JOINT RESEARCH CENTRE
EUROPEAN COMMISSION

Institute for Environment and Sustainability

The STARS FAPAR Algorithm: A Consolidated and Generalized Software Package

Users' Guide

by Malcolm Taberner, Nadine Gobron, Frédéric Mélin, Bernard Pinty and Michel M. Verstraete

Institute for Environment and Sustainability
Joint Research Centre, TP 440
I–21020 Ispra (VA), Italy

Version 1.0, January 22, 2002

JRC Publication No. EUR 20145 EN

Chapter **Page**

Chapter 1

Introduction

1.0 Purpose

This guide summarises the background, design, and use of the STARS group FAPAR algorithm software. The code was developed in IDL and C++, and is available as a package. The code was developed in such a manner that the end user will be able to manipulate the user interface easily, whilst the main algorithm components are protected behind the interface. The addition of new sensors, using existing generalised code has been facilitated. The addition of new, sensor specific code, when developments require it, has been incorporated, *especially within the C++* code. The intention is that this code be made available on the internet for downloading by potential users.

1.1 Requirements and Context

The STARS group has developed procedures for producing the Fraction of Absorbed Photosynthetically Active Radiation (FAPAR) using a technique which is optimised for various sensors. The technique incorporates corrections for angular, soil, and atmospheric variations. The sensors for which the routines have been optimised include GLobal Imager (GLI, on ADEOS−2), MEdium Resolution Imaging Spectrometer Instrument (MERIS, on ENVISAT), Sea Wide Field−of−view Sensor (SeaWiFS on ORBVIEW−2), and SPOT VEGETATION, reflecting the "Global Vegetation Monitoring" perspective of the group.

The current package is a consolidation and development of various research codes produced by N. Gobron, in general, F. Mélin, for SeaWIfs, and M. Verstraete, for the GLI sensor. The code, having been made more user friendly and robust, is made available to others. The code is designed to be of assistance to two levels of users: the "expert" scientific user who would understand the production of the product and would be able to readily assimilate the code into their own procedures; and a more general user who would only be interested in deriving the product with as little difficulty as possible. The code is made available in the IDL programming language, and in C++. This is a core product, intended to be downloaded from the GVM website, with a generic input structure and a recommended output format in a form that is both stand−alone yet also easily integrated into users own operational code, *whilst at the same time not jeopordising the integrity of the computational code.* The option to save the intermediate rectified channels is available.

The IDL code has been compiled and run under versions 5.2 and 5.4 of IDL. C++ code was compiled using version 2.95.2 of the GNU gcc compiler suite (g++). These programs have been run on AIX IBM 6000 and Sun Ultra 60 machines. The aim has been to write code that, in general, conforms to IDL programming guidelines ("Coding Style Guide for Interactive Visuals, Inc.", 1999, Interactive Visuals) exceptions to the recommendations were made, however, when efficiency or expediency were at stake. The C++ code conforms to ISO/IEC 14882 and uses features of the Standard Template Library. Where appropriate C code conforms to ISO9899:1990 with corrections in the Technical Corrigenda 1994 and 1996.

1.2 Outline

The report is split into four sections: the first describes the FAPAR algorithm as it affects the software design, the second and third sections provide an overview of the design, components, and use of the IDL and C++ packages, the last section presents examples of synthetic and real remotely sensed data sets.

1.3 Reference Documents

This document, which describes the software publicly available to generate FAPAR using algorithms optimised for individual sensors, is one of a series of documents covering this approach. The physics and optimisation is fully described in separate technical reports (Gobron et al. 2001, references 2 through 5 and 7) and are available on the web at http://www.enamors.org and at http://ies.jrc.cec.eu.int/Units/gvm under the STARS group.

1.4 Development

This code results from research and development carried out here, at the JRC. This research is ongoing, both to improve current techniques, and to exploit new instruments. The code is, therefore, a snapshot of the current state of the art and will be added to, and modified, in future according to developments − facilities for such adaptation is built into the code. As this is an ongoing development, some of the code incorporated in the software derives from earlier research. This code is maintained in the software package because it corresponds to one or more current, operational, implementations. For the general user, though, the code based on the latest research is implemented.

Chapter 2

The FAPAR Algorithm

2.0 Introduction

The scientific background to the algorithm is described in Gobron et al. (2000) and Gobron et al. (2001, ref:6). In this chapter a brief overview of the algorithm, primarily as it affects the code development, and consolidated tables of coefficients and polynomials, are presented. In so far as the software development is concerned, for processing data from a particular sensor, there are 7 main steps, as shown in figure 2.0.

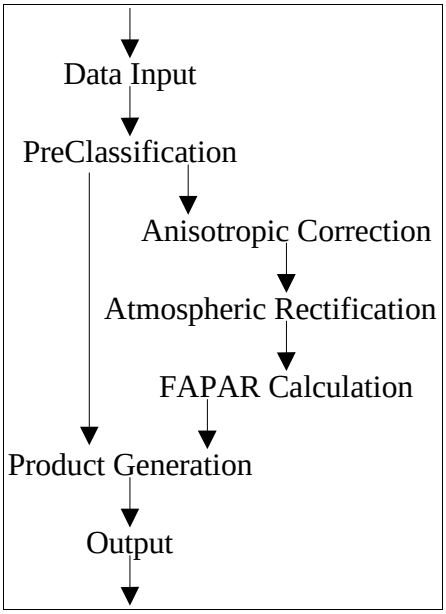

Figure 2.0 Processing Flow for FAPAR Product Generation for a Single Sensor.

2.1 Input Data

The data input is calibrated, top of the atmosphere, bidirectional reflectance factor (BRF), which should include compensation for the variable sun−earth distance. As well as reflectance, the illumination geometry is required (solar zenith and azimuth) and viewing geometry (viewing zenith and azimuth) for each pixel (strictly speaking relative azimuth is used).

2.2 Preclassification

Preclassification is used to define the bounds of the technique. There are 4 main tests (figure 2.1). As the proportion of pixels failing to get through these tests is usually high, and the FAPAR computation also being time consuming, the overhead caused by prefiltering is negligible compared to the time saved.

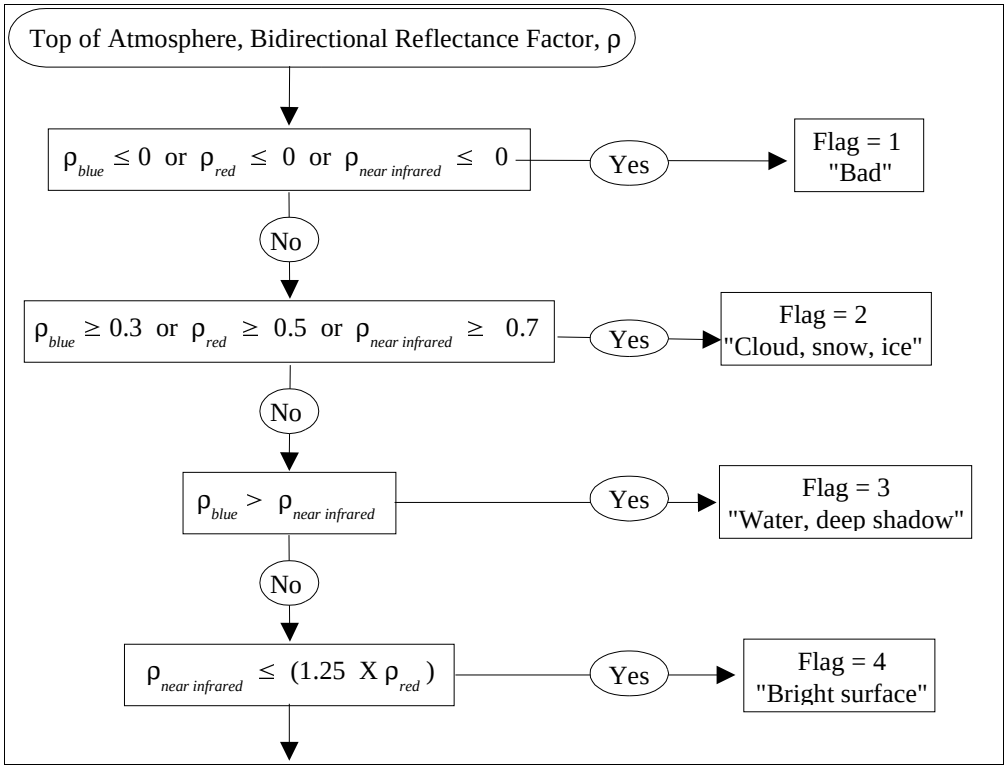

Figure 2.1 Spectral tests defining the FAPAR spectral domain (derived from Gobron et al, 2001 ref:6).

2.3 Anisotropic Normalisation

Signal variation caused by changes in the geometrical conditions are accounted for by a procedure based on the parametric bidirectional reflectance model (figure 2.2) of Rahman et al. 1993 − the Rahman, Pinty, Verstraete (RPV) model. This implementation,which is a generalised procedure, requires estimates of three parameters: ρ_{i0}, k_i, and Θ_i^{hg}, in order to characterise the vegetation canopy. These parameters are independent of geometry and illumination, but are specific to each sensor (Table 2.1). For greater details of the technique the reader is referred to the references.

$$\tilde{\rho}(\lambda_i) = \frac{\rho^{toa}(\Omega_0, \Omega_v, \lambda_i)}{F(\Omega_0, \Omega_v, k_{\lambda i}, \Omega_{\lambda i}^{HG}, \rho_{\lambda ic})}$$

λ_i = central wavelength (blue, red, near infrared) of band *i*

$\rho^{toa}(\Omega_0, \Omega_v, l_i)$ = BRF values measured by the sensor in band λ_i as a function of the illumination and observation geometry, defined by the appropriate zenith angles and their relative azimuth.

$F(\Omega_0, \Omega_v, k_{\lambda i}, \Omega_{\lambda i}^{HG}, \rho_{\lambda ic})$ = anisotropic reflectance function representing the shape of the reflectance field where:

$k_{\lambda i}, \Omega_{\lambda i}^{HG}, \rho_{\lambda i}$ = RPV parameters optimised, *a priori,* for each spectral band.

Figure 2.2 Anisotropic normalisation function.

Table 2.1 Band wavelengths and values of the anisotropic correction parameters.

Sensor	Central λ (nm)	Width (nm)	ρ_{ic}	k_i	Θ_i^{hg}
GLI	443	10	−0.13515	0.45696	0.01813
	678	10	−0.65625	0.78673	0.12335
	865	10	0.63484	0.87758	−0.00264
MERIS	441	10	−0.13515	0.45696	0.01813
	685	10	−0.65625	0.78673	0.12335
	865	10	0.63484	0.87758	−0.00264
SeaWiFS	443	20	0.23265	0.56184	−0.04125
	670	20	−0.44444	0.70535	0.03576
	865	40	0.63149	0.86644	−0.00102
SPOT VEGETATION	450	20	−0.25910	0.46011	0.02979
	640	30	−0.53764	0.77449	0.10338
	840	50	0.62335	0.88468	−0.00219

2.4 Atmospheric Rectification

Minimisation of atmospheric and soil perturbation factors is based on the model by Gobron et al., 1997. Rectified red and near infrared channels are produced using information from combinations of the blue with each of the prerectified red and near infrared channels respectively. Development and calibration has resulted in, sensor specific, polynomial coefficients for the band combinations, for each of the rectified channels. For the near infrared channel, ratios of polynomials are used (figure 2.3).

2.5 Calculation of the Fraction of Absorbed Photosynthetically Active Radiation (FAPAR)

FAPAR is calculated from the rectified red and near infrared channels. Calibration and optimisation has resulted in a set of polynomial coefficients for each sensor. FAPAR is calculated as a ratio of polynomials (figure 2.4).

2.6 Product Generation

In the research code, various output formats exist so it was decided to define a single output format which would be adopted in all the new codes, and would be recommended to other users. For reasons of space and portability it was decided that the flags should be included in the same data array as the FAPAR values. It was also decided that sufficient accuracy would be retained by using output arrays, or files, of byte (unsigned) data. The mapping is shown in figure 2.5. The limits imposed by the scaling ensure a maximum accuracy of ± 0.002.

$$B_{rect\ red} = l_{1,1}(B_{blue} + l_{1,2})^2 + l_{1,3}(B_{red} + l_{1,4})^2 + l_{1,5} B_{blue} B_{red}$$

GLI	*l*	-11.12000	-0.028923	1.77210	0.098161	11.05800
MERIS	*l*	-11.12000	-0.028923	1.77210	0.098161	11.05800
SeaWiFS	*l*	-9.87250	-0.027458	2.91440	0.059376	10.90400
Vegetation	*l*	-12.87700	-0.019822	1.01800	0.138320	14.59300

$$B_{rect\ nir} = \frac{l_{1,1}(B_{blue} + l_{1,2})^2 + l_{1,3}(B_{nir} + l_{1,4})^2 + l_{1,5} B_{blue} B_{nir}}{l_{2,1}(B_{blue} + l_{2,2})^2 + l_{2,3}(B_{nir} + l_{2,4})^2 + l_{2,5} B_{blue} B_{nir}}$$

GLI		-1.85260	1.10670	7.061700	-0.669650	23.01500
		0.40770	2.60720	-0.031230	14.901000	14.90100
MERIS		-1.85260	1.10670	7.061700	-0.669650	23.01500
		0.40770	2.60720	-0.031230	14.901000	14.90100
SeaWiFS		-0.66956	-0.16930	-0.071256	-0.090485	-0.81353
		-0.03544	-1.34380	-0.416730	-0.451230	-0.99648
Vegetation		-1.11350	2.16970	0.292930	4.261400	65.12800
		-204.33000	-0.13200	0.010911	23.812000	5.59270

Figure 2.3 Atmospheric rectification polynomials and coefficients.

$$FAPAR = \frac{l_{1,1} B_{R\ nir} - l_{1,2} B_{R\ red} - l_{1,3}}{(l_{2,1} - B_{R\ red})^2 + (l_{2,2} - B_{Rnir})^2 + l_{2,3}}$$

GLI		0.28787000	0.33326000	-0.0073898000
		-0.13661000	0.32259000	0.0983670000
MERIS		0.28787000	0.33326000	-0.0073898000
		-0.13661000	0.32259000	0.0983670000
SeaWiFS		0.25130709	0.30589629	-0.0048298022
		-0.32136740	0.31415914	-0.0107441800
Vegetation		0.37598000	0.50132000	-0.0109110000
		-0.17150000	0.29464000	0.1100900000

Figure 2.4 FAPAR polynomials and coefficients.

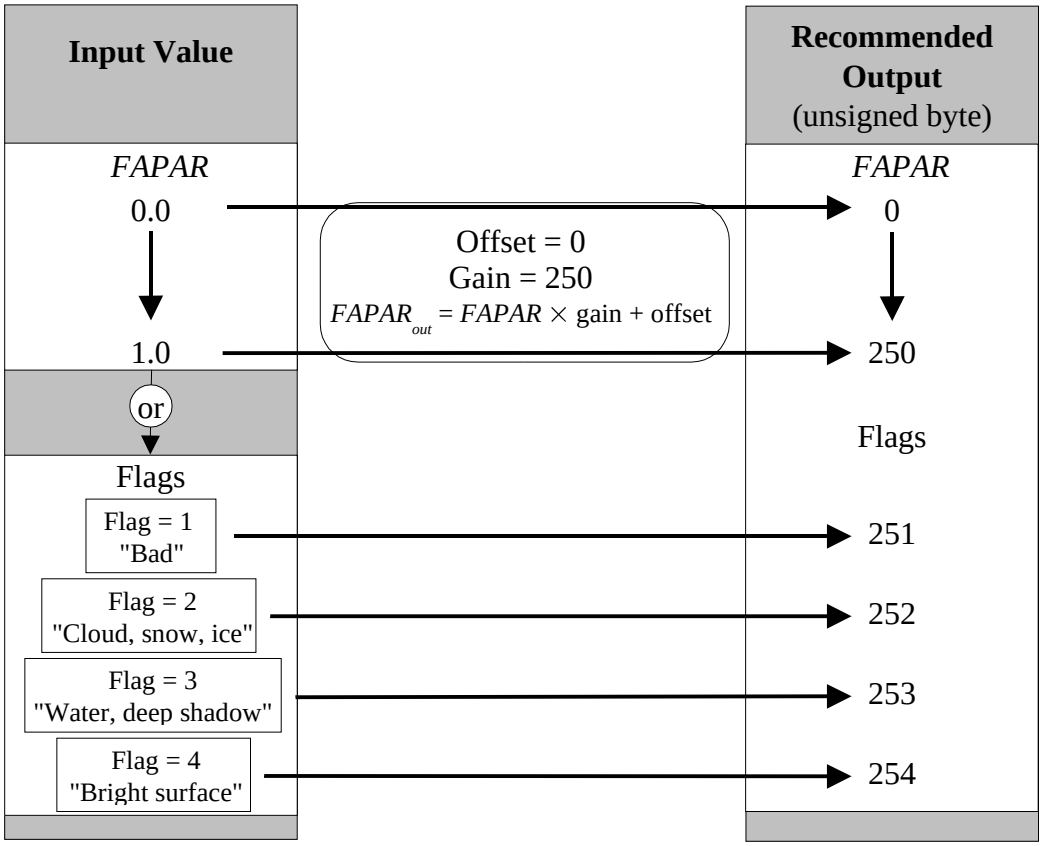

Figure 2.5 Mapping to the recommended output product.

Chapter 3

Generalised IDL Code

3.0 Introduction (Design issues)

The sizes of satellite sensor data sets can be, and in fact usually are, large. The prime operational constraint is, therefore, one of memory requirement. The code developed has, therefore, been optimised with this in mind, sometimes at the (slight) expense of computational efficiency.

There was a requirement to generalise the procedures as far as possible, and to minimise the sensor specific ones. The program structure reflects this requirement. Unfortunately there are differences amongst the implementations that make the task of generalisation more complicated. Furthermore, a general desire to update the code with improved routines as they are developed further complicates, and dictates, the form of the program structure.

The other main design issue is how to incorporate the sensor specific information both at the level of the program structure, and at the level of program control, whilst maintaining generality as much as possible.

The worst case for generality, yet the simplest case as far as implementation is concerned, is to use a switch statement early on in the control program to divert control to sensor specific code. After the switch statement it becomes difficult to use generic (control) code. For generality it is necessary to postpone sensor specific implementation as long as possible. In this software, selection of sensor specific code is achieved by passing a string defining the sensor type in the program flow and using this as a prefix to appropriate functions in combination with the "Call_Function" routine. This approach means that the controlling code can be written in a general manner, yet calls to sensor specific code can be made as and when necessary.

Generality also suffers as not only are the polynomial coefficients specific to each sensor, but the form of the polynomial and their implementation also varies (and is likely to be modified in the future). So, although the polynomials can be written as generic polynomial functions, there implementation is sensor specific. The problem arises when, after initialising the coefficients, control returns to the main code. Which polynomial should then be used? To simplify this, a structure is returned containing not only the coefficients (both numerator and denominator if implemented as a ratio) but the name of the polynomial which is to be associated with these coefficients. The correct polynomial, along with the appropriate coefficients is called from the control code also using "call_function".

3.1 The IDL Package.

The Fapar package is delivered as a tarred and gzipped file (Fapar_1.0.pro.tar.gz, 1.8 MB) which unpacks into a directory ../ Fapar_1.0. Figure 3.0 shows the package layout after unpacking (58.5 MB). In the top level directory: FaparStartUp is used to

load the Fapar program, Fapar.pro (figure 3.1) contains the main driver and user interface program; FaparGeneric.pro (figure 3.2) contains the generic routines; and FaparSensor.pro (figure 3.3) the sensor specific routines. This package format has been adopted to make the code and its handling simple and clean for the end user. Operationally, a better layout would be to have all the routines separate, in appropriate subdirectories, and to be compiled by IDL as required.

FaparColor.pro is code to implement a recommended colour lookup table. The color table is loaded automatically on compilation of the program.

