VSG Technical Report
OSMIA-01

Project	OSMIA: Open Source Medical Image Analysis EU Fifth Framework Programme (IST: Accompanying Measures)
Project number	IST-2001-34512
Deliverable number	D4.1 (i)
Revision	2
Report Title	Integration of TINA C libraries and the NeatVision Development Environment
Associated Software	aorta_tracker_example.tar.gz example.tar.gz seq_reader_example.tar.gz
Prepared by	Dr. Naser Prljaca
Approved / Edited by	Prof. Paul F Whelan
Distribution List	OSMIA project members
Date	June 2002

Contents

1. Motivation

TINA [1] has been written to provide a research environment for the machine vision software developers. It consists of a powerful set of C libraries to support algorithms and software development such as: basic system libraries, math libraries, image/geometry processing libraries and user interface and interactive graphics libraries under X Windows system.

<u>TINA assessment</u>

Our experiences in using the TINA open source software can be summarised as follows:

- TINA distribution package and installation instructions are good. We were able to install and run TINA without any problem under SUN Solaris
- Distribution and installation of TINA high level applications such as medical and machine vision applications went smoothly under SUN Solaris.
- The installation and compilation of TINA libraries under Linux (Red Hat) was successful accompanied by numerous compiler warnings
- The `tinatool` template application works but for some reason cannot close
- Some of the TINA medical and machine vision high-level applications do not work properly (ie they crash or do not work as expected) on Linux (e.g. Image co registration tool, Stereo tool)
- We have also noticed some conflicting function declarations and function definitions (for example `pgh_model.c` is conflicting with `pgh_funcs.h`)
- TINA templates are very useful for quick start up
- In general, distribution bundles and accompanying installation files are informative enough for new users to have TINA up and running on their machines
- TINA user's guide is useful
- TINA programmer's guide is modest
- TINA algorithmic guide is useful
- TINA source libraries are organised and structured well taking into account overall TINA architecture
- Unfortunately the source code is not well documented
- As a conclusion, the key disadvantage associated with learning and using TINA is lack of suitable documentation regarding TINA functions, data structures and algorithms

On the other hand the NeatVision [2] has been written in Java as a user-friendly package to support teaching and development of machine vision techniques and algorithms. The system support intuitive visual programming and is easily extendable by the end user developer

This work attempts to expose some of the TINA functionality to the NeatVision development environment [3]. Effectively this task can be reduced to the problem of calling C functions from Java.

2. The Java Native Interface

The *Java Native Interface* (JNI), which comes as part of the *Java Development Kit* (JDK), gives compile- and run-time support enable developers to call *native code* from a Java program. By *native code*, we mean non-Java code, typically C or C++; in this report we will assume C.

At compile time, the JNI defines the way that Java data types correspond to C data types – C programs get this information from JNI header files that come with the JDK. A tool, *javah* (supplied with the JDK) aids in creating application-specific header files which aim to eliminate mistakes in communication between Java and C routines.

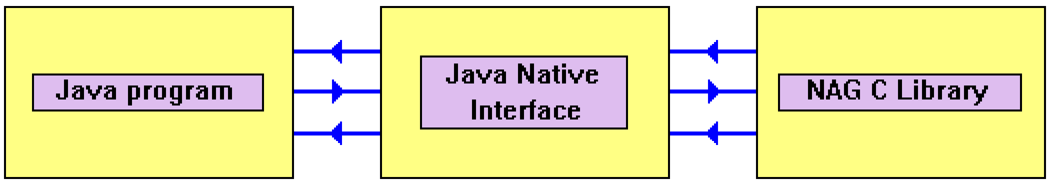
At run time, the JNI allows the passing of Java objects to the C code, and allows the C code access to Java *properties* and *methods*. Thus, for example, the C code can set properties of Java classes, and it is possible to call a Java method from C.

A good introduction to the Java Native Interface can be found at the Sun Microsystems Java Tutorial web site [4].

3. Use of an interface library

Use of the JNI entails creation of an intermediate shared library (on UNIX systems) or DLL (Microsoft Windows systems). This library acts as the interface between the Java code and the TINA C Library code.

The interface library is required because when a native method (i.e. *function* or *subroutine*) is called from Java, extra arguments are appended to the argument list of the native routine being called. These extra arguments give the native code access to Java methods and properties, but of course the TINA C Library was not designed to handle these arguments. Furthermore, the types of the arguments passed from Java do not always correspond exactly to standard C types, and so the TINA C Library cannot use them directly. The interface library must handle these issues, make its own calls to the TINA Library, and then send the results back to Java.



The Java Native Interface as a link between Java and C libraries

Implementation of a call from Java to the TINA C Library is a three-stage process

- Write a declaration, in Java, for the native method. This declaration will include the keyword *native* to signify to the Java compiler that it will be implemented externally.

