# Remote Sensing

# Raster Programming

*Dr. Yann H. Chemin*

**Remote Sensing Raster Programming**

Second Edition

Dr. Yann H. Chemin

ISBN 978-1-4092-6689-1

“How can I load input satellite imagery, compute input rasters into a given result and write that result as a new image to the hard disk”.

This book gives a range of programming options to answer this question, using high-level and low-level programming languages, some serial (C, Python, R) but also some in parallel (OpenMP, MPI-C, CUDA, OpenCL). Additionally, it also demonstrates how to perform various levels of integrations into few programming languages and environments having GUI functionalities (WxPython and GRASS GIS).

## Foreword

This book is about satellite imagery processing. It answers to the one single question: "How can I load input satellite imagery, compute input rasters into a given result and write that result as a new image to the hard disk". It gives a range of programming options to answer this question, using high-level and low-level programming languages, some serial but also some in parallel. Additionally, it also demonstrates how to perform various levels of integrations into few programming languages and environments having GUI functionalities.

Why Programming? If you need to research or have to make a custom-fit solution, there are few other ways to have full control on your data. Also, algorithm development is difficult without programming it. The languages used in this booklet are:

- In the C family: C, OpenMP, MPI C, CUDA, OpenCL and C in GRASS GIS
- In the Python family: Python and WxPython
- and R, the statistical high-level programming language

Both the GRASS GIS and wxPython codes include Graphical User Interfaces (GUIs). All of these codes execute the same functions:

- Loading satellite images,
- Calculating an index or a physical value,
- Writing the index or physical value to an image file on the hard disk.

Once you have mastered this very basic functionality in the programming context of your choice/need, you can easily modify/add additional processing functions, so as to reach the output you are looking for.

Greetings and good luck,
Yann Chemin

# Contents

# Chapter 1

# Introduction

In this age of increasing information recording, geographically-aware information (geoInformation) is becoming of paramount importance for many parts of our civilizations. More and more Earth Observing Satellites (EOS) are circumnavigating the various depths of our Earth orbits. Airborne imagery which has long been recording high spectral and temporal information, is rising to a new level with the availability of low-cost drones. These drones, most of the time equipped with spectrum-corrected digital cameras, provide basic spectral and very high-resolution in terms of space and time. All of these are raster-based geoInformation that need storage, correction, and eventually to find a use, have to be computed into some value that the users needs to classify/quantify/identify.

This booklet addresses this rising need of computation of raster-based geoInformation by leveraging the entry point to data access, modification and writing to disk. Most of the time, we are using an external library to facilitate our data access, this library is called the Geospatial Data Abstraction Layer (GDAL; gdal.osgeo.org). You will have to install it on your computer to make most of this booklet code come to life. In other cases, GDAL is most certainly used in the background (as it is in R, through Rgdal plugin) or GDAL is used to initially import data (through the r.in.gdal module in the GRASS GIS environment).

GDAL is not only facilitating access to the data inside a geographical data, it also manages the geographical meta-data for the user, so as to provide support while the programmer can concentrate on data computation. When GDAL is writing the output file to disk, the geographical meta-data along with other meta-data of interest are passed to the output file with a minimum of programming.

## 1.1 Typical remote sensing work-flow

Human perception is how we conceptualize what we need to make out of our image data. In a simple case, we would formulate such a mental setup: “Those two images should be summed together so as to make a third image.” Though it is clear, the structure needs clearly to be split into atomic actions. According to the level of programming, the atomicity expansion will be different.

### 1.1.1 High-Level programming

High-level programming is designed to reduce systematic operations for the user. Our problem ends up being a short script consisting of 5 steps.

- Load appropriate libraries
- Specify locations/filenames for image 1, 2 and output.
- Load image number 1 and 2 in memory
- Perform the algebra operation of image 1 on image 2, and store in memory
- Save the output image on disk

As we will see later, some computer memory maybe smaller than the input data. Some high-level languages have functions to avoid step 1 and 2 to read all data, but instead load file description, in order to perform step 3 on a row by row basis.

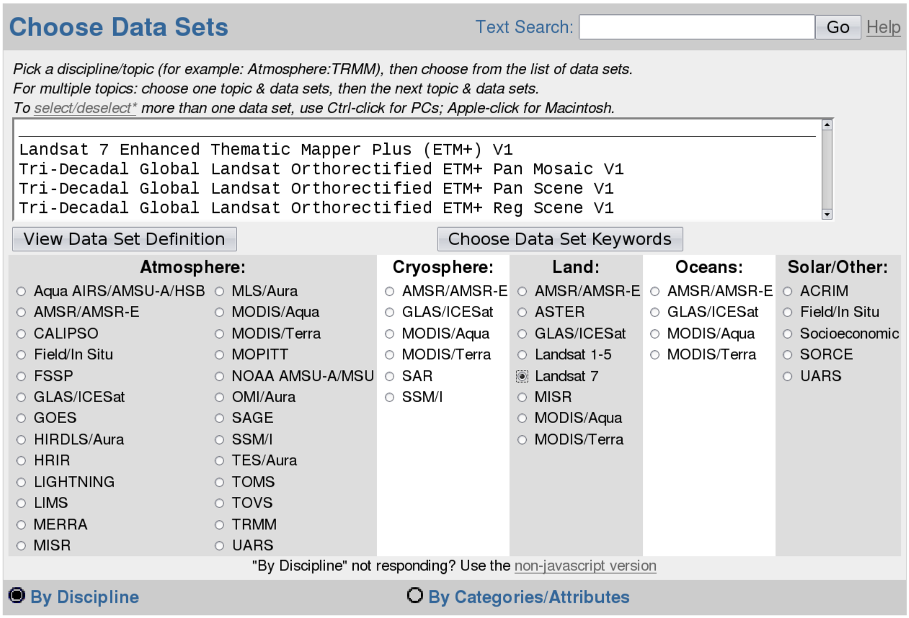
### 1.1.2 Lower-Level programming

Low-level programming is lengthier than high-level programming. It takes more time to reach, but it rewards with high control on operations at pixel level.

- Specify locations/filenames for image 1, 2 and output.
- Set up image file handlers, load row&column numbers, allocate row data memories
- Start row loop, load row data for each input
- Start column loop, load pixel data values for input 1 and 2.
- Perform the addition operation of image 1 pixel on image 2 pixel, store in row memory at column location.
- Leave column loop, save row data on file.
- Leave row loop.
- Free memory, close input and output image files.

## 1.2 Downloading free satellite data from Internet

Downloading satellite imagery from Internet is now common. From 2009, Landsat-class satellites is having all its imagery freely downloadable online. Terra and Aqua satellites have always been freely downloadable since their launch. Typically, a safe way to start with is to reach the Warehouse Inventory Search Tool (WIST), a NASA website (wist.echo.nasa.gov) holding a large amount of free satellite imagery. Below is the initial choice available on WIST.



**Choose Data Sets** Text Search: Go Help

*Pick a discipline/topic (for example: Atmosphere:TRMM), then choose from the list of data sets.*
*For multiple topics: choose one topic & data sets, then the next topic & data sets.*
*To select/deselect* more than one data set, use Ctrl-click for PCs; Apple-click for Macintosh.*

```
Landsat 7 Enhanced Thematic Mapper Plus (ETM+) V1
Tri-Decadal Global Landsat Orthorectified ETM+ Pan Mosaic V1
Tri-Decadal Global Landsat Orthorectified ETM+ Pan Scene V1
Tri-Decadal Global Landsat Orthorectified ETM+ Reg Scene V1
```

View Data Set Definition | Choose Data Set Keywords

| Atmosphere: | | Cryosphere: | Land: | Oceans: | Solar/Other: |
|---|---|---|---|---|---|
| Aqua AIRS/AMSU-A/HSB | MLS/Aura | AMSR/AMSR-E | AMSR/AMSR-E | AMSR/AMSR-E | ACRIM |
| AMSR/AMSR-E | MODIS/Aqua | GLAS/ICESat | ASTER | GLAS/ICESat | Field/In Situ |
| CALIPSO | MODIS/Terra | MODIS/Aqua | GLAS/ICESat | MODIS/Aqua | Socioeconomic |
| Field/In Situ | MOPITT | MODIS/Terra | Landsat 1-5 | MODIS/Terra | SORCE |
| FSSP | NOAA AMSU-A/MSU | SAR | [✓ Landsat 7] | | UARS |
| GLAS/ICESat | OMI/Aura | SSM/I | MISR | | |
| GOES | SAGE | | MODIS/Aqua | | |
| HIRDLS/Aura | SSM/I | | MODIS/Terra | | |
| HRIR | TES/Aura | | | | |
| LIGHTNING | TOMS | | | | |
| LIMS | TOVS | | | | |
| MERRA | TRMM | | | | |
| MISR | UARS | | | | |

"By Discipline" not responding? Use the non-javascript version

[✓ By Discipline] By Categories/Attributes

If you are downloading Modis images, be sure to have at hand a proper tool to manipulate the .hdf file format. The Modis Reprojection Tool (MRT) is such a tool and is available at (lpdaac.usgs.gov/landdaac/tools/modis/). MRT can convert to a more standard GeoTiff file format the Modis images. It can also reproject them to more common projection systems than its storage-purpose projection system the Integrated Sinusoidal. Alternatively, you can use a combination of tools from GDAL.

GDAL has a set of small tools that are practical to access information from header file and to perform repeated information on a dataset. You can use gdalinfo (www.gdal.org/gdalinfo.html) for extracting detailed information from the image file header, it extracts proper information about MODIS subdatasets and their naming conventions, add to it gdal_translate (www.gdal.org/gdal_translate.html) to extract specific bands from the HDF format file into single bands common formats like GeoTiff. The script below automatically reads and exports each subdataset of any MODIS HDF file in the directory into a subdirectory called *processed*.

```
#This Unix Shell script automatically extracts MODIS subdatasets
# into processed/ subdirectory as GeoTiff (*.tif) files
#-----------------------------------------------------------------
#/bin/bash
mkdir processed
for file in *.hdf
do
        IFS=$'\012' #Change separator to '\n'
        SDS_list=$(gdalinfo $file | grep "SUBDATASET_.*_NAME.*")
        for subdataset in $SDS_list
        do
                out=$(echo ${subdataset#*=} | sed 's/[\"\:\.\ ]/\_/g') #Clean to alphanum
                gdal_translate -of GTiff ${subdataset#*=} processed/$out.tif
        done
done
```

Another tool, gdalwarp (www.gdal.org/gdalwarp.html) is dedicated to extract part of images and geolocate them properly. It would reproject the image data, it is useful to note that it also change the file format if you request it to do so. The script below does that with a target projection *EPSG:4326* which actually means the following proj4 crs: *+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs*.

```
#/bin/bash
for file in *.hdf
do
    IFS=$'\012' #Change separator to '\n'
    SDS_list=$(gdalinfo $file | grep "SUBDATASET_.*_NAME.*")
    for subdataset in $SDS_list
    do
       out=$(echo ${subdataset#*=} | sed 's/[\"\:\.\ ]/\_/g') #Clean to alphanum
       gdalwarp -of GTiff -s_srs '+proj=sinu +R=6371007.181 +nadgrids=@null +wktext' \
        -t_srs EPSG:4326 ${subdataset#*=} processed/$out.tif
    done
done
```

The Landsat satellite is far more famous among a larger audience, bringing up a more numerous and diverse websites offering download.

- http://glovis.usgs.gov/
- http://landsat.gsfc.nasa.gov/
- http://eros.usgs.gov/products/satellite/landsat_ortho.html

For more information about Landsat and MODIS satellites, their Science websites are located respectively at:

- http://landsat.gsfc.nasa.gov/.
- http://modis.gsfc.nasa.gov/.

# Chapter 2

# GDAL

## 2.1 Image Processing in C with GDAL API

### 2.1.1 Introduction

C language is a most widely used computing language, its combination of control and robustness provides generations of programmers with standardized code-base. In this section, we will use the GDAL C API to provide the image handling capacity to our software. GDAL provides functions to open raster files and retrieve their meta-data as well as their image data. Once we have retrieved the image data, it is possible to apply mathematical operations to that data. This part is the algorithmic calculation that the raster input images undergo in order to provide an image result of a certain meaning sought by the user. Once the result is found, it is written to the output file using another GDAL function.

If you want to use C language on your Operating System, you have to know that it is always available. On Unix and Linux, the C compilers are generally installed by default and often called either "cc" or "gcc" (C compiler and Gnu C compiler). On MacOSX, "gcc" can be downloaded from www.mac.com or available in the companion DVD of your computer. On MS Windows, you can download MinGW (www.mingw.org). Standard compilation of a C file "cfile.c" for scientific purpose can be done (in case you use gcc) by gcc -o myProg.exe cfile.c -lm. In this case of scientific application, the trailing "-lm" is to tell the compiler that its needs to include mathematical software library which is needed in our case. A Makefile, is a special text file which name is exactly "Makefile" and which is holding information about the compilation of our software. It is convenient, as some software may require complex configurations. An example of a Makefile to replace the compilation command seen above would be as below. So a compilation command would now be reduce to typing “Makefile” in the Command Line Interface (CLI), also called command prompt in MS Windows. In italic is the specific GDAL Makefile requirement.

```
#A short Makefile contents
myProg.exe: cfile.c rs.c
    gcc -o myProg.exe cfile.c rs.c -lm -I/usr/include/gdal/ -L/usr/lib -lgdal1.7.0
```

This C code is the basis also for the parallel codes available in the next sections, where the parallelizations structure are emphasized more than the data access. Therefore, this section is a must read as a foundation to concentrate later on the parallelization structures of the next chapters.

### 2.1.2 Code explanation

Every C code starts with calling definitions of certain types of functions (header files hold them), they are used in the application written after being called. In all cases the header files called start with “stdio.h”, additionally we will also call two very specific header files. The first one (“gdal.h”) holds functions and variables dedicated to make us benefit from the GDAL library to access image datasets having various geographical format structures. The second one “rs.h”, is our header file, holding definition of our algorithms, it is expanded in Appendix A, alongside its C file holding the algorithms themselves.

```
#include<stdio.h>
#include<gdal.h>
#include "rs.h"
```

The main code holds two functions, one called usage(), being essentially a graceful exit in case of problem understanding the input arguments to the application. It has no additional reason to be else being informative, that is.
The second function, on the opposite, is the essential part of the application, and for that reason is rightfully called main(). This function, which is also the application itself, takes two sets of arguments. The first one int argc, is a count of all the arguments received by the application, char *argv[] is actual list of all arguments, starting with argv[0] being the application name itself. If the count of arguments argc in inferior to the number of input arguments necessary to run the application (in this case, if argc $< 6$), then the application launches the function usage(), and exits with a value $= 1$ (non-zero values on exit is an error report).

After that, several variables have to be defined so that they can be used later in the program. Row and columns are integers, access to x-axis and y-axis are always required when working on images. Float (a decimal number type) is used for various pixel values, and temporary ones too. Finally, char (character type) is used for variables holding the input filenames.

```
int main( int argc, char *argv[] )
{
        if( argc < 6 ) {
        usage();
        return 1;
    }
    int     row, col;
    float   ndvix, waterx;          /* Processing pixels */
    float   temp, tempval;          /* Temporary pixels */
    char    *inB1, *inB2, *inB7;    /* Input file names */
    char    *in250QC, *in500QC;     /* Input file names */
    char    *waterF;                /* Output file name */
```

Once this is done, we load the input and output raster file names into their respective name holders. Note argv index numbering.

```
//Loading the input files names
inB1        = argv[1];
inB2        = argv[2];
inB7        = argv[3];
in250QC     = argv[4];
in500QC     = argv[5];
waterF      = argv[6];
```

Define and load raster files using GDAL functions.

```
//Defining the GDAL input files holders
GDALDatasetH hDataset1;     //Red Band
GDALDatasetH hDataset2;     //NIR Band
GDALDatasetH hDataset7;     //SWIR2 Band
GDALDatasetH hDataset8;     //Quality Assessment 250m
GDALDatasetH hDataset9;     //Quality Assessment 500m

//Register all GDAL image drivers to memory
GDALAllRegister();

//Open the raster images through GDAL
hDataset1 = GDALOpen(inB1,GA_ReadOnly);
hDataset2 = GDALOpen(inB2,GA_ReadOnly);
hDataset7 = GDALOpen(inB7,GA_ReadOnly);
hDataset8 = GDALOpen(in250QC,GA_ReadOnly);
hDataset9 = GDALOpen(in500QC,GA_ReadOnly);

//Fail the application if unable to load any input file
if(hDataset1==NULL||hDataset2==NULL||hDataset7==NULL||
        hDataset8==NULL||hDataset9==NULL){
    printf("Unable to load one or more input file\n");
    exit(1);
}
```

Define holders and load individual image file format types.

```
//Loading the file infos
GDALDriverH hDriver1 = GDALGetDatasetDriver(hDataset1);
GDALDriverH hDriver2 = GDALGetDatasetDriver(hDataset2);
GDALDriverH hDriver7 = GDALGetDatasetDriver(hDataset7);
GDALDriverH hDriver8 = GDALGetDatasetDriver(hDataset8);
GDALDriverH hDriver9 = GDALGetDatasetDriver(hDataset9);
```

Define and create the output file and its constituting band. Note the use of input file number 1 information to create a copy for the output file.

```
//Creating output file: Water Mask
GDALDatasetH hDatasetOut0 = GDALCreateCopy( hDriver1, waterF,hDataset1,FALSE,NULL,NULL,NULL);
GDALRasterBandH hBandOut0 = GDALGetRasterBand(hDatasetOut0,1);
```

Defining and loading the file bands.

```
//Defining the file bands and loading them
GDALRasterBandH hBand1 = GDALGetRasterBand(hDataset1,1);
GDALRasterBandH hBand2 = GDALGetRasterBand(hDataset2,1);
GDALRasterBandH hBand7 = GDALGetRasterBand(hDataset7,1);
GDALRasterBandH hBand8 = GDALGetRasterBand(hDataset8,1);
GDALRasterBandH hBand9 = GDALGetRasterBand(hDataset9,1);
```

Once we have access to each image data (composed of one band in this example), the access to each row inside the band should be defined. We do that by discovering the amount of columns and rows in raster input number 1. By knowing the row dimension, we can allocate a row memory (called lines here) of that size for each of the input and output raster files. We would like to bring to your attention the size of line7 and line9, the memory allocation function malloc() uses a length value of nXSize1 divided by 2. This is due to the fact that our reference image for nXSize1 is having a resolution of 250m, and the images referred to by the line7 and line9 rows are at 500m spatial resolution (i.e. half the row size).

```
//Loading Input file number 1 rows and columns
int nXSize1     = GDALGetRasterBandXSize(hBand1);
int nYSize1     = GDALGetRasterBandYSize(hBand1);

//Preparing processing
float       *line1; //Red Band row
float       *line2; //NIRed Band row
float       *line7; //SWIR3 Band row
int         *line8; //Quality Assessment 250m Band row
int         *line9; //Quality Assessment 500m band row
float       *lineOut0; //Output Band row

//Defining the data rows
line1       = (float *) malloc(sizeof(float)*nXSize1);
line2       = (float *) malloc(sizeof(float)*nXSize1);
line7       = (float *) malloc(sizeof(float)*nXSize1/2);
line8       = (int *) malloc(sizeof(int)*nXSize1);
line9       = (int *) malloc(sizeof(int)*nXSize1/2);
lineOut0    = (float *) malloc(sizeof(float)*nXSize1);
```