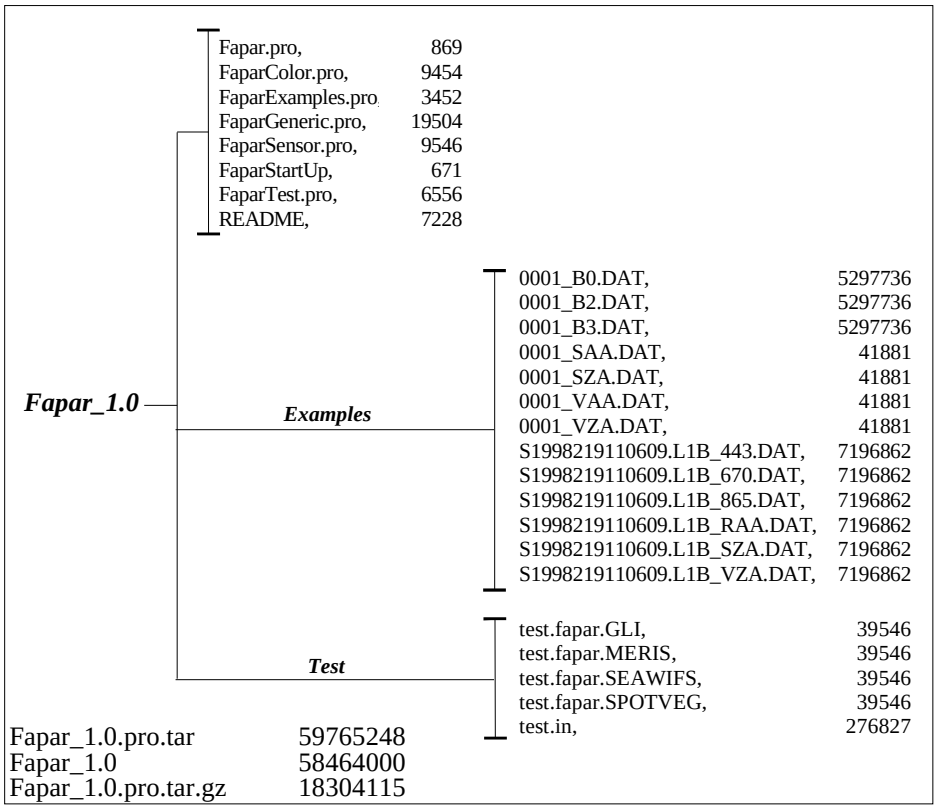

Figure 3.0 The IDL Fapar Package (file sizes in Bytes).

Fapar.pro

Main Driver Routine and User Interface

function **Fapar**, blue, red, nir, SZA=SZA, VZA=VZA, SAA=SAA, VAA=VAA, sensor=sensor, degrees=degrees, rectRed=rectRed, rectNir=rectNir

Figure 3.1 The main driver routine and user interface, *Fapar.pro.*

FaparTest.pro (figure 3.4) is run after the main Fapar package has been compiled. It calls on the compiled procedures to produce results for each sensor, and these are compared to a synthetic test reference data set. The tests are either passed, if within a certain tolerance figure, or failed. The test data set are in the *test* subdirectory and include *test.in* the input test data set, and the reference results *test.fapar.sensor*.

FaparGeneric.pro
General and generalised routines

Messages (error and usage)

pro **PrintAbort**
pro **PrintAtmosRectCoeffError**, abort=abort
pro **PrintNoBandsSelectedError**, name, abort=abort
pro **PrintInputDataError**, nPars, abort=abort
pro **PrintKeywordError**, name, abort=abort
pro **PrintNoProcessingError**, abort=abort
function **PrintDimError**, nDim0, nDim1, name=name, abort=abort

Checking Set Up

pro **CheckInputDims**, dims, names=pNames
pro **CheckInputKeywords**, bS, pNames=pNames, SZA=SZA, VZA=VZA, SAA=SAA, VAA=VAA, sensor=sensor, sNames=sNames
function **CheckData**, blue, red, nir
function **SubsetData**, blue, red, nir, SZA, VZA, SAA, VAA, mask

General Polynomial Functions

function **Poly0,** d0, d1, cf
function **Poly1**, d0, d1, cf
function **Poly2**, d0, d1, cf

Anisotropic Normalisation

pro **FRahman,** data, solarZenith, viewZenith, relAzim, sensor
pro **AnisotropicNormalisation**, data, sZ, vZ, rA, sensor, degrees=degrees

Atmospheric Rectification

function **AtmosRectPoly0**, d0, d1, cf
function **AtmosRectPoly1**, d0, d1, cf
function **AtmosphericRectification**, d0, d1, bands=bands, sensor=sensor
function **DoAtmosphericRectification**, data, sensor

Fapar Calculation

function **FaparPoly0**, d0, d1, cf
function **FaparPoly1**, d0, d1, cf
function **DoFapar**, data, sensor

Output Format

pro **RectificationOutput**, data, mask, missingVal, rectRed=rectRed, rectNir=rectNir
pro **FaparOutput**, data, mask, gain=gain

Figure 3.2 IDL functions and procedures in *FaparGeneric.pro.*

FaparExamples.pro (figure 3.5) loads example SeaWiFS and SPOT VEGETATION imagery, computes the Fapar, and displays the results on the screen. The example imagery is in the *examples* subdirectory and is in a modified format, not the original distribution format.

The README file provides a brief description of usage, references, copyright and contact information (figure 3.6).

3.2 Compilation.

The procedures can be compiled separately using the *.run* command (*.run FaparSensor FaparGeneris Fapar*), or the simple script, *FaparStartUp,* can be run from the command line by *@FaparStartUp.*

FaparSensor.pro
Sensor specific data (coefficients, polynomial type)

GLI Sensor
function **GLIAnisotropicReflectanceParameters,** nBands=nBands
function **GLIAtmosphericRectificationCoefficients**, band
function **GLIFaparCoefficients**

MERIS Sensor
function **MERISAnisotropicReflectanceParameters**, nBands=nBands
function **MERISAtmosphericRectificationCoefficients**, band
function **MERISFaparCoefficients**

SeaWiFS Sensor
function **SEAWIFSAnisotropicReflectanceParameters**, nBands=nBands
function **SEAWIFSAtmosphericRectificationCoefficients**, band
function **SEAWIFSFaparCoefficients**

SPOT Vegetation Sensor
function **SPOTVEGAnisotropicReflectanceParameters**, nBands=nBands
function **SPOTVEGAtmosphericRectificationCoefficients**, band
function **SPOTVEGFaparCoefficients**

Figure 3.3 IDL functions and procedures in *FaparSensor.pro.*

FaparTest.pro
Routines to test FAPAR results

Function **CalcFapar,** brfdata, sensor
function **FaparTestData**, name, columns, rows
function **FaparRefDat**a, name, columns, rows, float=float
pro **FaparTest**, SPOTVEG=SPOTVEG, GLI=GLI, MERIS=MERIS, SEAWIFS=SEAWIFS, DIR=DIR

Figure 3.4 IDL functions and procedures in *FaparTest.pro.*

FaparExamples.pro

Routines to create and display FAPAR examples.

function **ReadFile**, fname
pro **FaparLoadExampleSPOTVEG**, blue, red, nir, SZA, SAA, VZA, VAA, dir=dir
pro **FaparLoadExampleSEAWIFS**, blue, red, nir, SZA, SAA, VZA, VAA, dir=dir
pro **FaparExampleDisplay**, fr, sensor=sensor
function **FaparExamples**, dir=dir, sensor=sensor

Figure 3.5 IDL functions and procedures in *FaparExamples.pro*.

3.3 Input and Output.

The IDL code is designed to run from the command line. Input data, corresponding to *blue, red, near infrared, solar zenith, solar azimuth, view zenith*, and *view azimuth*, are in IDL fltarr() format. The type of sensor is identified with an IDL string to the input keyword *sensor*. If the illumination and viewing geometry is in degrees, then these will be converted to radians if the *degrees* keyword is set. A byte array, of identical size to the input arrays, is returned in the recommended Fapar product format.

Solar zenith, solar azimuth, view zenith, and *view azimuth* are input as keywords. This is not recommended as good practice in IDL, however, it facilitates error checking and enables error messages, sent to the user, to be more informative.

Optionally, the anisotropically normalised, atmospherically rectified, data can also be returned by setting the keywords *rectRed* and *rectNir* to a variable which has a positive value. On completion these variables will contain the appropriate data in floating point arrays.

The use of flat binary arrays as input was a constraint imposed: to avoid the need to develop multifarious ingest routines; to present a robust and clean input interface to the user; and to facilitate incorporating the code into the users' programs. It does, however, put the onus of converting sensor distribution formats into this generalised format onto the user.

3.4 Main Driver and User Interface (Fapar.pro).

The main driver and user interface is designed to be a straightforward linear set of commands which call on the appropriate procedures to fulfill the 5 main functions: error checking, anisotropic normalisation, atmospheric rectification, Fapar computation, and Fapar product output (figure 3.7).

It is expected that this routine will be modified and manipulated by the users to meet their own ends. This constraint has dictated its linear and uncomplicated structure and the basic level of the interface with the main concepts. It calls on procedures, representing the concepts, which are not expected to be, nor should be, altered by the end user.

NAME:

README

PURPOSE:

This file provides general information about the package, written using IDL, to calculate the Fraction of Absorbed Photosynthetically Active Radiation (FAPAR) from satellite sensor data, optimised for individual sensors (GLI, MERIS, SEAWIFS, SPOT VEGETATION).

APPLICABILITY:

This FAPAR estimation technique is applicable only to land surfaces under clear skies. Angular, background, and atmospheric effects are taken into account by the physical algorithm. The algorithm requires Top of Atmosphere spectral reflectances, as they are measured by the sensor, after radiometric calibration and after correction for the variable Sun−Earth distance.

AUTHORS:

Malcolm Taberner, Nadine Gobron and Frederic Mélin.

REFERENCES:

Gobron,N., Pinty,B., Verstraete,M, Widlowski,J−W, (2000). 'Advanced Vegetation Indices Optimized for Up−Coming Sensors: Design, Performance, and Applications', IEEE Transactions on Geoscience and Remote Sensing, 38, 2489−2505.

Gobron, N., F. Mélin, B. Pinty, M. M. Verstraete, J.−L. Widlowski, and G. Bucini (2001) 'A Global Vegetation Index for SeaWiFS: Design and Applications', in Remote Sensing and Climate Modeling: Synergies and Limitations, Edited by M. Beniston and M. M. Verstraete, Kluwer Academic Publishers, Dordrecht, 5−21.

COPYRIGHT:

(C) Nadine Gobron, Bernard Pinty, and Michel M. Verstraete 2001 The copyrights for these programs and files remain with the authors.

Academic users:
Are authorized to use this code for research and teaching, but must acknowledge the use of these routines explicitly and refer to the references in any publication or work. Original, complete, unmodified versions of these codes may be distributed free of charge to colleagues involved in similar activities. Recipients must also agree with and abide by the same rules. The code may not be sold, nor distributed to commercial parties, under any circumstances.

Commercial and other users:
Use of this code in commercial applications is strictly forbidden without the written approval of the authors. Even with such authorization the code may not be distributed or sold to any other commercial or business partners under any circumstances.

CONTENTS:

The Fapar package contains:

README: this file.

Fapar.pro: contains the main driver routine and is intended as the interface with the user.

FaparGeneric.pro contains the generalised procedures for input data checking, error processing, anisotropic normalisation, atmospheric correction, fapar estimation, and output production.

Figure 3.6 IDL package *README* file (continued).

FaparStartUp batch file used to compile the above. Initiated by @FaparStartUp at the IDL command level.

FaparTest.pro a set of procedures which use the test data set to check the operation of the installed routines. Initiated by .run at the IDL command level.

test.in a 169 columns x 234 rows test dataset containing 4 byte floating point, band sequential, binary, synthetic, data. The bands, in order, are: blue, red, near IR, solar zenith, solar azimuth, view zenith, view azimuth. The angles are in degrees.

test.fapar.gli
test.fapar.meris
test.fapar.seawifs
test.fapar.spotveg
are 169 columns x 234 rows, flagged, sensor specific, binary, byte, reference data, in the recommended output format (see below).

FaparExamples.pro a set of procedures to show how to produce the FAPAR product illustrated with (converted) SeaWiFS and SPOT Vegetation data.

0001_B0.DAT
0001_B2.DAT
0001_B3.DAT
0001_SAA.DAT
0001_SZA.DAT
0001_VAA.DAT
0001_VZA.DAT
example SPOT Vegetation data (modified format).

S1998219110609.L1B_443.DAT
S1998219110609.L1B_670.DAT
S1998219110609.L1B_865.DAT
S1998219110609.L1B_RAA.DAT
S1998219110609.L1B_SZA.DAT
S1998219110609.L1B_VZA.DAT
example SeaWiFS data (modified format).

IMPLEMENTATION:

Ensure that either the programs are in the IDL path, or that they are in the current directory.

At the IDL prompt, type @FaparStartUp to compile the procedures.

To test that everything works type .run FaparTest which will automatically run the program, for all sensors, with the test data set provided these data are in the current directory. If the test data are elsewhere set variable: testdir= "appropriate location" (or edit testdir in FaparTest.pro).

EXECUTION:

See the heading in Fapar.pro or call Fapar() without any arguments for a detailed description. Basically, the function Fapar can be called from the command line or from other procedures without adaptation

Figure 3.6 (continued).

INPUTS:

1. arrays containing the top of the atmosphere, radiance calibrated, solar distance compensated, bidirectional reflectance factors for the appropriate blue, red, and near−infrared bands (input separately).
2. arrays containing the solar zenith, solar azimuth, viewing zenith, and viewing azimuth angles (keywords SZA, SAA, VZA, VAA respectively) for each location in the spectral arrays. If these are in degrees, then the degrees keyword needs to be set.
3. the sensor name, by the keyword sensor, which is currently one of: "GLI", "MERIS", SEAWIFS", or "SPOTVEG".

OUTPUT:

Returns a bytarr in the recommended format. The recommended output format is scaled, flagged, unsigned byte data. Fapar is scaled from (0.0 − 1.0) to (0 − 250). Values 251 to 254 are unprocessed pixels outside the fapar confidence limits. (see detailed description for more information). Optionally returns the anisotropically, atmospherically, corrected red and near infrared bands as fltarrs.

QUESTIONS:

Nadine Gobron
Institute for Environment and Sustainability (IES)
EC Joint Research Centre, TP 440
I−21020 Ispra (VA)
Italy
Tel: [39] 03 32 78 63 38
FAX: [39] 03 32 78 90 73
E−mail: nadine.gobron@jrc.it

or

Bernard Pinty
Institute for Environment and Sustainability (IES)
EC Joint Research Centre, TP 440
I−21020 Ispra (VA)
Italy
Tel: [39] 03 32 78 61 40
FAX: [39] 03 32 78 90 73
E−mail: bernard.pinty@jrc.it

or

Michel M. Verstraete
Institute for Environment and Sustainability (IES)
EC Joint Research Centre, TP 440
I−21020 Ispra (VA)
Italy
Tel: [39] ((0)332) 78 55 07 (direct line)
Tel: [39] ((0)332) 78 98 30 (secretariat)
FAX: [39] ((0)332) 78 90 73
E−mail: michel.verstraete@jrc.it

VERSION:

1.0

MODIFICATION HISTORY:

Written by: Malcolm Taberner, 20th August, 2001

Figure 3.6 (continued).

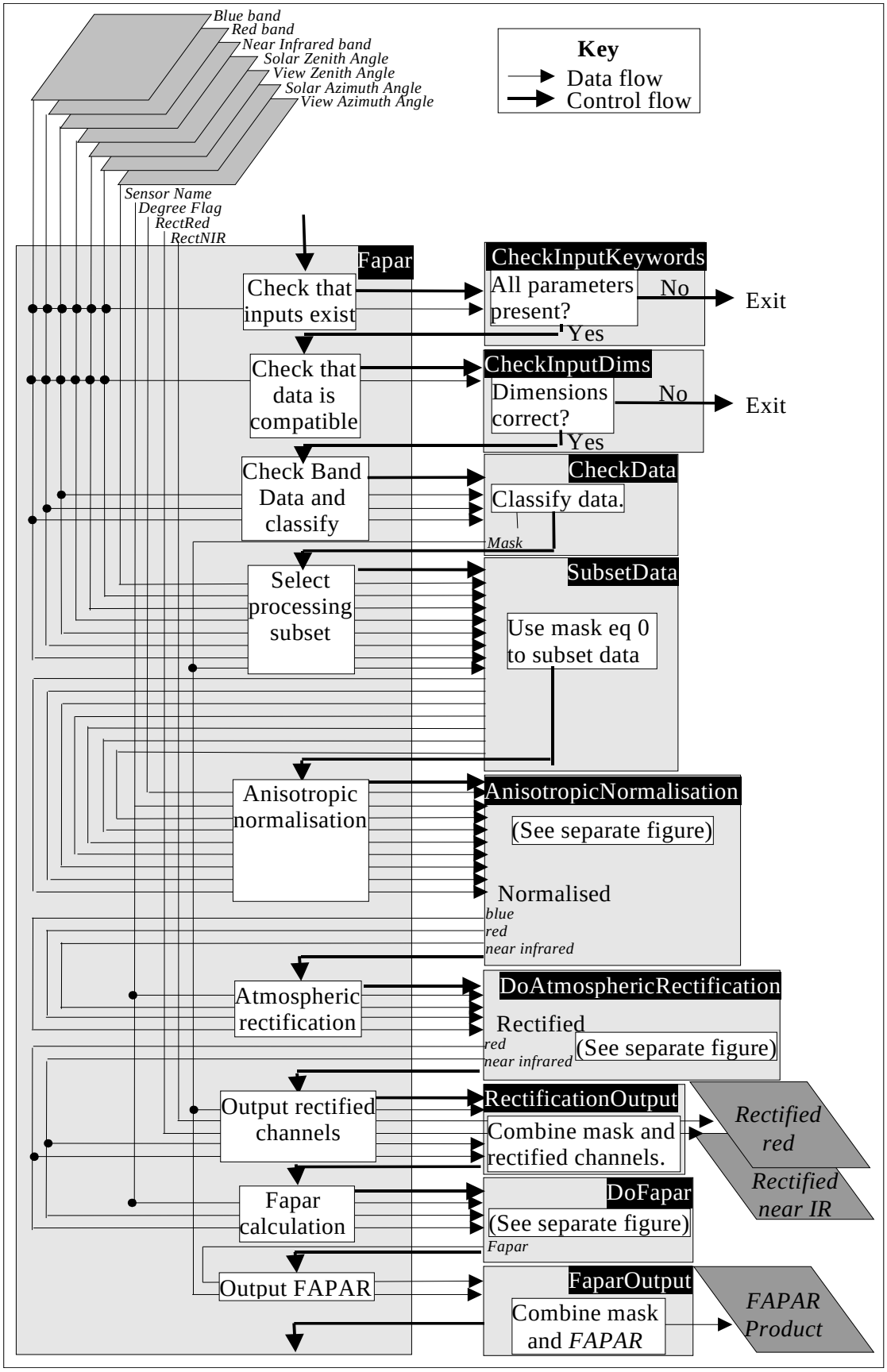

Figure 3.7 Main IDL driver and user interface routine *Fapar.pro.*

3.5 Error Checking.

Error checking consists of checking for a complete command line, checking that the dimensions of the arrays are compatible, and checking the band data itself. If, in the first two cases, an error is identified, then the program aborts with an appropriate error message. The third case is used to define the limits for Fapar calculation by preclassifying the data and creating a mask. Results which pass through the classification are used to subset the data. If no data pass the tests, then no data will be processed for Fapar, but the preclassification will be returned so that the classification results can be examined.

