

Sentinel-2 Agriculture

Design Definition File

Algorithm Theoretical Basis Document for L3 biophysical product



Milestone	Milestone 2
Version	1.0
Authors	CESBIO - Jordi Inglada, Marcela Arias, David Morin

© UCL-Geomatics 2015

This document is the property of the Sen2-Agri partnership, no part of it shall be reproduced or transmitted without the express prior written authorisation of UCL-Geomatics (Belgium).

Contents

1	Introduction	2
2	System components	2
2.1	LAI retrieval	2
2.1.1	Overview of the subsystem	2
2.1.2	The applications	2
2.1.2.1	Input Variable Generation	3
2.1.2.2	ProSail Simulator	3
2.1.2.3	Inverse Model Learning	4
2.1.2.4	Inversion	4
2.1.2.4.1	Inversion from ASCII files	4
2.1.2.4.2	Image Inversion	5
2.1.2.5	Profile Reprocessing	5
2.1.2.5.1	On-line reprocessing	5
2.1.2.5.2	CSDM fitting	5
2.1.3	The Python module	6
2.2	Phenological NDVI metrics	8
2.2.1	Overview of the subsystem	8
2.2.2	NDVI profile computation	8
2.2.3	Double logistic fitting	9
2.2.4	Metric estimations	15
2.2.4.1	Date of the maximum positive gradient	15
2.2.4.2	Starting date	15
2.2.4.3	Length of the plateau	16
2.2.4.4	Senescence date	16
2.2.4.5	Metrics estimation implementation	17
3	Appendix	17
3.1	The OTB-BV library	17
3.1.1	Type definitions	17
3.1.2	Utilities	18
3.1.3	Simulation	18
3.1.4	Regression	21
3.1.5	Reprocessing	21
3.1.5.1	On-line reprocessing	21
3.1.5.2	Fit CSDM	23

1 Introduction

This document describes the proposed processing chain for the production of the vegetation status products for the Sentinel-2 Agriculture project.

The algorithm description and justification of choices have been documented in the Design Justification File¹. The present document describes the processing chain and its subsystems.

Where possible, we use standard components available in the Orfeo Toolbox version 4.4². When no equivalent component is available in the Orfeo Toolbox, the algorithm is described using either pseudo-code or the example implementations in Python and C++ available at <http://tully.ups-tlse.fr/jordi/otb-bv> (LAI retrieval and profile reprocessing) and <http://tully.ups-tlse.fr/jordi/phenotb> (NDVI temporal metrics).

2 System components

2.1 LAI retrieval

2.1.1 Overview of the subsystem

The LAI retrieval is performed by using machine learning to build a non-linear regression model. The regression model is estimated using simulated satellite reflectances. These reflectances are simulated using the ProSail model. This approach is based on the LAI procedure S2PAD³, the main difference being that the model is applied to every acquisition date, which allows removing the solar and sensor angles from the predictor variables. Figure 1 illustrates this procedure.

After that, 2 reprocessing options are available in order to improve the LAI retrieval by taking into account the multi-temporal information. The first option is an on-line algorithm which uses the n last acquisitions to estimate the LAI value for the last one. It uses the error estimation of the LAI retrieval to weight the LAI values. The second option can be applied at the end of the season and consists in fitting a phenological model (double logistic function) to the LAI retrieval time series. Figure 2 illustrates the multi-temporal retrieval approaches.

The LAI retrieval processing chain is implemented as a set of applications corresponding to the individual processing blocks. These applications are based on a C++ library which relies on the Orfeo Toolbox (see section 3.1). A Python module is also provided in order to better integrate the applications into a processing chain.

2.1.2 The applications

There are 6 applications used to implement the LAI retrieval processing chain.

1. **BVInputVariableGeneration**: generates a distribution of input variables for the ProSail model which have the correct statistics and correlations.
2. **ProSailSimulator**: simulates satellite reflectances using the ProSail variables, the acquisition configuration and the satellite spectral bands.
3. **InverseModelLearning**: uses the simulated reflectances for each input LAI in order to build a regression model. This application can also be used to build a regression model for the LAI retrieval error.
4. **BVInversion**: application which applies the regression models to a set of input reflectances. It is mainly used for validation purposes.
5. **BVImageInversion**: like **BVInversion** but applied to full images.
6. **ProfileReprocessing**: application which applies the multi-temporal algorithms to an LAI time series.