- Create a header file for use by the native (C) code. This header file contains the declaration of the native method as viewed by the C compiler, i.e. it includes the extra arguments required for the C function to access Java methods and properties, and also has argument types defined in terms of standard C types.
- Implement the native method in C. This function will use the header file created above, make calls to the TINA C Library and possibly back to Java methods, and return results to Java. The C code is compiled to build the interface library.

When the interface library has been built, the Java code that uses it is still machine-independent even though the interface library is not. Thus, we need to build the interface library on all platforms that we are interested in, but we do not have to edit or rebuild the Java code that uses it.

4. Function prototype from the TINA C Library (Example)

The process of creating an interface library is most easily understood by demonstration. Here, we give an example.

According to the TINA C Library Manual, the prototype function is illustrated below:

```
#include <tina/all_tina.h>
Vector *x  matrix_solve(Matrix *A , Vector *b);
```

The function is designed to solve the set of *n* linear equations *A x* = *b*, where *A* is an *n* by *n* matrix, and *b* and *x* are vectors of length *n*.   The matrix *A* an vectors *b* and *x* are stored in Tina user defined data types Matrix and Vector respectively.

5. Declaring the native function in our Java program

In our Java program, we will declare the function as follows:

```
 // Declaration of the Native (C) function
  private native  void solve_equations (int n,
  double A[], double b[], double x[]);
```

The reason we declared the new C function in our Java program instead of the function declared in Tina C library is due to the fact that Java does not recognise C user defined data types.

Although the matrix *A* is two dimensional, we choose to store it in a one-dimensional Java array of type double[]. This makes the Java code slightly harder to read than it would be otherwise, because we need to deal with array subscripting, but it makes the C code that we need to write in the interface library much simpler.

6. Compiling the Java program

Here is the complete source code of our Java program tina.java

```
public class tina
{
      //Declaration of the native C function
      private native  void   solve_equations(int n,
      double  aa[], double bb[], double xx[]);
      static
      {
            System.loadLibrary("JCtinamath");
      }
      public static void main(String [] args)
      {
            double aa[], bb[], xx[];
            int n = 10;
            tina lineq = new tina();
            aa = new double[n*n];
            bb = new double[n];
            xx = new double[n];
            int k;
            for(k = 0; k < n; k++) {
            aa[k*(n+1)] = 4.0;
            bb[k] = -1.0;
            }
            System.out.println("Call of TINA linear equation
          solver routine");
            lineq.solve_equations(n, aa, bb, xx);
            int i;
            for(i = 0; i < n ; i++) {
                  System.out.println(" " + xx[i]);
            }
    }
}
```

We compile our Java program with the command

```
% javac tina.java
```

7. Generating a header file for use by C

Having compiled tina.java, we can use the *javah* utility to create a C header file:

```
  % javah -jni  tina
```

The generated header file, tina.h, contains this function prototype:

```
 JNIEXPORT void  JNICALL Java_tina_solve_equations
    (JNIEnv *, jobject, jint, jdoubleArray,jdoubleArray,
     jdoubleArray);
```

As before, from the C point of view, our function has two extra arguments: the Java environment pointer and the Java object.

8. Implementing the native function in C code

Now that we have created the header file tina.h, we can write our C code implementation of solve_equations as follows:

```
#include <jni.h>
#include "tina.h"
#include <tina/all_tina.h>


JNIEXPORT void JNICALL Java_tina_solve_equations
  (JNIEnv *env, jobject obj, jint n, jdoubleArray aa, jdoubleArray
bb, jdoubleArray xx)

{
 Matrix *a;
  double **c;
  Vector *b, *x;
  double *d;
  int i, j;

  jdouble *apt;
  jdouble *bpt;
  jdouble *xpt;


  /*JNI data transfer calls */

  apt = (*env)->GetDoubleArrayElements(env, aa, 0);
  bpt = (*env)->GetDoubleArrayElements(env, bb, 0);
  xpt = (*env)->GetDoubleArrayElements(env, xx, 0);

  /*internal Tina data operation */

  a = (Matrix *)matrix_alloc(n, n, matrix_full,
       double_v);
  c = a->el.double_v;
  b = vector_alloc(n, double_v);
  d = b->data;

  for(i = 0; i < n; i++) {
          for(j = 0; j < n; j++) {
                  c[i][j] = apt[i*n +j];
                  d[i] = bpt[i];
          }
  }

/* call to Tina function */

  x = matrix_solve(a, b);
  d = x->data;

  for(i = 0; i < n; i++) xpt[i] = d[i];


  /*End of Tina operation */
  /* JNI data transfer calls */

  (*env)->ReleaseDoubleArrayElements(env,aa, apt,0);
  (*env)->ReleaseDoubleArrayElements(env,bb, bpt,0);
  (*env)->ReleaseDoubleArrayElements(env,xx, xpt,0);

  /* JNI end of transfer*/
}
```