Finally, we access the data row by row, here again the row size is not common to all files, watch out for resizing requirements of line7 and line9. The data is read from hBandx using GDALRasterIO() with the GF_Read specification and put in linex, x being a input raster file number. Note the data in line8 and line9 are kept as integer type, since they are not a satellite image physical measurement but in this case a quality control flag binary bit field (i.e. MODIS surface reflectance products Quality Assessment flags).

```
//Accessing the data rowxrow
for(row=0;row<nYSize1;row++){
GDALRasterIO(hBand1,GF_Read,0,row,nXSize1,1,line1,nXSize1,1,GDT_Float32,0,0);
GDALRasterIO(hBand2,GF_Read,0,row,nXSize1,1,line2,nXSize1,1,GDT_Float32,0,0);
GDALRasterIO(hBand7,GF_Read,0,row/2,nXSize1/2,1,line7,nXSize1/2,1,GDT_Float32,0,0);
GDALRasterIO(hBand8,GF_Read,0,row,nXSize1,1,line8,nXSize1,1,GDT_Int32,0,0);
GDALRasterIO(hBand9,GF_Read,0,row/2,nXSize1/2,1,line9,nXSize1/2,1,GDT_Int32,0,0);
```

At this point the row data is in memory, and we may now access the data at the pixel level. The following code goes through the columns (X-axis) dimension, it starts with loading temporary variables with quality assessment data (from MODIS surface reflectance products), then it assesses if the quality flags are failed. For each failure, a separate error value is issued to the output pixel for clarity reasons. A “no data” case (value=-28672) is also there. If it not “no data” and if quality flags are indicating the data is good, we can then process the data with our algorithm. The output data, in this case waterx is sent to the output row data (lineOut0 at X-axis position [col]).

```
    //Processing the data cellxcell
    for(col=0;col<nXSize1;col++){
        //Load temporary values with quality flags
        tempval = stateqa500a(line9[col/2]);
        temp = qc250a(line8[col]);
        //if no data or quality flag fail then assign respective fail values
        if(line1[col]==-28768||temp>1.0||tempval>=1.0){ /*skip it*/
            if(temp>1.0)
                lineOut0[col]=10.0;
            else if (tempval>=1.0)
                lineOut0[col]=100.0;
            else
                lineOut0[col]=-28768;
        }
        // if pixel are correct, process our algorithms
        else {
            //NDVI calculation
            ndvix = ndvi(line1[col],line2[col]);
            //Water calculation
            waterx = water_modis(line7[col/2]*0.0001,ndvix);
            lineOut0[col] = waterx;
        }
    }
```

Once the output row data is in lineOut0 it can be copied to the output raster file using GDALRasterIO() this time with the GF_Write specification.

```
//Write image to disk through GDAL
GDALRasterIO(hBandOut0,GF_Write,0,row,nXSize1,1,lineOut0,nXSize1,1,GDT_Float32,0,0);
```

The processing is now over, we need to free the row data memory holders and close the input and output raster files properly with GDALClose().

```
if( line1 != NULL )
    free( line1 );
...
GDALClose(hDataset1);
....
```

### 2.1.3 C code

```
#include<stdio.h>
#include<gdal.h>
#include rs.h

void usage()
{
    printf( "-----------------------------------------\n");
    printf( "--Modis Processing chain--Serial code----\n");
    printf( "-----------------------------------------\n");
    printf( "./wm inB1 inB2 inB7\n");
    printf( "     in250QC in500stateQA\n");
    printf( "     outWATER\n");
    printf( "-----------------------------------------\n");
    printf( "inB[1-7] files are surface reflectance files (250&500)\n");
    printf( "inB1 and inB2 are Modis 250m\n");
    printf( "inB3-7 are Modis 500m, they will be split to 250m\n");
    printf( "in250QC is Modis 250/500m Quality Assessment\n");
    printf( "in500stateQA is Modis 500m State Quality Assessment\n");

    printf( "outWATER is the Water mask output [0-1]\n");
    return;
}

int main( int argc, char *argv[] )
{
        if( argc < 6 ) {
        usage();
        return 1;
    }
    int row,col;
    float   ndvix, waterx;      /* Processing pixels */
    float   temp, tempval;      /* Temporary pixels */
    char    *inB1, *inB2, *inB7;        /* Input file names */
    char    *in250QC, *in500QC;         /* Input file names */
    char    *waterF;                /* Output file name */

    //Loading the input files names
    inB1        = argv[1];
    inB2        = argv[2];
    inB7        = argv[3];
    in250QC     = argv[4];
    in500QC     = argv[5];
    waterF      = argv[6];

    //Defining the GDAL input files holders
    GDALDatasetH hDataset1; //Red Band
    GDALDatasetH hDataset2; //NIR Band
    GDALDatasetH hDataset7; //SWIR3 Band
    GDALDatasetH hDataset8; //Quality Assessment 250m Band
    GDALDatasetH hDataset9; //Quality Assessment 500m Band

    //Register all GDAL image drivers to memory
    GDALAllRegister();

    //Open the raster images through GDAL
    hDataset1 = GDALOpen(inB1,GA_ReadOnly);
    hDataset2 = GDALOpen(inB2,GA_ReadOnly);
    hDataset7 = GDALOpen(inB7,GA_ReadOnly);
    hDataset8 = GDALOpen(in250QC,GA_ReadOnly);
    hDataset9 = GDALOpen(in500QC,GA_ReadOnly);
```

```
//Fail the application if unable to load any input file
if(hDataset1==NULL||hDataset2==NULL||
    hDataset7==NULL||hDataset8==NULL||hDataset9==NULL){
    printf("Unable to load one or more input file\n");
    exit(1);
}

//Loading the file infos
GDALDriverH hDriver1 = GDALGetDatasetDriver(hDataset1);
GDALDriverH hDriver2 = GDALGetDatasetDriver(hDataset2);
GDALDriverH hDriver7 = GDALGetDatasetDriver(hDataset7);
GDALDriverH hDriver8 = GDALGetDatasetDriver(hDataset8);
GDALDriverH hDriver9 = GDALGetDatasetDriver(hDataset9);

//Creating output file
//Water Mask
GDALDatasetH hDatasetOut0 = GDALCreateCopy(hDriver1,waterF,hDataset1,FALSE,NULL,NULL,NULL);
GDALRasterBandH hBandOut0 = GDALGetRasterBand(hDatasetOut0,1);

//Defining the file bands and loading them
GDALRasterBandH hBand1 = GDALGetRasterBand(hDataset1,1);
GDALRasterBandH hBand2 = GDALGetRasterBand(hDataset2,1);
GDALRasterBandH hBand7 = GDALGetRasterBand(hDataset7,1);
GDALRasterBandH hBand8 = GDALGetRasterBand(hDataset8,1);
GDALRasterBandH hBand9 = GDALGetRasterBand(hDataset9,1);

//Loading Input file number 1 rows and columns
int nXSize1     = GDALGetRasterBandXSize(hBand1);
int nYSize1     = GDALGetRasterBandYSize(hBand1);

//Preparing processing
float       *line1; //Red Band row
float       *line2; //NIRed Band row
float       *line7; //SWIR3 Band row
int         *line8; //Quality Assessment 250m Band row
int         *line9; //Quality Assessment 500m band row
float       *lineOut0; //Output Band row

//Defining the data rows
line1       = (float *) malloc(sizeof(float)*nXSize1);
line2       = (float *) malloc(sizeof(float)*nXSize1);
line7       = (float *) malloc(sizeof(float)*nXSize1/2);
line8       = (int *) malloc(sizeof(int)*nXSize1);
line9       = (int *) malloc(sizeof(int)*nXSize1/2);
lineOut0    = (float *) malloc(sizeof(float)*nXSize1);

//Accessing the data rowxrow
for(row=0;row<nYSize1;row++){
    GDALRasterIO(hBand1,GF_Read,0,row,nXSize1,1,line1,nXSize1,1,GDT_Float32,0,0);
    GDALRasterIO(hBand2,GF_Read,0,row,nXSize1,1,line2,nXSize1,1,GDT_Float32,0,0);
    GDALRasterIO(hBand7,GF_Read,0,row/2,nXSize1/2,1,line7,nXSize1/2,1,GDT_Float32,0,0);
    GDALRasterIO(hBand8,GF_Read,0,row,nXSize1,1,line8,nXSize1,1,GDT_Int32,0,0);
    GDALRasterIO(hBand9,GF_Read,0,row/2,nXSize1/2,1,line9,nXSize1/2,1,GDT_Int32,0,0);
    //Processing the data cellxcell
    for(col=0;col<nXSize1;col++){
        //Load temporary values with quality flags
        tempval = stateqa500a(line9[col/2]);
        temp = qc250a(line8[col]);
        //if no data or quality flag fail then assign respective fail values
        if(line1[col]==-28768||temp>1.0||tempval>=1.0){ /*skip it*/
            if(temp>1.0)
                lineOut0[col]=10.0;
                else if (tempval>=1.0)
                    lineOut0[col]=100.0;
                else
                    lineOut0[col]=-28768;
            }
            // if pixel are correct, process our algorithms
            else {
                //NDVI calculation
                ndvix = ndvi(line1[col],line2[col]);
                //Water calculation
                waterx = water_modis(line7[col/2]*0.0001,ndvix);
                lineOut0[col] = waterx;
            }
        }
        //Write image to disk through GDAL
        GDALRasterIO(hBandOut0,GF_Write,0,row,nXSize1,1,lineOut0,nXSize1,1,GDT_Float32,0,0);
    }
    if( line1 != NULL ) free( line1 );
    if( line2 != NULL ) free( line2 );
    if( line7 != NULL ) free( line7 );
    if( line8 != NULL ) free( line8 );
    if( line9 != NULL ) free( line9 );
    if( lineOut0 != NULL ) free( lineOut0 );
    GDALClose(hDataset1);
    GDALClose(hDataset2);
    GDALClose(hDataset7);
    GDALClose(hDataset8);
    GDALClose(hDataset9);
    GDALClose(hDatasetOut0);
}
```

## 2.2 Python-GDAL scripting

### 2.2.1 Introduction

Python (www.python.org), is an object-oriented high-level programming language, also called by some a scripting language, because it does not need explicit compilation into an application binary file to be used. Python-GDAL bindings (pypi.python.org/pypi/GDAL/) are a number of tools for programming and manipulating geospatial raster data.
The purpose of this example code is to calculate band-wise top of atmosphere reflectance, and use them to calculate NDVI and Albedo for a Landsat 7 ETM+ image of around Ubon Ratchathani in North Eastern Thailand. The image can be downloaded from the Global Land Cover Facility website (glcf.umiacs.umd.edu), or alternatively a direct download link is available in the beginning of the code itself. It has to be added that top of atmosphere reflectance can be converted to surface reflectance by using an atmospheric correction model such as 6S (6s.ltdri.org), this is not covered in here.

### 2.2.2 Code explanation

The initial part of the program requires to load GDAL libraries and its dependent friend, the numerical python (numpy.scipy.org) library. This provides the supporting functions for opening files (gdal_array. LoadFile()), saving the output array (SaveArray()), and towards the end of the code, some more functions used for saving the last output file (Albedo) with projection and georeference information.

```
# For Image Processing
from math import *
import numpy
from osgeo import gdalnumeric
from osgeo import gdal
from osgeo.gdal_array import *
from osgeo.gdalconst import *
```

Since we are going to convert Level 1G Digital Numbers (DN) into Top of Atmosphere reflectance, we need to load appropriate constants. There are DN to radiance conversion parameters (L1Max, L1min, etc.), exo-atmospheric band-wise irradiances (KEXO1, etc.), sun elevation (sun_elevation), day of year (doy), and a combined solar/astronomical constant (constant).

```
# DN to radiance conversion factors
#(Check if they are correct for your image)
LMax = [191.6,196.5,152.9,241.1,31.060,10.8]
Lmin = [-6.2,-6.4,-5.0,-5.1,-1.0,-0.35]
#Exo-Atmospheric band-wise irradiance
KEXO = [1970.0,1843.0,1555.0,1047.0,227.1,80.53]
# Unless you use p127r049 2000/11/04, these must be changed
sun_elevation = 51.3428710
```

```
doy = 311
# Cos(sun zenith angle) / (pi * (Sun-Earth distance)^2)
constant=(cos ( (90-sun_elevation)  *  pi/180 )  / ( pi * (1+0.01672 *
        sin ( 2 * pi * (doy - 93.5 ) / 365))**2))
```

Let us suppose we try to do the same processing on each band. It starts with the creation of an object "b", by loading the file with python-GDAL into an array. "b" is then used into the creation of "result", an array holding the top of atmosphere reflectance in the respective band processed. Finally, the array "result" is saved in a file on disk with an appropriate filename. It has to be noted in this case that the file will not be having any projection and other georeference. Finally, all the objects in memory are flushed out of the RAM (del result,b).

```
# Landsat Band 1
b =gdal_array. LoadFile('b1.tif')
result = ((b * ((L1Max-L1min)/255.0) + L1min) / (constant * KEXO1))
SaveArray(result,'b1_ref.tif','GTiff')
#Empty memory objects
del result,b
```

This is nice, but not really efficient to repeat several times the same for each of the input band files. So we implement a loop over the filenames and input parameters. This makes more sense then, since we now have the power to import Digital Number (DN) values and compute them into reflectance at top of atmosphere, this for each band in one list of tasks repeated over all input files. We start by creating a object have 5 subscript slots, each one to hold the 5 bands to be processed. After that, we can start our loop, loading each of the 5 input files using LoadFile() with their filenames. We return the output array "b[i]" and we set ourselves to create the output file name, making a case to skip the number 6, since band 6 of Landsat 5TM is a temperature band, thus not of our interest for NDVI and Albedo processing.
Once the object "b[i]" received its data, we are interested in saving it to file with projection and georeference information, so that it can be used in GIS and remote sensing software later. This involves copying this specific information from another file, here it is band 1 ("b1.tif"). We use GetGeoTransform() and GetProjection() for that purpose. Once this information in memory, we open the albedo array with gdal_array.OpenArray() and set the projection/georeference of its object ("out") using SetProjection() and SetGeoTransform(). We then set the definition of the output image format driver (here ENVI), once this is initialized, we can use the driver to create a GeoTiff output file, using the filename "albedo" and the projected/georeferenced GDAL array "A". Once we are through we clear the memory from the unnecessary objects, including the GDAL array "A".

```
#satellite has 5 bands, so 5 array handle objects
b=[0,0,0,0,0,0]
#Top Of Atmosphere Reflectance
for i in range (0,6):
    DN = LoadFile(F[i])
    b[i] = (DN * ((LMax[i]-Lmin[i])/255.0) + Lmin[i]) / (constant * KEXO[i])
```

```
        #Create output file name
        if i != 5:
            output_filename='b'+str(i+1)+'_ref'
        else: #skip band 6 which is temperature, not shortwave reflectance
            output_filename='b'+str(i+2)+'_ref'
        #Create output file with projection
        A = OpenArray(b[i])
        geot=A.SetGeoTransform( geoT )
        proj=A.SetProjection( proJ )
        output=driver.CreateCopy(output_filename, A)
        #Empty memory objects
        del A, DN, output, geot, proj
```

Once all bands are processed to their respective Top Of Atmosphere Reflectance (TOAR), NDVI is calculated in the same way, using Landsat 5TM band 3 and 4 at TOAR, which are respectively "b[2]" and "b[3]" in our memory handling object "b".

```
    #NDVI=(NIR-Red)/(NIR+Red)
    ndvi = (b[3]-b[2])/(b[3]+b[2])
    #prepare/create output file with projection
    ndvi = OpenArray(ndvi)
    ndvi.SetGeoTransform( geoT )
    ndvi.SetProjection( proJ )
    driver.CreateCopy('ndvi', ndvi)
    #Empty memory objects
    del ndvi
```

Eventually, the Albedo TOAR can also be calculated and written on disk with geographical information attached.

```
    #Broadband Albedo (0.3-3 micrometers)
    albedo = (0.293 * b[0]) + (0.274 * b[1]) + (0.233 * b[2]) + ( 0.156 * b[3])
            + (0.033 * b[4]) + ( 0.011 * b[5])
    #prepare/create output file with projection
    albedo = OpenArray(albedo)
    albedo.SetGeoTransform( geoT )
    albedo.SetProjection( proJ )
    driver.CreateCopy('albedo', albedo)
```

Finally, free the last memory objects.

```
    #Empty memory objects
    del albedo, b
```

## 2.2.3 Python Code

```
# Landsat 7 ETM + image downloaded from GLCF
#ftp.glcf.umiacs.umd.edu/glcf/Landsat/WRS2/p127/r049/
#       p127r049_7x20001104.ETM-EarthSat-Orthorectified/
#p127r049 2000/11/04 around Ubon Ratchathani, North East Thailand.
# Metadata in the text file: p127r049_7x20001104.met

#Requires >8Gb RAM to run completely

#Purpose: Calculate NDVI and Albedo
#Usage: /usr/bin/python -u "ndvi_albedo.py"

#if problem of libgrass: MAPSET is not set
#then: set GDAL_SKIP = GRASS

F=[]
F.append('p127r049_7t20010518_z48_nn10.tif')
F.append('p127r049_7t20010518_z48_nn20.tif')
F.append('p127r049_7t20010518_z48_nn30.tif')
F.append('p127r049_7t20010518_z48_nn40.tif')
F.append('p127r049_7t20010518_z48_nn50.tif')
F.append('p127r049_7t20010518_z48_nn70.tif')

# For Image Processing
from math import *
import numpy
from osgeo import gdalnumeric
from osgeo import gdal
from osgeo.gdal_array import *
from osgeo.gdalconst import *

# Set our output file format driver
driver = gdal.GetDriverByName( 'ENVI' )

# Set our output files Projection parameters
# from input file number 1
tmp = gdal.Open(F[0])
geoT = tmp.GetGeoTransform()
proJ = tmp.GetProjection()
del tmp

#DN to radiance conversion factors
#(Check if they are correct for your image)
LMax = [191.6,196.5,152.9,241.1,31.060,10.8]
Lmin = [-6.2,-6.4,-5.0,-5.1,-1.0,-0.35]
#Exo-Atmospheric band-wise irradiance
KEXO = [1970.0,1843.0,1555.0,1047.0,227.1,80.53]
#Unless you use p127r049 2000/11/04, these must be changed
sun_elevation = 51.3428710
doy = 311
#Cos(sun zenith angle) / (pi * (Sun-Earth distance)^2)
constant=(cos((90-sun_elevation) * pi/180 ) / (pi * (1+0.01672 * sin
(2 * pi * (doy - 93.5) / 365)) ** 2))

#satellite has 5 bands, so 5 array handle objects
b=[0,0,0,0,0,0]
#Top Of Atmosphere Reflectance
for i in range (0,6):
    DN = LoadFile(F[i])
    b[i] = (DN * ((LMax[i]-Lmin[i])/255.0) + Lmin[i]) / (constant * KEXO[i])
    if i != 5:
        output_filename='b'+str(i+1)+'_ref'
    else: #skip band 6 which is temperature, not shortwave reflectance
        output_filename='b'+str(i+2)+'_ref'
    #Create output file with projection
    A = OpenArray(b[i])
    geot=A.SetGeoTransform( geoT )
    proj=A.SetProjection( proJ )
    output=driver.CreateCopy(output_filename, A)
    #Empty memory objects
    del A, DN, output, geot, proj


#NDVI=(NIR-Red)/(NIR+Red)
ndvi = (b[3]-b[2])/(b[3]+b[2])
#prepare/create output file with projection
out = OpenArray(ndvi)
out.SetGeoTransform( geoT )
out.SetProjection( proJ )
driver.CreateCopy('ndvi', out)
#Empty memory objects
del out, ndvi

#Broadband Albedo (0.3-3 micrometers)
albedo = (0.293 * b[0]) + (0.274 * b[1]) + (0.233 * b[2]) + ( 0.156 *
b[3]) + (0.033 * b[4]) + ( 0.011 * b[5])
#prepare/create output file with projection
albedo = OpenArray(albedo)
albedo.SetGeoTransform( geoT )
albedo.SetProjection( proJ )
driver.CreateCopy('albedo', albedo)
#empty memory objects
del albedo, b
```

# Chapter 3

# R Statistical Language

## 3.1 R raster scripting

### 3.1.1 Introduction

R statistical software (www.r-project.org), is actually a high-level programming language in itself, initially designed after the S statistical language, it has now become a large open source repository of complex algorithms and scientific models. Because of its high-level, R is short-handed, yet a powerful analytical tool that new generation of Life scientists should have in their toolbox.