3.6 Anisotropic Normalisation

The procedure code for implementing anisotropic normalisation is shown in figure 3.8. The function *AnisotropicNormalisation,* called from the main driver program, is the anisotropic normalisation interface and driver procedure. If the input data is in degrees it is first converted to radians and the function *Frahman* is subsequently called to compute the normalisation coefficients.

Conceptually, *FRahman,* should only compute the coefficients and return them, however, the normalisation coefficients are computed for each pixel, for each band, so, in order to save memory, the band normalisation itself occurs in *FRahman.*

The Rahman sensor coefficients are loaded on entry to *FRahman* using the IDL *call_function* with *sensor+'AnisotropicReflectanceParameters'*, where *sensor* is the name of the sensor passed as a keyword.

The normalised band data overwrites the input band data and, unless modified, on returning to the main driver, the angle array dimensions are reset to 0 to save memory.

3.7 Atmospheric Rectification

The atmospheric rectification procedure is outlined in Figure 3.9. The interface is the function *DoAtmosphericRectification*. On entering this function the 3 band data array is split, for computational efficiency, and the band pairings used to rectify the red and near infrared channels are passed through successive calls to the main rectification function *AnisotropicRectification.*

In *AnisotropicRectification*, the sensor rectification coefficients, both numerator and denominator coefficients if a ratio is being applied, and the name of the sensor specific polynomial function are loaded with *sensor+'AtmosphericRectificationCoefficients'*. The selected polynomial is then calculated using *call_function* with the loaded name and sensor coefficients. The limits of the rectified bands are set to $0 \leq Brect \leq 1.0$.

The resulting rectified bands are combined in *DoAtmosphericRectification*before control is passed back to the main routine where, unless the code is modified, the original band data dimensions are set to zero to save memory.

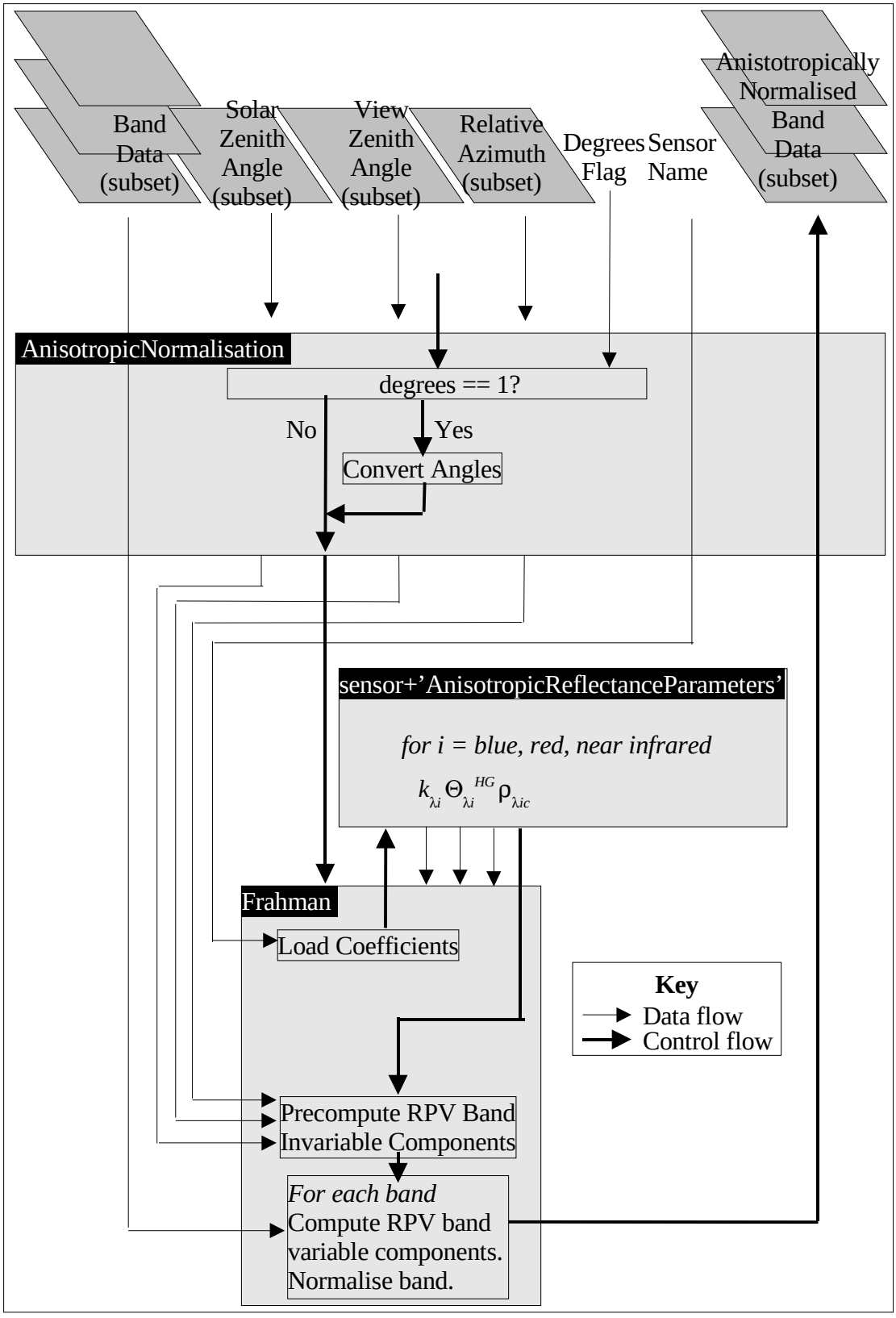

Figure 3.8 Anisotropic Normalisation (IDL).

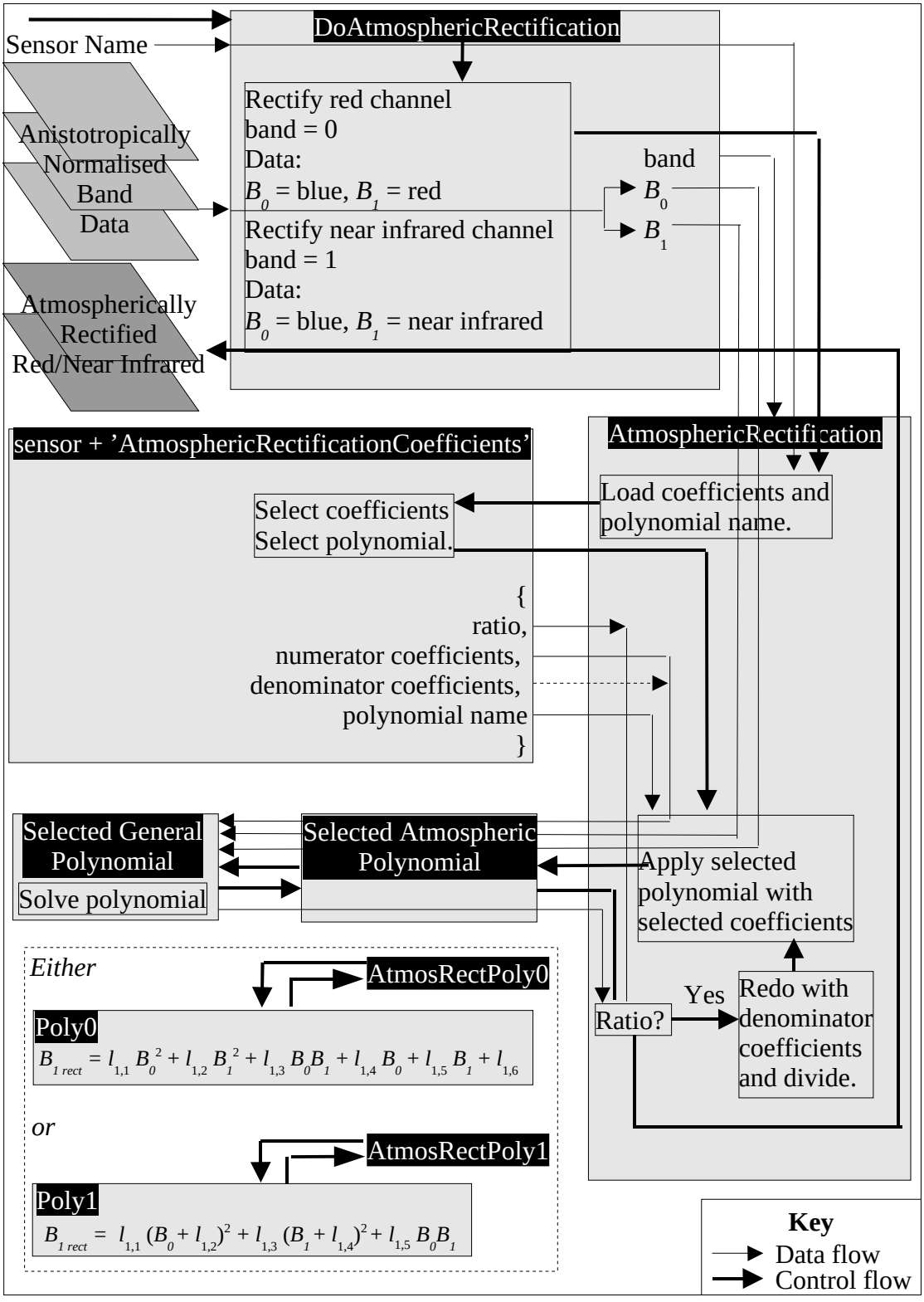

Figure 3.9 Atmospheric Rectification (IDL).

3.8 Fapar Calculation.

The Fapar calculation module is shown in figure 3.10. *DoFapar* is the interface with the Fapar calculation module. The sensor specific coefficients and polynomial name are loaded using *call_function* with *sensor+FaparCoefficients* and, as in the previous module, the selected polynomial is applied using *call_function* with the retrieved name and sensor coefficients. The Fapar results are returned to the main routine.

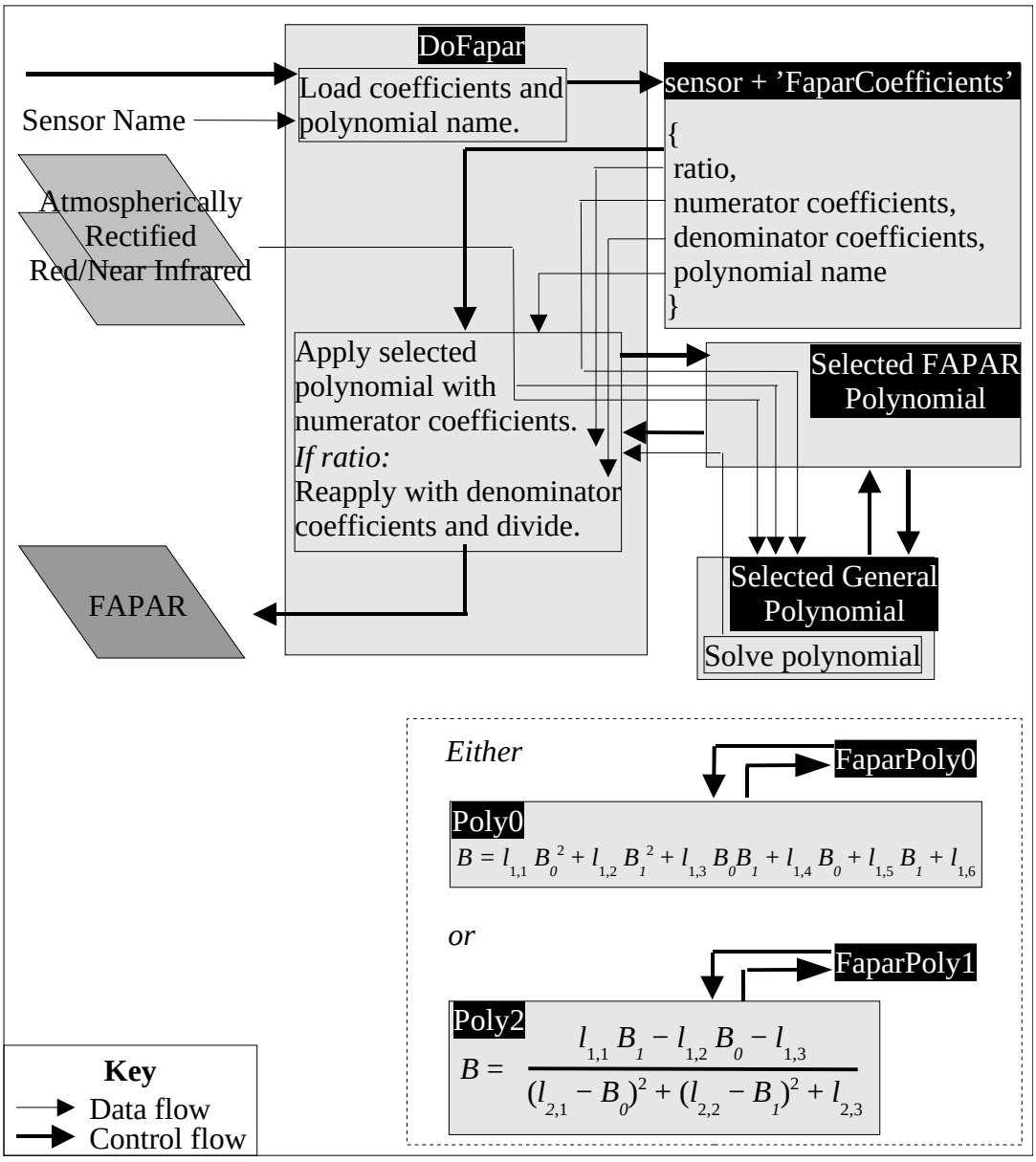

Figure 3.10 Fapar Calculation (IDL).

3.9 Product Generation

The *mask* created by the classification, along with the Fapar results (computed from the band data subset) are combined in *FaparOutput* into the recommended format, and the resulting array returned.

If the option to return the rectified channels has been set, then the mask and rectified data are combined in *RectificationOutput* and set to the variable established by the rectification keywords.

3.10 Code modification and updating.

The code has been structured to facilitate modification at three levels:

1. At the user level, the code is organised so that the user only need (should) modify the input/output sections etc. of the main driver program *Fapar.pro* and their interactions with the concepts.
2. At the new sensor level. New sensors, using existing generic code, can be added by simply copying, renaming, and modifying the 3 routines necessary for each sensor in *FaparSensor.pro* and adding the new sensor name in the name array in *Fapar.pro.*
3. At the algorithm improvement level, by adding new generic routines in *FaparGeneric.pro* which are buffered from the other parts of the program by the appropriate interfaces and control is simplified by the polynomial implementation method.

Chapter 4

Generalised C++ Code

4.0 Introduction (Design Issues)

The design issues for the C++ development are similar to those for the IDL code, however, the object oriented approach permits far greater (apparent) generalisation with much greater flexibility for incorporating sensor specific code. This power is at the expense of code length which, as is common for C++ programs, is much greater than using other languages. This is due, in part, to the fact that, in order to function properly, a complete system has to be developed and coded in C++. Code written in C, on the other hand, can be written very concisely, and very specifically targeted, but it is harder to maintain and modify and lacks the flexibility and ease of code management and reusability. As with the IDL code, care has been taken to ensure the code is as memory efficient, and computationally as fast, as possible, undergoing several optimisations.

Object oriented programming permits programs to be developed on the basis of concepts as defined by classes. In this implementation the core concept (class) is that of a sensor. The sensor class has a structure and contains information that enables the sensor data to carry out the basic functions of anisotropic normalisation, atmospheric rectification, and derived products. In this case, of course, the only product to derive is the FAPAR product.

The user interface implemented is as abstract and generalised as this suggests, an instance of a sensor is created, and then the sensor is required (requested) to anisotropically normalise, atmospherically correct, and calculate the Fapar product, for the sensor data.

Operationally it is implemented in two stages, the first stage establishes the sensor system with an instance of a specific sensor (GLI, MERIS, etc.), and the second phase uses the sensor in the generalised fashion.

4.1 The C++ Package

The C++ package is provided in a gzipped tarred file, *Fapar−1.0.tar.gz* (58.6 kb.) which, when unpacked, has the structure and manifest shown in figure 4.0. The package uses the *GNU* package handling facilities. In the main directory are the basic configuration details and messages. The *src* directory contains all the C++ source code, and the test program and reference files are in the *test* subdirectory. Both subdirectories contain their local *makefile*s.

At the top level, the *COPYING* file usually contains the 'GNU General Public License' but this has been substituted with a version modified from copyright notices included with other STARS code (figure 4.1). The *README* file is shown in figure 4.2. The *ChangeLog* is provided to record modifications to the software, and is currently empty. The NEWS file is used to publish updates of features in a user friendly manner. The *AUTHORS* file contains the name(s) of the original author(s)

and to record any subsequent authors. The *INSTALL* file contains information about the installation procedure.

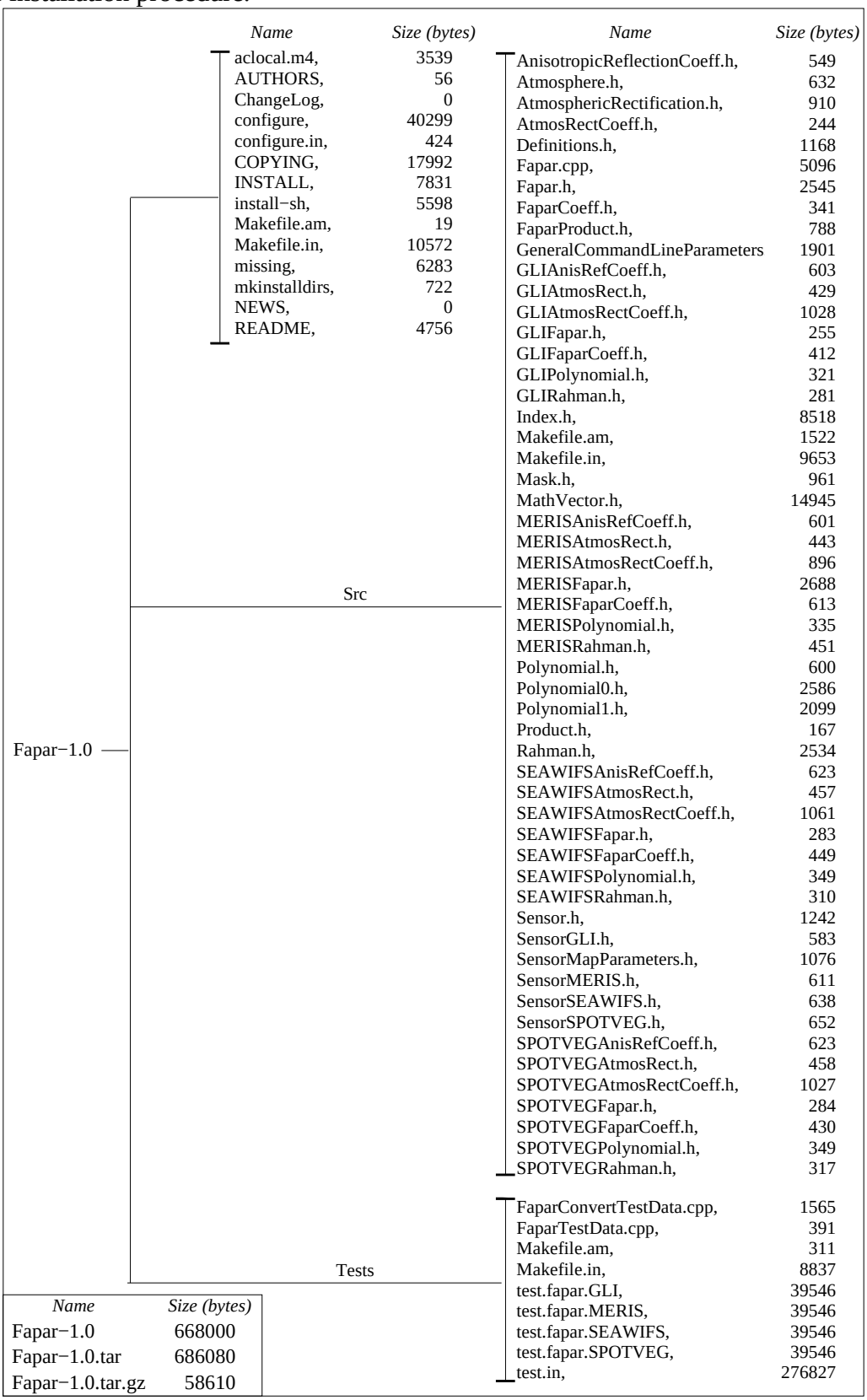

Figure 4.0 The C++ FAPAR package.