Points to note:

- As before, we must include the appropriate Tina C Library header files
- We cannot access the elements of array arguments *aa*, *bb* and *xx* directly, because they are not C-style arrays but rather Java-style arrays of type *jdoubleArray*. Trying to access the array elements directly would lead to catastrophe. Instead, we must convert them to C-style double arrays, using the JNI function *GetDoubleArrayElements*. This function is declared in the JNI header file jni.h as follows:

  ```
  jdouble * (JNICALL *GetDoubleArrayElements)
  (JNIEnv *env, jdoubleArray array, jboolean
  *isCopy);
  ```

  GetDoubleArrayElements is accessed through the *JNIEnv* pointer, **env*. Given the array of type *jdoubleArray*, it returns a pointer to an array of elements of type *jdouble*, which can safely be manipulated by C. The output argument *isCopy* tells us whether Java made a copy of the array, or just passed us a pointer to the elements *in situ*. This is not of interest to us at this stage.

  Our C program therefore makes three calls of *GetDoubleArrayElements*, one for each array argument. The returned pointers are passed directly to the C function solve_equations which in turn calls Tina C library function matrix_solve after argument preparations.

- After return from the Tina Library, we need to tell Java that we are finished with the array pointers that it gave us. We therefore make three calls of function *ReleaseDoubleArrayElements*, declared in *jni.h* as

```
void (JNICALL *ReleaseDoubleArrayElements)
   (JNIEnv *env, jdoubleArray array, jdouble *elems,
   jint mode);
```

  We need to do this for two reasons: to ensure that our results get copied back to the appropriate Java arrays, and so that Java garbage collection can work properly (if we did not do it, Java might leak the memory that it allocated for us).

9. Building the shareable library or DLL

This step is operating-system dependent.

Building on Linux (Solaris)

```
% gcc -c -I/usr/java/jdk1.3.1_03/include  \
  -I/usr/java/jdk1.3.1_03/include/linux \
  -I/home/prljacan/Tina/include  tina.c

% ld -G  tina.o -o \
  libJCtinamath.so \
  -L/home/prljacan/Tina/lib-linux –ltinamath –ltinasys\
  –lc –lm
```

10. Running the program

Assuming that all has gone well, we can run the program using the command

```
% java tina
```

11. Quick summary of how to build the Tina linear equation solver

Given the two source files tina.java and tina.c issue the following commands:

Compile the Java class:

```
% javac tina.java
```

Create the header file:

```
 % javah -jni tina
```

Compile and build interface library:
(Linux/Solaris)

```
% gcc -c -I/usr/java/jdk1.3.1_03/include - \
          I/usr/java/jdk1.3.1_03/include/linux \
          -I/home/prljacan/Tina/include  tina.c

% ld -G tina.o -o libJCtinamath.so \
            /home/prljacan/Tina/lib-linux –ltinamath \
            -ltinasys  – lm – lc
```

where

```
/usr/java/jdk1.3.1_03/include, /usr/java/jdk1.3.1_03/include/linux,
/home/prljacan/Tina/include and /home/prljacan/Tina/
 lib-linux
```

are directory names appropriate to your Java and TINA library installations.

Above example files are available as: example.tar.gz

12. General considerations of calling Tina C library functions from Java

In the previous section the steps necessary to call C functions from the Java program (Java Native Interface Methods) were presented. In the case of Tina C library calls, the following things should be highlighted:

- Tina library consists of tens of user defined data types relaying heavily on pointers and dynamic memory allocation
- Tina library consists of hundreds of low-level to high-level functions, with parameters ranging from primitive data types to function pointers

Regarding JNI the following things must be highlighted:

- Primitive date types of Java and C (int, float,...) get passed from-to Java-C in one to one correspondence
- One-dimensional arrays of primitive data types also get passed in one to one correspondence by the help of a few JNI methods accessible to C code. This allows fast access to Java arrays from C code which does not differ from C access to its own arrays
- For more complex data types such as Java objects and arrays of objects, JNI methods gets much more complex and slower. For example, in order to access an java object, C code must at first access the object itself, then it must access each member using separate JNI method and member signature

As a conclusion JNI works at its highest when a called C function has parameters consisting of primitive data types and one dimensional arrays of primitive data types.

If one is bound to develop shareable Tina library which would expose all Tina C functions (*as it is)* to Java code it will mount to a huge amount of manual wrapping code for almost each C function and user defined data types in the library, since this process due to big number of user defined data types and various function signatures can not be easily automated.