In this example we will use the "RemoteSensing" (remote sensing operations) package, available directly from inside the R command prompt by typing:

*install.packages("RemoteSensing",repos="http://R-Forge.R-project.org")*

The "RemoteSensing" package has the following dependencies: "raster" (raster-based operations), "bitops" (bit operations), "sp" (spatial operations) and "rgdal" (GDAL for R). All dependencies are installable in the same way as any other R package using *install.packages("package_name").*

The "raster" dependency package deals with basic spatial raster (grid) data access and manipulation. It can deal with very large files and includes standard raster calculations such as overlay, aggregation, and merge. It builds on two other packages, a spatial data handler "sp" and our most known GDAL library wrapped into R under the name of "rgdal". The "RemoteSensing" package provides basic satellite image processing functions such as pre-processing, some indices like Albedo, water mapping and vegetation indices (NDVI, SAVI, etc...).

### 3.1.2 Code explanation

In R, you first start by setting your working directory, using the function setwd(). This will define where your input/output data will be found/generated. The example below show a typical Unix/Linux working directory.

```
# Set Working Directory
setwd("/home/user/RSDATA/Modis")
```

After that, you need to load the geoInformation handling library "raster" and the remote sensing library "RemoteSensing". This will enable you to call, process and write geoInformation on the fly.

```
# Load Libraries
library(raster)
library(RemoteSensing)
```

Once this is ready, load your satellite images in R memory, using the rasterFromFile() and rasterReadAll() functions. The first one sets up the raster information the second one loads data into the R memory object.

```
# Load files from Hard Disk
band1 <- raster("MODIS_band1.tif")
band2 <- raster("MODIS_band2.tif")
band1 <- readAll(band1)
band2 <- readAll(band2)
```

In case of MODIS data, the original files are stored in integer, with a multiplying factor of 10000. Thus, we have to rescale the data found in memory.

```
# Prepare Data
b1 <- band1/10000.0
b2 <- band2/10000.0
```

At this point all is ready to process a given vegetation index as NDVI. As you can notice, this is a single line of code.

```
# Process a Vegetation Index
ndvi <- ndvi(b1,b2)
```

We complete the processing by writing the raster file to disk, which requires first to define the output file name. The writeRaster() function is also a single line and requires the raster image output format, in this case GeoTiff. The full list of output format is available from the common GDAL sources.

```
# Write to disk
filename(ndvi) <- 'ndvi.tif'
ndvi <- writeRaster(ndvi, format='GTiff')
```

You may want to display the NDVI output map, using the plot() function. Here we add also a greyscale color palette.

```
plot(ndvi, col=grey(0:255/255))
```

As simple as it can be...

Additionally, If you want to display the Thailand country boundary over your NDVI image, you have to download the Thailand country boundary, reproject it and add it in red color to the NDVI image plot. Any country abbreviated name can be called and downloaded on the fly. Levels are corresponding to administrative levels of www.gadm.org.

```
#Admin boundaries plotting over the image
#In this case, Thailand boundary is loaded
tha <- getData("GADM", country="THA", level=1)
plot(tha,add=T,border="red")
```

If you are using data not in Geographical (Lat/long) projection, you can reproject the administrative boundary on the fly with:

```
#Landsat Data: lat/long to UTM for country boundaries
thailand<-spTransform(tha,
        CRS("+proj=utm +zone=51 +ellps=WGS84 +datum=WGS84 +units=m +no_defs"))
```

### 3.1.3 R code

```
# Set Working Directory
setwd("/home/user/RSDATA/Modis")

# Load Libraries
library(raster)
library(RemoteSensing)

# Load files from Hard Disk
band1 <- raster(MODIS_band1.tif)
band2 <- raster(MODIS_band2.tif)
band1 <- readAll(band1)
band2 <- readAll(band2)

# Prepare Data
b1 <- band1/10000.0
b2 <- band2/10000.0

# Process a Vegetation Index
ndvi <- ndvi(b1,b2)

# Write to disk
filename(ndvi) <- 'ndvi.tif'
ndvi <- writeRaster(ndvi, format='GTiff')

# Display
plot(ndvi, col=grey(0:255/255))

# Clean up memory
rm(b1,b2,ndvi)

#Admin boundaries plotting over the image
#In this case, Thailand boundary is loaded
tha <- getData("GADM", country="THA", level=1)
plot(tha,add=T,border="red")
```

# Chapter 4

# Parallel & Distributed Computing

## 4.1 OpenMP parallel computing

### 4.1.1 Introduction

OpenMP (openmp.org) is an easy parallel programming addition to C, C++ and Fortran languages for shared-memory computers. What are shared-memory computers? The most common is the multi-core CPU of current generation of desktop/laptop. By multi-core, understand that the processing units are many (2, 4, 8, etc...) inside a single CPU. They share the computer memory (RAM). OpenMP is actually a set of pre-processor instructions (also called directives when dealing with Fortran), enabling serial appplication to be converted to parallel applications with a minimal amount of code modification using the concept of threads. It is easy to imagine a string being a collection of threads. Likewise, a thread of execution, in computer science, is a computer program that can be forked into smaller execution tasks.

One of the most important concept in OpenMP implementation is directly linked to the physical architecture of the n-core CPU, the shared-memory. Variables declared before the code becomes parallel (i.e. before *#pragma omp parallel*) can be sent into each core either as *private* or *shared*. A private variable will be duplicated for each thread. All links to the original variable re-routed to the new private duplicate. No other thread will be able to access it. On the opposite scenario, a shared variable is going to be accessed by all threads at the same time. If one considers a dual-core machine (0 and 1), a shared variable has a direct implication for loop execution as a loop control variable (i.e. row<10) will be executed on core number 0 in range of 0<row<4 and on core number 1 in range of 5<row<10.

The files holding remote sensing algorithms functions (rs.c) and algorithms functions prototypes (rs.h) can be found in Appendix A and are joined to the compilation process in the Makefile. An example Makefile is below:

```
ndvi_omp: ndvi_omp.c rs.c
    gcc -o ndvi_omp ndvi_omp.c rs.c -fopenmp -I/usr/include/gdal/ -L/usr/lib -lgdal1.7.0
```

In this example, the application does not take any argument, because file names are “hard-coded” into the application source code. It can be run by:

```
./ndvi_omp
```

## 4.1.2 Code explanation

A C source file that needs to enable OpenMP capacities will first load its header file, as seen in the second line of includes below.

```
#include<stdio.h>
#include<omp.h>
#include"gdal.h"
#include"rs.h"
```

As we have seen before in the chapter 2, we start our simplified coding with a default loading of fixed names of images (“b3.tif” and “b4.tif”), after testing for their existance, loading the file geographical formats drivers, creating an output file (“ndvi.tif”) and loading the image band handles, we are set to start the row by row processing. We load the rows with the GDALRasterIO() function.

We are going to instruct OpenMP that the pixels should be processed in parallel on each of the core of the CPU. This means that a given pixel will go to a given core, which we do not know beforehand, as jobs are distributed to cores on ”first come first serve” basis, depending on core availability. OpenMP uses the pre-processor instruction #pragma omp, to indicate to the compiler that this information has to be interpreted separately (by the OpenMP additional library of the compiler) to efficiently distribute each of the loop sub-processes (the raster pixels) to any available core as they indicate so.

Once the for loop at “col” level is started we have to consider this as being an independent process, possibly in a remote core. Each process in each core will access the same row memory slot and read its own col value to get the pixel data from the shared row memory slots.

After reading each input row image with the GDALRasterIO() function, we can now load the pixel data from the image row in parallel inside a shared column-based loop to access each pixel, and then process the NDVI output pixel. Once all the pixels in the row are processed, we use GDALRasterIO() function again with a GF_Write argument to write the row to the output raster image file on disk. We also free the memory allocated each time.

```
        //Processing the data cellxcell in parallel
        #pragma omp parallel for default(none) private(col) \
                shared(nX,line1,line2,lineOut)
        for(col=0;col<nX;col++){
            if(line1[col] < -1.0)
                lineOut[col] = 0.0;
            else
                lineOut[col] = ndvi(line1[col],line2[col]);
        }
```

Finally, we force all row processes on all cores to finish before closing the input and output files. We use the *barrier* instruction, that is a standard name in parallel and distributed programming to indicate the end of a parallel code section. What *barrier* is doing is called *synchronizing working threads*. Threads are *instructions streams*. They handle a set of calculations, and in our case, OpenMP sends threads to each CPU core. Thus, *barrier* is synchronizing the completion of all of the instruction streams sent to each core before giving the control back to the software.

```
        #pragma omp barrier
        GDALRasterIO(hBandOut,GF_Write,0,row,nX,1,lineOut,nX,1,GDT_Float32,0,0);
    }
    if( line1 != NULL )
        free( line1 );
    if( line2 != NULL )
        free( line2 );
    if( lineOut != NULL )
        free( lineOut );
    GDALClose(hDataset1);
    GDALClose(hDataset2);
    GDALClose(hDatasetOut);
```

### 4.1.3 OpenMP C code

```
#include<stdio.h>
#include<omp.h>
#include"gdal.h"
#include"rs.h"

int main()
{
    GDALAllRegister();
    GDALDatasetH hDataset1 = GDALOpen("b3.tif",GA_ReadOnly);
    GDALDatasetH hDataset2 = GDALOpen("b4.tif",GA_ReadOnly);
    if(hDataset1==NULL||hDataset2==NULL){
        printf("Unable to load at least one input file\n");
        exit(EXIT_FAILURE);
    }

    //Loading the file infos
    GDALDriverH hDriver1 = GDALGetDatasetDriver(hDataset1);
    GDALDriverH hDriver2 = GDALGetDatasetDriver(hDataset2);

    //Creating output file
    GDALDatasetH hDatasetOut=GDALCreateCopy(hDriver1,"ndvi.tif",hDataset1,FALSE,NULL,NULL,NULL);
    GDALRasterBandH hBandOut = GDALGetRasterBand(hDatasetOut,1);

    //Loading the file bands
    GDALRasterBandH hBand1 = GDALGetRasterBand(hDataset1,1);
    GDALRasterBandH hBand2 = GDALGetRasterBand(hDataset2,1);

    //Parameters for loading the data rowxrow
    int nX = GDALGetRasterBandXSize(hBand1);
    int nY = GDALGetRasterBandYSize(hBand1);

    //Allocate Memory for line Buffers
    float *line1 = (float *) CPLMalloc(sizeof(float)*nX);
    float *line2 = (float *) CPLMalloc(sizeof(float)*nX);
    float *lineOut = (float *) CPLMalloc(sizeof(float)*nX);
    int row,col;

    //Accessing the data rowxrow
    for(row=0;row<nY;row++){
        //Fetch line buffers from image files
        GDALRasterIO(hBand1,GF_Read,0,row,nX,1,line1,nX,1,GDT_Float32,0,0);
        GDALRasterIO(hBand2,GF_Read,0,row,nX,1,line2,nX,1,GDT_Float32,0,0);
        //Processing the data cellxcell and in parallel
        #pragma omp parallel for default(none) private(col) \
            shared(nX,line1,line2,lineOut)
        for(col=0;col<nX;col++){
            if(line1[col] < -1.0){
                /*skip it*/
                lineOut[col] = 0.0;
            } else {
                lineOut[col] = ndvi(line1[col],line2[col]);
                lineOut[col] += 1.0;
                lineOut[col] *= 100.0;
            }
        }
        #pragma omp barrier
        GDALRasterIO(hBandOut,GF_Write,0,row,nX,1,lineOut,nX,1,GDT_Float32,0,0);
    }
    if( line1 != NULL )
        free( line1 );
    if( line2 != NULL )
```

```
        free( line2 );
    if( lineOut != NULL )
        free( lineOut );
    GDALClose(hDataset1);
    GDALClose(hDataset2);
    GDALClose(hDatasetOut);
}
```

## 4.2 MPI cluster computing

### 4.2.1 Introduction

In this section, we are dealing with what is called “cluster computing”. Cluster computing is the ability of several computers to act as if they were a single one. A computer cluster is generally set up by networking several computers together through a high speed network switch. The software installed in all of the computers is setup in such a way that, generally, one computer can order all other computers to perform tasks on demand. Cluster computing is a useful/fascinating science and technology, permitting advances in many sciences and book-keeping in many financial institutions. More recently, it came to the public as the technology behind animation movies, which are rendered on “cluster computing farms”. Message Passing Interface (MPI; www.mpi-forum.org) is an addendum to the C (and FORTRAN) language to enable the communication between any two computers, both part of a “cluster computer”. Each computer that is a part of a cluster computer is called a “node”. In our type of processing, we will differentiate one node from all of the others. This node will be the one used to launch the program, and is often the same used for most management operations on the cluster. This node is referred to as the “master node” or also “front node”. The latter name being that this computer is the visible access to the cluster, in front of all the other “working nodes” or “slave nodes”. Sometimes the combination of “master-worker” is used to describe the broadcasting and gathering/harvesting of data in between the dispatcher of computing jobs and the nodes actually crunching the numbers.

If you need to setup your own cluster computer, you will need a minimum of 3 computers, one switch (preferably a fast one), and a cluster computing operating system. This code was used in a ROCKS cluster setup, which can be downloaded from the ROCKS website (www.rocksclusters.org). The installation is mostly automatic, the master node first, and then auto-install the two others by booting them from the network installation server of the master node. It takes about one day for a first timer.

To run this code on a cluster computer, you will first need to install GDAL library only on your “front” or “master” node. Once the row data from the image has been given to its row size memory, GDAL has fulfilled its use. To create the application, copy the code from this booklet in a file mpi_ndvi.c and compile it with mpicc:

```
mpicc -o ndvimpi mpi_ndvi.c rs.c -I/usr/local/include -L/usr/local/lib -lgdal1.7.0
```

and run it:

```
mpirun -np 10 mpindvi inB1 inB2 in250QC in500stateQA outNDVI
```

The argument “-np 10” indicates the number of nodes you want to run your application on. In this case we suppose that we have 11 computers in the cluster, 1 master node and 10 worker nodes. The files rs.c and rs.h can be found in Appendix A. An example Makefile is below:

```
mpindvi: mpi_ndvi.c rs.c
    mpicc -o mpindvi mpi_ndvi.c rs.c -I/usr/local/include -L/usr/local/lib -lgdal1.7.0
```

### 4.2.2 Code explanation

There are two important informations to remember when working on a MPI code. The first one is that the total number of communications from the master node should always be equal to the sum of receiving communications of the workers nodes. The same is true on the returning of processed data from the workers to the master. This has a direct coding corollary, a distribution loop in the master will loop n times, likewise, the reception/processing loop for the workers will also loop the same n times. It is easy to check, and ensures a working code quality. The second is that Message Passing can be reduced to two functions, MPI_Send() and MPI_Recv(). All other data communications functions are definitely interesting, important and sometimes simplifying a lot our computations, but they are all derived from these atomic data transport functions. Knowing those two well, will already bring you a long way into RS cluster computing.
Any MPI-enabled C code has to include the incantations below, all MPI C code starts with the function “MPI_Init()” and finishes by the closing function “MPI_Finalize()”. Those two critical functions set up and destroy the interface for the communication of data in between nodes.

```
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    >>>>> Include MPI managed parallel processing here <<<<<
    MPI_Finalize();
```

The MPI functions MPI_Comm_size() is designed to automatically assess the number of nodes operational in the cluster. After MPI_Init(), all the remaining code will be simultaneously read by all nodes in the cluster, so when MPI_Comm_rank() will be called, each node will store its own identification number locally, corresponding to its own activation rank within the internal distribution algorithm of the cluster.
Before actually processing row data, you maybe want to send values that will be constant throughout the image processing. It is judicious to do that once and for all, because all communication has a cost in time of processing. Below is the loop sending the image dimensions to all worker nodes. What happens is that this is a loop done from inside a code section answering only to id=0, meaning, that this is master node section only. From the master node, this loop will send to all

nodes with rank identifications from 1 to p (p=10 in our example), the image size in X and Y dimensions.

```
for (n=1;n<p;n++){
    MPI_Send(&nXSize1,1,MPI_INT,n,1,MPI_COMM_WORLD);
    MPI_Send(&nYSize1,1,MPI_INT,n,1,MPI_COMM_WORLD);
}
```

On the other side, when id != 0 (see further down in the code), we are in the worker nodes, and they should also receive the image dimensions.

```
//Receive from Master max rows and Max columns
MPI_Recv(&nXSize1,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
MPI_Recv(&nYSize1,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
```

Now that we have received the constants, we can start a loop in the master node to send the row data to the worker nodes. To do so, we first loop through the rows with an increment equal to the number of worker nodes. Indeed, we will sub-loop through each worker nodes to distribute jobs.

```
//Accessing the data rowxrow. We have p-1 slaves
for(row=0;row<nYSize1;row+=p-1){
    for(n=1;n<p;n++){
```

