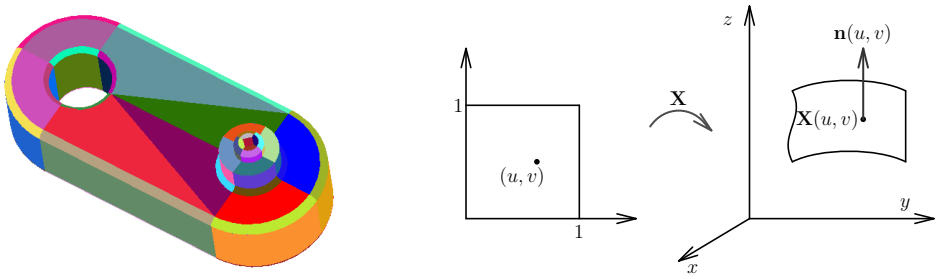
The ITL programming interface toolkit (API functions Version 0.9.1)

Maharavo Randrianarivony

1 Introduction

This document serves as a reference for the beta version of our evaluation library ITL. There are two main objectives for the ITL toolkit. First, it describes a library which gives an easy way for programmers to evaluate the 3D image and the normal vector corresponding to a parameter value (u, v) which belongs to the unit square as illustrated in Fig. 1(b). The API functions which are described in this document let programmers make those evaluations without the need to understand the underlying CAD complications. As a consequence, programmers can concentrate on their own scientific interests. Our second objective is to describe the input which is a set of parametric four-sided surfaces that have the structure required by some integral equation solvers as in Fig. 1(a). Note that the task of this toolkit is not to split a set of CAD surfaces into four-sided subsurfaces. Such a task can be done by another program whose output is used by this toolkit. We will show four sample programs for illustrating the use of the API functions. Note that this toolkit has been completely written in ANSI C. Therefore, the illustrating samples are also written in C. In later versions, we might provide FORTRAN examples (if needed).



(a) (b)

Figure 1: (a)Input: foursided surfaces (b)Evaluation of image and normal vector

2 The API functions

We will describe in this section a set of API functions, which we call henceforth ITL functions, that programmers can use and link in their programs. All of those functions have the prefix `itl_` in order to recognize them. In order that one can use the following ITL functions, the short header file `"itl.h"` must be included. A complete listing of that header file can be found toward the end of this document. As a general rule, the content of the program which uses ITL functions must be included between the functions `itl_start_session()` and `itl_terminate_session()`.

2.1 itl_compute_image()

Usage:

```
point3D itl_compute_image(int s,double u,double v);
s=index of the considered patch
u=first coordinate of a parameter from the unit square
v=second coordinate of a parameter from the unit square
```

Description:

Suppose we have as input a set of N foursided patches which are the images of the unit square by parametric functions $\mathbf{X}_i(\cdot,\cdot)$ $(i = 0, ..., N-1)$. The function `itl_compute_image()` considers the four-sided patch which is identified by the index in the first argument. Then, it computes the 3D image of the parameter variable $(u, v) \in [0, 1] \times [0, 1]$. If the value of u or v resides outside the interval $[0, 1]$, then $[0, 0, 0]$ is returned by the function.

2.2 itl_compute_normal()

Usage:

```
vector3D itl_compute_normal(int s,double u,double v,int mode);
s=index of the considered patch
u=first coordinate of a parameter from the unit square
v=second coordinate of a parameter from the unit square
mode=method of computing the normal
```

Description:

This function is similar to `itl_compute_image()` but it computes now the

normal vector corresponding to the parameter value (u, v)

$$\mathbf{n}(u, v) = \partial_u \mathbf{X}_s(u, v) \times \partial_v \mathbf{X}_s(u, v). \tag{1}$$

Although the set of foursided surfaces could describe the boundary of a solid, the normal vector could be pointed inward or outward according to the parametrization. The value of the mode variable in the fourth argument of the function could be one the following macros:

RAW_NORMAL_VEC

UNIT_NORMAL_VEC

depending on whether we obtain a plain or a unit normal vector. If the paremeter (u, v) is not located inside the unit square, then the zero vector is returned.