COPYRIGHT:
(C) Nadine Gobron, Bernard Pinty, and Michel M. Verstraete
The copyrights for these programs and files remain with the authors.

Academic users:
Are authorized to use this code for research and teaching, but must acknowledge the use of these routines explicitly and refer to the references in any publication or work. Original, complete, unmodified versions of these codes may be distributed free of charge to colleagues involved in similar activities. Recipients must also agree with and abide by the same rules. The code may not be sold, nor distributed to commercial parties, under any circumstances.

Commercial and other users:
Use of this code in commercial applications is strictly forbidden without the written approval of the authors. Even with such authorization the code may not be distributed or sold to any other commercial or business partners under any circumstances.

Figure 4.1 The copyright notice provided with the C++ package.

The other files in the top level directory relate to the compilation and installation procedures.

The *src* subdirectory contains *.h* files containing one object class per file, and the main driver routine in *Fapar.cpp*. These will be described in more detail later.

The *test* subdirectory contains the reference test data, *test.fapar.sensor*, the test input data, *test.in*, and the programs to convert the test data to local machine format, *FaparConvertTestData.cpp*, and run the tests, *FaparTestData.cpp*.

4.2 Compilation

As the package uses the GNU package handling mechanism, the reader is directed to the Fapar−1.0 directory files for specific compilation information, and the website: http://www.gnu.org for details of the package management software.

Briefly, the C++ package is set up with a set of standard reference files and layout which interface with the GNU mechanism. Code, variables, libraries, and other requirements to compile the code are established when the package is being created. After the package is downloaded and unpacked, the user, usually, only needs to cd to the top level directory, run the file *./configure*, run *make*, and run *make check* if required to make the tests, and run *make install* to install the directory into the default, or requested, locations.

NAME:

README

PURPOSE:

This file provides general information about the package, written using C++, to calculate the Fraction of Absorbed Photosynthetically Active Radiation (FAPAR) from satellite sensor data, optimised for individual sensors (GLI, MERIS, SEAWIFS, SPOT VEGETATION).

APPLICABILITY:

This FAPAR estimation technique is applicable only to land surfaces under clear skies. Angular, background, and atmospheric effects are taken into account by the physical algorithm. The algorithm requires Top of Atmosphere spectral reflectances, as they are measured by the sensor, after radiometric calibration and after correction for the variable Sun−Earth distance.

AUTHORS:

Malcolm Taberner, Nadine Gobron and Frederic Mélin.

REFERENCES:

Gobron,N., Pinty,B., Verstraete,M, Widlowski,J−W, (2000). 'Advanced Vegetation Indices Optimized for Up−Coming Sensors: Design, Performance, and Applications', IEEE Transactions on Geoscience and Remote Sensing, 38, 2489−2505.

Gobron, N., F. Mélin, B. Pinty, M. M. Verstraete, J.−L. Widlowski, and G. Bucini (2001) 'A Global Vegetation Index for SeaWiFS: Design and Applications', in Remote Sensing and Climate Modeling: Synergies and Limitations, Edited by M. Beniston and M. M. Verstraete, Kluwer Academic Publishers, Dordrecht, 5−21.

COPYRIGHT:

(C) Nadine Gobron, Bernard Pinty, and Michel M. Verstraete 2001 The copyrights for these programs and files remain with the authors.

Academic users:
Are authorized to use this code for research and teaching, but must acknowledge the use of these routines explicitly and refer to the references in any publication or work. Original, complete, unmodified versions of these codes may be distributed free of charge to colleagues involved in similar activities. Recipients must also agree with and abide by the same rules. The code may not be sold, nor distributed to commercial parties, under any circumstances.

Commercial and other users:
Use of this code in commercial applications is strictly forbidden without the written approval of the authors. Even with such authorization the code may not be distributed or sold to any other commercial or business partners under any circumstances.

CONTENTS:

The Fapar package contains:

In the top level directory (Fapar−1.0) the configuration files.
Two subdirectories: src and test.

The src directory contains the C++ program source code files
and local makefiles.

The test subdirectory contains a test data set and routines
to test the implementation.

Figure 4.2 The C++ package *README* (continued).

The test data files are:

test.in a 169 columns x 234 rows test dataset containing 4 byte floating point, band sequential, binary, synthetic, data. The bands, in order, are: blue, red, near IR, solar zenith, solar azimuth, view zenith, view azimuth. The angles are in degrees.

test.fapar.gli
test.fapar.meris
test.fapar.seawifs
test.fapar.spotveg
are 169 columns x 234 rows, flagged, sensor specific, binary, byte, reference data, in the recommended output format (see below).

IMPLEMENTATION:

cd to the main Fapar−1.0 directory and follow the instructions in the *INSTALL* file.

make check ensures that everything works.
It will automatically run the program, for all sensors with the test data set provided in the *test* subdirectory.

EXECUTION:

See the heading in Fapar.cpp or call Fapar() without any arguments, or with the help flag, for a usage message.

USAGE:

FAPAR blue= red= nearIR= solarAzimuth= solarZenith= viewAzimuth= viewZenith= s= of= [rf=, −−degrees, −−byteswap, −−verbose] [−−help]

blue=, red=, nearIR=
(required) are binary files of floating point data corresponding to top of the atmosphere (TOA), bidirectional, spectral reflectance factor ($0.0 < r < 1.0$). These data should be previously calibrated to radiance and corrected for the variable solar distance.
solarAzimuth=, solarZenith=, viewAzimuth=, viewZenith= (required) are binary files of floating point data corresponding to the solar and satellite view angles (azimuth and zenith), for each pixel location. They are assumed to be in radians, unless the flag −−degrees is used.
s= (required) corresponds to the sensor from which the data were acquired.
of= (required) full path and name of the output file. This format is the "recommended" format for the Fapar product. The file consists of unsigned, single byte, Fapar data scaled from between 0 and 1.0 to between 0 and 250. Values of 251−254 correspond to spectral values outside the confidence limits. Semantically, these loosely correspond:
251 to bad data;
252 to clouds, snow, ice;
253 to water bodies, deep shadow; and
254 to bright (unvegetated) surfaces.
rf= (optional) full path and name stem for saving anisotropically, and atmospherically, corrected red (suffix .red) and near infrared (suffix .nir) bands (binary floating point format).
−−degrees means that input angles are in degrees.
−−byteswap swap the byte ordering of the input files when necessary.
−−help (or no parameters) print this message (no processing).
−−verbose prints out the settings.

Figure 4.2 (continued).

QUESTIONS:

Nadine Gobron
Institute for Environment and Sustainability (IES)
EC Joint Research Centre, TP 440
I−21020 Ispra (VA)
Italy
Tel: [39] 03 32 78 63 38
FAX: [39] 03 32 78 90 73
E−mail: nadine.gobron@jrc.it

or

Bernard Pinty
Institute for Environment and Sustainability (IES)
EC Joint Research Centre, TP 440
I−21020 Ispra (VA)
Italy
Tel: [39] 03 32 78 61 40
FAX: [39] 03 32 78 90 73
E−mail: bernard.pinty@jrc.it

or

Michel M. Verstraete
Institute for Environment and Sustainability (IES)
EC Joint Research Centre, TP 440
I−21020 Ispra (VA)
Italy
Tel: [39] ((0)332) 78 55 07 (direct line)
Tel: [39] ((0)332) 78 98 30 (secretariat)
FAX: [39] ((0)332) 78 90 73
E−mail: michel.verstraete@jrc.it

VERSION:

1.0

MODIFICATION HISTORY:

Written by: Malcolm Taberner, 20th August, 2001

Figure 4.2 (continued)

/configure checks the local machine, software locations, and software capability. It selects appropriate machine local software, options and establishes pathnames, and uses the information to create local makefiles in the package hierarchy.

The package requires a C++ compiler capable of handling templates and with a reasonable version of the standard library. The code was developed using the GNU C++ compiler (*g++*, version 2.95.2). This compiler, which is well maintained and up−to−date, works under the GNU Public License and is freely available.

4.3 Input and Output.

The C++ code is designed to run from the command line. Input data, corresponding to *blue, red, near infrared, solar zenith, solar azimuth, view zenith,* and *view azimuth,* are in machine local floating point format. These are flat binary files without headers − the dimensions of the files do not need to be defined as the complete file will be

processed on a linear basis. Their type does not need to be specified as floating point is expected. The files must, of course, be of identical lengths.

The type of sensor, identified by the sensor acronym (*GLI, MERIS, SEAWIFS, SPOTVEG)* is identified using the sensor keyword (see figure 4.2 for actual semantics). If the illumination and viewing geometry is in degrees, then these will be converted to radians if the *degrees* keyword is set.

A byte file, with the same number of elements as the input arrays, is returned in the recommended Fapar product format.

Optionally, the anisotropically normalised, atmospherically rectified, data can also be saved by setting the appropriate keyword to the pathname and name for the rectified files. The data will be saved in floating point format, with the preclassification flag values applied. The red and near infrared bands will have the suffixes *.red* and *.nir* respectively.

As with the IDL code, the use of flat binary arrays as input was a constraint imposed to avoid the need to develop multifarious ingest routines; to present a robust and clean input interface to the user; and to facilitate incorporating the code into the users' program. As stated before, it does put the onus of converting sensor distribution formats into the generalised format onto the user.

4.4 The main driver and user interface module (*Fapar.cpp*)

The layout of the program is constrained, to a certain extent, by the desire to have all the code that users are likely to want (be "allowed") to modify at this level. This has made the code in this module more complicated and less conceptually clean than it might otherwise have been.

The flow diagram for this module is given in figure 4.3. On entry into the program, 2 classes are instantiated: *sensorMap* and *cLine.* The first is of class *SensorMapParameters* the other is of class *GeneralCommandLineParameters*.

The class *GeneralCommandLineParameters* is a derived class of the standard library <*map*> with a <*string, string*> pairing (figure 4.4). This class parses the command line. It maps the values associated with the command line parameters to the command line parameters themselves. The association definitions between parameters and keywords are in the file *Definitions.h.* The class can find and return the value associated with a parameter, or simply test whether one exists. If key parameters don't exist, then an error message is printed and the program aborts.

The other class, *SensorMapParameters*, is also a derived class of the standard library <*map*> but this time with a <*string, fP*> pairing (figure 4.5), where the string is the sensor name, and *fP* is defined by: *typedef Sensor* (*fP)()* (i.e. *fP* is a pointer to a function creator object). Given the sensor name, it returns the appropriate function object pointer which can be used to create an instantiation of the specific sensor. If the sensor name cannot be found, because, for instance, it was misspelt on the command line, an error message is printed and the program aborted.

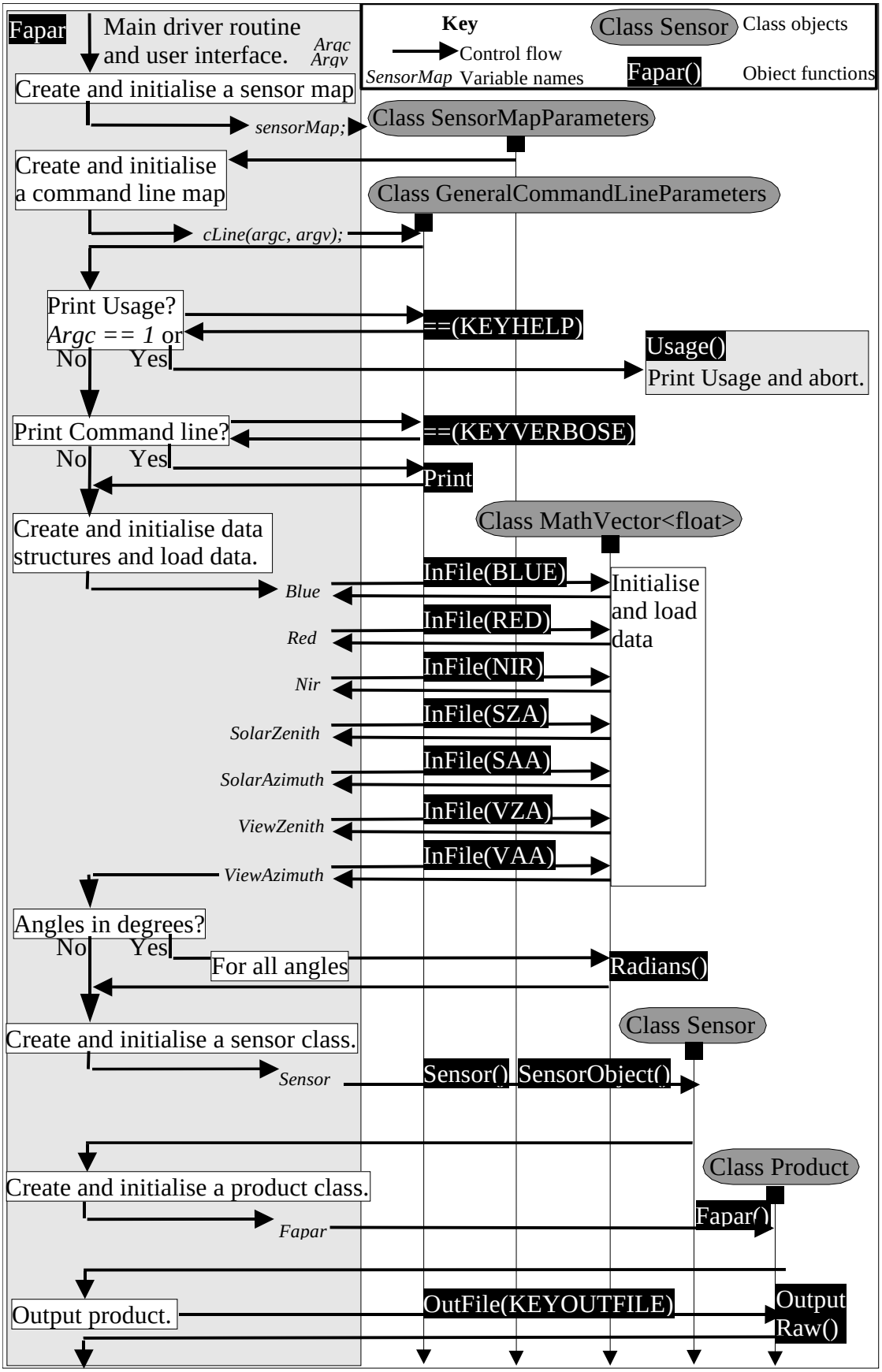

Figure 4.3 Control flow in the main driver and user interface, *Fapar.cpp*.

```
class GeneralCommandLineParameters: public map< string, string >{
//Purpose:  A class to parse the command line parameters.
public:
        GeneralCommandLineParameters();
        GeneralCommandLineParameters(int argc, char *argv[]);
        ~GeneralCommandLineParameters();
        void Print();

        //Find and return the sensor type.
        string Sensor();

        // Find and return the input file name associated with
        // the key name.
        string inFile(string key);

        // Find and return the output file name associated with
        // the key name.
        string outFile(string key);

        //Determine whether a parameter flag has been set.
        bool operator==(const string key);
```

Figure 4.4 The *GeneralCommandLineParameters* class.

```
typedef Sensor* (*fP)();
class SensorMapParameters: public map< string, fP >{
//Purpose:  Map the sensor type to the appropriate constructor function objects.
Public:
        SensorMapParameters();
        ~SensorMapParameters();
        void Print();

        //Match the sensor type to the sensor constructor function
        //object and return it.
        fP SensorObject(const string sensor);
};
```

Figure 4.5 The *SensorMapParameters* class.

The program continues on to check whether to display the usage message (and abort) and whether to print the program's interpretation of the command line.

The data is then loaded into band and angle *float* instantiations of the **template** data structure class *Math_Vector<T>*. The *Math_Vector<T>* class (figure 4.6) is one of two main data structures used in the program, the other is Index<size_t>.

```
template<class T>
class MathVector: public vector<T> {
//Purpose:  A vector data structure capable of subsetting, indexing,
                    and maths operations.
  Public:
  /***********  Construction *************************/
  MathVector() : vector<T>();
  MathVector(const size_t n, const T& value) : vector<T>(n, value);
  MathVector(const int n, const T& value) : vector<T>(n, value);
  MathVector(const long n, const T& value) : vector<T>(n, value);
  explicit MathVector(const size_t n) : vector<T>(n);
  MathVector(T* b, T* e) : vector<T>(b, e);

  template<class T2>
  MathVector(size_t n, T2 *z): vector<T>(n);
  MathVector(string filename);

  ~MathVector();

  /***********  Memory Manipulation *********************/
  template<class T2>
  //Allocate space.
  void reserve(T2 val);

  /***********  Type Conversion ************************/
  // It is the responsibility of the user to establish
  // whether the conversion makes sense!
  template <class T2>
  MathVector(const vector<T2>& z): vector<T>(z.size());

  //Swap the order of adjacent bytes in a pair.
  void swapBytes2();
  //Swap the order of adjacent bytes in a pair and adjacent byte pairs.
  void swapBytes4();
  //Automatically select 2 or 4 byte swapping.
  void swapBytes();

  /***********  Copy, generation, replacement.  **************/
  MathVector<T> operator=(const T value);
  template<class T2>
  const MathVector<T> &operator=(const vector<T2> &right)const;
  template<class T2>
  MathVector<T> operator=(const vector<T2> &right);

  // Generate a Series.
  MathVector<T> &indgen(size_t value);

  //Insert a value at locations defined by index.
                MathVector<T> &insert(const Index &ix, const T value);
                                                          continued ...
```

Figure 4.6 The Math_Vector<T> class.

```
class MathVector: public vector<T> {            continued

/*********** Logical operators. *************/
Index operator==(const T &value)const;
Index operator!=(const T &value);
Index operator>(const T &value)const;
Index operator<(const MathVector<T> &vec)const;
Index operator>(const MathVector<T> &vec)const;
Index operator<(const T &value)const;
Index operator<=(const T &value)const;
Index operator>=(const T &value)const;
Index operator>=(const MathVector<T> &vec)const;

/*********** Indexing operators. *************/
T operator[](const size_t ix) const;
MathVector<T> operator[](const Index &ix) const;

/*********** Basic Maths Operations ******************/
//Addition
template <class T2>
const MathVector<T> &operator+=(const T2 value);
template <class T2>
const MathVector<T> &operator+=(const class MathVector<T2> vec);
template <class T2>
MathVector<T> operator+(const T2 value) const;
template <class T2>
MathVector<T> operator+ (const class MathVector<T2> vec1) const ;

//Subtraction
template <class T2>
MathVector<T>& operator−=(const T2 value);
template <class T2>
MathVector<T>& operator−=(const class MathVector<T2> vec);
template <class T2>
MathVector<T> operator−(const  T2 value)const;
template <class T2>
MathVector<T> operator−(const class MathVector<T2> vec1)const;

//Multiplication
template <class T2>
MathVector<T> &operator*=(const class MathVector<T2> &vec);
template <class T2>
const MathVector<T> &operator*=(const T2 value);
template <class T2>
MathVector<T> operator*(const T2 value)const ;

                                                continued ...
```

Figure 4.6 (continued)

```
class MathVector: public vector<T> {            continued

//Division
template <class T2>
MathVector<T>& operator/=(T2 value);
template <class T2>
MathVector<T>& operator/=(const class MathVector<T2> &vec);
template <class T2>
MathVector<T> operator/(const T2 value)const;
template <class T2>
MathVector<T> operator/ (const class MathVector<T2>& vec1)const;
template <class T2>
MathVector<T> operator%(const T2 value)const;
template <class T2>
MathVector<T>& operator%=(T2 value);
template <class T2>
MathVector<T>& operator%=(const class MathVector<T2> &vec);
template <class T2>
MathVector<T> operator% (const class MathVector<T2>& vec1)const;

/****************** Limits. **********************/
const MathVector<T> &gt(const T value);
const MathVector<T> &gt(const T value, const T replacement);
const MathVector<T> &lt(const T value);
const MathVector<T> &lt(const T value, const T replacement);

/****************** Maths conversion. ************/
void abs();
void Radians();

/****************** Input/Output .****************/
void  inputRaw(string fileName, size_t numElements);
void  inputRaw(string fileName);
void  outputRaw(const string fileName)const;

void Print();
};
```

Figure 4.6 (continued).