Since this is a simplistic application, we do not control for the exact number of the remaining number of available rows in the image divided by the number of available worker nodes. To avoid any calling of row number higher than the maximum image row size, we use an if statement. Once this is done, we can use GDAL to get the row data (row+n-1) and MPI_Send() it to the according worker node (n). Reversely, MPI_Recv() the output row from worker node n and saving it to the appropriate image row using GDAL. Remark the MPI_Send()/MPI_Recv() 4th & 5th arguments are both n & 1, meaning that from a worker node perspective, you receive data from 1 (master node) to your n rank, and you send data from your n rank to 1 (master node).

```
if((row+n-1)<nYSize1){
    //GET IMAGE ROW DATA
    GDALRasterIO(hBand1,GF_Read,0,row+n-1,nXSize1,1,line1,
        nXSize1,1,GDT_Float32,0,0);
    ...
    //SEND TO WORKERS
    MPI_Send(line1,nXSize1,MPI_FLOAT,n,1,MPI_COMM_WORLD);
    ...
    //RECEIVE FROM WORKERS
    MPI_Recv(lineOut1,nXSize1,MPI_FLOAT,n,1,MPI_COMM_WORLD,&status);
    //PUT IMAGE ROW DATA
    GDALRasterIO(hBandOut1,GF_Write,0,row+n-1,nXSize1,1,lineOut1,
        nXSize1,1,GDT_Float32,0,0);
```

In the workers nodes, we simply need to loop the same number of times as in the master node and for each loop receive the data for it, process and send the output to the master node.

```
//Accessing the data rowxrow
    for(row=0;row<nYSize1/(p-1);row++){
        //Receive from Master the row data
        MPI_Recv(data1,nXSize1,MPI_FLOAT,0,1,MPI_COMM_WORLD,
                &status);
```

## 4.2.3 MPI C code

```
#include<stdio.h>
#include"gdal.h"
#include"mpi.h"
#include"rs.h"

void usage()
{
    printf( "-----------------------------------------\n");
    printf( "--Modis Processing example --MPI code----\n");
    printf( "-----------------------------------------\n");
    printf( "mpirun -np 10 mp  inB1 inB2 in250QC in500stateQA outNDVI\n");
    printf( "-----------------------------------------\n");
    printf( "inB1 and inB2 are Modis 250m\n");
    printf( "in250QC is Modis 250m Quality Assessment\n");
    printf( "in500stateQA is Modis 500m State Quality Assessment\n");
    printf( "outNDVI is the NDVI output [-1;+1]->[0-100]\n");
    return;
}

int main(int argc , char* argv[]){
    int i, row, col;
    int n=0;    //worker processor allocation variable
    int nXSize1, nYSize1;
    if( argc < 5 ) {
        usage();
        exit(EXIT_FAILURE);
    }
    //Loading the input files names
    char    *inB1, *inB2;
    char    *in250QC, *in500QC;
    char    *ndviF;
    float   temp, tempval;
    inB1        = argv[1];
    inB2        = argv[2];
    in250QC     = argv[3];
    in500QC     = argv[4];
    ndviF       = argv[5];

    // MPI STUFF
    int     id;                                 // will be master if id = 0
    int     p;                                  // number of processors online
    int     name_length;                        // length of name of processor
    char    proc_name[MPI_MAX_PROCESSOR_NAME];  // Processor name
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    MPI_Get_processor_name(proc_name,&name_length);

    if(id==0)
        printf("I am HPC \"%s\"\n",proc_name);
    else
        printf("I am processor %i of %i\n",id,p);

    // We are in MASTER NODE (MPI_Comm_rank=id=0)
    if(id==0){
        GDALAllRegister();

        //Loading the input files
        GDALDatasetH hDataset1;  //Red Band
        GDALDatasetH hDataset2;  //NIR Band
        GDALDatasetH hDataset8;  //Quality Assessment 250m
        GDALDatasetH hDataset9;  //Quality Assessment 500m

        hDataset1 = GDALOpen(inB1,GA_ReadOnly);
        hDataset2 = GDALOpen(inB2,GA_ReadOnly);
        hDataset8 = GDALOpen(in250QC,GA_ReadOnly);
        hDataset9 = GDALOpen(in500QC,GA_ReadOnly);

        if(hDataset1==NULL||hDataset2==NULL||hDataset8==NULL||hDataset9==NULL){
            printf("One or more input files could not be loaded\n");
            exit(1);
        }

        //Loading the file infos
        GDALDriverH hDriver1;
        GDALDriverH hDriver2;
        GDALDriverH hDriver8;
        GDALDriverH hDriver9;

        hDriver1 = GDALGetDatasetDriver(hDataset1);
        hDriver2 = GDALGetDatasetDriver(hDataset2);
        hDriver8 = GDALGetDatasetDriver(hDataset8);
        hDriver9 = GDALGetDatasetDriver(hDataset9);

        //Creating output file (NDVI)
        GDALDatasetH    hDatasetOut1;
        hDatasetOut1 = GDALCreateCopy(hDriver1,ndviF,hDataset1,FALSE,NULL,NULL,NULL);
        GDALRasterBandH     hBandOut1;
        hBandOut1   = GDALGetRasterBand(hDatasetOut1,1);

        //Loading the file bands
        GDALRasterBandH hBand1;
        GDALRasterBandH hBand2;
        GDALRasterBandH hBand8;
        GDALRasterBandH hBand9;

        hBand1  = GDALGetRasterBand(hDataset1,1);
        hBand2  = GDALGetRasterBand(hDataset2,1);
        hBand8  = GDALGetRasterBand(hDataset8,1);
        hBand9  = GDALGetRasterBand(hDataset9,1);

        nXSize1     = GDALGetRasterBandXSize(hBand1);
        nYSize1     = GDALGetRasterBandYSize(hBand1);

        float   *line1;     //Red Band
        float   *line2;     //NIR Band
        int     *line8;     //250m QC
        int     *line9;     // 500m QC
        float   *lineOut1;  //NDVI output

        line1       = (float *) CPLMalloc(sizeof(float)*nXSize1);
        line2       = (float *) CPLMalloc(sizeof(float)*nXSize1);
        line8       = (int *) CPLMalloc(sizeof(int)*nXSize1);
        line9       = (int *) CPLMalloc(sizeof(int)*nXSize1/2);
        lineOut1    = (float *) CPLMalloc(sizeof(float)*nXSize1);

        for (n=1;n<p;n++){
            MPI_Send(&nXSize1,1,MPI_INT,n,1,MPI_COMM_WORLD);
            MPI_Send(&nYSize1,1,MPI_INT,n,1,MPI_COMM_WORLD);
        }
        //Accessing the data rowxrow. We have p-1 slaves
        for(row=0;row<nYSize1;row+=p-1){
            for(n=1;n<p;n++){
                if((row+n-1)<nYSize1){
                //GET IMAGE ROW DATA
                GDALRasterIO(hBand1,GF_Read,0,row+n-1,nXSize1,1,line1,
                                    nXSize1,1,GDT_Float32,0,0);
                GDALRasterIO(hBand2,GF_Read,0,row+n-1,nXSize1,1,line2,
                                    nXSize1,1,GDT_Float32,0,0);
                GDALRasterIO(hBand8,GF_Read,0,row,nXSize1,1,line8,
                                    nXSize1,1,GDT_UInt32,0,0);
                GDALRasterIO(hBand9,GF_Read,0,row/2,nXSize1/2,1,line9,
                                    nXSize1/2,1,GDT_UInt32,0,0);
                //SEND TO WORKERS
                MPI_Send(line1,nXSize1,MPI_FLOAT,n,1,MPI_COMM_WORLD);
                MPI_Send(line2,nXSize1,MPI_FLOAT,n,1,MPI_COMM_WORLD);
                MPI_Send(line8,nXSize1,MPI_INT,n,1,MPI_COMM_WORLD);
                MPI_Send(line9,nXSize1/2,MPI_INT,n,1,MPI_COMM_WORLD);
                //RECEIVE FROM WORKERS
                MPI_Recv(lineOut1,nXSize1,MPI_FLOAT,n,1,MPI_COMM_WORLD,&status);
                //PUT IMAGE ROW DATA
                GDALRasterIO(hBandOut1,GF_Write,0,row+n-1,nXSize1,1,lineOut1,
                                    nXSize1,1,GDT_Float32,0,0);
                }else{
                //Excess workers
                MPI_Send(line1,nXSize1,MPI_FLOAT,n,1,MPI_COMM_WORLD);
                MPI_Send(line2,nXSize1,MPI_FLOAT,n,1,MPI_COMM_WORLD);
                MPI_Send(line8,nXSize1,MPI_INT,n,1,MPI_COMM_WORLD);
                MPI_Send(line9,nXSize1/2,MPI_INT,n,1,MPI_COMM_WORLD);
                MPI_Recv(lineOut1,nXSize1,MPI_FLOAT,n,1,MPI_COMM_WORLD,&status);
                }
            }
        }
        if( line1 != NULL )
            free( line1 );
        if( line2 != NULL )
            free( line2 );
        if( line8 != NULL )
            free( line8 );
        if( line9 != NULL )
            free( line9 );
        if( lineOut1 != NULL )
            free( lineOut1 );
        GDALClose(hDatasetOut1);
        GDALClose(hDataset1);
        GDALClose(hDataset2);
        GDALClose(hDataset8);
        GDALClose(hDataset9);
    } // END OF MASTER
    // WE ARE IN WORKER NODES
    else {
        int     col = 0;
        float   ndvix,*data1,*data2,*data7,*data8,*data9,*dataOut1;

        //Receive from Master max rows and Max columns
        MPI_Recv(&nXSize1,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
        MPI_Recv(&nYSize1,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);

        //Allocate memory for the row data
        data1   = (float *) CPLMalloc(sizeof(float)*nXSize1);
        data2   = (float *) CPLMalloc(sizeof(float)*nXSize1);
        data8   = (int *) CPLMalloc(sizeof(int)*nXSize1);
        data9   = (int *) CPLMalloc(sizeof(int)*nXSize1/2);
        dataOut1    = (float *) CPLMalloc(sizeof(float)*nXSize1);
```

```
        //Accessing the data rowxrow
        for(row=0;row<nYSize1/(p-1);row++){
            //Receive from Master the row data
            MPI_Recv(data1,nXSize1,MPI_FLOAT,0,1,MPI_COMM_WORLD,&status);
            MPI_Recv(data2,nXSize1,MPI_FLOAT,0,1,MPI_COMM_WORLD,&status);
            MPI_Recv(data8,nXSize1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
            MPI_Recv(data9,nXSize1/2,MPI_INT,0,1,MPI_COMM_WORLD,&status);
            for(col=0;col<nXSize1;col++){
                tempval = stateqa500a(data9[col/2]);
                temp = qc250a(data8[col]);
                if(data1[col]==-28672||temp>1.0||tempval>=1.0){ /*skip it*/
                    if(temp>1.0)
                        dataOut1[col]=10.0;
                    else if (tempval>=1.0){
                        dataOut1[col]=100.0;
                    else
                        dataOut1[col]=-28672;
                } else {
                    //NDVI Calculation
                    ndvix = ndvi(data1[col],data2[col]);
                    //Store NDVI =(1+ndvi)*100
                    dataOut1[col] = ndvix + 1.0;
                    dataOut1[col] *= 100.0;
                }
            }
            //Send the output row to the Master
            MPI_Send(dataOut1,nXSize1,MPI_FLOAT,0,1,MPI_COMM_WORLD);
        }
        if( data1 != NULL )
            free( data1 );
        if( data2 != NULL )
            free( data2 );
        if( data8 != NULL )
            free( data8 );
        if( data9 != NULL )
            free( data9 );
        if( dataOut1 != NULL )
            free( dataOut1 );
    }
    MPI_Finalize();
}
```

# Chapter 5

# GPGPU Computing

## 5.1 CUDA with GDAL

### 5.1.1 Introduction

The new generation of graphic cards are having on-board what is called a Graphical Processing Unit (GPU). These GPUs, have commonly hundreds of processing cores, very high speed parallel architecture, and RAM by the Gigabytes. Derived essentially by the need of processing virtual representation of reality in the gaming industry, they are now also having general physics accelerated algorithms for environmental modeling like fluids dynamics.

General-Purpose computation on GPUs (GPGPU) is a relatively new type of computation possibility derived from the increasingly varied types of computations available on those graphic cards. They are what is called in computer engineering "coprocessors", their specific high-speed high-parallel architecture makes them very attractive for heavy RAM-based computations. More information on GPGPU in general can be found at www.gpgpu.org.

For our example computing water maps from MODIS datasets, we will use a language from NVIDIA GPUs called Compute Unified Driver Architecture (CUDA). This is a C/C++ language addendum that enables the code to send your data to your GPGPU-enabled NVIDIA graphic card, to be processed there, and retrieve your results back from the graphic card RAM memory to your computer hard disk.

For your computer to understand this code, it needs an additional compiler than your already available C compiler. That compiler can be downloaded from www.nvidia.com, look for "CUDA Zone".

Copy the code from this booklet in a file cuda_water.cu, compile it with nvcc:

```
nvcc -o ndvicu cuda_water.cu
```

and run it:

```
./watercu
```

Example Makefile:

```
ndvicu: cuda_water.cu
    nvcc -o watercu cuda_water.cu -I/usr/include/gdal/ -L/usr/lib -lgdal1.7.0
```

### 5.1.2 Code explanation

Most of the time, when in High Performance computing, care is given to limit communication overhead. In this case, if we can load all the data into the GPU memory it is best. Luckily, *GDALRasterIO* provides with that possibility.
Say we load the MODIS datasets, define holders for our Red (red), Near Infrared (nir), Band 7 (band7) and water (water) image bands with GDAL. Then we load the red, nir and band7 bands into into line buffers (one dimensional arrays). Using line buffers here is just a choice, since the processing is not column nor it is row dependent. If needed though, GPUs have clear and easy ways to access and work with 2D and 3D localization.

```
/* N = col x row */
int N = nX * nY;

/* one dimensional arrays, since processing is same for all pixels */
float *red=(float *) malloc(sizeof(float)*N);
float *nir=(float *) malloc(sizeof(float)*N);
float *band7=(float *) malloc(sizeof(float)*N);
float *water=(float *) malloc(sizeof(float)*N);

/* Load input datasets */
GDALRasterIO(hBandRed,GF_Read,0,0,nX,nY,red,nX,nY,GDT_Float32,0,0);
GDALRasterIO(hBandNir,GF_Read,0,0,nX,nY,nir,nX,nY,GDT_Float32,0,0);

/* Band 7 is 500m pixel size <!=!> Red and NIR are 250m pixel size */
GDALRasterIO(hBandBand7,GF_Read,0,0,nX/2,nY/2,band7,nX,nY,GDT_Float32,0,0);
```

In GPGPU programming, the main point is to be able to manage the memory allocation in the GPU itself. After that, it is just a question of copying the data into the GPU memory before processing and copying it back to the computer after processing.

We start by allocating variables with the suffix "_d" (d for device) to remind us of their location of use, in the GPU device. As everything is a grid (2D or 3D matrix) in a GPU, we allocate an integer N as our image total dimension, this is actually the total length of the grid allocated inside the GPU (it could be written like this: N=row_d x col_d).

```
/* pointers to GPU device memory */
float *red_d, *nir_d, *band7_d, *water_d;
```

Following that, we allocate arrays on the host (our computer itself). So we can fetch the data from our images.

```
/* Allocate arrays on host*/
red = (float*) malloc(sizeof(float) * N);
nir = (float*) malloc(sizeof(float) * N);
band7 = (float*) malloc(sizeof(float) * N);
water = (float*) malloc(sizeof(float) * N);
```

Once the data is in our computer memory, it is time to send the row data into the GPU. To do that, we have to use a specific function called

```
cudaMemcpy(GPU_memory,PC_memory,size_of_data,cudaMemcpyHostToDevice)
```

The last argument indicates the direction of the copy of the data. In this case we send the data from the computer to the GPU, so the direction is "cudaMemcpy-HostToDevice", to retrieve the data after computation, we will use the opposite direction "cudaMemcpyDeviceToHost".

```
/* Copy data from host memory to GPU device memory */
cudaMemcpy(red_d, red, sizeof(float)*N, cudaMemcpyHostToDevice);
cudaMemcpy(nir_d, nir, sizeof(float)*N, cudaMemcpyHostToDevice);
cudaMemcpy(band7_d, band7, sizeof(float)*N, cudaMemcpyHostToDevice);
```

When the data has reached the GPU memory, it is time to apply calculation on it, this is done in GPGPU computing by applying a kernel. The incantation for doing so is shown below:

```
/* Add arrays red, nir, band7 and store result in water */
waterid<<<dimGrid,dimBlock>>>(red_d, nir_d, band7_d, water_d, N);
```

In this case, the kernel to be used is called "waterid", it is applied on a grid. That grid is sub-divided in blocks, which are the units of work allocation within the GPU. The grid in our example is N, our row data. It is visualized as a grid by the GPU and is split into computing blocks limited by the architecture of the GPGPU.

At the time of the writing of this booklet, lower-end graphic cards are capable of 256 or 512 size blocks. The bigger the block, of course, the less number of job distributions to complete the processing of our row data. If the row data has a size of N=nXSize1=1024, it will take 4 blocks of 256 or 2 blocks of 512 to compute it. One last remark, as you can witness, Block and Grid allocation are defined into dim3, GPUs, being graphical devices operate natively on 3D vectors.

```
/* Compute the Blocks of data to be sent to GPU */
// On GeForce 8600 Galaxy x=256
// On GeForce 9500 Galaxy/GT & 9800 GT x=512
int x=512;
dim3 dimBlock(x);
dim3 dimGrid ( (N / dimBlock.x) + (!(N % dimBlock.x)?0:1));
```

The kernel was initially a standard C function, eventually, it was modified to appropriately take benefit of the GPGPU architecture. You can refer to Appendix A for the original C versions of the NDVI and Water functions.

```
__global__ void waterid(float *red, float *nir, float *band7, float *water, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x ;
    if ( i < N ){
        water[i]=(nir[i]-red[i])/(red[i]+nir[i]);
        if(water[i] < 0.1 && band7[i] < 0.04)
            water[i] = 1;
        else
            water[i] = 0;
    }
}
```

An additional step required for streamlining the processing, is to synchronize the data movement with the job completion in the GPU, this is done by applying the function “cudaThreadSynchronize()”, simply.

```
/* Block until device completed processing */
cudaThreadSynchronize();
```

From this point onwards, data is copied back to the computer from the device using the “cudaMemcpy” function. Additionally, the GPU memory is also freed.

```
/* Copy data from GPU device to CPU host memory */
cudaMemcpy(water, water_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

/* Free the GPU memory */
cudaFree(red_d);
cudaFree(nir_d);
cudaFree(water_d);
```

After this point, this is a standard C program, as seen earlier, where GDAL takes over the row data and writes it to the disk. Eventually closing the output file and freeing the host memory.

```
    /* Copy the output data in to the output raster */
    GDALRasterIO(hBandNdvi,GF_Write,0,0,nX,nY,nir,nX,nY,GDT_Float32,0,0);

    /* Free the memory from the host */
    if( red != NULL ) free( red );
    if( nir != NULL ) free( nir );
    if( band7 != NULL ) free( nir );
    if( water != NULL ) free( water );

    /* Close input and output raster files */
    GDALClose(hDatasetRed);
    GDALClose(hDatasetNir);
    GDALClose(hDatasetBand7);
    GDALClose(hDatasetWater);
```

As a last note, the beauty of GPU-based processing is that there is no need of loops, and this only because of the hardware architecture, which is inherently parallel.

