if BUILDMODE == 0

    if (nbIn <= 0) {

        NeuraMorphErr->_type = PErrTypeInvalidArg;
        sprintf(
            NeuraMorphErr->_msg,
            "'nbIn' is invalid (%ld>0)",
            nbIn);
        PErrCatch(NeuraMorphErr);

    }

#endif

    // Declare a variable to memorise the result
    long nb = 0;

    // Calculate the number of values in the triangular matrix of size
    // nbIn
    for (
        long i = nbIn;
        i >= 0;
        nb += (i--));

    // Return the result
    return nb;

}

// Calculate the outputs for the 'inputs' with the NeuraMorphUnit 'that'
// Update 'that->outputs'
void NMUnitEvaluate(
    NeuraMorphUnit* that,
    const VecFloat* inputs) {

#if BUILDMODE == 0

    if (that == NULL) {

        NeuraMorphErr->_type = PErrTypeNullPointer;
        sprintf(

```

```

        NeuroMorphErr->_msg,
        "'that' is null");
    PBErriCatch(NeuraMorphErr);
}

if (VecGetDim(inputs) != VecGetDim(that->iInputs)) {

    NeuroMorphErr->_type = PBErriTypeInvalidArg;
    sprintf(
        NeuroMorphErr->_msg,
        "'inputs' has invalid dimension (%ld!=%ld)",
        VecGetDim(inputs),
        VecGetDim(that->iInputs));
    PBErriCatch(NeuraMorphErr);
}

#endif

// Reset the outputs
VecSetNull(that->outputs);

// Update the active flags and scaled inputs (skip the constant)
for (
    long iInput = 1;
    iInput < VecGetDim(that->scaledInputs);
    ++iInput) {

    // Get the input value and its low/high filters
    float val =
        VecGet(
            inputs,
            iInput - 1);
    float low =
        VecGet(
            that->lowFilters,
            iInput);
    float high =
        VecGet(
            that->highFilters,
            iInput);

    // If the value is inside the filter
    if (
        low <= val &&
        val <= high &&
        (high - low) > PBMATH_EPSILON) {

        // Set this value as active
        that->activeInputs[iInput] = true;

        // Scale the value according to the filter
        float scaled = 2.0 * (val - low) / (high - low) - 1.0;
        VecSet(
            that->scaledInputs,
            iInput,
            scaled);

    // Else the value is outside the filter
    } else {

```



```

        // Set this value as inactive
        that->activeInputs[iInput] = false;

    }

}

// Loop on the pair of active inputs
for (
    long iInputA = 0;
    iInputA < VecGetDim(that->scaledInputs);
    ++iInputA) {

    if (that->activeInputs[iInputA] == true) {

        for (
            long iInputB = 0;
            iInputB <= iInputA;
            ++iInputB) {

            if (that->activeInputs[iInputB] == true) {

                // Loop on the outputs
                for (
                    long iOutput = 0;
                    iOutput < VecGetDim(that->outputs);
                    ++iOutput) {

                    // Calculate the components for this output and pair of inputs
                    float comp =
                        VecGet(
                            that->scaledInputs,
                            iInputA) *
                        VecGet(
                            that->scaledInputs,
                            iInputB) *
                        NMUnitGetCoeff(
                            that,
                            iInputA,
                            iInputB,
                            iOutput);

                    // Add the component to the output
                    float cur =
                        VecGet(
                            that->outputs,
                            iOutput);
                    VecSet(
                        that->outputs,
                        iOutput,
                        cur + comp);

                }

            }

        }

    }

}
}

```

```

// If the low and high values for outputs don't exist yet
if (that->lowOutputs == NULL) {

    // Create the low and high values by cloning the current output
    that->lowOutputs = VecClone(that->outputs);
    that->highOutputs = VecClone(that->outputs);

// Else, the low and high values for outputs exist
} else {

    // Loop on the outputs
    for (
        long iOutput = 0;
        iOutput < VecGetDim(that->outputs);
        ++iOutput) {