Math_Vector<T> is a derived class from the standard template library base class *vector<T>*. *vector<T>* is an efficient, fast, implementation of a dynamic, one dimensional array, of type *<T>*. *Math_Vector* is designed to support array based mathematical, logical and set operations, assignment, copying, and advanced subscripting with *Index<size_t>* vectors. It also supports basic, binary, input/output (dumps), unconverted, to and from type *<T>* files.

The *Index<size_t>* class (figure 4.7) is also a derived class from the standard template library base class *vector<T>*. It is an array of zero based offsets used to index and subset elements of *Math_Vector<T>* or other *Index<size_t>* arrays. It allows basic

assignment, copying, subscripting, arithmetic, logical, and set operations on subscripts arrays.

```
class Index: public vector<size_t> {

//Purpose:  A vector data structure for indexing vectors.

    Public:
    /********************* Construction *********************/
    Index():vector<size_t>();
    Index(size_t sz):vector<size_t>(sz);
    //Copy/assignment
    Index(const Index &ix):vector<size_t>(ix);
    const Index operator=(const Index &right);
    //Subset
    Index operator[](const Index &ix) const;
    const size_t operator[](const size_t value) const;
    size_t &operator[](const size_t value);
    //Insertion
    Index &insert(const Index &ix, const size_t value);
    //Generation
    const Index &indGen(size_t num);

    /********************** Logical **********************/
    //Set operators
    const    Index operator||(const Index &ix)const;
    const Index operator&&(const Index &ix)const;
    const Index operator!=(const Index &ix);
    //Unary operators
    template<class T>
    Index operator<(const T &value);
    template<class T>
    Index operator>(const T &value);
    template<class T>
    Index operator<=(const T &value);
    template<class T>
    Index operator>=(const T &value);
    template<class T>
    Index operator==(const T &value);
    template<class T>
    Index operator!=(const T &value);
                                                    continued ...
```

Figure 4.7 Definition of the *Index<size_t>* class (continued).

```
class Index: public vector<size_t> {            continued

    /******************** Arithmetic ********************/
    template <class T2>
    const Index &operator+=(const T2 value) ;
    const Index &operator+=(const class Index vec);
    template <class T2>
    Index operator+(const T2 value) const;
    template <class T2>
    Index &operator−=(const T2 value);
    template <class T2>
    Index operator−(const  T2 value)const;
    template <class T2>
    Index &operator*=(T2 value);
    template <class T2>
    Index operator*(const T2 value)const;
    template <class T2>
    Index &operator/=(T2 value);
    template <class T2>
    Index operator/(const T2 value)const;
    template <class T2>
    Index &operator%=(const T2 value);
    template <class T2>
    Index operator%(const T2 value)const;
    long product();

    /******************** Location/Seek ********************/
    size_t min()const;
    size_t max()const;

    /************* Input/Output *****************/
    void  outputRaw(string fileName);
    void Print();
    };
```

Figure 4.7 (continued).

The band and angle instantiations are restricted to type *float* but user modification of data type input is simply carried out by changing the template format *T* from *MathVector<float>* to that desired. The data is loaded from the file whose name is retrieved from *cLine* and passed to the *MathVector<float>* on instantiation. The size of *MathVector<float>* is dynamically altered to accommodate all elements from the file.

If the parameter flag denoting that angles are in degrees, then the angles are converted to radians. *Radians()*, which is one of the mathematical operations in *MathVector<T>*, does this conversion.

The processes of anisotropic normalisation, atmospheric rectification, and FAPAR calculation all take place in the context of the *sensor* class which knows how to

produce a *Product* class which is of type *Fapar*. The *Product* class knows how to output its contents to file.

4.5 Error Checking

Unlike the IDL code, error checking, *per se*, is not a separate operation in the C++ code but comprehensive error control is built into the class operations. Parameter errors, for example, are caught by the *GeneralCommandLineParameters* and *SensorMapParameters* classes when a mismatch occurs. If the data file doesn't exist, or does not exist in the place specified, then this is caught when attempting to load the data by the *MathVector<T>* class. If the dimensions of the data do not agree, then this is caught on instantiation of the *Sensor* class.

4.6 The *Sensor* Class

The *Sensor* class is the main concept, and organisational unit, for uniting the operations possible on sensor data. Conceptually it is desirable to handle a sensor in a very general way. It is desirable that operations and procedures can be applied to a sensor, irrespective of the actual type of sensor. Given an instance of a class *Sensor*, for example, anisotropic normalisation, atmospheric rectification, etc. should be possible without knowing that it is a MERIS or SeaWiFS sensor that is being dealt with. This keeps the user interface simple and uncomplicated and is the core idea defining the program structure. This idea is implemented, operationally, by designing the program to work in two stages: an initialisation stage, and an operational stage. In the initialisation stage, an instantiation of a specific sensor is created but, using inheritance and virtual functions, only the general interface is presented to the user (and, indeed, a general interface is presented to the other concepts in the system maximising the modular aspect).

At the general level, the *Sensor* class itself (figure 4.8) knows how to correct for anisotropic effects, through the *Rahman* class member, for atmospheric effects, through the *Atmosphere* class member, and to estimate FAPAR, through the *Fapar* class member. It also knows how to produce a *Fapar Product* by manipulating these classes, through the *Fapar* function. This effectively defines the interface with the user.

Conceptually, the band and angle *MathVector<float>* data classes should also be members (in fact more specialised classes), however, these were abstracted into the main user interface to facilitate modification by the user. To avoid excessive use of memory and to avoid time loss these are not copied into the sensor, but passed, as references, when *Sensor* operations require them.

A specific sensor is instantiated by creating a sensor specific derived class of the *Sensor* class, such as the *SensorSEAWIFS* example in the figure. The instantiation of *SensorSEAWIFS* is carried out in the initialisation stage and is created from the main driver routine using the function creator object retrieved from the *SensorMapParameters* class. Once created, however, the instantiation is held in the main driver routine as the more general *Sensor* base class. This means that only the general operations available at the sensor level can be accessed by the user interface.

Inheritance, and, in particular, virtual inheritance and multiple inheritance, however, means that the general functions at the level of the base class can (and will be) overridden by derived functions so that, although the call to Sensor uses general syntax, if appropriate, the actual call is to the hidden, more specialised, *SensorSEAWIFS*

```
class Sensor{
  protected:
  class Rahman *anisotropyCoeffs;
  class Atmosphere atmos;
  class Fapar *fapar;
  class Mask *mask;

  public:
  Sensor(){}
  virtual ~Sensor();

  FaparProduct *Fapar(MathVector<float> &blue, MathVector<float> &red,
            MathVector<float> &nearIR, const MathVector<float> &sZ, const
            MathVector<float> &sA, const MathVector<float> &vZ, const
            MathVector<float> &vA, string rectOut="");
  `
```

```
class SensorSEAWIFS : public Sensor {
  public:
  SensorSEAWIFS();
  ~SensorSEAWIFS();
  static Sensor* new_SensorSEAWIFS();
  };
```

Figure 4.8 Definition of the *Sensor* class.

4.7 *Sensor* Initialisation

The following discussion uses SEAWIFS as an example sensor, but the issues discussed are similar for the other sensors, although the detailed implementation might vary slightly. A reference to the *Sensor* is kept in the main driver routine. The *Sensor*, itself, is a base class of *SensorSEAWIFS*. The *SensorSEAWIFS* is created from the main driver routine using the function creator object retrieved from the *SensorMapParameters* class. This creates a new instantiation of *SensorSEAWIFS* and is assigned to the pointer *sensor*.

During initialisation of *SensorSEAWIFS*, the *Sensor* member instantiations of the *Rahman*, *atmosphere*, and *Fapar* are created by the sensor (figure 4.9). As these instantiations are created by the *SensorSEAWIFS* initialisation phase, they are created as instantiations of the classes *SEAWIFSRahman*, *SEAWIFSAtmosRect*, and *SEAWIFSFapar*. *Rahman*, *atmosphere*, and *Fapar* are the interfaces to, obviously, the anisotropic normalisation, atmospheric rectification, and FAPAR estimation components.

Creating instantiations of *Rahman*, *atmosphere*, and *Fapar* in turn, creates member instantiations of *AnisotropicReflectionCoeff* in *Rahman*, *Polynomial* in *AtmosRect* and *FaparCoeff* in *Fapar*. As these are created from *SEAWIFSRahman*, *SEAWIFSAtmosRect*, and *SEAWIFSFapar*, however, they are created as *SEAWIFSAnisRefCoeff*, *SEAWIFSPolynomial*, and *SEAWIFSFapar*. *SEAWIFSPolynomial*, in turn, creates an instantiation of *SEAWIFSAtmosRectCoeff* held as an *AtmosRectCoeff* class in its base class.

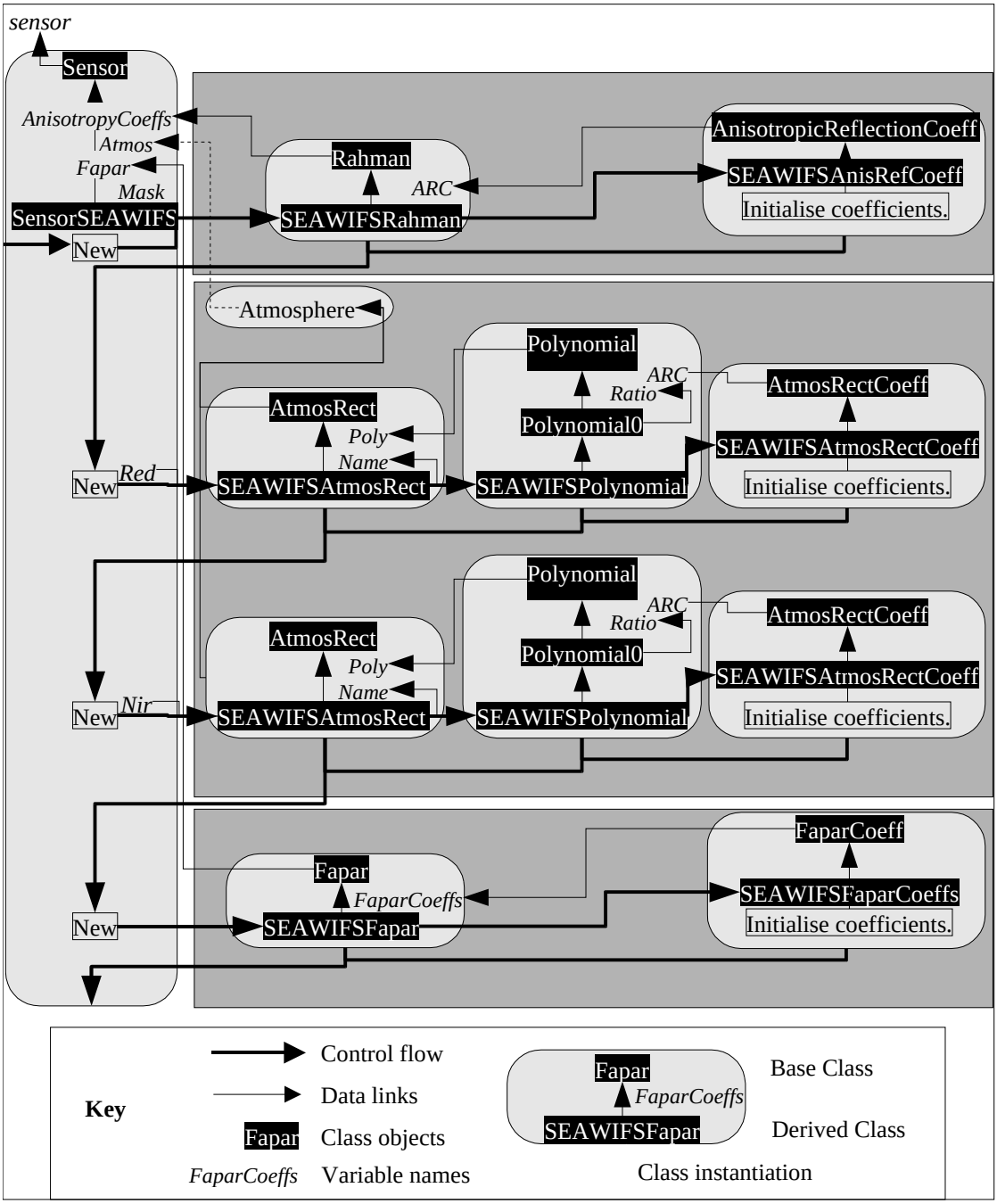

Figure 4.9 Instantiation of the *SensorSEAWIFS* class.

After instantiation, *AnisotropicReflectionCoeff*, *Polynomial*, *SEAWIFSAtmosRectCoeff*, and *FaparCoeff* contain all the information (sensor coefficients and polynomial specifications) required to carry out anisotropic normalisation, atmospheric rectification, and FAPAR calculation using the generalised infrastructure. The outline definition of these classes is given in the appendix.

4.8 *Sensor* operations − calculating FAPAR (*sensor.Fapar(...)*)

The *Sensor* class provides a general framework for sensor based computations and product generations. In this implementation *sensor* only knows the requirements to calculate FAPAR. The instance of *sensor* is held by the main driver and a *Product* class, which is a base class of class *FaparProduct*, is created, from the main driver, using the generalised control and function structure using the publicly accessible *sensor.Fapar(...)* member function. The outline of this process is illustrated in figure 4.10.

On entry to *sensor.Fapar()*, an instance of the *mask* class is created which carries out, and contains the results from, the preclassification testing. This information is subsequently used to subset the band and angle data. The anisotropic normalisation coefficients are computed from the angle data using *AnisotropyCoeffs.Rahman()* and are used to normalise the band data. The normalised bands are atmospherically rectified using the appropriate *blue, red* or *blue, nir* band pairings using *AtmosphericRectification.Rectify()*. The rectified bands are then used by *Fapar.Calculate()* to estimate FAPAR.

A *FaparProduct* class is instantiated which creates the recommended output format from the *Fapar* and *Mask* information. It is returned to the main driver, though, as the more general, base class, *Product* (figure 4.11). *FaparProduct* has multiple inheritance from both the *Product* and *MathVector<T>* classes. Only the base class *Product* is seen at the main driver level, however, through virtual function inheritance of *OutputRaw()* from *FaparProduct*, it knows how to write the product data to file. *FaparProduct*, in turn, knows how to output the data as it inherits the *MathVector<T>.OutputRaw()* function.

4.8 Code Modification and Updating.

Modification by the user is at the main driver and interface level in *Fapar.cpp*. The program is structured so that, at this level, the interface to the users programs can be easily implemented by giving access to the parameter parsing, initialisation, data loading and formatting functions. These aspects are straightforward and readily modified. The interface to the sensor class and product generation is facile. The built in error identification aids this process.

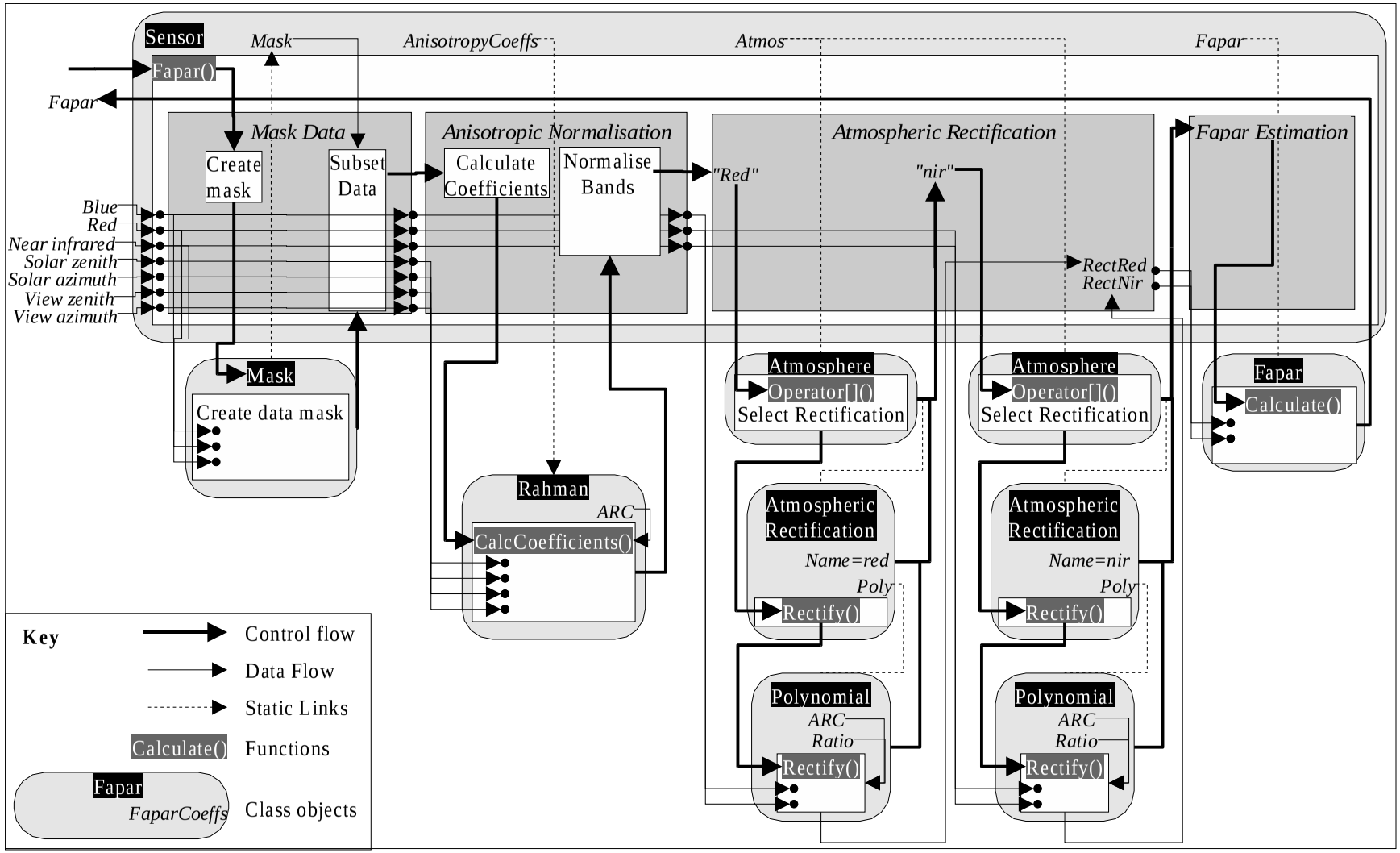

Figure 4.10 Operation of the *Sensor* class to calculate FAPAR

All the code that the user should not modify is behind the sensor interface. Additional sensors are added by:

1. incorporating a new function creator object in *SensorMapParameters* and defining a keyword to identify it.
2. Creating sensor specific derived classes of *Rahman, Atmospheric Rectification*, and *Fapar* − programs which are simple to modify.

The intention is that, because of the generalisation and generalised interfaces, there should be no need to modify any of the other control code to implement *Fapar* for a new sensor. The power of C++ is such that this is the case even if the sensor requirements, to produce *FaparProduct*, are considerably different amongst sensors.

Extension of the system to other products is possible.