For example, the following are some of the matrix manipulation functions provided by the Tina library (out of 157)

```
extern Matrix *matrix_sum(Matrix * mat1, Matrix * mat2);
extern Matrix *matrix_add(Matrix * m1, Matrix * m2);
extern Matrix *imatrix_add(Matrix * mat1, Matrix * mat2);
extern Matrix *imatrix_add_inplace(Matrix * mat1, Matrix * mat2);
extern Matrix *fmatrix_add_inplace(Matrix * mat1, Matrix * mat2);
extern Matrix *dmatrix_add_inplace(Matrix * mat1, Matrix * mat2);
extern Matrix *mat_alloc(int m, int n);
extern void    imatrix_format_lower(Matrix * mat);
extern void    imatrix_format_gen(Matrix * mat);
extern void    fmatrix_format_full(Matrix * mat);
extern void    fmatrix_format_lower(Matrix * mat);
extern void    fmatrix_format_gen(Matrix * mat);
extern void    dmatrix_format(Matrix * mat);
extern void    dmatrix_format_full(Matrix * mat);
extern void    dmatrix_format_lower(Matrix * mat);
```

```
 58 extern void    ptr_default_print(void *ptr);
 59 extern void    ptr_set_print(void (*newprint) (    ));
 60 extern void    (*ptr_get_print(void))();
 61 extern void    pmatrix_format(Matrix * mat);
 62 extern void    pmatrix_format_full(Matrix * mat);
 72 extern void    matrix_set_default_zval(Complex zval);
 76 extern void    matrix_set_default_pval(void *pval);
 77 extern void   *matrix_getp(Matrix * mat, int i, int j);
 85 extern void   *matrix_getp_full(Matrix * mat, int i, int j);
 86 extern Matrix *matrix_invert(Matrix * mat);
 87 e98 extern void    fmatrix_pprint(FILE * fp, Matrix * mat);
105 extern Matrix *matrix_minus(Matrix * mat);
106 extern Matrix *fmatrix_minus(Matrix * mat);
107 extern Matrix *fmatrix_minus_inplace(Matrix * mat);
113 extern Matrix *matrix_diff(Matrix * mat1, Matrix * mat2);
126 extern Matrix *matrix_sub(Matrix * mat1, Matrix * mat2);
137 extern Matrix *matrix_transp(Matrix * mat);
138 extern Matrix *dmatrix_transp(Matrix * mat);
141 extern Matrix *matrix_transform2(Transform2 transf);
142 extern Transform2 trans2_matrix(Matrix * mat);
143 extern Matrix *matrix_transform3(Transform3 transf);
144 extern Transform3 trans3_matrix(Matrix * mat);
145  extern  Matrix  *matrix_unit(int  m,  int  n,  Matrix_shape  shape,
Vartype vtype);
```

One possible approach which might alleviate the previously mentioned problem of manual wrapping of a huge number of C functions is to reduce inter language communications and move burden of coding and data translation to C side. In fact it is possible to write a C generic high-level function corresponding to a part of Tina library, and then expose ***only*** this function to Java using JNI. It is important to mention that Java will still have the access to all C functions in the module through the generic function.

For example, consider the previously mentioned matrix manipulation functions (157 of them), it is possible to write a generic matrix function which encapsulate all particular functions with the following prototype

```
int  matrix_manipulations(char *particular_function, list of generic
parameters);
```

where, `particular_function` is the Tina name of the function, `list of generic parameters` is a chosen set of parameters which allows to pass parameters to any function.

13. NeatVision development environment

The NeatVision is a Java written machine vision package, which is easy to use and extend thanks to the features of Java language and package design itself. More details about the NeatVision can be found in [2]

14. Integration of the TINA sequence tool and the NeatVision

The TINA sequence reader is a powerful set (tool) of C functions which allows reading and manipulating of image sequences in different formats. Currently it supports seven image file formats. As such it secures input data for all image processing and analysis tasks within TINA package regarding image sequence processing. The TINA sequence tool provides the following functionality (excluding display functionality)

- reading and storing image sequence
- moving to the first frame in the sequence
- moving to the last frame in the sequence
- moving to the next frame in the sequence
- moving to the previous frame in the sequence
- getting the image data
- getting sequence length
- getting frame width
- getting frame height
- getting image scaling factors

We want to provide the same functionality to the NeatVision. In that regard it is necessary to define a Java class containing native methods for accessing the TINA functions. The following class definition was chosen for this purpose

```
public class seq {

  static {
          System.loadLibrary("seqNvTina");
        }

 // sequence manipulation methods

  public static native int   seq_input(String filename,
                                       int filetype);
  public static native void  moveto_firstframe();
  public static native void  moveto_nextframe();
  public static native void  moveto_lastframe();
  public static native void  moveto_prevframe();
  public static native int   get_sequencelenght();
  public static native int   get_frameheight();
  public static native int   get_framewidth();
  public static native void  get_frame(int slika[]);
 public static native void  get_scale(double scale[]);
}
```

- First step in exposing TINA sequence reading functionality to NeatVision is to identify TINA source files and relating functions which do this task in TINA, these are located at /../../Tina/src/tools/sequence