        // Update the low and high values for this output
        float val =
            VecGet(
                that->outputs,
                iOutput);

        float curLow =
            VecGet(
                that->lowOutputs,
                iOutput);
        if (curLow > val) {

            VecSet(
                that->lowOutputs,
                iOutput,
                val);

        }

        float curHigh =
            VecGet(
                that->highOutputs,
                iOutput);
        if (curHigh < val) {

            VecSet(
                that->highOutputs,
                iOutput,
                val);

        }

    }

}

}

// Get the coefficient for the pair of inputs 'iInputA', 'iInputB' in the
// NeuraMorphUnit 'that' for the output 'iOutput'
float NMUnitGetCoeff(
    const NeuraMorphUnit* that,
        long iInputA,
        long iInputB,
        long iOutput) {

```

```

#if BUILDMODE == 0

    if (that == NULL) {

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

    }

    if (
        iInputA < 0 ||
        iInputA >= VecGetDim(that->scaledInputs)) {

        NeuraMorphErr->_type = PBErrTypeInvalidArg;
        sprintf(
            NeuraMorphErr->_msg,
            "'iInputA' is invalid (0<=%ld<%ld)",
            iInputA,
            VecGetDim(that->scaledInputs));
        PBErrCatch(NeuraMorphErr);

    }

    if (
        iInputB < 0 ||
        iInputB >= VecGetDim(that->scaledInputs)) {

        NeuraMorphErr->_type = PBErrTypeInvalidArg;
        sprintf(
            NeuraMorphErr->_msg,
            "'iInputB' is invalid (0<=%ld<%ld)",
            iInputB,
            VecGetDim(that->scaledInputs));
        PBErrCatch(NeuraMorphErr);

    }

    if (iInputA < iInputB) {

        NeuraMorphErr->_type = PBErrTypeInvalidArg;
        sprintf(
            NeuraMorphErr->_msg,
            "The pair of indices is invalid (%ld>=%ld)",
            iInputA,
            iInputB);
        PBErrCatch(NeuraMorphErr);

    }

    if (
        iOutput < 0 ||
        iOutput >= VecGetDim(that->outputs)) {

        NeuraMorphErr->_type = PBErrTypeInvalidArg;
        sprintf(
            NeuraMorphErr->_msg,
            "'iInputB' is invalid (0<=%ld<%ld)",
            iInputB,
            VecGetDim(that->outputs));

    }

```

```

        PBErrCatch(NeuraMorphErr);

    }

#endif

    // Calculate the index of the coefficient
    long iCoeff = 0;
    for (
        long shift = 0;
        shift < iInputA;
        iCoeff += (shift++) + 1);
    iCoeff += iInputB;

    // Return the coefficient
    float coeff =
        VecGet(
            that->coeffs[iOutput],
            iCoeff);
    return coeff;
}

// ----- NeuraMorph

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

// Create a new NeuraMorph with 'nbInput' inputs and 'nbOutput' outputs
NeuraMorph* NeuraMorphCreate(
    long nbInput,
    long nbOutput) {

    // Allocate memory for the NeuraMorph
    NeuraMorph* that =
        PBErrMalloc(
            NeuraMorphErr,
            sizeof(NeuraMorph));

    // Init properties
    that->nbInput = nbInput;
    that->nbOutput = nbOutput;
    that->inputs = VecFloatCreate(nbInput);
    that->outputs = VecFloatCreate(nbOutput);
    that->hiddens = NULL;
    that->units = GSetCreateStatic();

    // Return the NeuraMorph
    return that;
}

// Free the memory used by the NeuraMorph 'that'
void NeuraMorphFree(NeuraMorph** that) {

    // Check the input
    if (that == NULL || *that == NULL) {

        return;
    }

    // Free memory

```

```

VecFree(&((*that)->inputs));
VecFree(&((*that)->outputs));
if ((*that)->hiddens != NULL) {

    VecFree(&((*that)->hiddens));

}

while (GSetNbElem(&((*that)->units)) > 0) {

    NeuroMorphUnit* unit = GSetPop(&((*that)->units));
    NeuroMorphUnitFree(&unit);