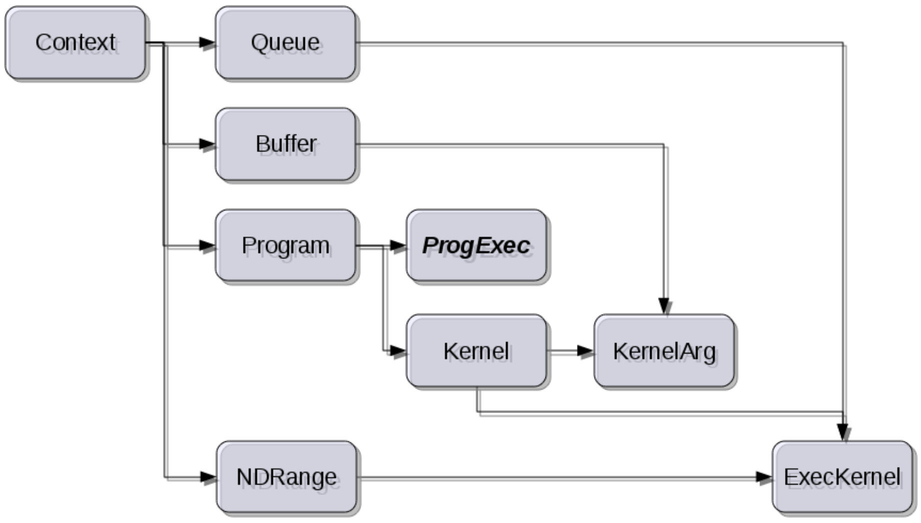
### 5.1.3 CUDA code

```
#include <stdio.h>
#include <gdal.h>
#include <stdlib.h>

__global__ void waterid(float *red, float *nir, float *band7, float *water, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x ;
    if ( i < N ){
        water[i]=(nir[i]-red[i])/(red[i]+nir[i]);
        if(water[i] < 0.1 && band7[i] < 0.04)
            water[i] = 1;
        else
            water[i] = 0;
    }
}

int main()
{
    /**GDAL STUFF***************/
    //Loading the input files
    GDALDatasetH hDatasetRed; //Red band
    GDALDatasetH hDatasetNir; //NIR band
    GDALDatasetH hDatasetBand7; //band7

    GDALAllRegister();

    hDatasetRed = GDALOpen("/home/user/MODIS_b1.tif",GA_ReadOnly);
    hDatasetNir = GDALOpen("/home/user/MODIS_b2.tif",GA_ReadOnly);
    hDatasetBand7 = GDALOpen("/home/user/MODIS_b7.tif",GA_ReadOnly);

    if(hDatasetRed==NULL||hDatasetNir==NULL||hDatasetBand7==NULL){
        printf("At least one input file could not be loaded\n");
        exit(1);
    }

    //Loading the file infos
    GDALDriverH hDriverRed;
    hDriverRed = GDALGetDatasetDriver(hDatasetRed);

    //Creating output file
    GDALDatasetH hDatasetWater;
    hDatasetWater=GDALCreateCopy(hDriverRed,"water.tif",hDatasetRed,FALSE,NULL,NULL,NULL);
    GDALRasterBandH hBandWater;
    hBandWater = GDALGetRasterBand(hDatasetWater,1);

    //Loading the file bands
    GDALRasterBandH hBandRed,hBandNir,hBandBand7;

    hBandRed = GDALGetRasterBand(hDatasetRed,1);
    hBandNir = GDALGetRasterBand(hDatasetNir,1);
    hBandBand7 = GDALGetRasterBand(hDatasetBand7,1);

    int nX = GDALGetRasterBandXSize(hBandRed);
    int nY = GDALGetRasterBandYSize(hBandRed);

    /**CUDA STUFF***************/
    /* pointers to host memory */
    float *red, *nir, *band7, *water;
    /* pointers to GPU device memory */
    float *red_d, *nir_d, *band7_d, *water_d;
    int N=nX*nY; N=row x col processing in Device Memory
    /* Allocate arrays on host*/
    red = (float*) malloc(sizeof(float) * N);
    nir = (float*) malloc(sizeof(float) * N);
    band7 = (float*) malloc(sizeof(float) * N);
    water = (float*) malloc(sizeof(float) * N);

    for(int i=0; i<N;i++)
        water[i]=0.0;

    /* Allocate arrays a_d, b_d and c_d on device*/
    cudaMalloc ((void **) &red_d, sizeof(float) * N);
    cudaMalloc ((void **) &nir_d, sizeof(float) * N);
    cudaMalloc ((void **) &band7_d, sizeof(float) * N);
    cudaMalloc ((void **) &water_d, sizeof(float) * N);

    /* Compute the Blocks of data to be sent to GPU */
    // On GeForce 8600 Galaxy x=256
    // On GeForce 9500 Galaxy/GT & 9800 GT x=512
    int x=512;
    dim3 dimBlock(x);
    dim3 dimGrid ( (N / dimBlock.x) + (!(N % dimBlock.x)?0:1)) ;

    /* Read input files through GDAL */
    GDALRasterIO(hBandRed,GF_Read,0,0,nX,nY,red,nX,nY,GDT_Float32,0,0);
    GDALRasterIO(hBandNir,GF_Read,0,0,nX,nY,nir,nX,nY,GDT_Float32,0,0);
    GDALRasterIO(hBandBand7,GF_Read,0,0,nX/2,nY/2,band7,nX,nY,GDT_Float32,0,0);
    /* Copy data from host memory to GPU device memory */
    cudaMemcpy(red_d, red, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(nir_d, nir, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(band7_d, band7, sizeof(float)*N, cudaMemcpyHostToDevice);
    /* Add arrays red, nir, band7 and store result in water */
    waterid<<<dimGrid,dimBlock>>>(red_d, nir_d, band7_d, water_d, N);
    /* Block until device completed processing */
    cudaThreadSynchronize();
    /* Copy data from device memory to host memory */
    cudaMemcpy(water, water_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    /* Write output file through GDAL */
    GDALRasterIO(hBandWater,GF_Write,0,row,nX,nY,water,nX,nY,GDT_Float32,0,0);
    /* Free the memory */
    free(red);
    free(nir);
    free(band7);
    free(water);
    cudaFree(red_d);
    cudaFree(nir_d);
    cudaFree(band7_d);
    cudaFree(water_d);
    GDALClose(hDatasetRed);
    GDALClose(hDatasetNir);
    GDALClose(hDatasetBand7);
    GDALClose(hDatasetWater);
}
```

## 5.2 OpenCL with Python & GDAL

### 5.2.1 Introduction

The Open Computing Language or OpenCL (www.khronos.org/opencl) is an open specification for parallel processing. A single code will run on CPU, GPU, embedded devices, etc. On the GPGPU side, it is compatible on both ATI and NVIDIA. OpenCL for NVIDIA GPU hardware is built on top of CUDA API, so if you are having a NVIDIA GPU and intend to keep on with it, there is less incentive to move from the easier to start CUDA language. ATI built their OpenCL for the GPU on top of their Compute Abstraction Layer (CAL), on which they previously created their Brook+ GPU parallel computing language (developer.amd.com).
The programming requirements for OpenCL are higher than CUDA, but more rewarding in terms of GPGPU support (ATI maybe cheaper than NVIDIA, or vice-versa at time). However, since OpenCL can work on many devices and is relatively low-level language, it is let to the programmer to manage memory and also to gather limitations of the GPU it is running the software on. Of course, that will require some programming to prevent overloading the device, but also to tell if certain tasks cannot be done on a given hardware. In terms of remote sensing processing, a first barrier is the GPU RAM size capacity to hold our satellite images in all.
Below is a flow-chart of the basic structure of an OpenCL program code.



From a C and Python programming background, it seemed more logical to start this rather different computing paradigm from a Python perspective, limiting the

information to the flow of the process rather than the intricacies of full detailed C code.
At the platform layer, an initial instance will query the Operating System for platform information, and available compute devices (CPUs, GPUs, FPGAs, etc.). This information will be part of the creation of the *Context*.
On a larger vision, the *Context* is part of most of steps required to have the OpenCL eco-system ready to compute your data (including Command Queue and Device Buffer creation).
The programs that will run in OpenCL eco-system are very often seen as a main instance (*main program*) and a set of *kernels*. They are often written in separated text files too. As previsously seen in CUDA section, there is a special API for memory buffers to be transfered to and from the device. In OpenCL, The *Kernel Arguments* definition is the process of linking host memory handles to the device memory handles through the *Input/Output Arguments* of the defined *Kernel*.
While OpenCL has a specific image handling memory objects, it is simpler for a start to use generic memory buffers with the GDAL API as we have done largely already in this book.
Finally, *NDRange* is an index space (1-D, 2-D or 3-D) that is requested with dimensionality of data in mind, so that work-groups and/or work-items are organised/sized to perform tasks in an optimised manner so as to reduce the number of clock cycles for a kernel to compute over all of the dataset in device memory.
In order to have a working version of python-opencl in Linux (here Debian), please type the following command as supe-user: *apt-get install python-pyopencl*. Other O/Ses please refer to the mathema.tician.de/software/pyopencl website.

### 5.2.2 Code explanation

In order to have a good understanding of this code, please first refer to section 2.2. From there two images are created *b3* and *b4* that are input to this code.
b3 and b4 Python Objects are loaded with data from there respective images with the *LoadFile* GDAL command. The objects are then resized to 1-D arrays in b3r and b4r.
From that onwards, the Python wrapper for the OpenCL library is loaded as *cl*.

```
import pyopencl as cl
```

The context and the Queue are created straight after that. Those are defined automatically in PyOpenCL.

```
#Create Context
ctx = cl.create_some_context()
#Create Commands Queue
queue = cl.CommandQueue(ctx)
```

At this point, the data should be transferred from the Python arrays in the RAM of the host (The Computer) to the RAM of the OpenCL selected device (i.e. GPU, FPGA, etc.). This is done using the *pyopencl.Buffer( context, memoryFlags, hostBuffer )* function. As it states, it requires defined *memoryFlags*, which we first generate:

```
#Create Memory Flags Objects
mf = cl.mem_flags

#Fill Device Memory with Data from Host
b3r_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b3r)
b4r_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b4r)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b4r.nbytes)
```

In Single Instruction Multiple Data (SIMD) architectures, a *Kernel* is applied to a data grid. This Kernel is in OpenCL called a *program* using different input data grids to generate an output data grid. It is time to develop an NDVI OpenCL *program* for this code.

```
#Create NDVI Kernel Program
prg = cl.Program(ctx, """
   __kernel void ndvi(__global const float *b3r,
   __global const float *b4r, __global float *dest)
   {
     int gid = get_global_id(0);
     dest[gid] = (b4r[gid] - b3r[gid])/(b3r[gid] + b4r[gid]+0.01);
   }
   """).build()
```

The program content is bounded by triple quotes, the *ndvi* algorithm is defined in a text format and given the program creation function *pyopencl.program(context, algorithmText)*. The suffix *.build()* function is triggering the generation of the binary file inside the context *ctx* for a program eventually called *prg*. The NDVI function defined into the program *prg* is then used to compute the new NDVI image using the OpenCL selected device.

```
#Execute Kernel Program
prg.ndvi(queue, b3r.shape, None, b3r_buf, b4r_buf, dest_buf)
```

The computation is being queued to the *queue*, the shape of b3r is a single value of number of pixels present and the last 3 arguments are the datasets input1, input2, and output. When this is computed, an output array has to be created inside the host (The Computer), here with the name of *ndvi_cl* to receive the output dataset generated in the device with the name of *destination buffer (dest_buf)*. We use Python numpy to get an empty copy of b3r original dataset.

```
#Create Empty array from first (reshaped to 1-D) input
ndvi_cl = numpy.empty_like(b3r)

#Return Device Array Content to Host
cl.enqueue_read_buffer(queue, dest_buf, ndvi_cl).wait()
```

Eventually, using *pyopencl.enqueue_read_buffer(queue, deviceBuffer, hostBuffer)* the device output dataset (*dest_buf*) is copied to the host output dataset (*ndvi_cl*). The suffix function *.wait()* is a synchronization barrier to make sure all dataset has been returned from the device before giving the *ndvi_cl* dataset for your future use. After that, a standard set of GDAL operations permit to save the *ndvi_cl* array to a raster formatted file. To note is the required reshape of the array back to the originally shaped 2-D b3 dataset, before writing to disk as 2-D raster.

```
#prepare/create output file with projection
out = OpenArray( ndvi_cl.reshape(b3.shape[0],b3.shape[1]) )
out.SetGeoTransform( geoT )
out.SetProjection( proJ )
driver.CreateCopy( 'ndvi_cl', out )
```

## 5.2.3 PyOpenCL code

```
# -*- coding: utf-8 -*-
#Purpose: Calculate NDVI
#Usage: /usr/bin/python -u "ndvi.py"

#if problem of libgrass: MAPSET is not set
#then: set GDAL_SKIP = GRASS

F=[]
F.append('b3_ref')
F.append('b4_ref')

# For Image Processing
from math import *
import numpy
from osgeo import gdalnumeric
from osgeo import gdal
from osgeo.gdal_array import *
from osgeo.gdalconst import *

# Set our output file format driver
driver = gdal.GetDriverByName( 'ENVI' )

# Set our output files Projection parameters
# from input file number 1
tmp = gdal.Open(F[0])
geoT = tmp.GetGeoTransform()
proJ = tmp.GetProjection()
del tmp

#Load ToAR bands (from Python-GDAL section)
b3 = LoadFile(F[0]).astype(numpy.float32)
b4 = LoadFile(F[1]).astype(numpy.float32)

#Reshape the data from 2-D to 1-D
b3r = b3.reshape(b3.shape[0]*b3.shape[1])
b4r = b4.reshape(b4.shape[0]*b4.shape[1])

#----------------------------------------------------------
#BEGIN OPEN_CL
import pyopencl as cl

#Create Context
ctx = cl.create_some_context()
#Create Commands Queue
queue = cl.CommandQueue(ctx)
#Create Memory Flag Objects
mf = cl.mem_flags
#Fill Device Memory with Data from Host
b3r_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b3r)
b4r_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b4r)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b4r.nbytes)

#Create NDVI Kernel Program
prg = cl.Program(ctx, """
    __kernel void ndvi(__global const float *b3r,
    __global const float *b4r, __global float *dest)
    {
      int gid = get_global_id(0);
      dest[gid] = (b4r[gid] - b3r[gid])/(b3r[gid] + b4r[gid]+0.01);
    }
    """).build()
```

```
#Execute Kernel Program
prg.ndvi(queue, b3r.shape, None, b3r_buf, b4r_buf, dest_buf)
#Create Empty array from first (reshaped to 1-D) input
ndvi_cl = numpy.empty_like(b3r)
#Return Device Array Content to Host
cl.enqueue_read_buffer(queue, dest_buf, ndvi_cl).wait()
#----------------------------------------------------------
#END OPEN_CL

#prepare/create output file with projection
out = OpenArray( ndvi_cl.reshape(b3.shape[0],b3.shape[1]) )
out.SetGeoTransform( geoT )
out.SetProjection( proJ )
driver.CreateCopy( 'ndvi_cl', out )
```

## 5.3 OpenCL with C & GDAL

### 5.3.1 Introduction

After reviewing basic structure and processing of OpenCL in Python, we can now have a more detailed coding with the OpenCL C language. We will process an NDVI product from two input raster files, a *red.tif* and a *nir.tif*. We will consider that using GDAL API the programmer has loaded two memory buffers *red* and *nir* respectively with those images as covered the use of GDAL API for C in section 2.1.

The contents of a short Makefile are expanded below, using the OpenCL library from the NVIDIA GPU computing SDK (www.nvidia.com).

```
#A short NVIDIA-bound Makefile
ndvi: main.c arrays.c
        gcc -o ndvi main.c arrays.c
                -I. #This is for arrays.h
                -I~/NVIDIA_GPU_Computing_SDK/OpenCL/common/inc/CL/
                -L~/NVIDIA_GPU_Computing_SDK/OpenCL/common/lib/
                -lOpenCL -L/usr/lib64
                -I/usr/include/gdal -lgdal
                -Wall
```

The same Makefile but using the AMD Accelerated Parallel Processing SDK (developer.amd.com).

```
#A short AMD-bound Makefile
ndvi: main.c arrays.c
        gcc -o ndvi main.c arrays.c
                -I. #This is for arrays.h
                -I~/AMD-APP-SDK-v2.4-lnx64/include/CL
                -L~/AMD-APP-SDK-v2.4-lnx64/lib/x86_64
                -lOpenCL -L/usr/lib64
                -I/usr/include/gdal -lgdal
                -Wall
```

Few upfront observations need to be mentioned here. First, the compilation is a standard C language compilation using gcc (for example) and GDAL as in section 2.1. Second, the only presence of OpenCL is in the form of *-I* for including headers and *-L* for locating libraries, thus declaring *-lOpenCL* eventually. One more observation is the include of the actual code directory, since *arrays.h* is also found there, pointing to the compiled file *arrays.c* being the file holding memory allocation of array types in C (see these files in section 7.3). Finally, and this is just a useful addition, the flag *-Wall* sets all warnings to be reported to the command line interface on compilation time.

There are three identified sections in this code, each one pertaining to either GDAL or OpenCL APIs. The initial section is using GDAL API to load raster data in memory, then the OpenCL API section is developed, finally a short GDAL API section is used to write the memory to the output raster data file.