2.3 itl_start_session()

Usage:

```
void itl_start_session(char *nm);
```

`nm=string containing the file name`

Description:

This function opens the ITL file which is specified by the string nm. It automatically determines the format of the file which could be binary or ascii. First, the file is loaded in a self-expanding buffer. Afterwards, the content of the file is loaded inside an implicit ITL object. This function takes care also of every required memory allocation to store the geometric information. Some message error is displayed if the ITL file is absent, wrong or incomplete and the program is terminated. Before calling any other ITL function, this function must already be executed. The file is closed by `itl_start_session()` when all required geometric information is loaded. The member variables of an ITL object are seen below:

```
1 typedef struct itl_object{
2       int nb_four_reg;        //number of four-sided regions
3       int *bel;               //parent trimmed surface
4       patch *surf;            //list of four-sided patches
5       FILE *fp;               //file storing the ITL object
6       char *file_name;        //file name of the ITL object
7       int storage_type;       //binary or ascii file
8       int *eval_type;         //use Coons or Gordon
9 }itl_object;
```

Since our primary goal is simplicity and prevention of complicated tasks for programmers using this toolkit, we have taken care of all dynamic memory managements required to store geometric information.

2.4 itl_terminate_session()

Usage:

```
void itl_terminate_session();
```

Description:

The purpose of this routine is to terminate the session of one ITL file. Its main function is to release all memories which have been allocated implicitly with `itl_start_session()` and to destroy the internal ITL object. Note that the closing of the file is already done by `itl_start_session()`. Thus, we need only handle the internal ITL object.

2.5 itl_number_patches()

Usage:

```
int itl_number_patches();
```

Description:

This function returns the number of four-sided patches of the geometry which is contained in the input file.

2.6 itl_parent_face()

Usage:

```
int itl_parent_face(int s)

s=index of the considered patch
```

Description:

Note that the surface of a CAD object generally consists of several multiply connected surfaces which we call faces. Each face has then several four-sided subsurfaces which we call patch. The function `itl_parent_face()` determines the parent face of the four-sided patch whose index `s` is specified in the argument. That is, we determine the face to which the s-th four-sided patch belongs. This function demonstrates itself useful when we want to visualize the current geometric model where every face should have its own drawing color.

2.7 itl_number_faces()

Usage:

```
int itl_number_faces();
```

Description:

This function determines the number of trimmed surfaces (faces) of the geometry contained in the current file.

2.8 itl_face_members()

Usage:

```
int itl_face_members(int s,int *list);

s=index of the considered face

list[*]=set of member patches
```

Description:

Consider the s-th face which is specified by the first argument. This function determines the list of patches which are derived from the face. Afterwards, it stores the result in the array list[*] and returns the length of the list. In many situations, the patches of all faces are arranged in lexicographical order. But it is not always the case. In fact, this is the opposite of the function `itl_parent_face()`.

2.9 itl_save_file()

Usage:

```
void itl_save_file(char * nm,int mode);

nm=name of the file to store the ITL file

mode=method of saving the file
```

Description:

Use `itl_save_file()` in order to store the current ITL object in the file which is specified by the string nm. For the value of the second variable mode, we can choose from one of the following predefined macros.

```
ASCII_VERBOSE_FORMAT

ASCII_SILENT_FORMAT

BINARY_LITTLE_ENDIAN_FORMAT

BINARY_BIG_ENDIAN_FORMAT
```

For ascii mode, we may choose between verbose and silent structure. If verbose is chosen, some alphabetical keywords are written in the content of the output file and it is easy for the reader to read it with a simple editor. Otherwise, almost only numerical data are written except in the

header section of the output file. For binary output, we may choose between little-endian or big-endian. The output file of a certain chosen endianess is independent of the native endianess of the current machine. This routine is for example useful when we want to convert an ITL file into another format.

3 The input files

The property of the input object can be summarized as bellow:

1. Four-sided surfaces,
2. No hanging nodes: the intersection of two different four-sided surfaces is either a complete side or a corner point,
3. Use of transfinite interpolation for the mappings,

(4). Globally continuous.

We put the fourth point inside parentheses in this beta version because we cannot yet always guarantee it without additional assumption. We call the input file which stores the above information an ITL file.

Before describing that file, let us discuss the reason why we want to introduce that type of file. The set of patches which are used in this library is the result of another program `integralCAD` which has two main tasks. First, it splits a CAD object into four-sided surfaces. Second, it searches for a diffeomorphic function from the unit square onto each foursided patch (Fig. 2(b)). Usually, Coons patch is enough to describe the second task. But sometimes we have to use Gordon patches.