- Source files of interests in the above mentioned directory are seq_tool.c and seq_io.c
- Use, rewrite and write necessary C functions in order to support interface functions(methods) as declared in class seq. All these new functions are marked and added at the end of original TINA source files seq_tool.c and seq_io.c
- Use make utility and present Makefile to rebuild TINA library including new C functions

After the previous steps are done, the rest is the application of JNI recipes (Sections 1-9) as follows

14.1 Compile seq.java class

```
public class seq {

  static {
           System.loadLibrary("seqNvTina");
         }

 // sequence manipulation methods

  public static native int   seq_input(String filename,
                                       int filetype);
  public static native void  moveto_firstframe();
  public static native void  moveto_nextframe();
  public static native void  moveto_lastframe();
  public static native void  moveto_prevframe();
  public static native int   get_sequencelenght();
  public static native int   get_frameheight();
  public static native int   get_framewidth();
  public static native void  get_frame(int slika[]);
 public static native void  get_scale(double scale[]);
}

% javac seq.java
```

14.2 Generate a header file for use by C

```
% javah –jni  seq
```

There will be seq.h generated header file

14.3 Implementing the native functions in C code

Now that we have created the header file seq.h, we can write C code, which implements native methods from seq class using functions from seq_tool.c and seq_io.c. The file is named seq.c.

14.4 Building the shareable library

Building on Linux (Solaris)

```
%gcc -c -I/usr/java/jdk1.3.1_03/include  \
        -I/usr/java/jdk1.3.1_03/include/linux  seq.c
```

```
%ld -G   seq.o -o libseqNvTina.so \
       -L/home/prljacan/Tina/lib-linux \
       -ltinatools -ltinadraw -ltinafile -ltinavision \
       -ltinatv -ltinaXm -ltinaX11 \
       -ltinamath -ltinasys -L/usr/openwin/lib \
       -L/usr/X11R6/lib -lXm -lXt -lX11 \
       -lm  -lc
```

14.5 Driver Java program example

In order to test seq.java class and libseqNvTina.so library, we have developed a simple driver Java program pogon.java which implements a simple sequence reader and image display. In order to run the example one has to compile pogon.java and run it

```
% javac pogon.java

% java pogon
```

Above mentioned files an instructions are available as:
seq_reader_example.tar.gz

14.6 Tina sequence reader in NeatVision

The full-blown version of the Tina sequence reader is implemented as a NeatVision component and was made a part of the NeatVision environment.

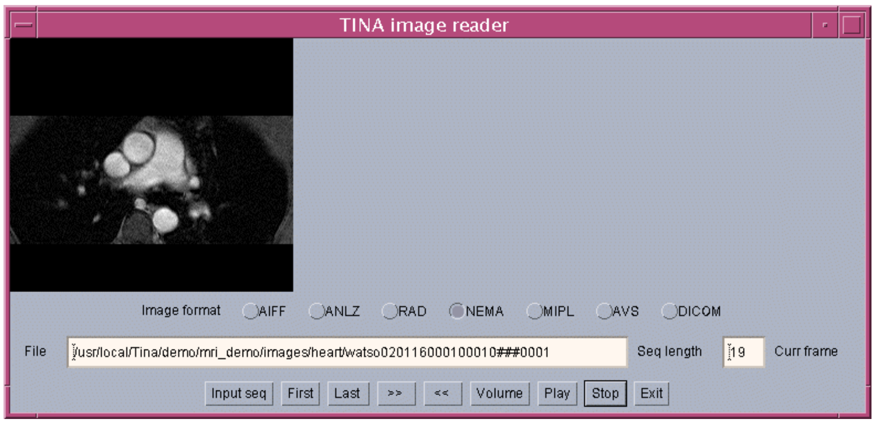

TINA image reader

Image format ( ) AIFF ( ) ANLZ ( ) RAD [✓ NEMA] ( ) MIPL ( ) AVS ( ) DICOM

File: /usr/local/Tina/demo/mri_demo/images/heart/watso020116000100010###0001

Seq length: 19

Curr frame

15. Integration of the Tina Aorta Tracking tool and the NeatVision

The TINA aorta-tracking tool is a set of C functions, which allows tracking (locating) of an aorta boundary in a sequence of MR images using active contour model. As such the tool could be used for functional cardiac analysis. The TINA aorta-tracking tool provides the following functionality

- entering initial model parameters
- changing initial model parameters
- searching the best model
- getting the best model parameters