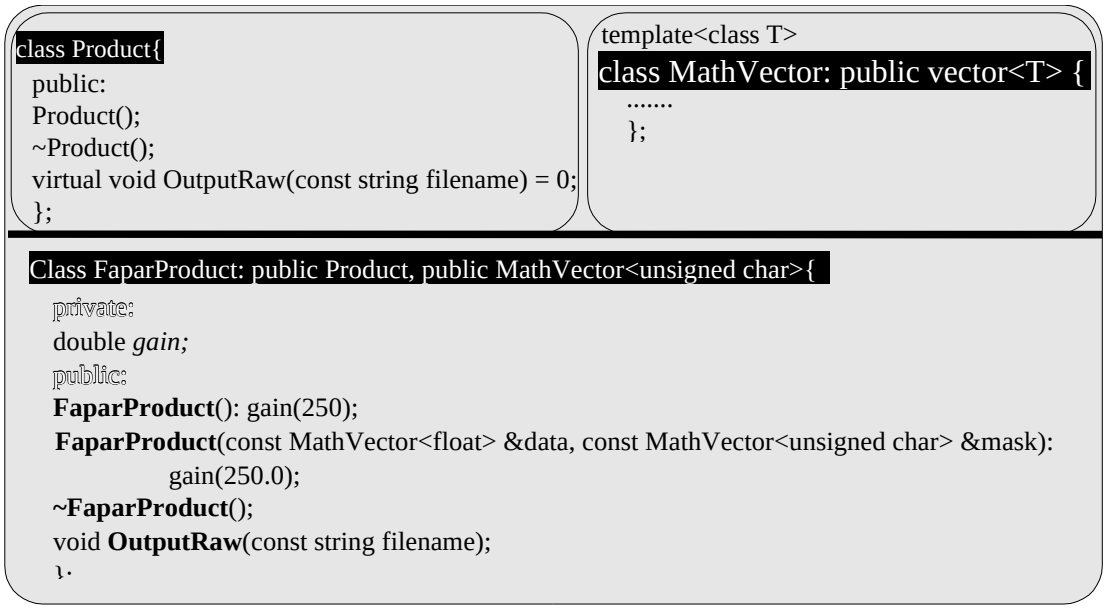

Figure 4.11 Definition of the *FaparProduct* class.

Chapter 5

Code Testing and Examples of Use

5.0 Introduction

The code has been tested against actual SeaWiFS and SPOT VEGETATION data, however, actual data does not necessarily cover all the test situations that are desirable, nor is it particularly easy to analyse any errors. As a result, synthetic test data set was established and is included in both packages. As a rule, the C++ code is generally about 3x faster than the IDL code.

5.1 The Synthetic Test Data Set.

The synthetic test data set is shown in figure 5.0. It includes 3 band files, representing the blue, red, and near infrared bands. There are 13 different values in each band ranging from −0.1 to +1.1 in 0.1 increments and they are intended to cover the expected data range plus values outside the expected range. The blue band sequences across this range for its entire length, the red band repeats an individual value 13 times before progressing to the next value, and the near infrared band repeats an individual value 13 x 13 times before progressing to the next value. In this manner, the entire 3−dimensional spectral space is covered, between −0.1 and +1.1, at a resolution of 0.1.

The angle files are set to the following angle specifications:

Solar Zenith Angle	20° and 50°.
Solar Azimuth Angle	0°, 45° and 90°.
View Zenith Angle	0°, 25° and 40°.
View Azimuth Angle	0°.

These angles are replicated in the files, and the bands files replicated accordingly, so that all combinations of angle values and band values exist in the data set.

5.2 The SeaWiFS Example Data Set.

The SeaWiFS data set was supplied by F. Mélin of the Inland and Marine Waters Unit and has been reprojected from the original data. The data set was received in *TeraScan TDF* format and converted to binary by the *TeraScan* procedure *expbin*. The resulting file was imported into IDL and reformatted and split into the appropriate files or arrays. The image specifications are given in table 5.0, and illustrated in figures 5.1 through 5.4. The FAPAR product produced from the C++ code is given in figure 5.5.

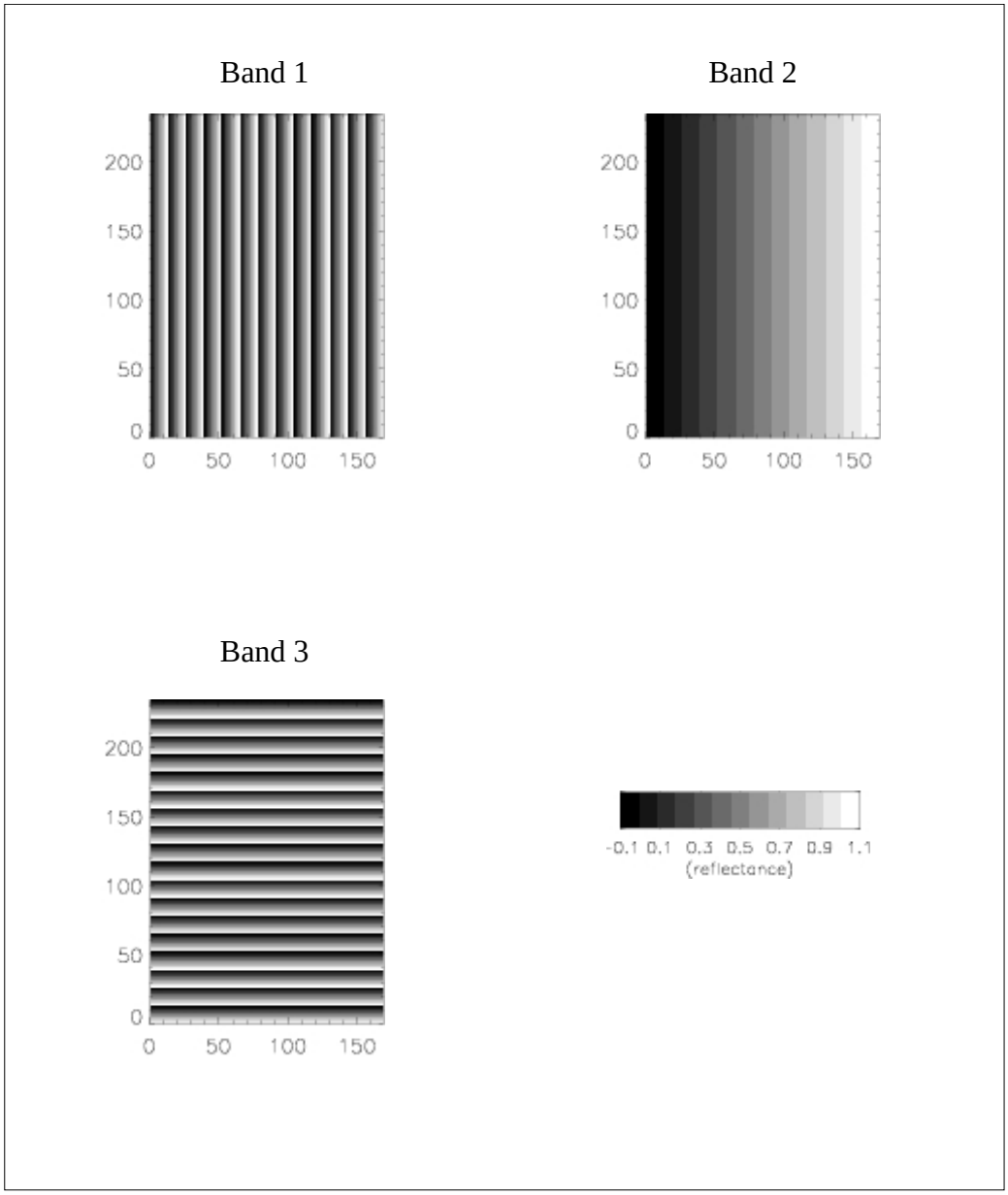

Figure 5.0 The Synthetic Test Data Set (continued).

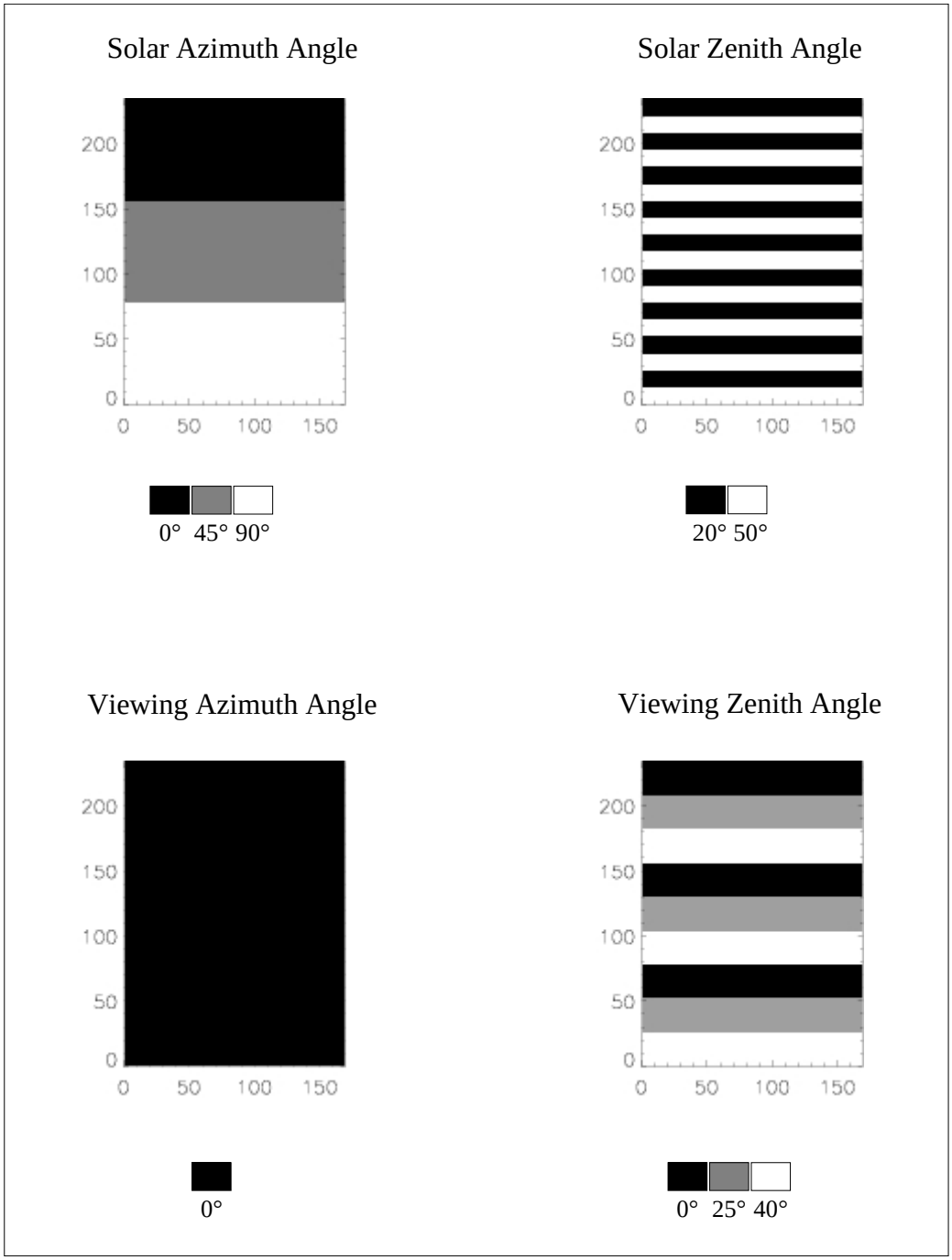

Figure 5.0 (continued).

Table 5.0 Specifications of the SeaWiFs image.

Specifications	
Sensor	SeaWiFs
File Name	S1998219_1998219.eur.bin
File Type	TeraScan
Date	7th August, 1998
Columns	1530
Rows	1376
Data Format	Integer (2 byte)
Band scaling	0.001
Angle Scaling	0.010
Angle Units	degrees

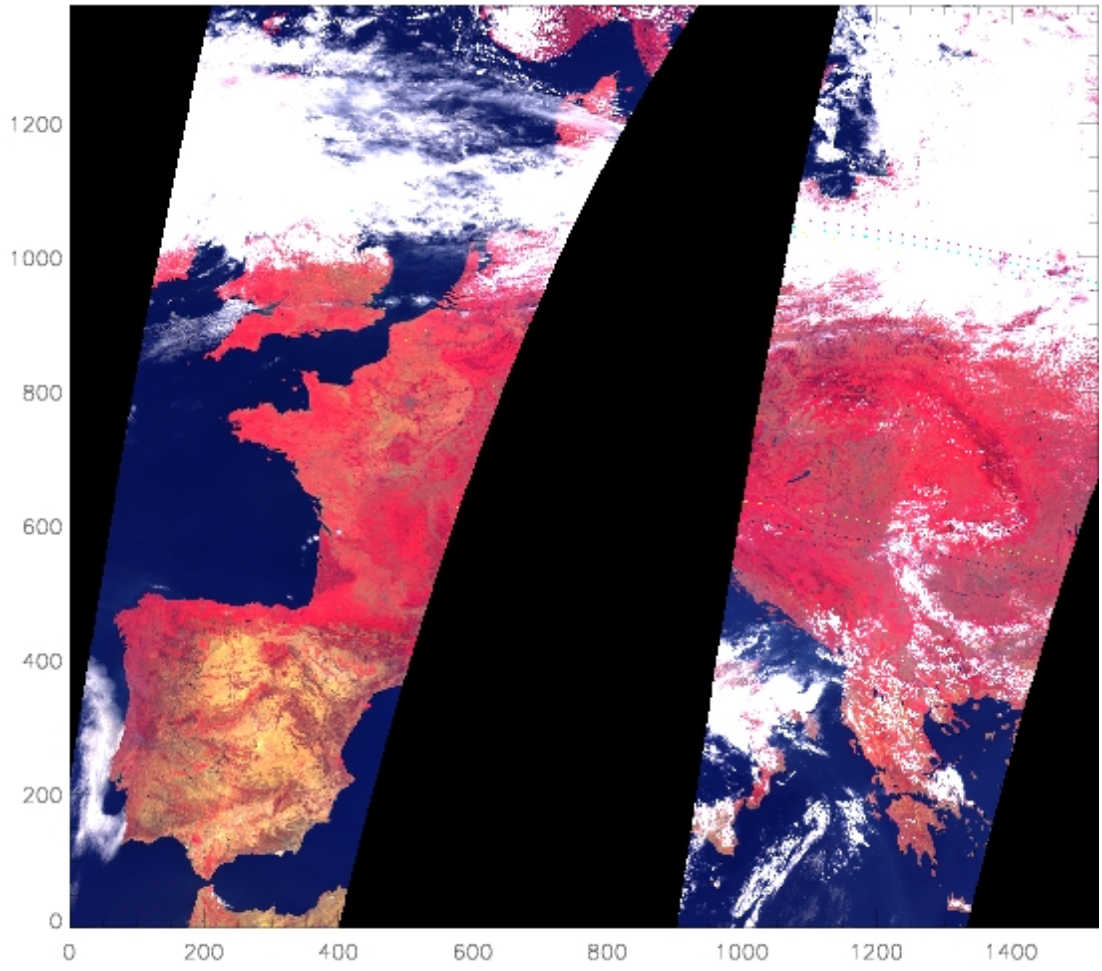

Figure 5.1 The SeaWiFs image, reflectance range 0−0.4, bands 443, 670, and 865 nm. in blue, green, and red channels respectively.

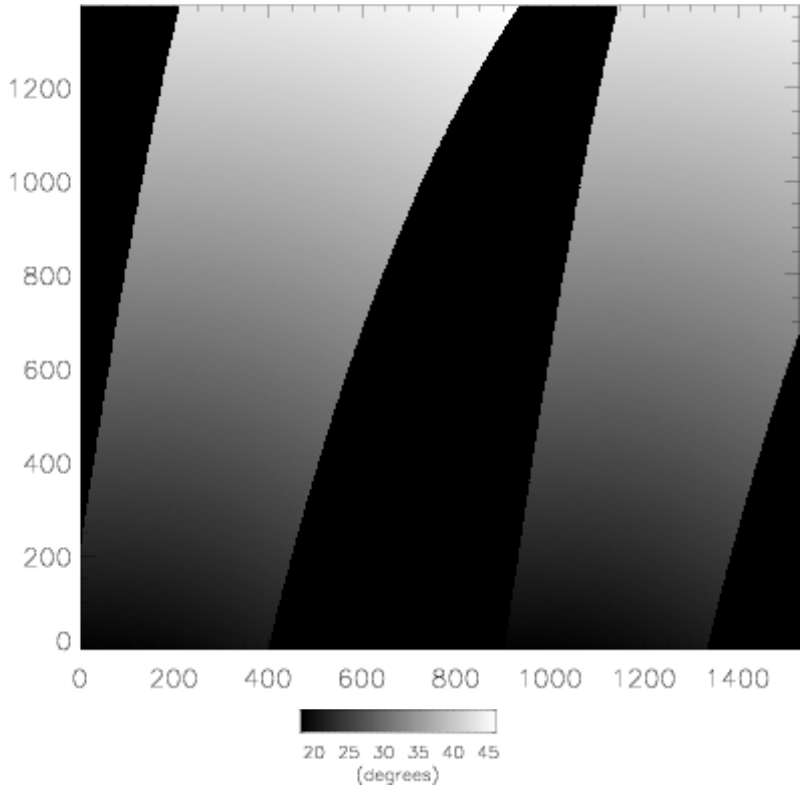

Figure 5.2 Solar zenith angles for the SeaWiFs image.

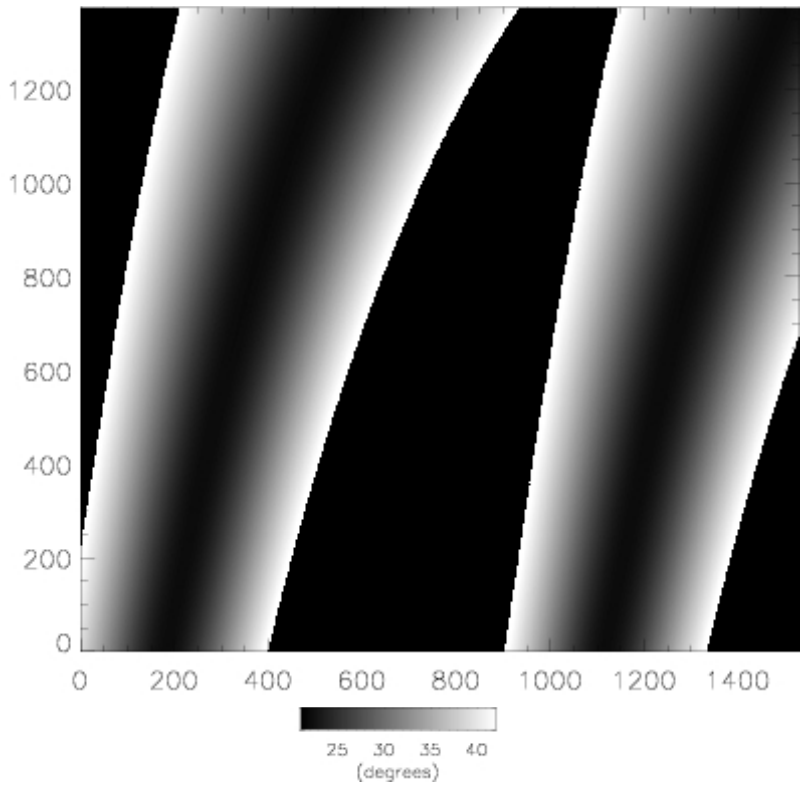

Figure 5.3 Viewing zenith angles for the SeaWiFs image.