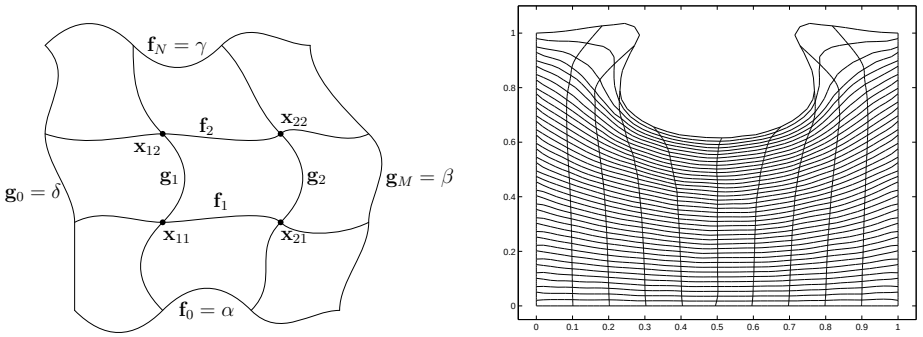
Let us recall the Coons patch in matrix form as:

$$\begin{aligned}\mathbf{X}(u,v) &= \begin{bmatrix} F_0(u) & F_1(u) \end{bmatrix} \begin{bmatrix} \delta(v) \\ \beta(v) \end{bmatrix} + \\ &\quad \begin{bmatrix} \alpha(u) & \gamma(u) \end{bmatrix} \begin{bmatrix} F_0(v) \\ F_1(v) \end{bmatrix} - \\ &\quad \begin{bmatrix} F_0(u) & F_1(u) \end{bmatrix} \begin{bmatrix} \alpha(0) & \gamma(0) \\ \alpha(1) & \gamma(1) \end{bmatrix} \begin{bmatrix} F_0(v) \\ F_1(v) \end{bmatrix}. \end{aligned} \tag{2}$$

The Gordon patch which is defined as

$$\mathbf{X}(u,v) := \sum_{i=0}^{M} \mathbf{g}_i(v)\varphi_i(u) + \sum_{j=0}^{N} \mathbf{f}_j(u)\psi_j(v) - \sum_{i=0}^{M}\sum_{j=0}^{N} \mathbf{x}_{ij}\varphi_i(u)\psi_j(v) \tag{3}$$

requires the knowledge of some curves f_j and g_i inside the four-sided domain as in Fig. 2(a).



(a) (b)

Figure 2: (a)A network of curves for Gordon patch (b)Diffeomorphism onto a four-sided domain

Several CAD interfaces do not support transfinite interpolations. That is, they do not have specified entities for Coons or Gordon patches. As a consequence, if we use those CAD interfaces then we have to recompute the internal curves g_i and f_j of equation (3). Such a task which has been done by `integralCAD` and which could be computationally expensive does not need to be repeated. Therefore, we store the equations of the internal curves g_i and f_j in the ITL file. Another reason for using the ITL file is its simplicity: it is very much adapted with our implicit data structure and the file is sequentially written. You do not need to make any jump or line search to find relevant information.

Input files are provided in ascii or binary modes. The use of binary formats has the advantage of smaller size and speed of input/output. But text formats might be more advantageous for new users who want to inspect the content of the files. For ascii mode, we may choose between *verbose* and *silent* option. For verbose option, some alphabetical keywords can be found in the ascii ITL file and it is easy for the reader to investigate the file. For silent option only numerical data can be found in the content section of the ascii ITL file. As for binary ITL files, little-endian or big-endian byte arrangements are available. Our program is neutral in terms of endianess in the sense that we store the same file of a certain endianess irrespective of the endianess of the system of the computer. Our implementation computes automatically the endianess of the current machine and computes byte reordering accordingly. In other words, for file access or file export, we determine first if we use MSB (Most Significant Byte) or LSB (Least Significant Byte) byte organization. To simplify things, we deal only with double precision floating number `double` (8 bytes), integer `int` (4 bytes) and `char` (1 byte). All other built-in data types are not handled for now.

An ITL file has two sections: the *header* section and the *content* section. The header section consists of some human readable introductory texts. The first line consisting of the string "itl file" identifies that we deal with an ITL file. The beginning of the header file is recognized by the keyword START_HEADER. We see in the header section the number of four-sided patches and the format with which the data is stored. The following four keywords are used to identify those formats:

ascii_verbose,

ascii_silent,

binary_little_endian,

binary_big_endian.