### 5.3.2 Code explanation

The necessary header files for exploiting the OpenCL library are:

```
#include <cl.h>
#include <cl_ext.h>
```

At this point, after declaring all necessary headers, could be a good place to locate the kernel if small enough. We rellocated the kernel in an external file called *kernels.cl* (see section 7.4) in this case for long term reasons of increasing kernels number. The kernel used in this example is called *nd_vi*. It has 2 input (*const float * red, const float * nir*) and 1 output (*float * ndvi*), all are set as *__global* memory.

```
//Define Kernel in external file ''kernels.cl''
__kernel void nd_vi(__global const float * red,__global const float * nir,__global float * ndvi)
{
        int i = get_global_id(0);
        if((nir[i]+red[i])<0.01) ndvi[i]=255;
        else ndvi[i] = (1+((nir[i]-red[i])/(nir[i]+red[i]));
};
```

It is read into a text buffer to be sent eventually to the OpenCL library for compilation into a binary kernel.

```
//----------------------------------------------------------
// OPENCL SECTION
// Load the kernel source code into the array source_str
FILE *fp                = fopen(inkernel,"r");
if (!fp){
        printf("Failed to load kernel file\n");
        exit(EXIT_FAILURE);
}
char *readSource        = (char*)malloc(MAX_SOURCE_SIZE);
fread(readSource,1,MAX_SOURCE_SIZE,fp);
fclose(fp);
const char *sProgSource = (const char *) readSource;
```

As seen in section 5.1, the workload partitioning is an important step in the definition of the work-groups in this type of distributed programming. OpenCL also defines them as *cnBlocks* and *cnBlockSize* and their combined dimension (*cnDim*).

```
///OPENCL
//Allocate constants memory
const unsigned int cnBlockSize = 16;
const unsigned int cnBlocks = (int) ceil(N / cnBlockSize);
const size_t cnDim = cnBlocks * cnBlockSize;
```

As of OpenCL v1.1, the definition of the PlatformID is compulsory, so it is acquired in this way.

```
//Create OpenCL Platform ID and Platform Number
cl_platform_id p_id     = NULL;
cl_uint p_num;
cl_int err              = clGetPlatformIDs(1,&p_id,&p_num);
```

```
//Get Device ID and Info
cl_device_id d_id       = NULL;
cl_uint d_num;
err                     = clGetDeviceIDs(p_id,CL_DEVICE_TYPE_GPU,1,&d_id,&d_num);
cl_context hContext     = clCreateContext(NULL,1,&d_id,NULL,NULL,&err);
```

The definition of the context of processing involves the targeting of the device (here a GPU as *CL_DEVICE_TYPE_GPU*) on the identified platform (*p_id*). In the code is a fallback to the CPU device (*CL_DEVICE_TYPE_CPU*) on that platform if no GPU is found, and a final fallback to a graceful failure with exit.

```
if(CL_DEVICE_NOT_FOUND==err){
        printf("trying CPU instead of GPU\n");
        err             = clGetDeviceIDs(p_id,CL_DEVICE_TYPE_CPU,1,&d_id,&d_num);
        hContext        = clCreateContext(NULL,1,&d_id,NULL,NULL,&err);
}
if (err){
        printf("Could not find Device to create context on\n");
        exit(EXIT_FAILURE);
}
```

In the event that more than one GPU device is available in the newly created context on the specific platform, this code will want to use the first GPU device available in the list returned from the query. This process is done by asking the context for its available information regarding enabled devices.

```
//Query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES, 0, 0, &nContextDescriptorSize);
cl_device_id * aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES, nContextDescriptorSize, aDevices, 0);

//Create a command queue for first device the context reported
cl_command_queue hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);
```

The command queue is then created using *aDevices[0]* being the first GPU device in the context query list. The command queue being enabled, the program holding kernels definitions (in this case there is only one kernel defined in it, the NDVI kernel) can be compiled into binary with the functions *clProgramWithSource() & clBuildProgram()*.

```
//Create and compile program
cl_program hProg = clCreateProgramWithSource(hContext, 1, &sProgSource, 0, 0);
clBuildProgram(hProg, 0, 0, 0, 0, 0);
```

Once the binary program is available, *clCreateKernel()* allocated a variable name to the handle of the binary part pertaining to the kernel definition name provided.

```
//Create Kernel
cl_kernel hKernel = clCreateKernel(hProg, "nd_vi", 0);
```

All is now ready to receive variables and run the kernel on the dataset. To do that, allocate memory on the device and copy the data to the GPU in a similar fashion as in section 5.1.

```
//Allocate device memory and copy datasets
cl_mem hDevMem[3];
hDevMem[0]= clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,\
        cnDim * sizeof(cl_int), red, &err);
hDevMem[1] = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,\
        cnDim * sizeof(cl_int), nir, &err);
hDevMem[2] = clCreateBuffer(hContext, CL_MEM_WRITE_ONLY,\
        cnDim * sizeof(cl_int), 0, &err);
```

The last device buffer (*hDevMem[2]*) being the output buffer for NDVI values, is initialized with zero values. Link the device memory buffer to the argument list of the kernel using *clSetKernelArg()*.

```
//Setup kernel parameter arguments values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDevMem[0]);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDevMem[1]);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDevMem[2]);
```

Execute the Kernel and retrieve the output buffer from the GPU device RAM into the computer RAM output buffer.

```
//Execute Kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, &cnDim, 0, 0, 0, 0);

// Wait for finish commands
clFinish(hCmdQueue);

//Copy results from device back to host
clEnqueueReadBuffer(hCmdQueue,hDevMem[2],CL_TRUE,0,cnDim*sizeof(cl_int),lOut,0,0,0);
```

Finally, free all OpenCL memory objects, and return the memory buffer to the GDAL API for writing the output raster file as in section 2.1.

### 5.3.3 OpenCL code

```
#include <stdio.h>
#include <CL/opencl.h>
#include <gdal.h>
#include <arrays.h>

#define MAX_SOURCE_SIZE (0x100000)


int main(int argc, char* argv[])
{
    //INPUT FROM CLI ARGUMENTS
    char *out   = argv[1]; // output NDVI raster file
    char *inred = argv[2]; // input red band raster file
    char *innir = argv[3]; // input NIR band raster file
    char *inkernel  = argv[4]; // name of file containing kernels

    ///GDAL STUFF
    //-------------------------------------------------------
    GDALAllRegister();
    GDALDatasetH hD1    = GDALOpen(inred,GA_ReadOnly);
    GDALDatasetH hD2    = GDALOpen(innir,GA_ReadOnly);
    if(hD1==NULL||hD2==NULL)exit(EXIT_FAILURE);
    GDALDriverH hDr     = GDALGetDatasetDriver(hD1);
    GDALDatasetH hDOut  = GDALCreateCopy(hDr,out,hD1,FALSE,NULL,NULL,NULL);
    GDALRasterBandH hBOut   = GDALGetRasterBand(hDOut,1);
    GDALRasterBandH hB1     = GDALGetRasterBand(hD1,1);
    GDALRasterBandH hB2     = GDALGetRasterBand(hD2,1);

    //GDAL Extracts info and prepare data memory
    int nX          = GDALGetRasterBandXSize(hB1);
    int nY          = GDALGetRasterBandYSize(hB1);
    int N           = nX * nY;
    int *red        = ai1d(N);
    int *nir        = ai1d(N);
    int *lOut       = ai1d(N);

    //GDAL fills host memory with raster data
    GDALRasterIO(hB1,GF_Read,0,0,nX,nY,red,nX,nY,GDT_Int32,0,0);
    GDALRasterIO(hB2,GF_Read,0,0,nX,nY,nir,nX,nY,GDT_Int32,0,0);

    //-------------------------------------------------------
    // OPENCL SECTION

    // Load the kernel source code into the array source_str
    FILE *fp        = fopen(inkernel,"r");
    if (!fp){
        printf("Failed to load kernel file\n");
        exit(EXIT_FAILURE);
    }
    char *readSource    = (char*)malloc(MAX_SOURCE_SIZE);
    fread(readSource,1,MAX_SOURCE_SIZE,fp);
    fclose(fp);
    const char *sProgSource = (const char *) readSource;

    ///OpenCL Block structure setup
    const unsigned int cnBlockSize = 16;
    const unsigned int cnBlocks = (int) ceil(N / cnBlockSize);
    const size_t cnDimension = cnBlocks * cnBlockSize;

    //Create OpenCL Platform ID and Platform Number
    cl_platform_id p_id = NULL;
```

```
cl_uint p_num;
cl_int err      = clGetPlatformIDs(1,&p_id,&p_num);

//Get Device ID and Info
cl_device_id d_id   = NULL;
cl_uint d_num;
err         = clGetDeviceIDs(p_id,CL_DEVICE_TYPE_GPU,1,&d_id,&d_num);
cl_context hContext = clCreateContext(NULL,1,&d_id,NULL,NULL,&err);

//Create OpenCL device and context
if(CL_DEVICE_NOT_FOUND==err){
    printf("trying CPU instead of GPU\n");
    err     = clGetDeviceIDs(p_id,CL_DEVICE_TYPE_CPU,1,&d_id,&d_num);
    hContext    = clCreateContext(NULL,1,&d_id,NULL,NULL,&err);
}
if (err){
    printf("Could not find Device to create context on\n");
    exit(EXIT_FAILURE);
}

//Query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES, 0, 0, &nContextDescriptorSize);
cl_device_id * aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES, nContextDescriptorSize, aDevices, 0);

//Create a command queue for first device the context reported
cl_command_queue hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);

//Create and compile program
cl_program hProg = clCreateProgramWithSource(hContext, 1, &sProgSource, 0, 0);
clBuildProgram(hProg, 0, 0, 0, 0, 0);

//Create Kernel
cl_kernel hKernel = clCreateKernel(hProg, "nd_vi", 0);

//Allocate device memory
cl_mem hDevMem[4];
hDevMem[0]= clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, \
    cnDimension * sizeof(cl_int),  (void*)&red, &err);
printf("err hDevMem[0]=%i\n",err);
hDevMem[1] = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, \
    cnDimension * sizeof(cl_int),  (void*)&nir, &err);
printf("err hDevMem[1]=%i\n",err);
hDevMem[2] = clCreateBuffer(hContext, CL_MEM_WRITE_ONLY, \
    cnDimension * sizeof(cl_int),  (void*) NULL, &err);
printf("err hDevMem[2]=%i\n",err);

//Setup kernel parameter arguments values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDevMem[0]);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDevMem[1]);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDevMem[2]);

//Execute Kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, &cnDimension, 0, 0, 0, 0);

// Wait for finish commands
clFinish(hCmdQueue);

//Copy results from device back to host
clEnqueueReadBuffer(hCmdQueue, hDevMem[2], CL_TRUE, 0, \
    cnDimension * sizeof(cl_int), lOut , 0, 0, 0);
    //--------------------------------------------------------
    //GDAL Copy ndvi to GDAL array and Write ndvi_gdal_array to file
    GDALRasterIO(hBOut,GF_Read,0,0,nX,nY,lOut,nX,nY,GDT_Int32,0,0);

    //GDAL Free Memory
    free(red);
    free(nir);
    free(lOut);
    GDALClose(hD1);
    GDALClose(hD2);
    GDALClose(hDOut);

    //--------------------------------------------------------
    // OPENCL FREE MEMORY
    clReleaseMemObject(hDevMem[0]);
    clReleaseMemObject(hDevMem[1]);
    clReleaseMemObject(hDevMem[2]);
    free(aDevices);
    clReleaseKernel(hKernel);
    clReleaseProgram(hProg);
    clReleaseCommandQueue(hCmdQueue);
    clReleaseContext(hContext);

    return(EXIT_SUCCESS);
}
```

# Chapter 6

# Algorithms with GUI

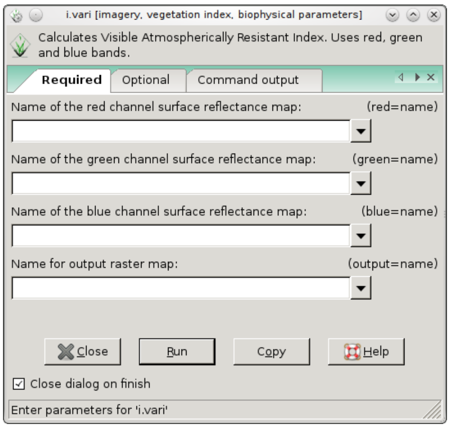
## 6.1 GRASS GIS

### 6.1.1 Introduction

GRASS GIS (grass.osgeo.org) is an Open source Geographical Information System software part of the Open Source Geospatial Foundation (www.osgeo.org). It is used for geospatial data management and analysis, image processing, graphics/maps production, spatial modeling, and visualization.

It provides with a programming support to image processing, as well as a combination of Command Line Interface and Graphical User Interface. Once an image processing algorithm is programmed within GRASS GIS, it can be used by any user like any other module, from the CLI or the GUI.

Appendix B provides additional information about how to build a module directory in the GRASS GIS main source tree. It also describes the companion files to the actual processing code we are concentrating in this section. Below is a screenshot of the graphical user interface and its help/manual tab. This is automatically generated by GRASS GIS upon compilation of the new module. We propose to walk you through the creation of this module in this section.



i.vari [imagery, vegetation index, biophysical parameters]

Calculates Visible Atmospherically Resistant Index. Uses red, green and blue bands.

Name of the red channel surface reflectance map: (red=name)

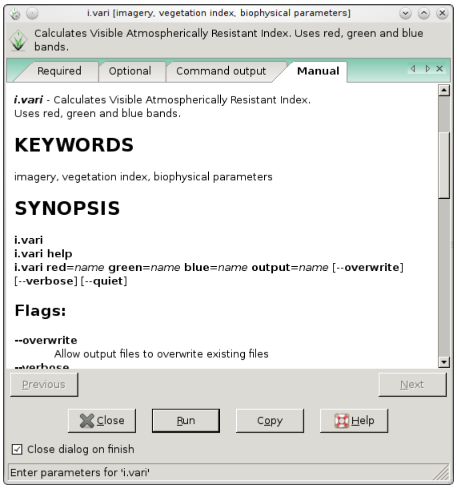
Name of the green channel surface reflectance map: (green=name)

Name of the blue channel surface reflectance map: (blue=name)

Name for output raster map: (output=name)

☑ Close dialog on finish

Enter parameters for 'i.vari'



i.vari [imagery, vegetation index, biophysical parameters]

Calculates Visible Atmospherically Resistant Index. Uses red, green and blue bands.

***i.vari*** - Calculates Visible Atmospherically Resistant Index.
Uses red, green and blue bands.

**KEYWORDS**

imagery, vegetation index, biophysical parameters

**SYNOPSIS**

**i.vari**
**i.vari help**
**i.vari red=***name* **green=***name* **blue=***name* **output=***name* [**--overwrite**] [**--verbose**] [**--quiet**]

**Flags:**

**--overwrite**
Allow output files to overwrite existing files

☑ Close dialog on finish

Enter parameters for 'i.vari'

It has to be noted that no special code is dedicated to the generation of the GUI, instead, GRASS is parsing the code for input, output, options, flags and other specific information and generates what is needed to create the GUI within its WxPython GUI architecture.

## 6.1.2 Code explanation

GRASS GIS code is generally written in C, therefore it is exhaustive. Fortunately, GRASS functions have been designed to lighten the programming load. To have access to such functions, it is necessary to load their descriptions in header files, having the typical C language suffix ".h". As you can witness, three header files loaded are GRASS header files, the fourth one is ours, as we also have specific remote sensing functions in a file called "grass_rs.c". Both "grass_rs.c" and "grass_rs.h" are described in Appendix A (section 7.1:83).

```
#include <grass/gis.h>
#include <grass/raster.h>
#include <grass/glocale.h>
#include "grass_rs.h"
```

Following this setup, a long list of variable initialization comes up, let us start with the Module definition file. As we can see it is a structure that is initialized later in the code by *G_define_module()*. Eventually its internal arguments (keywords, label, description) are allocated informative text. The text in _() is meant to be translated by a GRASS program.

```
struct  GModule *module;        /*GRASS GIS Module setup*/
G_add_keyword(_("imagery"));
G_add_keyword(_("vegetation index"));
G_add_keyword(_("biophysical parameters"));
```

```
module->keywords = _("imagery, vegetation index, biophysical parameters");
module->label =_("Calculates Visible Atmospherically Resistant Index.");
module->description = _("Uses red, green and blue bands.");
```

Another set of variables allocations is the "Option", this will define input and output files, they maybe any kind of files, though in this case the GRASS function *G_define_standard_option()* is set with G_OPT_R_INPUT or G_OPT_R_OUTPUT. The "_R_", specifies that our set of instructions is regarding input/output files of type raster. After that initialization, we can modify the structures elements (i.e. key, label, etc...) as we see fit to our purpose. In the case of the output structure, we just initialize it with default raster type output arguments: G_OPT_R_OUTPUT.

```
struct  Option *input1, *input2, *input3, *output;      /*CLI/GUI input options */
input1 = G_define_standard_option(G_OPT_R_INPUT);
input1->key = "red";
input1->label =_("Name of the red channel surface reflectance map");
input1->description = _("Range: [0.0;1.0]");

input2 = G_define_standard_option(G_OPT_R_INPUT);
input2->key = "green";
input2->label =_("Name of the green channel surface reflectance map");
input2->description = _("Range: [0.0;1.0]");

input3 = G_define_standard_option(G_OPT_R_INPUT);
input3->key = "blue";
input3->label =_("Name of the blue channel surface reflectance map");
input3->description = _("Range: [0.0;1.0]");

output = G_define_standard_option(G_OPT_R_OUTPUT);
```

Two more additional structures are set up, one holding metadata support, the other one holding a color table. Here we also attach their complete use (as can be seen at the end of the example code) to provide clarity about their use. The history incantation is self consistent and rarely needs change. The color structure is first initialized by *Rast_init_colors()* and then a greyscale is applied from -1.0 to +1.0 image range values. This is done using *Rast_add_c_color_rule()*, *c* for *CELL*. It associates a pixel value to a RGB triplet, in this case pixel value -1.0 is associated with RGB triplet (0,0,0) which is black color. Likewise, pixel value +1.0 is associated with RGB triplet (255,255,255) which is white color.

```
/* Beginning of code: Create memory slot for metadata */
struct  History history;

/* End of code: Copy metadata */
Rast_short_history(result, "raster", &history);
Rast_command_history(&history);
Rast_write_history(result, &history);

/* Beginning of code: Create memory slot for color rules */
struct  Colors colors;
CELL    val1, val2; /* For colors range*/

/*End of code: Color table from -1.0 to +1.0 in grey */
Rast_init_colors(&colors);
```

```
val1 = -1.0;
val2 = 1.0;
Rast_add_c_color_rule(&val1, 0, 0, 0, &val2, 255, 255, 255, &colors);
```

Next variable set to be initialized is relating to the input/output raster files, respectively, their file names variable holders, their file descriptors, file row values holders, and finally pixel values. It is to be noted here that generally row data and pixels can be set to DCELL (GRASS double type), as it is converted automatically when reading the data in from the raster files as we will see later.

```
/*Input/Output Files name holders*/
char    *bluechan, *greenchan, *redchan, *result;

/*Input/Output Files descriptors */
int     infd_redchan, infd_greenchan, infd_bluechan, outfd;

/*Input row data variables*/
DCELL   *inrast_redchan, *inrast_greenchan, *inrast_bluechan;

/*Output row data variable*/
DCELL   *outrast;

/*Input Pixel variables*/
DCELL   d_bluechan, d_greenchan, d_redchan;
```

After variables initialization, the module is being initialized within the GRASS processing environment with *G_gisinit()*. After this point, as we mentioned earlier, the module is defined, its inputs, outputs are set up, and the user can input the appropriate arguments to order the module to process raster files.

```
/* Initialize Module in GRASS GIS */
G_gisinit(argv[0]);
```

Once the user has provided the module with input arguments, the code is running *G_parser()* on all the input argument set. If not appropriate, the module exists gracefully with meaningful information about what it feels did not go well, and a full instruction set about the module appropriate input arguments.

```
/*Check module input arguments */
if (G_parser(argc, argv))
    /*if check failed, exit gracefully */
    exit(EXIT_FAILURE);
```

From this point onwards, the access to the raster files is starting, first by their file names taken from the module input arguments.

```
/* load file names */
redchan     = input1->answer;
greenchan   = input2->answer;
bluechan    = input3->answer;
result      = output->answer;
```

