A Note on Using the CHomP C++ Library Interface

Paweł Pilarczyk

August 9, 2007

Abstract

This note provides some very basic information on how to use the C++ interface to a few functions available in the CHomP library which allow to compute the homology of cubical sets, as well as the homomorphism induced in homology by a combinatorial cubical multivalued map.

1 Introduction

There are several programming interfaces included in the CHomP library for direct access to the homology computation functions from a program written in the C++ programming language. In professional computations, this saves a lot of time that would be wasted if one first saves the data to files, and then runs the programs `chomp` or `homcubes`. Here we will introduce a few most basic interfaces.

2 The Basic CHomP Library Interface

The basic CHomP Library interface allows one to use the homology computation engines which are available in the executable program `chomp` directly from a program in C++. The set of cubes must be prepared as a bitmap (explained below). The engine can be chosen by giving its name, exactly like in the case of the `chomp` program.

2.1 Binary Bitmaps for the Basic Interface

An n-dimensional bitmap that represents a full cubical set can be stored in a memory buffer directly. Bit set to 1 means that the corresponding (hyper)cube is present in the cubical set, bit set to 0 means that the corresponding (hyper)cube does not belong to the cubical set. A rectangular area in $\mathbf{R}^d$ of the size $n_1 \times \cdots \times n_d$ is stored in $n_1 \cdots n_d$ consecutive bits of contiguous memory. For technical reasons, n_1 must be a multiple of 32 on 32-bit machines, and a multiple of 64 on 64-bit machines. Each byte represents 8 consecutive cubes, the lower bits of the

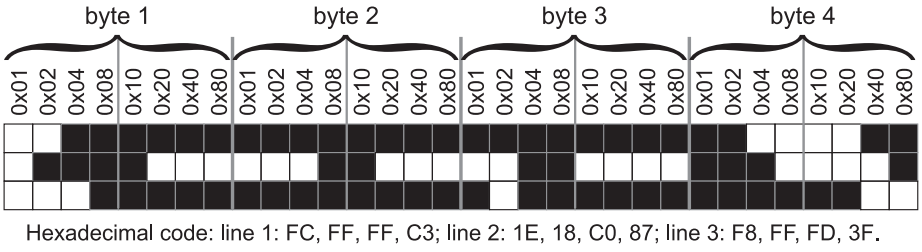

Figure 1: Sample bitmap encoded as a sequence of bytes. Bit masks for bits are indicated for each column.

byte represent cubes to the left (see Figure 1 for an illustration). The first bit in the bitmap represents cube $(0, \ldots, 0)$, the first n_1 bits represent the cubes whose first coordinate changes from 0 to $n_1 - 1$, and all the other coordinates are 0. These bits are followed by n_1 bits which represent cubes whose second coordinate is 1. The bits that represent all the cubes in the plane $\{x_3 = 0, \ldots, x_d = 0\}$ are followed by bits which represent cubes for which $x_3 = 1$, and so on.

The interface procedure for computing the homology of a full cubical set represented by means of a bitmap is as follows:

```
void ComputeBettiNumbers (const void *buffer, int *sizes, int dim,
int *result, const char *engine = 0, const int *wrapping = 0,
bool quiet = false);
```

The C++ code below illustrates a sample program that computes the homology of the bitmap depicted in Figure 1 using the function `ComputeBettiNumbers`.

Program 1 *Homology Computation of a Bitmap*

```
01 #include <iostream>
02 #include "capd/homengin/homology.h"
03
04 int main ()
05 {
06   const int dim = 2;
07   int sizes [] = {32, 3};
08   char buffer [] = {
09     '\xFC', '\xFF', '\xFF', '\xC3',
10     '\x1E', '\x18', '\xC0', '\x87',
11     '\xF8', '\xFF', '\xFD', '\x3F',
12   };
13
14   const char *engine = 0;
15   bool quiet = false;
16
17   int betti [dim + 1];
18   ComputeBettiNumbers (buffer, sizes, dim, betti, engine, quiet);
19   for (int i = 0; i <= dim; ++ i)
20     std::cout << (i ? " " : "") << betti [i];
21   std::cout << '\n';
22   return 0;
23 }
```