In the following sections we show how to use these applications from the command line.

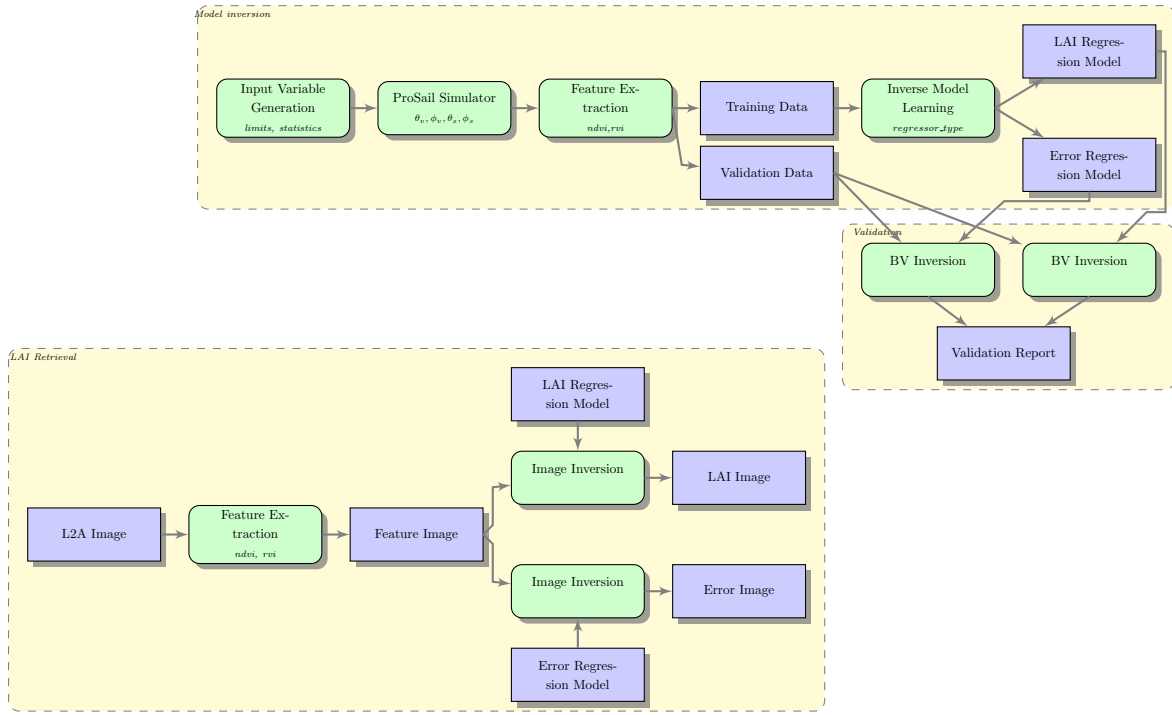


Figure 1: Block diagram for the mono-date LAI retrieval procedure.

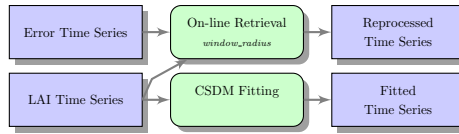


Figure 2: Block diagram for the 2 multi-date LAI retrieval procedures.

2.1.2.1 Input Variable Generation Table 1 lists the variables used by the `BVInputVariableGeneration` application. The application is used as follows:

```
1 | otbcli_BVInputVariableGeneration -samples $samples -out $outfile
```

2.1.2.2 ProSail Simulator Table 2 lists the variables used by the `ProSailSimulator` application. The application is used as follows:

```
1 | otbcli_ProSailSimulator -bvfile $bvFile -rsrfile $rsrFile -out $outputFile\  
2 | -solarzenith $solarZenithAngle\  
3 | -sensorzenith $sensorZenithAngle\  
4 | -azimuth $solarSensorAzimuth\  
5 | -noisevar $noisevar
```

¹Sentinel-2 for Agriculture Design Justification File (v1.1, 2015/04/15)

²This version corresponds to the changeset <http://hg.orfeo-toolbox.org/OTB/file/baf740ee2113>.