Once the filenames are stored, the files are accessed using the *Rast_open_old()* internal function and given each a file descriptor. Finally, a row data memory is allocated for each file using *Rast_allocate_d_buf()*, since GRASS GIS typically processes raster files on a row by row basis, which is a robust way to handle more raster files data than your computer memory can handle. Similarly, a new raster file is created to receive the module output data using the *Rast_open_raster_new()* function. Also, as for input raster files, the output raster has a row data being allocated for row-based processing. The various row data are allocated as double type using *Rast_allocate_d_buf()*, thus the "_d_".

```
/* Open access to image files and allocate row access memory */
infd_redchan = Rast_open_old(redchan, "");
inrast_redchan = Rast_allocate_d_buf();

/* Create New raster file and row access memory allocation */
outfd = Rast_open_new(result, DCELL_TYPE);
outrast = Rast_allocate_d_buf();
```

Once all this is set up, we can initialize the image processing, by getting the number of rows (Y-axis) and columns (X-axis) from the GRASS computational region window.

```
   /* Load max rows and max columns */
  nrows = Rast_window_rows();
  ncols = Rast_window_cols();
```

Once inside the row-based loop (in Y-Axis), the code loads the row data from raster images, opens a column-based loop (in X-axis) and loads the pixel values for each raster input based on their (col, row) pair identification.

```
/* Process pixels, access rowxrow first */
for (row = 0; row < nrows; row++)
{
/* Load rows for each input image  */
Rast_get_d_row(infd_redchan, inrast_redchan, row);
Rast_get_d_row(infd_greenchan, inrast_greenchan, row);
Rast_get_d_row(infd_bluechan, inrast_bluechan, row);

/* process the data on pixel basis*/
for (col = 0; col < ncols; col++)
{
    d_redchan       = inrast_redchan[col];
    d_greenchan     = inrast_greenchan[col];
    d_bluechan      = inrast_bluechan[col];
```

Once the pixel is accessed on a case by case basis, the code checks if any of the inputs is holding a NULL value using the *G_is_d_null_value()*, in the positive, the output pixel is directly set to NULL too, by the function *G_set_d_null_value()*. In the negative, the output pixel is filled by the algorithm result coming from the *vari()* function.

```
/* process NULL Values */
if (Rast_is_d_null_value(&d_redchan)||
    (Rast_is_d_null_value(&d_greenchan))||
    (Rast_is_d_null_value(&d_bluechan)))
    Rast_set_d_null_value(&outrast[col], 1);
```

```
        else
            /* if not NULL process index values */
            outrast[col] = vari(d_redchan, d_greenchan, d_bluechan);
```

Once the full output row data is given resulting values, it can be sent to the proper row in the output file with the function *Rast_put_d_row()*.

```
    /* Write data to disk */
    Rast_put_d_row(outfd, outrast);
```

Finally, after all the process finished, memory is freed using *G_free()*, and file handles are closed by *Rast_close()*.

```
    /* Free memory */
    G_free(inrast_redchan);
    Rast_close(infd_redchan);
    ...
```

### 6.1.3 GRASS GIS C code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <grass/gis.h>
#include <grass/raster.h>
#include <grass/glocale.h>
#include "grass_rs.h"

/* Declare requirements of vari() function */
// double vari(double redchan, double greenchan, double bluechan);

int main(int argc, char *argv[])
{
    int     nrows, ncols;                     /*max number of rows/columns*/
    int     row, col;
    struct  GModule *module;                  /*GRASS GIS Module setup*/
    struct  Option *input1, *input2, *input3;   /*CLI/GUI input options */
    struct  Option *output;                   /*CLI/GUI output options */
    struct  History history;                  /*metadata */
    struct  Colors colors;                    /*Color rules */

    char    *result;                          /*output raster name */
    int     infd_redchan, infd_greenchan;     /*File descriptors */
    int     infd_bluechan, outfd;             /*File descriptors */
    char    *bluechan, *greenchan, *redchan;  /*File names holders*/
    DCELL   *inrast_redchan, *inrast_greenchan; /*Input raster variables*/
    DCELL   *inrast_bluechan;                 /*Input raster variables*/
    DCELL   *outrast;                         /*Output raster variable*/
    DCELL   d_bluechan, d_greenchan, d_redchan; /*Input Pixel variables*/
    CELL    val1, val2;             /* For colors range*/
    G_gisinit(argv[0]);

    /* Define CLI/Graphical User Interface inputs and options */
    module = G_define_module();
    G_add_keyword(_("imagery"));
    G_add_keyword(_("vegetation index"));
    G_add_keyword(_("biophysical parameters"));
    module->label       = _("Calculates Visible Atmospherically Resistant Index.");
    module->description = _("Uses red, green and blue bands.");

    input1 = G_define_standard_option(G_OPT_R_INPUT);
    input1->key         = "red";
    input1->label       = _("Name of the red channel surface reflectance map");
    input1->description = _("Range: [0.0;1.0]");

    input2 = G_define_standard_option(G_OPT_R_INPUT);
    input2->key         = "green";
    input2->label       = _("Name of the green channel surface reflectance map");
    input2->description = _("Range: [0.0;1.0]");

    input3 = G_define_standard_option(G_OPT_R_INPUT);
    input3->key         = "blue";
    input3->label       = _("Name of the blue channel surface reflectance map");
    input3->description = _("Range: [0.0;1.0]");

    output = G_define_standard_option(G_OPT_R_OUTPUT);

    if (G_parser(argc, argv))
    exit(EXIT_FAILURE);

    /* load file names */
```

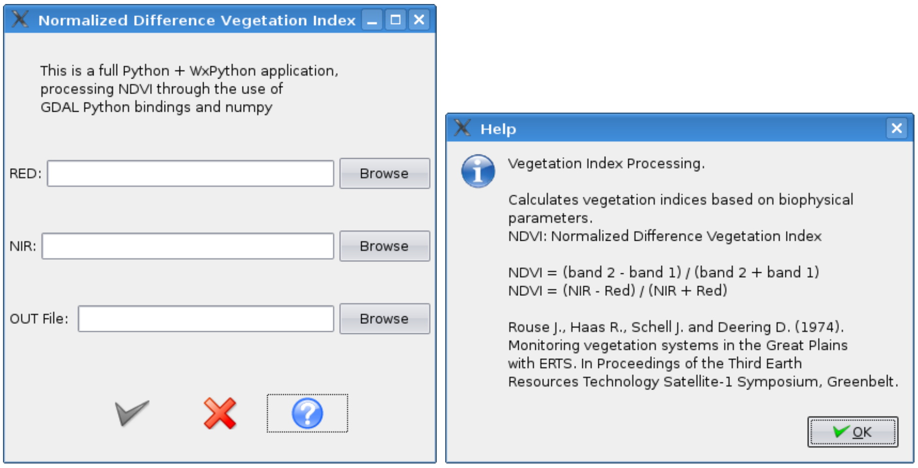
```
redchan     = input1->answer;
greenchan   = input2->answer;
bluechan    = input3->answer;
result      = output->answer;

/* Open access to image files and allocate row access memory */
infd_redchan = Rast_open_old(redchan, "");
inrast_redchan = Rast_allocate_d_buf();

infd_greenchan = Rast_open_old(greenchan, "");
inrast_greenchan = Rast_allocate_d_buf();

infd_bluechan = Rast_open_old(bluechan, "");
inrast_bluechan = Rast_allocate_d_buf();

/* Load max rows and max columns */
nrows = Rast_window_rows();
ncols = Rast_window_cols();

/* Create New raster file and row access memory allocation */
outfd = Rast_open_new(result, DCELL_TYPE);
outrast = Rast_allocate_d_buf();

/* Process pixels, access rowxrow first */
for (row = 0; row < nrows; row++)
{
/* Display row process percentage */
G_percent(row, nrows, 2);

/* Load rows for each input image  */
Rast_get_d_row(infd_redchan, inrast_redchan, row);
Rast_get_d_row(infd_greenchan, inrast_greenchan, row);
Rast_get_d_row(infd_bluechan, inrast_bluechan, row);

/* process the data on pixel basis*/
for (col = 0; col < ncols; col++)
{
    d_redchan       = inrast_redchan[col];
    d_greenchan     = inrast_greenchan[col];
    d_bluechan      = inrast_bluechan[col];

    /* process NULL Values */
    if (Rast_is_d_null_value(&d_redchan)||
        (Rast_is_d_null_value(&d_greenchan))||
        (Rast_is_d_null_value(&d_bluechan)))
        Rast_set_d_null_value(&outrast[col], 1);
    else
        /* if not NULL process index values */
        outrast[col] = vari(d_redchan, d_greenchan, d_bluechan);
}

/* Write data to disk */
Rast_put_d_row(outfd, outrast);
}

/* Free memory */
G_free(inrast_redchan);
Rast_close(infd_redchan);
G_free(inrast_greenchan);
Rast_close(infd_greenchan);
G_free(inrast_bluechan);
Rast_close(infd_bluechan);
```

```
    G_free(outrast);
    Rast_close(outfd);

    /* Color table from -1.0 to +1.0 in grey */
    Rast_init_colors(&colors);
    val1 = -1.0;
    val2 = 1.0;
    Rast_add_c_color_rule(&val1, 0, 0, 0, &val2, 255, 255, 255, &colors);

    /* Copy metadata */
    Rast_short_history(result, "raster", &history);
    Rast_command_history(&history);
    Rast_write_history(result, &history);

    exit(EXIT_SUCCESS);
}
```

## 6.2 WxPython

### 6.2.1 Introduction

Using python-GDAL bindings is attractive as we have seen earlier, but often a user-friendly interface is required. Creating a GUI is therefore often demanded to programmers while it is not their expertise and often not their interest. To be able to generate a single code, handling both processing and GUI, is a nice homogeneity exercise. Here we use WxPython (www.wxpython.org) alongside the Python-GDAL bindings and NumPy (Numeric Python, www.numpy.org) to load, process and write our raster-based geoInformation all in one code block, providing us with a graphical user interface for arguments input.



This is a full Python + WxPython application, processing NDVI through the use of GDAL Python bindings and numpy

Vegetation Index Processing.

Calculates vegetation indices based on biophysical parameters.
NDVI: Normalized Difference Vegetation Index

NDVI = (band 2 - band 1) / (band 2 + band 1)
NDVI = (NIR - Red) / (NIR + Red)

Rouse J., Haas R., Schell J. and Deering D. (1974).
Monitoring vegetation systems in the Great Plains with ERTS. In Proceedings of the Third Earth Resources Technology Satellite-1 Symposium, Greenbelt.

Above is the GUI generated by our example code, along with its Help dialog in a Linux Operating System. WxPython is using the Operating System GUI features to display your code exactly the same as any other native applications, using styles and coloring used for your display.
The code can be run in different ways, directly from the CLI with "python this-code.py", or on some Operating Systems, (double-)clicking on the file directly. It can also be compiled into an executable, on MS Windows O/S, it is handled by a software called py2exe that can be found in www.py2exe.org, for other O/S see www.python.org for more details.

### 6.2.2 Code explanation

The code (which is actually a script too) starts with indicating its nature and how the Operating System should set its interpretation to Python.

```
#!/usr/bin/python
```

After that, it needs to load libraries to manage for us different parts of our software, namely the GUI aspects first, the O/S libraries essentially to acquire the working directory. Later comes the image processing libraries, numpy for its array handling capacities and finally the GDAL bindings.

```
# For the GUI
import wx
import wx.lib.filebrowsebutton as filebrowse
# For getting the working directory
import os
# For Image Processing
import numpy
from osgeo import gdalnumeric
from osgeo import gdal
from osgeo import gdal_array
from osgeo.gdalconst import *
```

Following that, we need to define our basic global variable, the overview text used in the Help/Info.

```
# Define Info Message
overview = """Vegetation Index Processing.

Calculates vegetation indices based on biophysical parameters.
NDVI: Normalized Difference Vegetation Index

NDVI = (band 2 - band 1) / (band 2 + band 1)
NDVI = (NIR - Red) / (NIR + Red)

Rouse J., Haas R., Schell J. and Deering D. (1974).
Monitoring vegetation systems in the Great Plains
with ERTS. In Proceedings of the Third Earth
Resources Technology Satellite-1 Symposium, Greenbelt."""
```

The WxPython integration code has only few parts really, a window frame of type wx.Frame inside a main application definition of type wx.App. The widow frame is defined first, then the main application. At the end of the code, an “if statement” is set, so as to run the code if it is alone to be run.

```
# Main Window Frame Definition
class MyFrame(wx.Frame)

# Main Application Definition
class MainApp(wx.App)

#When the code is run alone, this gets activated
if __name__ == ’__main__’:
    app = wx.App()
    frame = MyFrame(None)
    frame.Show()
    app.MainLoop()
```

The Window frame is defined by overriding some of its default parameters such as giving it a title, an initial position on the screen and a display default size. All this is included in a new function definition, the function is an internal wxPython function specific of the wx.Frame, called __init__(). Additionally, this is where we are going to set up the look of the main GUI window, as you can read, we first use wx.LogNull() to hide any reporting (which may not be safe initially) for aesthetic reasons essentially. Then we declare the input filenames (initialized as null text) and construct the interface. The latter is made in 2 parts, first we call 3 functions (self.make_text(), self.make_buttons(), self.make_fb()) to construct the necessary objects to display inside the window frame. After that we create a wx.BoxSizer object to fit them all with automatic relative locations into the window frame. Once the wx.BoxSizer object called self.mbox is created, we can add object into it with self.mbox.Add(). Once all display object have been added, we set the self.mbox sizer to be activated with self.SetSizer(self.mbox). Finally, we call the function self.bindEvents(), which sets the connection (bindings) between mouse clicks on our display objects and functions we defined inside this application, without bindings, no user interaction !

```
# Main Window Frame Definition
class MyFrame(wx.Frame):
    def __init__(self,parent, id=-1, title='Normalized Difference Vegetation Index
                 Processing',
        pos=(0,0),
        size=(400,400),
        style=wx.DEFAULT_FRAME_STYLE):
        wx.Frame.__init__(self, parent, id, title, pos, size, style)
        # Hide warning reporting
        self.lognull = wx.LogNull()
        # Input Filenames
        self.redchan = ''
        self.nirchan = ''
        self.output = ''
        # Construct Interface
        # Create window frame contents
        self.make_text()
        self.make_buttons()
        self.make_fb()
        # Arrange objects position in the window frame
        self.mbox = wx.BoxSizer(wx.VERTICAL)
        self.mbox.Add((10,10))
        self.mbox.Add(self.text, 1, wx.EXPAND|wx.CENTER, 10)
        self.mbox.Add((10,10))
        self.mbox.Add((10,10))
        self.mbox.Add(self.cc2, 1, wx.EXPAND, 10)
        self.mbox.Add(self.cc3, 1, wx.EXPAND, 10)
        self.mbox.Add(self.cc6, 1, wx.EXPAND, 10)
        self.mbox.Add((10,10))
        self.mbox.Add((10,10))
        self.mbox.Add(self.bbox, 1, wx.CENTER, 10)
        self.mbox.Add((10,10))
        self.SetSizer(self.mbox)
        self.bindEvents()
```

The three Window Frame display functions are self.make_text(), self.make_buttons(), self.make_fb(). self.make_text() is displaying a text in the upper part of the window display frame. self.make_buttons() is displaying OK/Cancel/Help buttons in the lower part of the GUI. You can see that even the button images are default from the O/S you run it from. self.make_fb() is creating the main interaction, that is a set of three raster file access browsing boxes (filebrowse.FileBrowseButton()), two of them dedicated to opening files and one for saving the output file. For each of these, a companion function, a callback (fbbCallback6(self, evt)), handles the conversion of the raster file (path+name) into the appropriate format inside its allocated variable name.

```
# Front text display
def make_text(self):
    self.text = wx.StaticText(self, -1, "\n\tThis is a full Python + WxPython
            application,\n\tprocessing NDVI through the use
            of \n\tGDAL Python bindings and numpy")

# Bottom buttons
def make_buttons(self):
    self.bbox = wx.BoxSizer(wx.HORIZONTAL)
    # OnOK
    bmp0 = wx.ArtProvider.GetBitmap(wx.ART_TICK_MARK,
            wx.ART_TOOLBAR, (32,32))
    self.b0 = wx.BitmapButton(self, 20, bmp0, (20, 20),(bmp0.GetWidth()+50,
            bmp0.GetHeight()+10), style=wx.NO_BORDER)
            self.b0.SetToolTipString("Process")
    self.bbox.Add(self.b0,1,wx.CENTER,10)
    # OnCancel
    bmp1 = wx.ArtProvider.GetBitmap(wx.ART_CROSS_MARK,
            wx.ART_TOOLBAR, (32,32))
    self.b1 = wx.BitmapButton(self, 30, bmp1, (20, 20),(bmp1.GetWidth()+50,
            bmp1.GetHeight()+10), style=wx.NO_BORDER)
    self.b1.SetToolTipString("Abort")
    self.bbox.Add(self.b1,1,wx.CENTER,10)
    # OnInfo
    bmp2 = wx.ArtProvider.GetBitmap(wx.ART_HELP, wx.ART_TOOLBAR, (32,32))
    self.b2 = wx.BitmapButton(self, 40, bmp2, (20, 20),(bmp2.GetWidth()+50,
            bmp2.GetHeight()+10), style=wx.NO_BORDER)
            self.b2.SetToolTipString("Help/Info.")
    self.bbox.Add(self.b2,1,wx.CENTER,10)

# Path+filename seek and set
def make_fb(self):
    # get current working directory
    self.dirnm = os.getcwd()
    self.cc2 = filebrowse.FileBrowseButton(
        self, -1, size=(50, -1), labelText='RED:',
        startDirectory = self.dirnm,
        fileMode=wx.OPEN,
        changeCallback = self.fbbCallback2,
            )
    self.cc3 = filebrowse.FileBrowseButton(
        self, -1, size=(50, -1), labelText='NIR:',
        startDirectory = self.dirnm,
        fileMode=wx.OPEN,
        changeCallback = self.fbbCallback3
            )
    self.cc6 = filebrowse.FileBrowseButton(
        self, -1, size=(50, -1), labelText='OUT File: ',
        startDirectory = self.dirnm,
        fileMask='*.tif',
        fileMode=wx.SAVE,
        changeCallback = self.fbbCallback6
            )
# Collect path+filenames
def fbbCallback2(self, evt):
```

```
         self.redchan = str(evt.GetString())
    def fbbCallback3(self, evt):
         self.nirchan = str(evt.GetString())
    def fbbCallback6(self, evt):
         self.output = str(evt.GetString())
```

With that, it is necessary to set up the basic functionalities of the window frame, there are four of them, three being linked to the main buttons (OnCancel(), OnInfo(), OnOK()), the last one (OnCloseWindow()) is for closing the window frame itself from the Upper Right X button common to all windows. Here we will define three of them, of short and common description and use. The last one (OnOK()), is discussed in the next paragraph. OnCancel() has the function of exiting from the application, this is bound to a button defined in make_button(), called self.b1 using the function self.Bind(wx.EVT_BUTTON, self.OnCancel, self.b1) in BindEvent().