2.2 Cubical Sets for the Basic Interface

The class `CubicalSet` is a simple interface for creating and manipulating a bitmap representation of a full cubical set. To create an object of this class, one needs to declare the range of coordinates of cubes. For the area

$$[n_1^-, n_1^+] \times \cdots \times [n_d^-, n_d^+]$$

one must create the object `CubicalSet` and provide the numbers d, $n_1^-, \ldots, n_d^-$, $n_1^+, \ldots, n_d^+$. A cube is added to the bitmap with the function `Add`, a cube can be removed with the function `Delete`. Note that since full cubes are represented by their vertices with minimal coordinate values, the numbers $(k_1, \ldots, k_d)$ sent to these functions will satisfy the inequalities $n_i^- \leq k_i < n_i^+$. The homology of the cubical set can be computed with the function

```
void ComputeBettiNumbers (const CubicalSet &s,
int *result, const char *engine = 0, bool quiet = false);
```

An example application of this interface is illustrated by the following program:

Program 2 *Using the class* `CubicalSet`

```
01 #include <iostream>
02 #include "capd/homengin/cubiset.h"
03
04 int main ()
05 {
06   int left_coords [] = {-6, -5, 0};
07   int right_coords [] = {6, 1, 4};
08   CubicalSet Q (left_coords, right_coords, 3);
09
10   int cube1 [] = {1, -5, 0};
11   Q. Add (cube1);
12   int cube2 [] = {5, -2, 2};
13   Q. Add (cube2);
14
15   int betti [4];
16   ComputeBettiNumbers (Q, betti, "MM_CR", true);
17   for (int i = 0; i < 4; ++ i)
18     std::cout << (i ? " " : "") << betti [i];
19   std::cout << '\n';
20   return 0;
21 }
```

3 The Advanced CHomP Library Interface

The general interface to the CHomP library corresponds to using the engine `PP` in the program `chomp`. However, the functionality of the C++ interface is wider than using the pre-compiled software programs.

There are classes for representing full cubes and full cubical sets, elementary cubes (cubical cells) and cubical complexes, as well as combinatorial cubical multivalued maps. There are also functions for the homology computation of all these kinds of objects. Moreover, operators `<<` and `>>` allow to read these objects from input streams or write to output streams in the same format as used by the CHomP programs. Almost all the code is defined within the namespace `chomp::homology`.

The classes described below represent cubical sets with respect to the uniform integral lattice in $\mathbf{R}^d$. The classes are defined as templates whose parameter is the integer type to be used for storing the coordinates. By default, the type `coordinate` is used which is equivalent to `short int` and corresponds to 16-bit integers. This choice conserves memory, because in most applications there is no need to use numbers outside the range $[-32768, 32767]$. However, wider types can also be used if necessary. Please, refer to the source code for more information.

The homology groups H_q are finitely generated abelian groups. Each such group is represented as the direct sum

$$H_q \simeq \underbrace{\mathbf{Z} \oplus \cdots \oplus \mathbf{Z}}_{\beta_q} \oplus \mathbf{Z}_{p_1} \oplus \cdots \oplus \mathbf{Z}_{p_k}. \tag{1}$$

In the library interface to the CHomP software, the sequence of homology groups $(H_0, \ldots, H_n)$ is represented by an array of chains. The coefficient 1 in the chain corresponds to one group $\mathbf{Z}$ in (1); the number of such coefficients is the q-th Betti number β_q. The coefficients larger than 1 correspond to the torsion groups $\mathbf{Z}_{p_i}$ in (1). In fact, the user of the CHomP library doesn't need to know the actual structure of these chains or to access them directly, because convenient functions are available for extracting the Betti numbers (`BettiNumber`) and the torsion coefficients (`TorsionCoefficient`).

3.1 Full Cubes

The class `Cube` is used to represent a full cube in $\mathbf{R}^d$. The class has a constructor from an array of coordinates and the dimension of the cube. The class method `coord` extracts these coordinates to an array, and the method `dim` returns the dimension of the cube.

A set of cubes is represented by the class `SetOfCubes`. A cube can be added to this class with the method `add`, removed with the method `remove`. The verification of whether a given cube already belongs to the cubical set is very fast, as it uses the *hashing* technique; it is done by the method `check`.

The homology groups of a full cubical set X or the relative homology groups of the pair (X, A) is computed by the function `Homology`, as illustrated in Pro-

gram 3. By the way, this program is equivalent in what it does to Program 2. Note that in the CHomP library there are several functions with the same name `Homology` but different arguments, as it will be shown in the next subsection.