The end of the header section is specified by the keyword TERMINATE_HEADER. An example of the header section would look as follows:

```
1 itl file
2 START_HEADER
3 itl file type: ascii_verbose
4 number of four-sided surfaces: 46
5 #illustration of header
6 TERMINATE_HEADER
```

Everything after the symbol # is supposed to be a comment for the human reader. That is, the ITL parser ignores every line starting by #. A single space and multiple spaces are supposed to be the same. Thus, between two data there could only be space(s) or carriage return.

Usually the ITL files have suffix itl but that is is not mandatory. If the suffix is not present but the content of the file is correct, we can still use the file without change. But if the content is wrong, the itl suffix does not help. The suffix is exclusively meant for human users and not for the machines. We have chosen the labeling ITL which stands for *integral* because this toolkit is intended to be used for integral equations which is our primary interest. However, we believe that this format is general enough to be used for other programs which require similar geometric information in patch form.

In the next section, we will display several sample program fragments in order to illustrate the use of the ITL functions which we have described in section 2. The programs have no particular important applications but they serve as good examples.

4 Sample program 1 (organization and evaluation)

In the next simple program, we show a complete minimal routine for the computation of the 3D image and normal vector corresponding to a given parameter value (u, v) from the unit square.

```
1  #include <stdio.h>
2  #include "itl.h"
3
4  void main()
5  {int s;
6  double u,v;
7  point3D image;
8  vector3D nml;
9  itl_start_session("sample.itl");
10 printf("patch="); scanf("%d",&s);
11 printf("u="); scanf("%lf",&u);
12 printf("v="); scanf("%lf",&v);
13 image=itl_compute_image(s,u,v);
14 printf("image=[%f,%f,%f]\n",image.absi,image.ordo,image.cote);
15 nml=itl_compute_normal(s,u,v,UNIT_NORMAL_VEC);
16 printf("normal=[%f,%f,%f]\n",nml.absi,nml.ordo,nml.cote);
17 itl_terminate_session();
18 }
```

Let us take a quick look at what this program does. First, we have to include the header file `"itl.h"` so that we can use the ITL functions in our program. The input is loaded from the file `"sample.itl"` with which we implicitly generate an ITL object. Afterwards, the user is asked to specify a patch index s and a parameter value (u, v) belonging to the unit square. With the help of the ITL functions `itl_compute_image()` and `itl_compute_normal()`, we determine the corresponding 3D image and unit normal vector. As probably already noticed by the reader, the coordinates of the point and the components of the vector are stored in the following data structure:

```
1  typedef struct point__{
2        double absi;
3        double ordo;
4        double cote;
5  }point3D,vector3D;
```

Note that in our implementation, the self-expanding memory for the ITL object is very tight. We have only allocated an amount of memory which is exactly what is needed to load the object. As a consequence, a good memory management between `itl_start_session()` and `itl_terminate_session()` is essential. It is possible that a program having inaccurate memory organization between those two keywords does not terminate well. As an illustration, `itl_terminate_session()` might not function properly in the

following code. That is because we only allocate four `int`'s for the variable `z`. Even worse, the value of the 3D point `w` might be awry because of memory conflict.

```
1  void counter_example()
2  {int *z;
3  point w;
4  itl_start_session("sample.itl");
5  z=(int *)malloc(4*sizeof(int));
6  ...
7  z[4]=56;//access violation
8  w=itl_evaluate(0,0.5,0.5);
9  itl_terminate_session();
10 free(z);
11 }
```

It is worth being mentioned that files can be opened consecutively but the current ITL object must be closed before opening another one. That is, we must invoke `itl_terminate_session()` before opening another file. That situation is illustrated in the following code fragment.

```
1  void example()
2  {itl_start_session("sample1.itl");
3  ...//code using sample1
4  itl_terminate_session();//closing session for sample1
5  ...
6  itl_start_session("sample2.itl");
7  ...//code using sample2
8  itl_terminate_session();//closing session for sample2
9  ...
10 }
```

The following counter-example demonstrates an undesired situation where the session with the first file "`sample1.itl`" from line 2 is not yet closed before opening a new session with the file "`sample2.itl`" from line 4.

```
1  void counter_example()
2  {itl_start_session("sample1.itl");
3  ...
4  itl_start_session("sample2.itl");
5  ...
6  itl_terminate_session();//closing session for sample2
7  ...
8  itl_terminate_session();//closing session for sample1
9  ...
10 }
```

5 Sample program 2 (format conversion)

The next code takes an ITL file as input and it converts that file into a binary little endian ITL file without modifying the stored geometric information. If by chance the input file happens to be already a binary file of little endian type, then `itl_save_file()` simply copies it.

```
1  void convertion(char *input_itl,char *output_itl)
2  {itl_start_session(input_itl);
3  itl_save_file(output_itl,BINARY_LITTLE_ENDIAN_FORMAT);
4  itl_terminate_session();
5  }
```