```
    # Main GUI Operative functionalities (OnCloseWindow,OnCancel,OnInfo,OnOK)
    #-----------------------------------------------------------------------
    # Main Window Close button clicked, close application
    def OnCloseWindow(self, event):
        self.Destroy()

    # Cancel button clicked, close application
    def OnCancel(self, event):
        self.Destroy()

    # Display Info/Help
    def OnInfo(self,event):
        dlg = wx.MessageDialog(self, overview,'Help', wx.OK | wx.ICON_INFORMATION)
        dlg.ShowModal()
        dlg.Destroy()

    # Process Equations, Handling and saving of output
    def OnOK(self,event):
```

The function OnOK() is the main function, as it is where raster processing is happening. It first check if both input raster filenames are not empty, if they are, it fails gracefully through self.OnFileInError() otherwise it loads the raster files into GDAL arrays. Once loaded, it processes the raster files through the self.ndvi() functions. Finally, the creation, preparation and loading of data into the output file is taken care by first copying geo-transform and projection information from one input file, then loading the processing result GDAL array and copying geo-transform and projection data into it. Once the file data is ready, it finds out the output driver info (GeoTiff here), and it creates a copy into a file on disk having a filename and a GDAL array with all proper geographical metadata enabled.

```
    # Process Equations, Handling and saving of output
    def OnOK(self,event):
        print "red: ", self.redchan, " nir:",self.nirchan, " out:", self.output
        if(self.redchan==''):
            self.OnFileInError()
        elif(self.nirchan==''):
            self.OnFileInError()
        else:
            self.redband = gdal_array.LoadFile(self.redchan)
            self.nirband = gdal_array.LoadFile(self.nirchan)
        # NDVI
        self.result=self.ndvi(self.redband, self.nirband)

        # prepare/create output file
        tmp = gdal.Open(str(self.redchan))
        geoT = tmp.GetGeoTransform()
        proJ = tmp.GetProjection()
        out = gdal_array.OpenArray(self.result )
        out.SetGeoTransform( geoT )
        out.SetProjection( proJ )
        driver = gdal.GetDriverByName( 'GTiff' )
        driver.CreateCopy( self.output, out )
        self.Destroy()
```

Finally, we will define the main application that will handle our window frame. We define inside of it a function called OnInit(), creating a frame object, and setting it up to display when the application is launched. Finally we make the frame object the highest level display object so the we are sure no other window will hide it on start.

```
    # Main Application Definition
    class MainApp(wx.App):
        def OnInit(self):
            frame = MainFrame(None)
            frame.Show(True)
            self.SetTopWindow(frame)
            return True
```

### 6.2.3 WxPython, Python and GDAL code

```
#!/usr/bin/python
import wx
import wx.lib.filebrowsebutton as filebrowse
import os

# For Image Processing
import numpy
from osgeo import gdalnumeric
from osgeo import gdal
from osgeo import gdal_array
from osgeo.gdalconst import *

# Define satellite bands
redchan = ''
nirchan = ''

# Define output file name
output = ''

# Define Info Message
overview = """Vegetation Index Processing.

Calculates vegetation indices based on biophysical parameters.
NDVI: Normalized Difference Vegetation Index
NDVI = (band 2 - band 1) / (band 2 + band 1)
NDVI = (NIR - Red) / (NIR + Red)

Rouse J., Haas R., Schell J. and Deering D. (1974).
Monitoring vegetation systems in the Great Plains
with ERTS. In Proceedings of the Third Earth
Resources Technology Satellite-1 Symposium, Greenbelt."""

class MyFrame(wx.Frame):
    def __init__(self,parent, id=-1,
            title='Normalized Difference Vegetation Index Processing',
            pos=(0,0),
            size=(400,400),
            style=wx.DEFAULT_FRAME_STYLE):
        wx.Frame.__init__(self, parent, id, title, pos, size, style)
        self.lognull = wx.LogNull()
        # Input Filenames
        self.redchan = redchan
        self.nirchan = nirchan
        self.output = output
        # Construct Interface
        self.make_text()
        self.make_buttons()
        self.make_fb()
        self.mbox = wx.BoxSizer(wx.VERTICAL)
        self.mbox.Add((10,10))
        self.mbox.Add(self.text, 1, wx.EXPAND|wx.CENTER, 10)
        self.mbox.Add((10,10))
        self.mbox.Add((10,10))
        self.mbox.Add(self.cc2, 1, wx.EXPAND, 10)
        self.mbox.Add(self.cc3, 1, wx.EXPAND, 10)
        self.mbox.Add(self.cc6, 1, wx.EXPAND, 10)
        self.mbox.Add((10,10))
        self.mbox.Add((10,10))
        self.mbox.Add(self.bbox, 1, wx.CENTER, 10)
        self.mbox.Add((10,10))
        self.SetSizer(self.mbox)
        self.bindEvents()

    # Process Equations, Handling and saving of output
    def OnOK(self,event):
        print "red: ", self.redchan, " nir:",self.nirchan, " out:", self.output
        if(self.redchan==''):
            self.OnFileInError()
        elif(self.nirchan==''):
            self.OnFileInError()
        else:
            self.redband = gdal_array.LoadFile(self.redchan)
            self.nirband = gdal_array.LoadFile(self.nirchan)
        # NDVI
        self.result=self.ndvi(self.redband, self.nirband)

        # prepare/create output file
        tmp = gdal.Open(str(self.redchan))
        geoT = tmp.GetGeoTransform()
        proJ = tmp.GetProjection()
        tmp = None
        out = gdal_array.OpenArray(self.result )
        out.SetGeoTransform( geoT )
        out.SetProjection( proJ )
        driver = gdal.GetDriverByName( 'GTiff' )
        driver.CreateCopy( self.output, out )
        self.Destroy()

    def ndvi( self, redchan, nirchan ):
            """
            Normalized Difference Vegetation Index
            ndvi( redchan, nirchan )
            """
            result = 1.0*( nirchan - redchan )
            result /= 1.0*( nirchan + redchan )
            return result

    def OnFileInError(self):
        dlg = wx.MessageDialog(self,
                'Minimum files to add:\n\n
                Input files => Red and NIR\n  One Output file',
                'Error',wx.OK | wx.ICON_INFORMATION)
        dlg.ShowModal()
        dlg.Destroy()

    # Path+filename seek and set
    def make_fb(self):
        # get current working directory
        self.dirnm = os.getcwd()
        self.cc2 = filebrowse.FileBrowseButton(
            self, -1, size=(50, -1), labelText='RED:',
            startDirectory = self.dirnm,
            fileMode=wx.OPEN,
            changeCallback = self.fbbCallback2,
                )
        self.cc3 = filebrowse.FileBrowseButton(
            self, -1, size=(50, -1), labelText='NIR:',
            startDirectory = self.dirnm,
            fileMode=wx.OPEN,
            changeCallback = self.fbbCallback3
                )
        self.cc6 = filebrowse.FileBrowseButton(
            self, -1, size=(50, -1), labelText='OUT File: ',
            startDirectory = self.dirnm,
            fileMask='*.tif',
            fileMode=wx.SAVE,
            changeCallback = self.fbbCallback6
                )
    # Collect path+filenames
    def fbbCallback2(self, evt):
         self.redchan = str(evt.GetString())
    def fbbCallback3(self, evt):
         self.nirchan = str(evt.GetString())
    def fbbCallback6(self, evt):
         self.output = str(evt.GetString())
    # Front text
    def make_text(self):
        self.text = wx.StaticText(self, -1, "\n\tThis is a full Python +
            WxPython application,\n\tprocessing NDVI through the use
            of \n\tGDAL Python bindings and numpy")

    # Bottom buttons
    def make_buttons(self):
        self.bbox = wx.BoxSizer(wx.HORIZONTAL)
        # OnOK
        bmp0 = wx.ArtProvider.GetBitmap(wx.ART_TICK_MARK, wx.ART_TOOLBAR, (32,32))
        self.b0 = wx.BitmapButton(self, 20, bmp0, (20, 20),
            (bmp0.GetWidth()+50, bmp0.GetHeight()+10), style=wx.NO_BORDER)
            self.b0.SetToolTipString("Process")
        self.bbox.Add(self.b0,1,wx.CENTER,10)
        # OnCancel
        bmp1 = wx.ArtProvider.GetBitmap(wx.ART_CROSS_MARK, wx.ART_TOOLBAR, (32,32))
        self.b1 = wx.BitmapButton(self, 30, bmp1, (20, 20),
            (bmp1.GetWidth()+50, bmp1.GetHeight()+10), style=wx.NO_BORDER)
        self.b1.SetToolTipString("Abort")
        self.bbox.Add(self.b1,1,wx.CENTER,10)
        # OnInfo
        bmp2 = wx.ArtProvider.GetBitmap(wx.ART_HELP, wx.ART_TOOLBAR, (32,32))
        self.b2 = wx.BitmapButton(self, 40, bmp2, (20, 20),
            (bmp2.GetWidth()+50, bmp2.GetHeight()+10), style=wx.NO_BORDER)
            self.b2.SetToolTipString("Help/Info.")
        self.bbox.Add(self.b2,1,wx.CENTER,10)

    def bindEvents(self):
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)
            self.Bind(wx.EVT_BUTTON, self.OnOK, self.b0)
        self.Bind(wx.EVT_BUTTON, self.OnCancel, self.b1)
        self.Bind(wx.EVT_BUTTON, self.OnInfo, self.b2)

    def OnCloseWindow(self, event):
        self.Destroy()

    def OnCancel(self, event):
        self.Destroy()

    def OnInfo(self,event):
        dlg = wx.MessageDialog(self, overview,'Help', wx.OK | wx.ICON_INFORMATION)
        dlg.ShowModal()
        dlg.Destroy()

class MainApp(wx.App):
    def OnInit(self):
        frame = MainFrame(None)
        frame.Show(True)
        self.SetTopWindow(frame)
        return True

if __name__ == '__main__':
    app = wx.App()
    frame = MyFrame(None)
    frame.Show()
    app.MainLoop()
```

# Chapter 7

# Appendices

## 7.1 Appendix A: Remote Sensing functions in C

### 7.1.1 Remote sensing functions compendium: rs.c and rs.h

```
/* File rs.c
 * Compendium of RS functions for C language
 */

/* MODLAND QA Bits 250m Unsigned Int bits[0-1]
 * 00 -> class 0: Corrected product produced at ideal quality -- all bands
 * 01 -> class 1: Corrected product produced at less than ideal quality --
some or all bands
 * 10 -> class 2: Corrected product NOT produced due to cloud effect -- all bands
 * 11 -> class 3: Corrected product NOT produced due to other reasons --
some or all bands maybe fill value (Note: value of [11] overrides value of [01])
 */

int qc250a (int pixel)
{
    return (pixel & 0x03);
}

/*MODLAND QA Bits 500m State QA  Cloud State unsigned int bits[0-1]
 * 00 -> class 0: clear
 * 01 -> class 1: cloudy
 * 10 -> class 2: mixed
 * 11 -> class 3: not set, assumed clear
 */

int stateqa500a(int pixel)
{
    /* Select bit 0 and 1 (right-side). hexadecimal "0x03"
    => binary "11" this will set all other bits to null */
    return (pixel & 0x03);
}

/* Normalized Difference Vegetation Index. */
/* Rouse J., Haas R., Schell J. and Deering D. (1974).
Monitoring vagetation systems in the Great Plains with ERTS.
```

```
In Proceedings of the Third Earth Resources Technology
Satellite-1 Symposium, Greenbelt.*/

float ndvi(float red, float nir)
{
    double result;
    if(red+nir<=0.001){
        result=-1.0;
    }else{
        result=(nir-red)/(red+nir);
    }
    return result;
}

/*VARI: Visible Atmospherically Resistant Index, it was designed
to introduce an atmospheric self-correction.
 * Gitelson A.A., Kaufman Y.J., Stark R., Rundquist D., 2002. Novel
algorithms for estimation of vegetation fraction Remote Sensing of
Environment (80), pp76-87. */

double vari(double redchan, double bluechan, double greenchan)
{
    double result;
    result = (greenchan - redchan ) / (greenchan + redchan - bluechan);
    return result;
}

/* Water identification by using MODIS imagery */
/*Xiao, X., Boles, S., Frolking, S., Li, C., Babu, J.Y., Sala, W.,
Morre III, Berrien, 2006, Mapping paddy rice agriculture in South and
South-East Asia using multi-temporal MODIS images.
Remote Sensing of Environment. 100, 95-113. */
int water_modis(double surf_ref_7, double ndvi)
{
    double result;
    if (surf_ref_7<0.04&&ndvi<0.1){
        result = 1 ;
    } else {
        result = 0 ;
    }
    return result;
}


/* File rs.h
 * Compendium of RS functions headers for C language
 */
int qc250a (int pixel);
int stateqa500a(int pixel);
float ndvi(float red, float nir);
double vari(double redchan, double bluechan, double greenchan);
int water_modis(double surf_ref_7, double ndvi);
```

### 7.1.2 GRASS GIS dedicated functions

```
/* File grass_rs.c
 * Compendium of RS functions for C language in GRASS GIS
 */

* MODLAND QA Bits 250m Unsigned Int bits[0-1]
 * 00 -> class 0: Corrected product produced at ideal quality -- all bands
 * 01 -> class 1: Corrected product produced at less than ideal quality --
some or all bands
 * 10 -> class 2: Corrected product NOT produced due to cloud effect -- all bands
 * 11 -> class 3: Corrected product NOT produced due to other reasons --
some or all bands maybe fill value (Note: value of [11] overrides value of [01])
 */

#include "grass/gis.h"
CELL qc250a (CELL pixel)
{
    CELL qctemp;
    qctemp = pixel & 0x03;
    return qctemp;
}

/*MODLAND QA Bits 500m State QA  Cloud State unsigned int bits[0-1]
 * 00 -> class 0: clear
 * 01 -> class 1: cloudy
 * 10 -> class 2: mixed
 * 11 -> class 3: not set, assumed clear
 */

#include "grass/gis.h"
CELL stateqa500a(CELL pixel)
{
    CELL qctemp;
    /* Select bit 0 and 1 (right-side). hexadecimal "0x03"
    => binary "11" this will set all other bits to null */
    qctemp = pixel & 0x03;
    return qctemp;
}

/* File grass_rs.h
 * Compendium of RS functions headers for C language
 */
CELL qc250a (CELL pixel);
CELL stateqa500a(CELL pixel);
```

## 7.2 Appendix B: GRASS GIS module & companion files

Besides from main.c which is our example file, the remote sensing functions dedicated files (grass_rs.c, grass_rs.h), building a GRASS GIS module requires a Makefile and a help file named after the module it is describing (i.vari.html). These two files, are joining the three other files into the module directory i.vari/
If you need to have your new module accessed from the main GRASS GIS menu, place the following XML code in grass/gui/wxpython/xml/menudata.xml file, where you feel suitable:

```
<menuitem>
  <label>VARI</label>
  <help>Visible Atmospherically Resistant Index.</help>
  <handler>self.OnMenuCmd</handler>
  <command>i.vari</command>
</menuitem>
```

### 7.2.1 Module directory

Our module is called i.vari, in GRASS GIS, i.* denotes an imagery related module and remote sensing processing of a vegetation index fits that description. Within the GRASS GIS source tree, we will create a directory (i.e. grass/imagery/i.vari/) where we can put the following files in. The checklist for i.vari module directory:

- main.c
- grass_rs.c
- grass_rs.h
- Makefile
- i.vari.html

### 7.2.2 Makefile

```
MODULE_TOPDIR = ../..
PGM = i.vari
LIBES = $(GISLIB) $(RASTERLIB)
DEPENDENCIES = $(GISDEP) $(RASTERDEP)
include $(MODULE_TOPDIR)/include/Make/Module.make
ifneq ($(USE_LARGEFILES),)
    EXTRA_CFLAGS = -D_FILE_OFFSET_BITS=64
endif
default: cmd
```

### 7.2.3 i.vari.html

```
<H2>DESCRIPTION</H2>
<EM>i.vari</EM> calculates VARI: Visible Atmospherically Resistant Index
<pre>
VARI: Visible Atmospherically Resistant Index
VARI = (green - red ) / (green + red - blue)
it was designed to introduce an atmospheric self-correction
Gitelson A.A., Kaufman Y.J., Stark R., Rundquist D., 2002.
Novel algorithms for estimation of vegetation fraction
Remote Sensing of Environment (80), pp76-87.
</pre>
<H2>SEE ALSO</H2>
<em>
  <a href="i.vari.html">i.vari</a>
</em>
<H2>AUTHORS</H2>
Yann Chemin, International Water Management Institute, Sri Lanka<br>
<p>
<i>Last changed: $Date:$</i>
```

## 7.3 Appendix C: Arrays functions in C

```
/* File arrays.c
 * Compendium of arrays functions for C language
 */

#include<stdio.h>
#include<stdlib.h>

/*Create 1D arrays*/

int* ai1d(int X)
{
    int* A = (int*) malloc(X*sizeof(int*));
    return A;
}

float* af1d(int X)
{
    float* A = (float*) malloc(X*sizeof(float*));
    return A;
}

double* ad1d(int X)
{
    double* A = (double*) malloc(X*sizeof(double*));
    return A;
}

/*Create 2D arrays*/

int** ai2d(int X, int Y)
{
    int i;
    int** A = (int**) malloc(X*sizeof(int*));
    for (i = 0; i < X; i++)
        A[i] = (int*) malloc(Y*sizeof(int));
    return A;
}
```

```
float** af2d(int X, int Y)
{
    int i;
    float** A = (float**) malloc(X*sizeof(float*));
    for (i = 0; i < X; i++)
        A[i] = (float*) malloc(Y*sizeof(float));
    return A;
}

double** ad2d(int X, int Y)
{
    int i;
    double** A = (double**) malloc(X*sizeof(double*));
    for (i = 0; i < X; i++)
        A[i] = (double*) malloc(Y*sizeof(double));
    return A;
}

/* File arrays.h
 * Compendium of arrays functions headers for C language
 */

#include<stdio.h>

/*Create 1D arrays*/

int*        ai1d(int X);
float*      af1d(int X);
double*     ad1d(int X);

/*Create 2D arrays*/

int**       ai2d(int X, int Y);
float**     af2d(int X, int Y);
double**    ad2d(int X, int Y);
```

## 7.4 Appendix D: kernels holding file for OpenCL

```
/* File kernels.cl
 * Compendium of RS kernel functions for OpenCL language
 */

//Define NDVI Kernel
//This kernel outputs in (1+NDVI)*100, range [0-200]

__kernel void nd_vi(    __global const float * red, __global const float * nir, __global float * ndvi)
{
    int i = get_global_id(0);
    if((nir[i]+red[i])<0.01)
        ndvi[i]=255;
    else
        ndvi[i] = 100*(1+((nir[i]-red[i])/(nir[i]+red[i])));
}
```

## 7.5 License for the code in this book

All code in this book is Public Domain, if you really need to include information about the author the following can be used.

```
################################################################################
# Public Domain 2008-2011, Yann Chemin <yann.chemin@gmail.com>
################################################################################
```