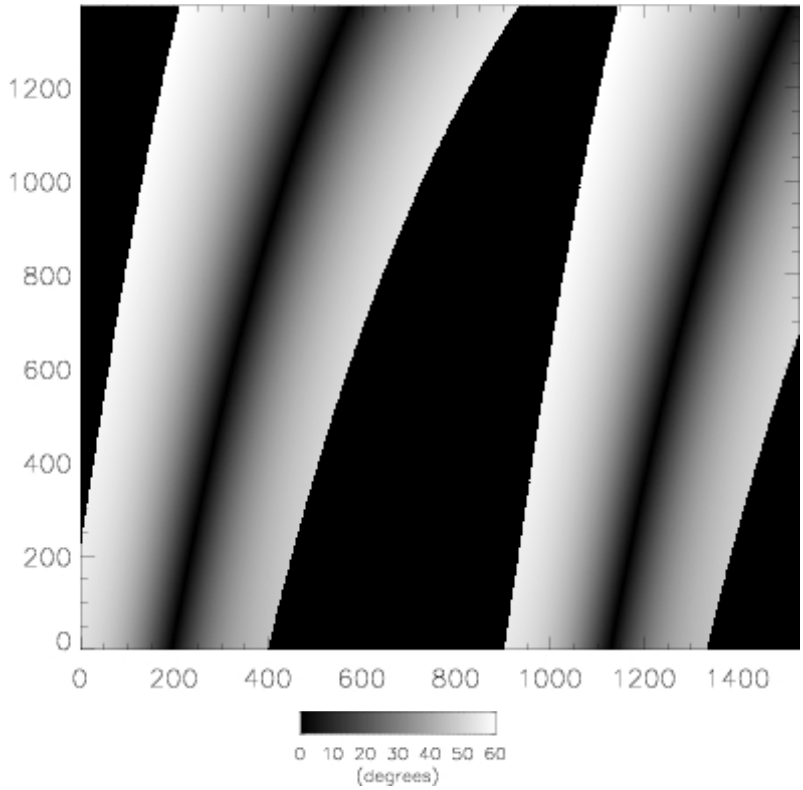

Figure 5.4 Relative azimuth angles for the SeaWiFs image.

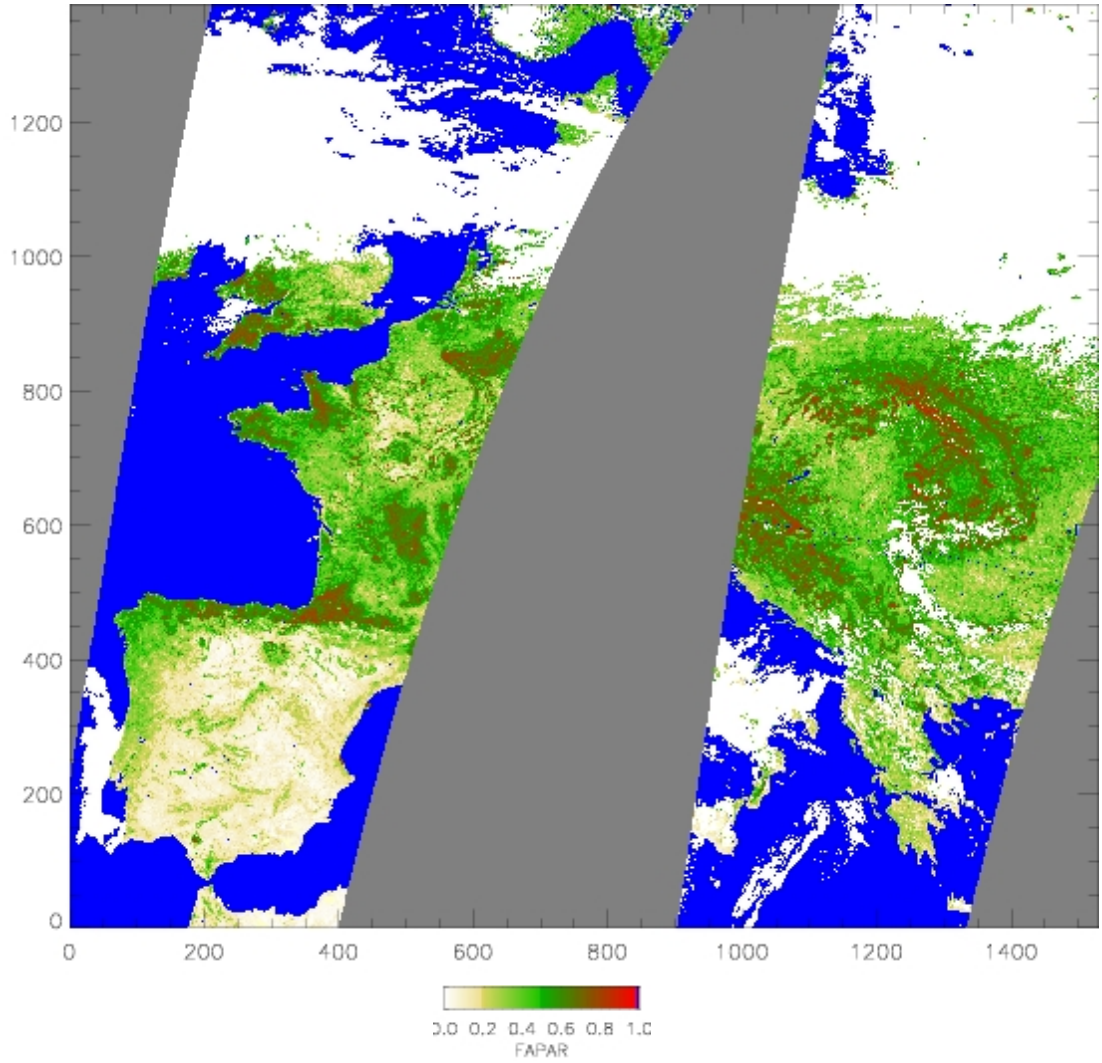

Figure 5.5 SeaWiFs image: Fraction of Absorbed Photosynthetically Active Radiation (FAPAR).

5.3 The SPOT VEGETATION Data Set.

The SPOT VEGETATION data were in P format, the only format usable with the FAPAR algorithm, as other formats have undergone further processing. The data were supplied in the original SPOT VEGETATION HDF distribution (CD) format. Typically the image data is stored as an *HDF SDS* data set and can be dumped (running under Unix) using the *hdp* utility supplied with the HDF distribution. To dump the satellite azimuth angles to a flat binary file:

hdp dumpsds −d 0001_SAA.HDF >0001_SAA.RAW

was used, and to dump the tag (image) information, held in SPOT VEGETATION as *VD* data:

hdp dumpvd 0001_SAA.HDF

The resulting files were imported into IDL and scaled appropriately. In the original SPOT VEGETATION data, angular information is only provided for every eighth pixel on a given line, and for every eighth line. The angles files were, therefore, interpolated to the same dimensions as the image data. These were then dumped as

flat binary floating point files for input into the C++ code.

Two examples are shown. The specifications of the first example, covering Burkino Fasso, are given in table 5.1. The image data is shown in figure 5.6, the angle data in figures 5.7 through 5.9, and the FAPAR product in figure 5.10. 919540 pixels were processed for FAPAR and there were no "bad" pixels.

Code performance is similar to that for SeaWiFS which is to be expected as the data sets are similar in size.

Table 5.1 Specifications of the SPOT VEGETATION test image (Burkino Fasso).

Specifications	
Sensor	SPOT VEGETATION
Product Type	P
File Name	0001_**.HDF
File Type	SPOT HDF
Date	13th December, 1999
Band Columns	1009
Band Rows	992
Band Format	Integer (2 byte)
Band scaling	0.001
Angle Columns	127
Angle Rows	125
Angle Format	Byte
Zenith Angle Scaling	0.500
Azimuth Angle Scaling	1.500
Angle Sample Rate	8 x 8
Angle Units	degrees

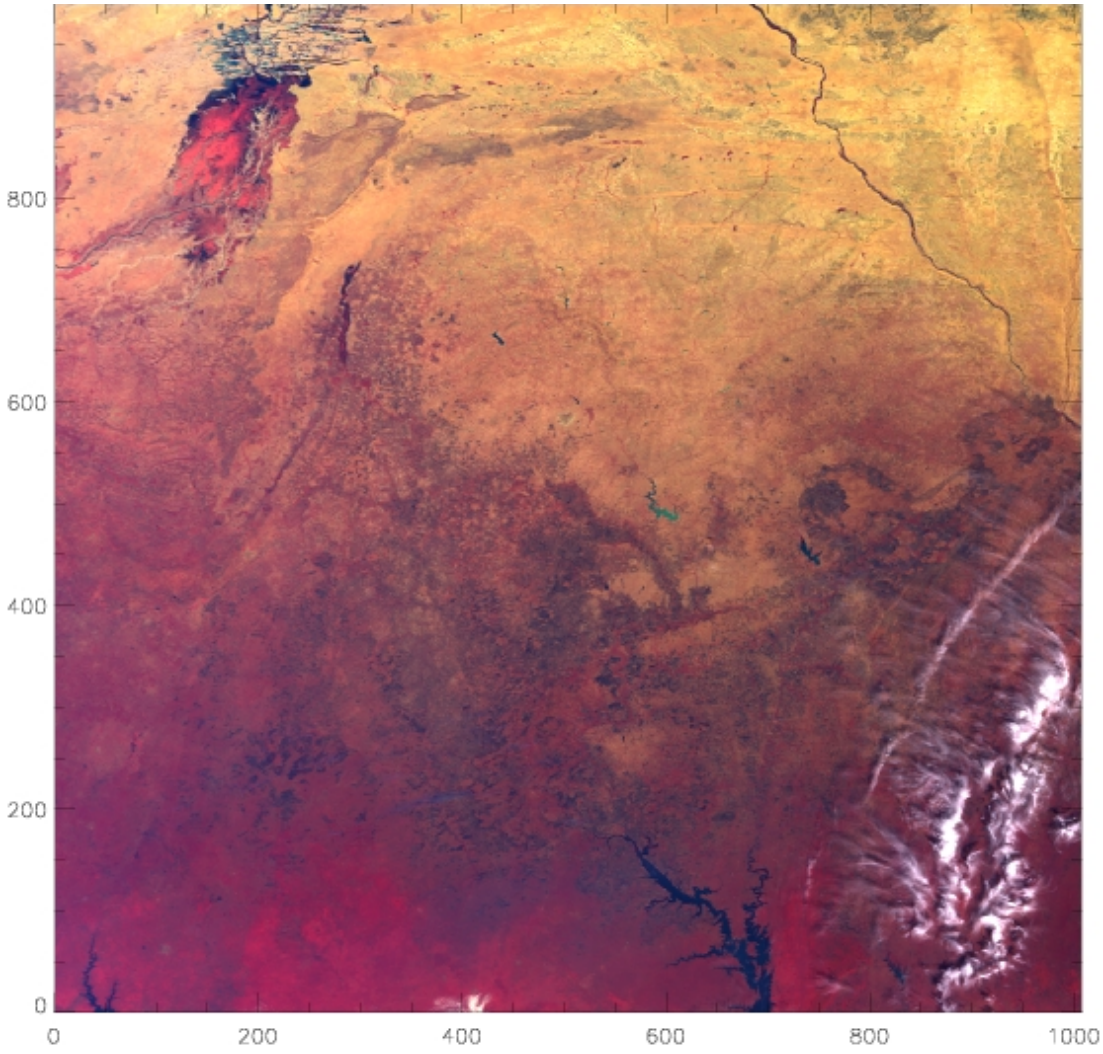

Figure 5.6 The SPOT VEGETATION Burkino Fasso image, reflectance range 0−0.4, bands 450, 640, and 840 nm. in blue, green, and red channels respectively.

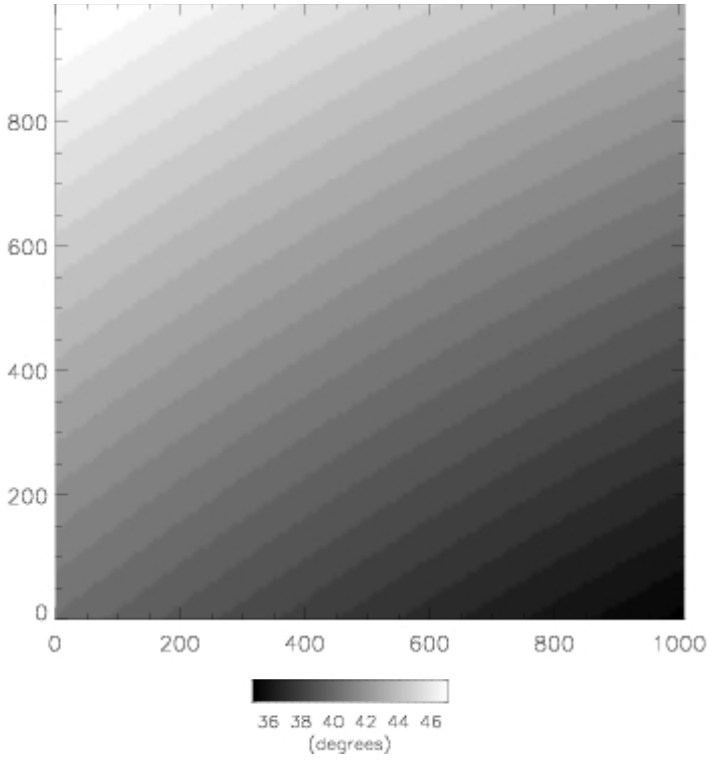

Figure 5.7 Solar zenith angles for the SPOT VEGETATION Burkino Fasso image.

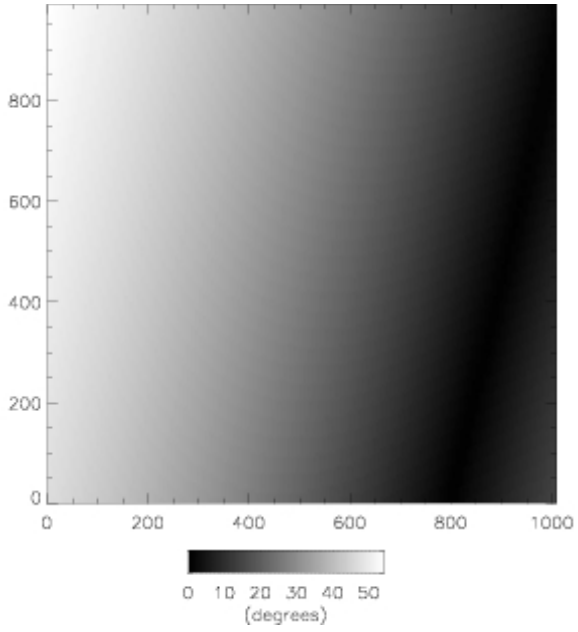

Figure 5.8 View zenith angles for the SPOT VEGETATION Burkino Fasso image.

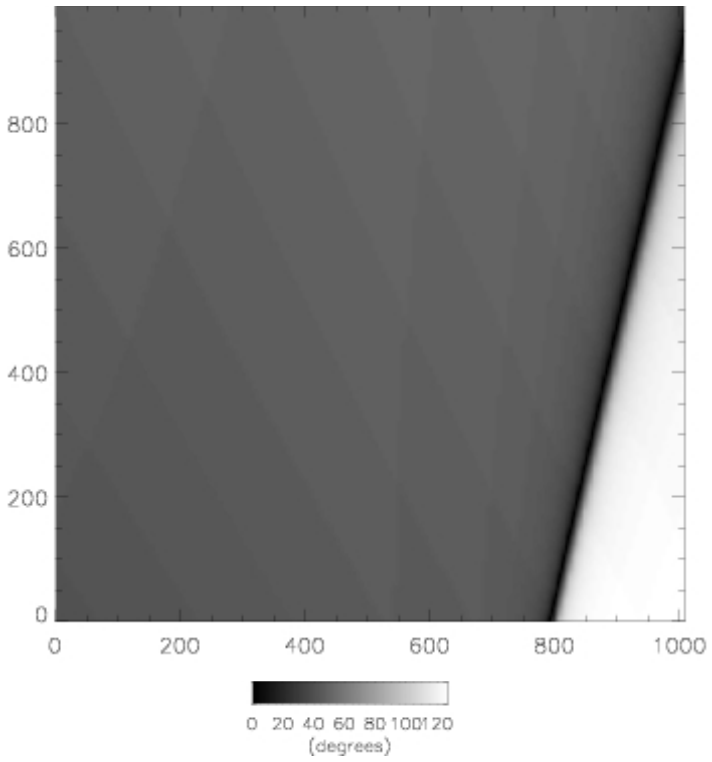

Figure 5.9 Relative azimuth angles for the SPOT VEGETATION Burkino Fasso image.

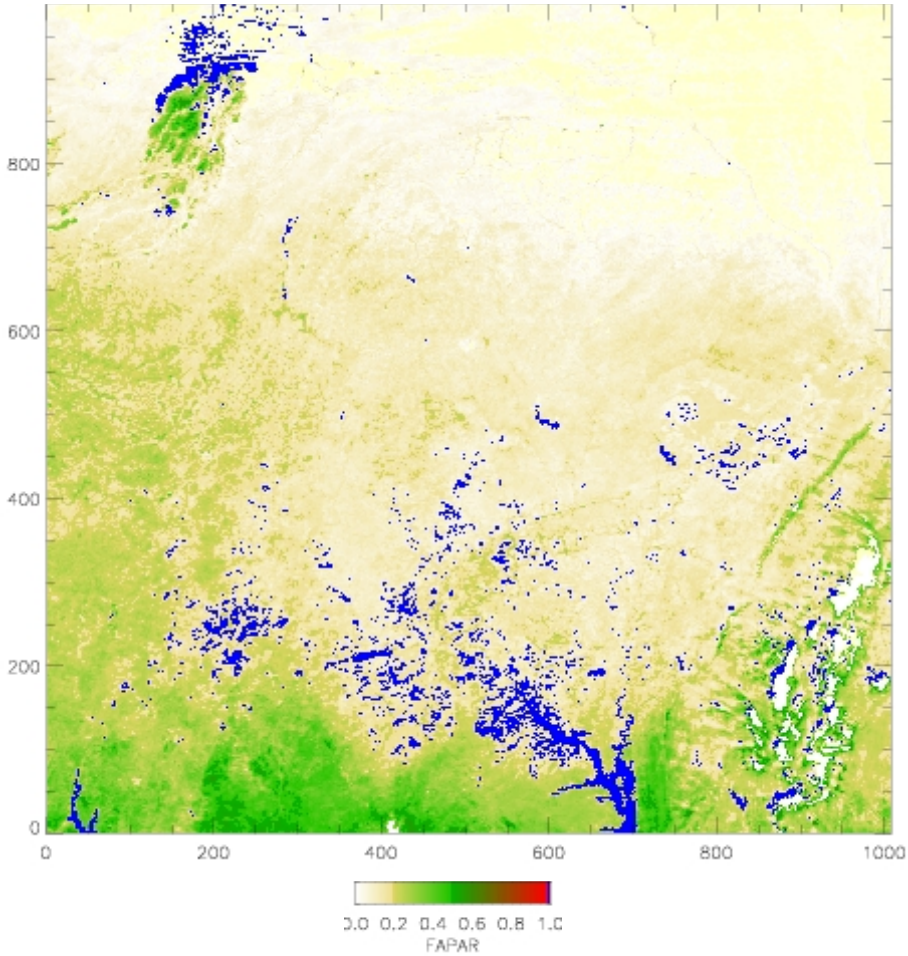

Figure 5.10 SPOT VEGETATION Burkino Fasso image: Fraction of Absorbed Photosynthetically Active Radiation (FAPAR).

The second SPOT VEGETATION example data set has the specifications listed in table 5.2, and covers Libya and Algeria north to the Netherlands (figure 5.11). The size, at more than 7×10^6 pixels, is much larger than the other data sets. The angle data is shown in figures 5.12 through 5.14, and the FAPAR product in figure 5.15.

IDL code performance, table 5.1, is much poorer with this image than for the smaller images (by an order of magnitude). The main reason for this is that computation is taking place on a machine with 512 Mb of RAM and has 512 Mb of disk based "RAM". IDL appears to perform poorly when disk based RAM is used, as chip based RAM is exhausted, spending a considerable amount of time paging in and out. The C++ code, however, computed the FAPAR in only 3 x the time taken for the Burkino Fasso data set.