}

free(*that);
*that = NULL;

}

// Add one NeuroMorphUnit with input and output indices 'iInputs'
// and 'iOutputs' to the NeuroMorph 'that'
// Return the created NeuroMorphUnit
NeuroMorphUnit* NMAAddUnit(
    NeuroMorph* that,
    const VecLong* iInputs,
    const VecLong* iOutputs) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        NeuroMorphErr->_type = PErrTypeNullPointer;
        sprintf(
            NeuroMorphErr->_msg,
            "'that' is null");
        PErrCatch(NeuroMorphErr);

    }

    if (iInputs == NULL) {

        NeuroMorphErr->_type = PErrTypeNullPointer;
        sprintf(
            NeuroMorphErr->_msg,
            "'iInputs' is null");
        PErrCatch(NeuroMorphErr);

    }

    if (iOutputs == NULL) {

        NeuroMorphErr->_type = PErrTypeNullPointer;
        sprintf(
            NeuroMorphErr->_msg,
            "'iOutputs' is null");
        PErrCatch(NeuroMorphErr);

    }

#endif
}

```

```

    // Create the NeuraMorphUnit
    NeuraMorphUnit* unit =
        NeuraMorphUnitCreate(
            iInputs,
            iOutputs);

    // Append the new NeuraorphUnit to the set of NeuraMorphUnit
    GSetAppend(
        &(that->units),
        unit);

    // Return the new unit
    return unit;
}

// Remove the NeuraMorphUnit 'unit' from the NeuraMorph 'that'
// The NeuraMorphUnit is not freed
void NMRemoveUnit(
    NeuraMorph* that,
    NeuraMorphUnit* unit) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

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

    }

#endif

    // Remove the NeuraorphUnit from the set of NeuraMorphUnit
    GSetRemoveAll(
        &(that->units),
        unit);
}

```

3.2 neuramorph-inline.c

```

// ===== NEURAMORPH-INLINE.C =====

// ----- NeuraMorphUnit

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

// Get the input indices of the NeuraMorphUnit 'that'
#ifdef BUILDMODE != 0
static inline
#endif
const VecLong* NMUnitIInputs(const NeuraMorphUnit* that) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

```

```

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

    }

#endif

    return that->iInputs;

}

// Get the output indices of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecLong* NMUnitIOutputs(const NeuraMorphUnit* that) {

#if BUILDMODE == 0

    if (that == NULL) {

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

    }

#endif

    return that->iOutputs;

}

// Get the output values of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* NMUnitOutputs(const NeuraMorphUnit* that) {

#if BUILDMODE == 0

    if (that == NULL) {

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

    }

#endif

    return that->outputs;

}

```

```

// ----- NeuraMorph

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

// Get the number of input values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbInput(NeuraMorph* that) {

#if BUILDMODE == 0

    if (that == NULL) {

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

    }

#endif

    return that->nbInput;

}

// Get the number of output values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbOutput(NeuraMorph* that) {

#if BUILDMODE == 0

    if (that == NULL) {

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

    }

#endif

    return that->nbOutput;

}

// Get the input values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* NMInputs(NeuraMorph* that) {

#if BUILDMODE == 0

    if (that == NULL) {

```



```

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

#endif

    return that->inputs;
}

// Get the output values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* NMOutputs(const NeuraMorph* that) {

#if BUILDMODE == 0

    if (that == NULL) {

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

#endif

    return that->outputs;
}

// Get the number of hidden values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbHidden(NeuraMorph* that) {

#if BUILDMODE == 0

    if (that == NULL) {

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

#endif

    long nb = 0;
    if (that->hiddens != NULL) {

```

```

        nb = VecGetDim(that->hiddens);
    }

    return nb;
}

// Set the number of hidden values of the NeuraMorph 'that' to 'nb'
#if BUILDMODE != 0
static inline
#endif
void NMSetNbHidden(
    NeuraMorph* that,
    long nb) {