We want to provide the same functionality to the NeatVision. In that regard it is necessary to define a Java class containing native methods for accessing the TINA functions. The previously defined seq.java class was expanded to accommodate for aorta tracking functionality as follows

```
public class seq
{

  static {
         System.loadLibrary("seqNvTina");
         }

 //sequence reader functions
  public static native int   seq_input(String filename,
 int filetype);
  public static native void  moveto_firstframe();
  public static native void  moveto_nextframe();
  public static native void  moveto_lastframe();
  public static native void  moveto_prevframe();
  public static native int   get_sequencelenght();
  public static native int   get_frameheight();
  public static native int   get_framewidth();
  public static native void  get_frame(int slika[]);
  public static native void  get_scale(double scale[]);

 //aorta tracking functions
  public static native void  input_model(String filename);
  public static native void  parameters_getset(double p[],
 double pa[], double par[], int para[], int flag);
  public static native void  search();
  public static native int   get_model(int model[]);

}
```

- First step in exposing TINA aorta tracking functionality to NeatVision is to identify TINA source files and relating functions which do this task in TINA, these are located at /../../Tina/src/tools/smartROI
- Source files of interests in the above mentioned directory are sroi_tool.c and sroi_io.c
- Use, rewrite and write necessary C functions in order to support interface functions(methods) as declared in class seq. All these new functions are marked and added at the end of original TINA source files sroi_tool.c and sroi_io.c
- Use make utility and present MakeFile to rebuild TINA library including new C functions

After the previous steps are done, the rest is the application of JNI recipes (sections 1-9) as follows

15.1 Compile seq.java class

```
% javac seq.java
```

15.2 Generate a header file for use by C

```
% javah –jni  seq
```

There will be seq.h generated header file

15.3 Implementing the native functions in C code

Now that we have created the header file seq.h, we can add C code which implements native methods from seq class using functions from sroi_tool.c and sroi_io.c. (Functions regarding sequence manipulation were developed in Section 14). The file is called seq.c.

15.4 Building the shareable library

Building on Linux (Solaris)

```
%gcc -c -I/usr/java/jdk1.3.1_03/include  \
        -I/usr/java/jdk1.3.1_03/include/linux  seq.c


%ld -G   seq.o -o libseqNvTina.so \
       -L/home/prljacan/Tina/lib-linux \
       -ltinatools -ltinadraw -ltinafile -ltinavision \
       -ltinatv -ltinaXm -ltinaX11 \
       -ltinamath -ltinasys -L/usr/openwin/lib \
       -L/usr/X11R6/lib -lXm -lXt -lX11 \
       -lm  -lc
```

15.5 Driver Java program example

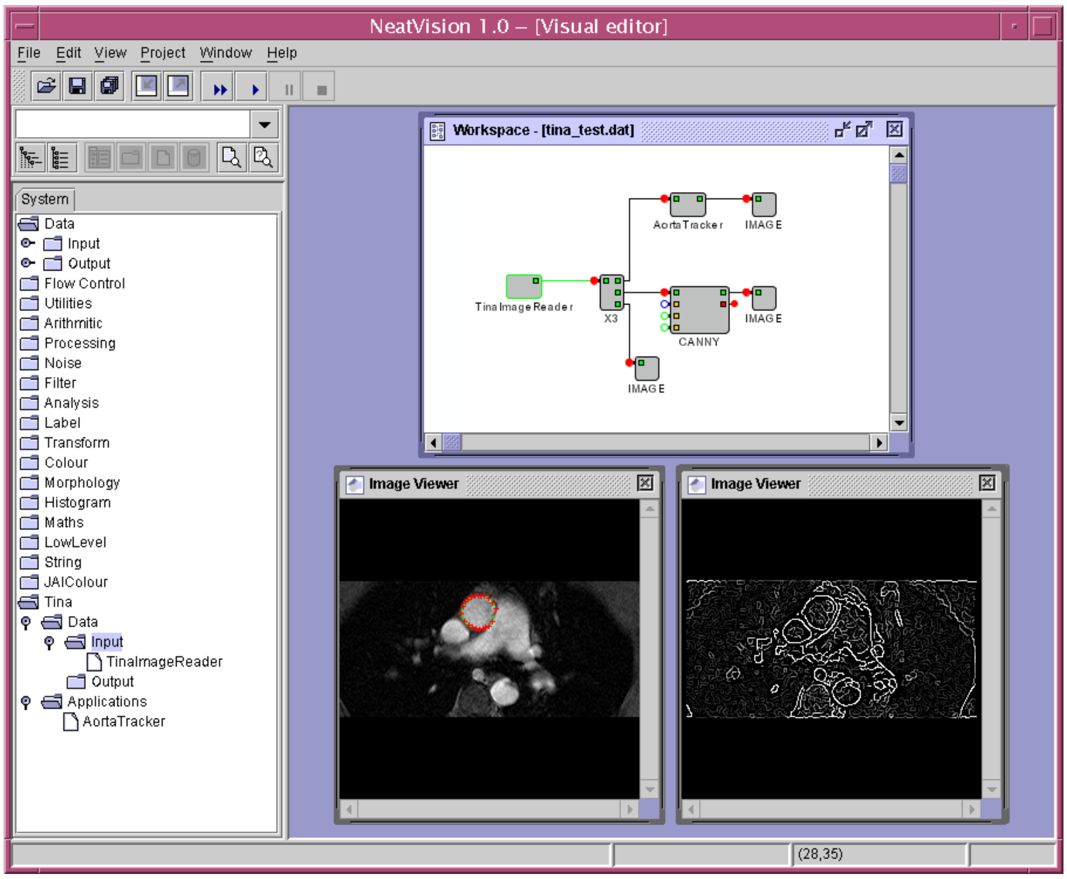
In order to test seq.java class and libseqNvTina.so library , we have developed a simple driver Java program pogon.java which uses sequence reader and aorta tracking functionalities. In order to run the example one has to compile pogon.java and run it