Program 3 *Homology computation with the Advanced CHomP Interface.*

```
01 #include <iostream>
02 #include "chomp/homology/homology.h"
03
04 using namespace chomp::homology;
05
06 int main ()
07 {
08   coordinate coords1 [] = {1, -5, 0};
09   Cube Q (coords1, 3);
10   SetOfCubes S;
11   S. add (Q);
12   coordinate coords2 [] = {5, -2, 2};
13   S. add (Cube (coords2, 3));
14
15   Chain *hom = 0;
16   int maxLevel = Homology (S, "S", hom);
17   for (int q = 0; q <= maxLevel; ++ q)
18     std::cout << (q ? " " : "") << BettiNumber (hom [q]);
19   std::cout << '\n';
20   delete [] hom;
21   return 0;
22 }
```

It is also possible to compute the homology generators. For that purpose, another function `Homology` should be used, which in addition to homology groups also saves the computed generators in terms of chains and related cubical cells. This will be discussed in the case of cubical cells in the next subsection, for full cubical sets it is very similar.

For computing relative homology of (X, A), an alternative function `Homology` must be used which takes the set of cubes `A` as an additional argument.

3.2 Cubical Cells

The class `CubicalCell` is used to represent an elementary cube in $\mathbf{R}^d$. It has a constructor from an object of the class `Cube` which creates a cubical cell that corresponds to the full cube, a constructor from two arrays of coordinates which creates a cell with given opposite corners (one with minimal, and the other with maximal coordinates), and a constructor of a boundary cell.

A cellular complex based on on cubical cells is represented by the class `CubicalComplex`. A cubical cell can be added to this complex by the method `add`. The dimension of the complex is obtained by the method `dim`. The set of cells of the given dimension is retrieved by using the operator `[]`. For the purpose of homology computation, all the faces of cells that belong to the cellular complex are automatically added during the computation, so there is no need to generate them beforehand and add to the complex explicitly.

The following example program shows how to create a simple cellular complex and how to compute its homology. Additionally, the homology generators

are also computed. This program uses standard CHomP Library interface routines to show the homology groups to the standard output stream, as well as the comptued homology generators. Note that each homology generator is a chain which can be understood as a formal combination of cubical cells with integral coefficients.

Program 4 *Computation of homology and generators of a cubical complex.*

```
01 #include <iostream>
02 #include "chomp/homology/homology.h"
03
04 using namespace chomp::homology;
05
06 int main ()
07 {
08   coordinate left [] = {1, 2, 0};
09   coordinate right [] = {2, 2, 1};
10   CubicalCell Q (left, right, 3);
11   CubicalComplex C;
12   C. add (Q);
13
14   Chain *hom = 0;
15   Chain **gen = 0;
16   int maxLevel = Homology (C, "C", hom, &gen);
17   ShowHomology (hom, maxLevel);
18   ShowGenerators (gen, hom, maxLevel);
19
20   delete [] hom;
21   for (int i = 0; i <= maxLevel; ++ i)
22     delete (gen [i]);
23   delete [] gen;
24   return 0;
25 }
```

For computing relative homology of the pair of cellular complexes (X, A), an alternative function `Homology` must be used which takes the cubical complex `A` as an additional argument.

3.3 Combinatorial Cubical Multivalued Maps

The class `CubicalMap` is used to represent a combinatorial cubical multivalued map. The domain of this map is a set of full cubes. The image of each cube is also a set of full cubes. If `Q` is an object of class `Cube` and `F` is an object of class `CubicalMap`, then `F [Q]` is a set of cubes of type `SetOfCubes` which corresponds to the image of `Q`. One can use the method `add` of a set of cubes to add cubes to the image of `Q`. Alternatively, one can use the assignment operator `=` to define the images of consecutive cubes.

The homomorphism induced in homology by a combinatorial cubical multivalued map is computed with the function `Homology`, whose use is illustrated in Program 5. Note that also this function is capable of extracting homology generators to additional data structures, like in Program 4, but we skip this feature here for the simplicity of presentation.

Program 5 *Computation of the homomorphism induced in homology.*

```
01 #include <iostream>
02 #include "chomp/homology/homology.h"
03
04 using namespace chomp::homology;
05
06 int main ()
07 {
08   coordinate c1 [] = {1, 1, 0}, c2 [] = {1, 1, 1};
09   CubicalMap F;
10   SetOfCubes S;
11   S. add (Cube (c1, 3));
12   S. add (Cube (c2, 3));
13   Cube Q (c1, 3);
14   F [Q] = S;
15
16   SetOfCubes X = F. getdomain (), A, Y = S, B;
17   Chain *homX = 0, *homY = 0;
18   ChainMap *homF = 0;
19   int maxX = 0, maxY = 0;
20   Homology (F, X, A, Y, B, homX, maxX, homY, maxY, homF);
21
22   delete homF;
23   delete [] homX;
24   delete [] homY;
25   return 0;
26 }
```