6 Sample program 3 (cloud of points)

Let us consider a code fragment for generating a random cloud of points. In fact, we want to create m points for each four-sided patch. We then display the result in the standard output.

```
1  #include <stdlib.h>
2  #include "itl.h"
3
4  // Generates a random number of value in [0,1]
5  double random_unit()
6  {int rd;
7  double res;
8  rd=1+(int) (100.0*rand()/(RAND_MAX+1.0));
9  res=(double)rd/100.0;
10 return res;
11 }
12
13 void cloud_points(int m)
14 {int N,s,j,k;
15 double u,v;
16 point3D *cloud;
17 itl_start_session("sample.itl");
18 N=itl_number_patches();
19 cloud=(point3D *)malloc(N*m*sizeof(point3D));
20 k=0;
21 for(s=0;s<N;s++)
22 for(j=0;j<m;j++)
23       {u=random_unit();
24       v=random_unit();
25       cloud[k]=itl_compute_image(s,u,v);
26       printf("cloud[%d]=[%f,%f,%f]\n",k,cloud[k].absi,
27       cloud[k].ordo,cloud[k].cote);
28       k++;
```

```
29      }
30  free(cloud);
31  itl_terminate_session();
32  }
```

7 Sample program 4 (patch inventory)

The next program illustrates the determination of entity numbers: number of patches, number of faces. Additionally, it shows how the memory for them should be allocated/released.

```
1  void patch_invetory()
2  {int N,m,i,j,p,*list;
3  itl_start_session("sample.itl");
4  m=itl_number_faces();
5  N=itl_number_patches();
6  list=(int *)malloc(N*sizeof(int));//worst case
7  for(i=0;i<m;i++)
8       {p=itl_face_members(i,list);
9       printf("Face=%d  number_of_its_patches=%d\n",i,p);
10      for(j=0;j<p;j++)
11           printf("\tpatch=%d\n",list[j]);
12      }
13 free(list);
14 itl_terminate_session();
15 }
```

8 Listing of the header file "itl.h"

```
1  #ifndef __ITL_H__
2  #define __ITL_H__
3
4  #if defined DLL_EXPORT
5  #define ITL_API __declspec(dllexport)
6  #else
7  #define ITL_API __declspec(dllimport)
8  #endif
9
10 typedef ITL_API struct point__{
11      double absi;
12      double ordo;
13      double cote;
14 }point3D,vector3D;
15
16 #define    ASCII_VERBOSE_FORMAT              100
```

```
17 #define    ASCII_SILENT_FORMAT              101
18 #define    BINARY_LITTLE_ENDIAN_FORMAT      102
19 #define    BINARY_BIG_ENDIAN_FORMAT         103
20 #define    UNIT_NORMAL_VEC                  104
21 #define    RAW_NORMAL_VEC                   105
22
23 #ifdef __cplusplus
24 extern "C"
25 {
26 #endif
27
28 ITL_API  void       itl_start_session(char *);
29 ITL_API  void       itl_terminate_session();
30 ITL_API  int        itl_number_patches();
31 ITL_API  int        itl_number_faces();
32 ITL_API  int        itl_face_members(int,int *);
33 ITL_API  int        itl_parent_face(int);
34 ITL_API  point3D    itl_compute_image(int,double,double);
35 ITL_API  vector3D   itl_compute_normal(int,double,double,int);
36 ITL_API  void       itl_save_file(char *,int);
37
38 #ifdef __cplusplus
39 }
40 #endif
41
42 #pragma comment(lib,"itl.lib")
43
44 #endif //__ITL_H__
```

9 Shipment and AS IS statement

We would like to discuss briefly about the items which are shipped in the setup. We ship the header file "`itl.h`" and the dynamic link library "`itl.dll`" in order to enable compilation with your routines. Those are enough to compile a program under windows using Visual Developper Studio. Additionally the setup program contains a few simple ITL files for testing purposes. The above code examples are also included in the package in form of developper studio project file as pattern. For Linux users, the static library "`itl.a`" will be provided in the next release. You might need to retouch the accompanying Makefiles so that they are adapted with your systems.

This library is part of the result of the CAGD research that I have done with Prof. Brunnett (University of Chemnitz). This documents and the accompanying library are provided "as is", without any warranty, expressed or implied.