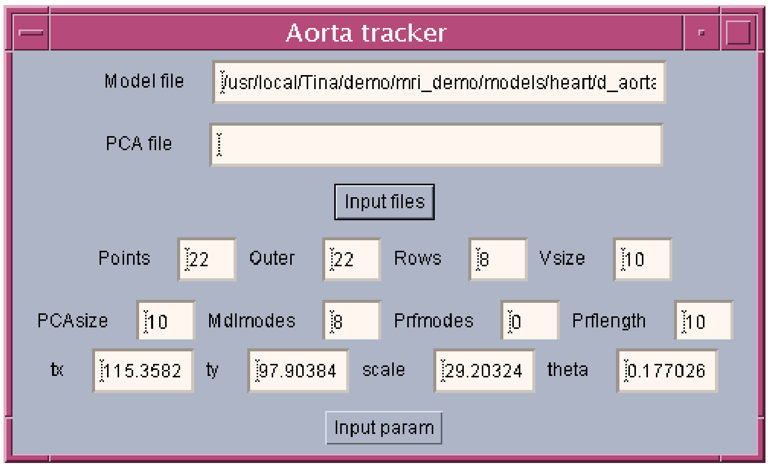
```
% javac pogon.java

% java pogon
```

Above mentioned files and instructions are available as:
aorta_tracker_example.tar.gz

15.6 Tina aorta tracker in NeatVision

The full-blown version of the Tina aorta tracker is implemented as a NeatVision component and was made a part of the NeatVision environment.





Model file	/usr/local/Tina/demo/mri_demo/models/heart/d_aorta						
PCA file							
Points	22	Outer	22	Rows	8	Vsize	10
PCAsize	10	Mdlmodes	8	Prfmodes	0	Prflength	10
tx	115.3582	ty	97.90384	scale	29.20324	theta	0.177026

16. Integration of the Tina NMR Segmentation tool and the NeatVision

The TINA NMR segmentation tool is a set of C functions, which allows segmentation of brain tissues from MR images. The TINA NMR segmentation tool provides the following functionality

- Fitting of a composite Gaussian density function to image histogram
- Computing probabilistic maps of brain tissues

We want to provide the same functionality to the NeatVision. In that regard it is necessary to define a Java class containing native methods for accessing the TINA functions. The previously defined seq.java class was expanded to accommodate for NMR segmentation functionality as follows

```
public class seq
{

  static {
          System.loadLibrary("seqNvTina");
          }

 //sequence reader functions
  public static native int   seq_input(String filename,
 int filetype);
  public static native void  moveto_firstframe();
  public static native void  moveto_nextframe();
  public static native void  moveto_lastframe();
  public static native void  moveto_prevframe();
  public static native int   get_sequencelenght();
  public static native int   get_frameheight();
  public static native int   get_framewidth();
  public static native void  get_frame(int image[]);
  public static native void  get_scale(double scale[]);

 //aorta tracking functions
  public static native void  input_model(String filename);
  public static native void  parameters_getset(double p[],
 double pa[], double par[], int para[], int flag);
  public static native void  search();
  public static native int   get_model(int model[]);

  // NMR segmentation functions
   public static native void nmr_fit_proc();
  public static native void fit_param_dialog(double par[],
                                int flag);
  public static native void rusenik_param_dialog(double
                             par[], int flag);
  public static native void rusenik();
  public static native void push_seq_frame();
  public static native void probabilistic_map(double
                          image[], int choice);

}
```

- First step in exposing TINA NMR segmentation functionality to NeatVision is to identify TINA source files and relating functions which do this task in TINA, these are located at /../../Tina/src/tools/nmr-segment
- Source files of interests in the above mentioned directory are nmr_segment_tool.c

- Use, rewrite and write necessary C functions in order to support interface functions (methods) as declared in class seq. All these new functions are marked and added at the end of original TINA source files nmr_segment_tool.c
- Use make utility and present Makefile to rebuild TINA library including new C functions

After the previous steps are done, the rest is the application of JNI recipes (paragraphs 1-9) as follows

16.1 Compile seq.java class

```
% javac seq.java
```

16.2 Generate a header file for use by C

```
% javah –jni  seq
```

There will be seq.h generated header file

16.3 Implementing the native functions in C code

Now that we have created the header file seq.h, we can add C code which implements native methods from seq.java class using functions from nmr_segment_tool.c (Functions regarding sequence manipulation were developed in Section 14 and aorta tracker in Section 15). The file is named seq.c.