Table 5.2 Specifications of the SPOT VEGETATION, W. Europe, image.

Specifications	
Sensor	SPOT VEGETATION
Product Type	P
File Name	0001_**.HDF
File Type	SPOT HDF
Date	27th October, 1999
Band Columns	1961
Band Rows	3585
Band Format	Integer (2 byte)
Band scaling	0.001
Angle Columns	246
Angle Rows	449
Angle Format	Byte
Zenith Angle Scaling	0.500
Azimuth Angle Scaling	1.500
Angle Sample Rate	8 x 8
Angle Units	degrees

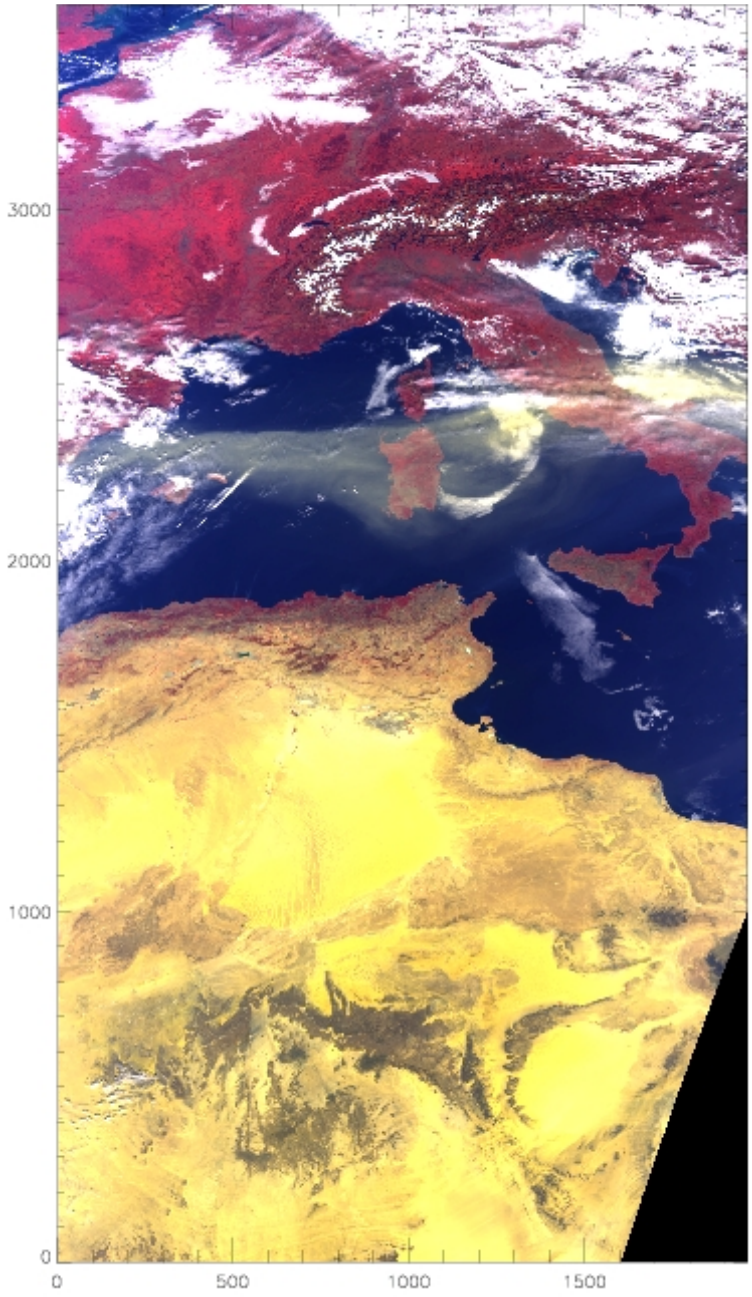

Figure 5.11 The SPOT VEGETATION, W. Europe, image, reflectance range 0−0.4, bands 450, 640, and 840 nm. in blue, green, and red channels respectively.

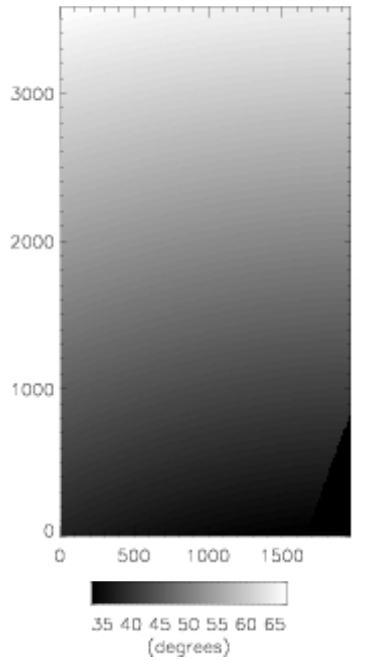

Figure 5.12. Solar Zenith Angles for the SPOT VEGETATION, W. Europe, image.

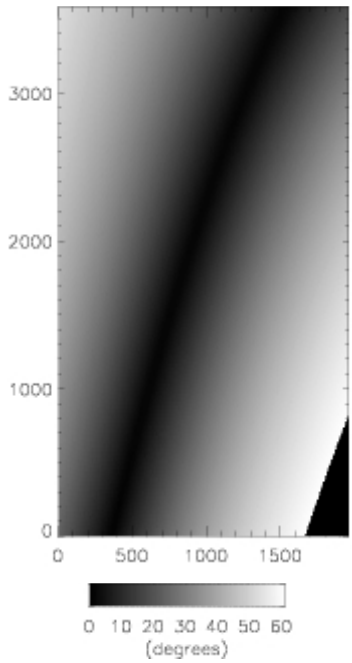

Figure 5.13 Viewing zenith angles for the SPOT VEGETATION, W. Europe, image.

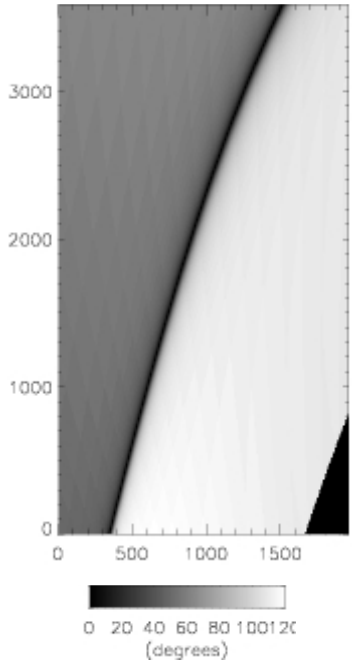

Figure 5.14 Relative azimuth angles for SPOT VEGETATION, W. Europe, image.

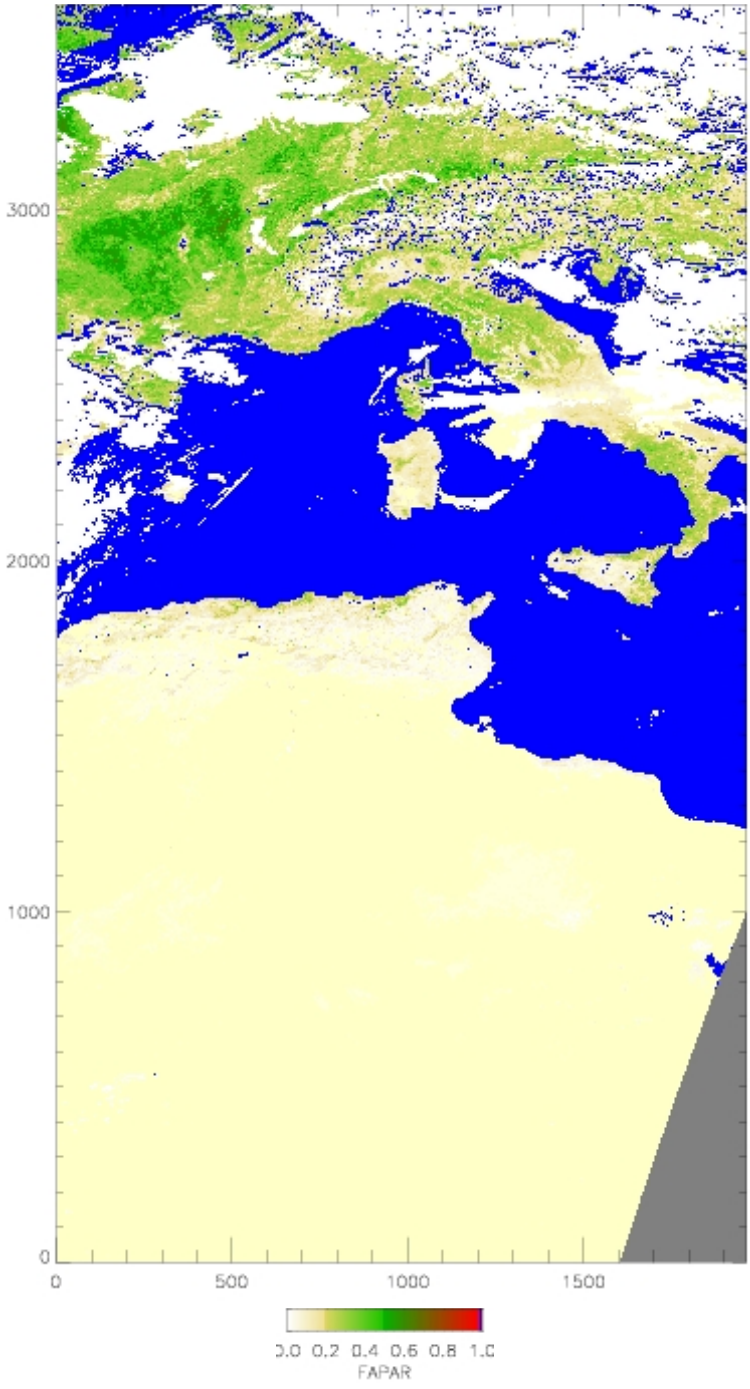

Figure 5.15 Fraction of Absorbed Photosynthetically Active Radiation (FAPAR) for the SPOT VEGETATION, W. Europe, image:

References

1. Gobron, N., Pinty, B., Verstraete, M. M., Widlowski, J−L., (2000). 'Advanced Vegetation Indices Optimized for Up−Coming Sensors: Design, Performance, and Applications', *IEEE Transactions on Geoscience and Remote Sensing*, 38, 2489−2505.

2. Gobron, N., Pinty, B., Verstraete, M. M., Taberner, M., (2002). 'Medium Resolution Imaging Spectrometer (MERIS) An optimized FAPAR Algorithm − Theoretical Basis Document.' − *Joint Research Centre, Institute for Environment and Sustainability, Publication no.: EUR 20143 EN.*

3. Gobron, N., Pinty, B., , Mélin, F., Taberner, M., Verstraete, M. M., (2002). 'Sea Wide Field−of−View Sensor (SeaWiFS) An optimized FAPAR Algorithm − Theoretical Basis Document.' − *Joint Research Centre, Institute for Environment and Sustainability, Publication no.: EUR 20148 EN.*

4. Gobron, N., Pinty, B., Verstraete, M. M., Taberner, M., (2002). 'VEGETATION An optimized FAPAR Algorithm − Theoretical Basis Document.' − *Joint Research Centre, Institute for Environment and Sustainability, Publication no.: EUR 20146 EN.*

5. Gobron, N., Pinty, B., Verstraete, M. M., Taberner, M., (2002). 'Global Land Imager (GLI) An optimized FAPAR Algorithm − Theoretical Basis Document.' − *Joint Research Centre, Institute for Environment and Sustainability, Publication no.: EUR 20147 EN.*

6. Gobron, N., Mélin, F., Pinty, B., Verstraete, M. M., Widlowski, J.−L., Bucini, G., (2001). 'A Global Vegetation Index for SeaWiFS: Design and Applications', in *Remote Sensing and Climate Modeling: Synergies and Limitations*, Edited by M. Beniston and M. M. Verstraete, Kluwer Academic Publishers, Dordrecht, 5−21.

7. Pinty, B., Gobron, N., Mélin, F., Pinty, B., Verstraete, M. M., (2002). 'A Time Composite Algorithm − Theoretical Basis Document. ' *Joint Research Centre, Institute for Environment and Sustainability, Publication no.: EUR 20150 EN.*

8. Rahman, H., Pinty, B., Verstraete, M. M., (1997). 'Coupled surface−atmosphere reflectance (CSAR) model. 2. Semiempirical surface model usable with NOAA advanced very high resolution radiometer data.' *J. Geophys. Res.* 102:9431−9446.

Appendix

Definition of the C++ Classes

```
typedef Sensor* (*fP)();

class SensorMapParameters: public map< string, fP >{

//Purpose:  Map the sensor type to the appropriate constructor function objects.

Public:
        SensorMapParameters();
        ~SensorMapParameters();
        void Print();

        //Match the sensor type to the sensorconstructor function
        //object and return it.
        fP SensorObject(const string sensor);
`
```

```
class GeneralCommandLineParameters: public map< string, string >{

//Purpose:  A class to parse the command line parameters.

public:
        GeneralCommandLineParameters();
        GeneralCommandLineParameters(int argc, char *argv[]);
        ~GeneralCommandLineParameters();
        void Print();

        //Find and return the sensor type.
        string Sensor();

        // Find and return the input file name associated with
        // the key name.
        string inFile(string key);

        // Find and return the output file name associated with
        // the key name.
        string outFile(string key);

        //Determine whether a parameter flag has been set.
        bool operator==(const string key);
`
```

```
class Sensor{
  protected:
  class Rahman *anisotropyCoeffs;
  class Atmosphere atmos;
  class Fapar *fapar;
  class Mask *mask;

  public:
  Sensor(){}
  virtual ~Sensor();

  FaparProduct *Fapar(MathVector<float> &blue, MathVector<float> &red,
            MathVector<float> &nearIR, const MathVector<float> &sZ, const
            MathVector<float> &sA, const MathVector<float> &vZ, const
            MathVector<float> &vA, string rectOut="");
  `
```

```
class SensorSEAWIFS : public Sensor {
   public:
   SensorSEAWIFS();
   ~SensorSEAWIFS();
   static Sensor* new_SensorSEAWIFS();
   };
```

```
class Rahman : public vector< MathVector<float> >{
    protected:
    class AnisotropicReflectionCoeff *aRC;

    template<class T>
    void Coefficients(const MathVector<T> &solarZenith, const MathVector<T>
            &viewZenith, const MathVector<T> &solarAzimuth,const
            MathVector<T> &viewAzimuth);

    public:
    Rahman();
    ~Rahman();
    template<class T>
    void CalcCoefficients(const MathVector<T> &sz, const MathVector<T> &vz,
            const MathVector<T> &sa, const MathVector<T> &va);
    void Print();
```

```
class SEAWIFSRahman : public Rahman {
    public:
    SEAWIFSRahman();
    ~SEAWIFSRahman();
    void Print();
    };
```

```
class Atmosphere : public vector<AtmosphericRectification *>{
    public:
    Atmosphere(){}
    ~Atmosphere(){}

    AtmosphericRectification *operator[](size_t i);
    AtmosphericRectification *operator[](string id);
    };
```

```
class Fapar: public MathVector<float>{
    private:
    float irN, redN, constN;
    float irC, redC, constD;

    protected:
    FaparCoeff faparCoeffs;
    public:
    Fapar();
    ~Fapar();

    virtual void Calculate(const MathVector<float> &red, const
                MathVector<float> &ir);
    virtual void Calculate(const MathVector<float> &red, const
                MathVector<float> &ir, const MathVector<unsigned char> &mask);
    FaparProduct *Product(const MathVector<unsigned char> &mask);
    void Print();
```

```
class SEAWIFSFapar: public Fapar{
    public:
    SEAWIFSFapar();
    ~SEAWIFSFapar();
    void Print();
    };
```

```
class Product{
  public:
  Product();
  ~Product();
  virtual void OutputRaw(const string filename) = 0;
  };
```

```
template<class T>
class MathVector: public vector<T> {
    .....
    };
```

```
Class FaparProduct: public Product, public MathVector<unsigned char>{
    private:
    double gain;
    public:
    FaparProduct(): gain(250);
    FaparProduct(const MathVector<float> &data, const MathVector<unsigned char> &mask):
                gain(250.0);
    ~FaparProduct();
    void OutputRaw(const string filename);
    };
```

```
struct AnisoRefCoeff {float r, p, k;};
class AnisotropicReflectionCoeff : public vector<AnisoRefCoeff> {
    public:
    AnisotropicReflectionCoeff();
    ~AnisotropicReflectionCoeff();
    void Print()const;
    };
```

```
class SEAWIFSAnisRefCoeff : public AnisotropicReflectionCoeff {
    public:
    SEAWIFSAnisRefCoeff()
    ~SEAWIFSAnisRefCoeff(){}
     void Print();
    };
```

```
struct AtmosRectRatio{float numerator, denominator;};
class AtmosRectCoeff : public vector<AtmosRectRatio> {
    public:
    AtmosRectCoeff();
    ~AtmosRectCoeff();
    };
```

```
Class SEAWIFSAtmosRectCoeff : public AtmosRectCoeff{
    public:
    SEAWIFSAtmosRectCoeff();
    SEAWIFSAtmosRectCoeff(string band);
    ~SEAWIFSAtmosRectCoeff();
    };
```

```
class FaparCoeff : public vector<float> {
    public:
    FaparCoeff();
    ~FaparCoeff();
    void Print()const;
    };
```

```
class SEAWIFSFaparCoeff : public FaparCoeff {
    public:
    SEAWIFSFaparCoeff();
    ~SEAWIFSFaparCoeff();
    void Print();
    };
```

```
Class Polynomial {
    protected:
    AtmosRectCoeff aRC;
    bool ratio;

    public:
    Polynomial();
    ~Polynomial();
    virtual void Rectify(const MathVector<float> &band1, const MathVector<float>
            &band2, MathVector<float> *output);
    virtual void Rectify(const MathVector<float> &band1, const MathVector<float>
            &band2, const MathVector<unsigned char> &mask, MathVector<float>
            *output);
```

```
class Polynomial0 : public Polynomial {
    private:
      float mB1N, cB1N, mB2N, cB2N, mB1B2N;
      float mB1D, cB1D, mB2D, cB2D, mB1B2D;
    protected:
      void SetCoefficients();
    public:
    Polynomial0();
    ~Polynomial0();
    void Rectify(const MathVector<float>
            &band1, const MathVector<float>
            &band2, MathVector<float>
            *output);
    void Rectify(const MathVector<float>
            &band1, const MathVector<float>
            &band2, const
            MathVector<unsigned char>
            &mask, MathVector<float>
            *output);
```

```
class Polynomial1 : public Polynomial {
    private:
    float l1, l2, l3, l4, l5, l6;
    protected:
    void SetCoefficients();
    public:
    Polynomial1();
    ~Polynomial1();
    void Rectify(const MathVector<float>
            &band1, const
            MathVector<float> &band2,
            MathVector<float> *output);
    void Rectify(const MathVector<float>
            &band1, const
            MathVector<float> &band2,
            const MathVector<unsigned
            char> &mask, MathVector<float>
            *output);
```

```
class SEAWIFSPolynomial : public Polynomial0 {
    public:
    SEAWIFSPolynomial(const string bandID)
    ~SEAWIFSPolynomial(){}
    };
```

```
class MERISPolynomial : public Polynomial1 {
    public:
    MERISPolynomial(const string bandID);
    ~MERISPolynomial(){}
    };
```

```
template<class T>
class MathVector: public vector<T> {
    .....
    };
```

```
class Mask : public MathVector<unsigned char> {
    Public:
    Mask();
    explicit Mask(size_t sz) : MathVector<unsigned char>(sz);
    Mask(const MathVector<float> &bc, const MathVector<float> &rc,const
            MathVector<float> &nc);
    ~Mask();
    };
```