#if BUILDMODE == 0

    if (that == NULL) {

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

    }

    if (nb <= 0) {

        NeuraMorphErr->_type = PBErrTypeInvalidArg;
        sprintf(
            NeuraMorphErr->_msg,
            "'nb' is invalid (%ld>0)",
            nb);
        PBErrCatch(NeuraMorphErr);

    }

#endif

    if (that->hiddens != NULL) {

        VecFree(&(that->hiddens));

    }

    that->hiddens = VecFloatCreate(nb);

}

```

4 Makefile

```

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

```

```

all: pbmake_wget main

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

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

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

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

```

5 Unit tests

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "neuramorph.h"

void UnitTestNeuraMorphUnitCreateFree() {

    VecLong* iIn = VecLongCreate(3);
    VecSet(
        iIn,
        0,
        0);
    VecSet(
        iIn,
        1,
        1);
    VecSet(
        iIn,
        2,
        2);
    VecLong* iOut = VecLongCreate(2);
    VecSet(
        iOut,
        0,
        0);
    VecSet(
        iOut,
        1,

```

```

1);
NeuraMorphUnit* unit =
    NeuraMorphUnitCreate(
        iIn,
        iOut);
if (
    VecGetDim(unit->coeffs[0]) != 10 ||
    VecGetDim(unit->outputs) != 2 ||
    VecGetDim(unit->lowFilters) != 4 ||
    VecGetDim(unit->highFilters) != 4 ||
    VecGetDim(unit->scaledInputs) != 4 ||
    unit->lowOutputs != NULL ||
    unit->highOutputs != NULL) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NeuraMorphUnitCreate failed (1)");
    PBErrCatch(NeuraMorphErr);

}

bool isSame =
    VecIsEqual(
        unit->iInputs,
        iIn);
if (isSame == false) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NeuraMorphUnitCreate failed (2)");
    PBErrCatch(NeuraMorphErr);

}

isSame =
    VecIsEqual(
        unit->iOutputs,
        iOut);
if (isSame == false) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NeuraMorphUnitCreate failed (3)");
    PBErrCatch(NeuraMorphErr);

}

NeuraMorphUnitFree(&unit);
if (unit != NULL) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NeuraMorphUnitFree failed");
    PBErrCatch(NeuraMorphErr);

}

VecFree(&iIn);

```

```

    VecFree(&iOut);
    printf("UnitTestNeuraMorphUnitCreateFree OK\n");
}

void UnitTestNeuraMorphUnitGetSet() {

    VecLong* iIn = VecLongCreate(3);
    VecLong* iOut = VecLongCreate(2);
    NeuraMorphUnit* unit =
        NeuraMorphUnitCreate(
            iIn,
            iOut);

    if (NMUnitIInputs(unit) != unit->iInputs) {

        NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            NeuraMorphErr->_msg,
            "NMUnitIInputs failed");
        PBErrCatch(NeuraMorphErr);
    }

    if (NMUnitIOutputs(unit) != unit->iOutputs) {

        NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            NeuraMorphErr->_msg,
            "NMUnitIOutputs failed");
        PBErrCatch(NeuraMorphErr);
    }

    if (NMUnitOutputs(unit) != unit->outputs) {

        NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            NeuraMorphErr->_msg,
            "NMUnitOutputs failed");
        PBErrCatch(NeuraMorphErr);
    }

    NeuraMorphUnitFree(&unit);
    VecFree(&iIn);
    VecFree(&iOut);
    printf("UnitTestNeuraMorphUnitGetSet OK\n");
}

void UnitTestNeuraMorphUnitEvaluate() {

    VecLong* iIn = VecLongCreate(3);
    VecLong* iOut = VecLongCreate(2);
    NeuraMorphUnit* unit =
        NeuraMorphUnitCreate(
            iIn,
            iOut);

    for (
        long iInput = 3;