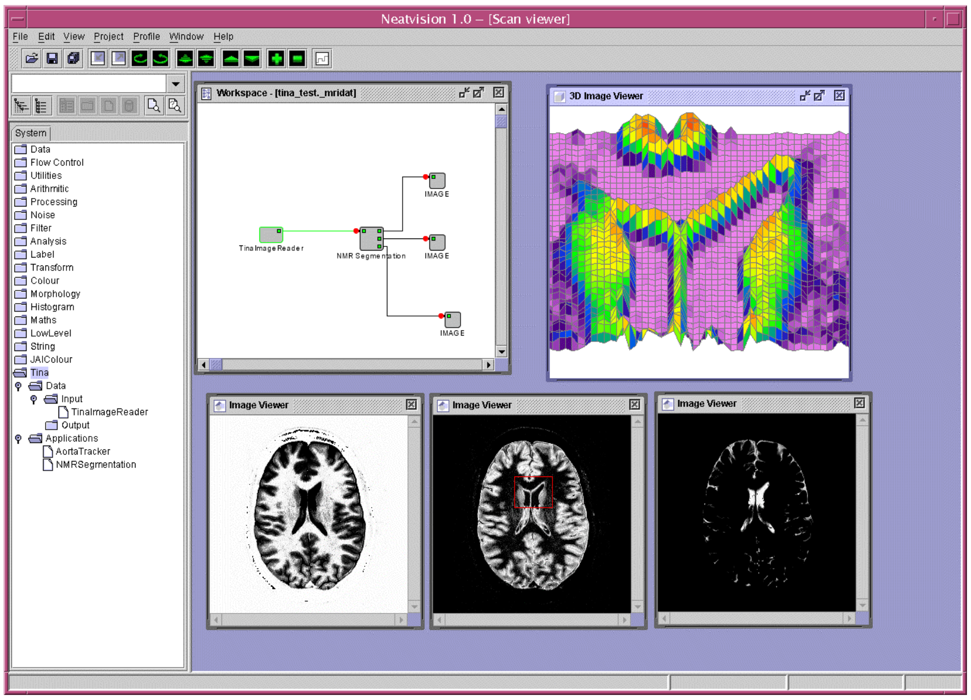
16.4 Building the shareable library

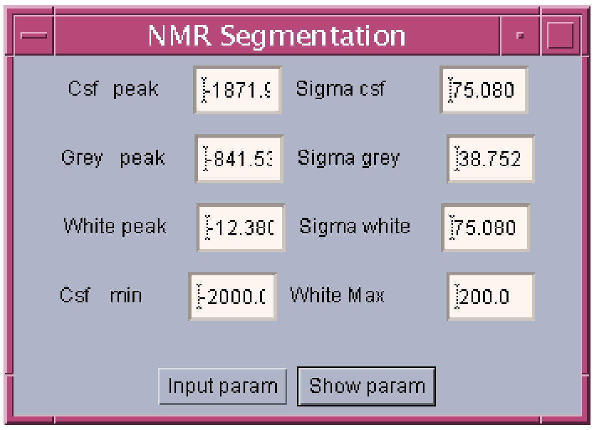
Building on Linux (Solaris)

```
%gcc -c -I/usr/java/jdk1.3.1_03/include  \
        -I/usr/java/jdk1.3.1_03/include/linux  seq.c


%ld -G   seq.o -o libseqNvTina.so \
       -L/home/prljacan/Tina/lib-linux \
       -ltinatools -ltinadraw -ltinafile -ltinavision \
       -ltinatv -ltinaXm -ltinaX11 \
       -ltinamath -ltinasys -L/usr/openwin/lib \
       -L/usr/X11R6/lib -lXm -lXt -lX11 \
       -lm  -lc
```

16.5 Tina NMR segmentation in NeatVision

The full-blown version of the Tina NMR segmentation tool was implemented as a NeatVision component and was made a part of the NeatVision environment.




NMR Segmentation

Csf peak	-1871.9	Sigma csf	75.080
Grey peak	-841.53	Sigma grey	38.752
White peak	-12.380	Sigma white	75.080
Csf min	-2000.0	White Max	200.0

Input param   Show param

References

[1] http://www.niac.man.ac.uk/Tina/tina.html
[2] http://www.neatvision.com/
[3] http://www.eeng.dcu.ie/~whelanp/osmia/
[4] http://java.sun.com