```

```

        iInput--;) {

    VecSet(
        unit->lowFilters,
        iInput + 1,
        0.0);
    VecSet(
        unit->highFilters,
        iInput + 1,
        2.0);

}

// iOutput == 0 -> 1.0+x+y+z+x^2+xy+xz+y^2+yz+z^2
// iOutput == 1 -> x^2-xy+2xz+3y^2-4yz+5z^2
float coeffs[2][10] = {

    { 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0},
    { 0.0, 0.0, 1.0, 0.0, -1.0, 3.0, 0.0, 2.0, -4.0, 5.0}

};

for (
    long iOutput = 2;
    iOutput--;) {

    for (
        long iCoeff = 10;
        iCoeff--;) {

        VecSet(
            unit->coeffs[iOutput],
            iCoeff,
            coeffs[iOutput][iCoeff]);

    }

}

VecFloat* inputs = VecFloatCreate(3);
VecSet(
    inputs,
    0,
    1.0);
VecSet(
    inputs,
    1,
    3.0);
VecSet(
    inputs,
    2,
    1.5);

NMUnitEvaluate(
    unit,
    inputs);

float check[2];
float x = 2.0 * (1.0 - 0.0) / (2.0 - 0.0) - 1.0;
float y = 0.0; //2.0 * (3.0 - 0.0) / (2.0 - 0.0) - 1.0;
float z = 2.0 * (1.5 - 0.0) / (2.0 - 0.0) - 1.0;
check[0] = 1.0 + x + y + z + x * x + x * y + x * z + y * y + y * z + z * z;
check[1] =

```

```

    x * x - x * y + 2.0 * x * z + 3.0 * y * y - 4.0 * y * z + 5.0 * z * z;
VecFloat2D checkHigh = VecFloatCreateStatic2D();
VecSet(
    &checkHigh,
    0,
    check[0]);
VecSet(
    &checkHigh,
    1,
    check[1]);
VecFloat2D checkLow = checkHigh;
for (
    long iOutput = 2;
    iOutput--;) {

    float v =
        VecGet(
            unit->outputs,
            iOutput);
    bool same =
        ISEQUALF(
            v,
            check[iOutput]);
    if (same == false) {

        NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            NeuraMorphErr->_msg,
            "NMUnitEvaluate failed (1)");
        PBErrCatch(NeuraMorphErr);

    }

}

bool sameLow =
    VecIsEqual(
        &checkLow,
        unit->lowOutputs);
bool sameHigh =
    VecIsEqual(
        &checkHigh,
        unit->highOutputs);
if (
    sameLow == false ||
    sameHigh == false) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NMUnitEvaluate failed (2)");
    PBErrCatch(NeuraMorphErr);

}

NeuraMorphUnitFree(&unit);
VecFree(&iIn);
VecFree(&iOut);
VecFree(&inputs);
printf("UnitTestNeuraMorphUnitEvaluate OK\n");

}

```

```

void UnitTestNeuraMorphUnit() {

    UnitTestNeuraMorphUnitCreateFree();
    UnitTestNeuraMorphUnitGetSet();
    UnitTestNeuraMorphUnitEvaluate();
    printf("UnitTestNeuraMorphUnit OK\n");

}

void UnitTestNeuraMorphCreateFree() {

    NeuraMorph* nm =
        NeuraMorphCreate(
            3,
            2);
    if (
        nm->nbInput != 3 ||
        nm->nbOutput != 2 ||
        VecGetDim(nm->inputs) != 3 ||
        VecGetDim(nm->outputs) != 2 ||
        nm->hiddens != NULL ||
        GSetNbElem(&(nm->units)) != 0) {

        NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            NeuraMorphErr->_msg,
            "NeuraMorphCreate failed");
        PBErrCatch(NeuraMorphErr);

    }

    NeuraMorphFree(&nm);
    if (nm != NULL) {

        NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            NeuraMorphErr->_msg,
            "NeuraMorphFree failed");
        PBErrCatch(NeuraMorphErr);

    }

    printf("UnitTestNeuraMorphCreateFree OK\n");

}

void UnitTestNeuraMorphGetSet() {

    NeuraMorph* nm =
        NeuraMorphCreate(
            3,
            2);
    if (NMGetNbInput(nm) != 3) {

        NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
        sprintf(
            NeuraMorphErr->_msg,
            "NMGetNbInput failed");
        PBErrCatch(NeuraMorphErr);

    }

}

```



```

if (NMGetNbOutput(nm) != 2) {

    NeuraMorphErr->_type = PErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NMGetNbOutput failed");
    PErrCatch(NeuraMorphErr);

}

if (NMGetNbHidden(nm) != 0) {

    NeuraMorphErr->_type = PErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NMGetNbHidden failed");
    PErrCatch(NeuraMorphErr);

}

NMSetNbHidden(
    nm,
    5);
if (NMGetNbHidden(nm) != 5) {

    NeuraMorphErr->_type = PErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NMSetNbHidden failed");
    PErrCatch(NeuraMorphErr);

}

if (NMInputs(nm) != nm->inputs) {

    NeuraMorphErr->_type = PErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NMInputs failed");
    PErrCatch(NeuraMorphErr);

}

if (NMOutputs(nm) != nm->outputs) {

    NeuraMorphErr->_type = PErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NMOutputs failed");
    PErrCatch(NeuraMorphErr);

}

NeuraMorphFree(&nm);

printf("UnitTestNeuraMorphGetSet OK\n");

}

void UnitTestNeuraMorphAddRemoveUnit() {

```

```

VecLong3D iInputs = VecLongCreateStatic3D();
VecSet(
    &iInputs,
    0,
    0);
VecSet(
    &iInputs,
    1,
    1);
VecSet(
    &iInputs,
    2,
    2);
VecLong2D iOutputs = VecLongCreateStatic2D();
VecSet(
    &iOutputs,
    0,
    0);
VecSet(
    &iOutputs,
    1,
    1);

NeuraMorph* nm =
    NeuraMorphCreate(
        3,
        2);

NeuraMorphUnit* unit =
    NMAddUnit(
        nm,
        (VecLong*)&iInputs,
        (VecLong*)&iOutputs);

bool isSameA =
    VecIsEqual(
        &iInputs,
        unit->iInputs);
bool isSameB =
    VecIsEqual(
        &iOutputs,
        unit->iOutputs);
if (
    GSetNbElem(&(nm->units)) != 1 ||
    GSetHead(&(nm->units)) != unit ||
    isSameA == false ||
    isSameB == false) {

    NeuraMorphErr->_type = PErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NMAddUnit failed");
    PErrCatch(NeuraMorphErr);

}

NeuraMorphFree(&nm);

nm =
    NeuraMorphCreate(
        3,
        2);

```

```

unit =
    NMAddUnit(
        nm,
        (VecLong*)&iInputs,
        (VecLong*)&iOutputs);

NMRemoveUnit(
    nm,
    unit);

if (GSetNbElem(&(nm->units)) != 0) {

    NeuraMorphErr->_type = PErrTypeUnitTestFailed;
    sprintf(
        NeuraMorphErr->_msg,
        "NMRemoveUnit failed");
    PErrCatch(NeuraMorphErr);

}

NeuraMorphUnitFree(&unit);
NeuraMorphFree(&nm);

printf("UnitTestNeuraMorphAddRemoveUnit OK\n");

}

void UnitTestNeuraMorph() {

    UnitTestNeuraMorphCreateFree();
    UnitTestNeuraMorphGetSet();
    UnitTestNeuraMorphAddRemoveUnit();
    printf("UnitTestNeuraMorph OK\n");

}

void UnitTestAll() {

    UnitTestNeuraMorphUnit();
    UnitTestNeuraMorph();
    printf("UnitTestAll OK\n");

}

int main() {

    UnitTestAll();

    // Return success code
    return 0;

}

```

6 Unit tests output

```

UnitTestNeuraMorphUnitCreateFree OK
UnitTestNeuraMorphUnitGetSet OK
UnitTestNeuraMorphUnitEvaluate OK

```

```
UnitTestNeuraMorphUnit OK  
UnitTestAll OK
```