³S2PAD-VEGA-ATBD-0003-2₁L2BATBD. S2PAD - Sentinel-2 MSI - Level 2B Products Algorithm Theoretical Basis Document. Issue 2.1. 2010/10/13.

Table 1: Variables for the input variable generation

Input variable	role
samples	Number of samples to be generated
Output variable	role
outfile	Name of the ASCII file to store the samples

Table 2: Variables for the ProSail simulation

Input variable	role
bvfile	File containing the ProSail variables
rsrfile	File containing the satellite spectral responses
solarzenith	Solar zenithal angle
sensorzenith	Sensor zenithal angle
azimuth	Relative azimuth between sensor and Sun
noisevar	Noise variance to be added to the reflectances
Output variable	role
out	Name of the ASCII file to store the simulated reflectances

Table 3: Variables for the inverse model learning

Input variable	role
training	File containing the input/output pairs
regression	Regression algorithm: svr, nn, mlr
normalization	File containing the min and max for each variable
Output variable	role
out	Name of the file containing the regression model for the variable
errest	Name of the file containing the regression model for the error estimation

2.1.2.3 Inverse Model Learning Table 3 lists the variables used by the `InverseModelLearning` application. The first column of the training file corresponds to the dependent variable and the following columns are the predictors. The normalization file contains 2 columns corresponding to the minimum and maximum values to be used for the normalization of each variable. Each row corresponds to a variable in the same order of the columns of the training file.

The application is used as follows:

```

1 otbcli_InverseModelLearning -training $trainingFile -out $modelFile\
2                             -errest ${modelFile}_errest\
3                             -regression svr\
4                             -normalization $normalizationFile\
5                             -bestof 1

```

2.1.2.4 Inversion There are 2 applications for the inversion. The first one operates on ASCII files as those used by the `InverseModelLearning` and is used for validation purposes. The second one operates on images and is used for the LAI retrieval.

It is worth noting that the same application is used for the LAI retrieval and for the error estimation, the only difference being the regression model provided to it.

Table 4: Variables for the variable inversion on ASCII files

Input variable	role
reflectances	File containing reflectances
model	Regression model file
normalization	File containing the min and max for each variable
Output variable	role
out	Name of the file containing the variable retrieval

2.1.2.4.1 Inversion from ASCII files Table 4 lists the variables used by the `BVInversion` application. The application is used as follows:

```

1 otbcli_BVInversion -reflectances $reflectanceFile -model $modelFile\
2 -normalization $normalizationFile\
3 -out $outputFile

```

Table 5: Variables for the variable inversion using images

Input variable	role
in	Input image
model	Regression model file
normalization	File containig the min and max for each variable
Output variable	role
out	Name of the image file containing the variable retrieval

2.1.2.4.2 Image Inversion Table 4 lists the variables used by the BVImageInversion application. The application is used as follows:

```

1 otbcli_BVImageInversion -in $inputimage -model $modelname \
2 -out $laiimage -normalization $normname

```

Table 6: Variables for the profile reprocessing

Input variable	role
ipf	Input time profile
algo	Reprocessing algorithm: local or fit
algo.local.bwr	Backward radius of the window for the local algorithm
algo.local.fwr	Forward radius of the window for the local algorithm
Output variable	role
opf	Output profile

2.1.2.5 Profile Reprocessing Table 6 lists the variables used by the ProfileReprocessing application.

It is worth noting that the current version of the otb-bv package does not have a version of this application which is able to operate on image time series and the input and output profiles are ASCII files containing the values for each date for individual pixels. Adapting the application to operate on images is trivial by using the `itk::UnaryFunctorImageFilter` and passing it the appropriate function defined in `otbProfileReprocessing.h`. See section 3.1.5 for details.

2.1.2.5.1 On-line reprocessing In the case of the on-line reprocessing, the application is used as follows:

```

1 otbcli_ProfileReprocessing -ipf $inputprofile\
2 -opf $outputprofile\
3 -algo local\
4 -algo.local.bwr 2\
5 - algo.local.fwr 0

```

In this case, the choice of radiuses implies that the last date is reprocessed ($fwr = 0$) using also the 2 previous dates ($bwr = 2$).

2.1.2.5.2 CSDM fitting In the case of the fitting reprocessing, there are no specific parameters and the application is used as follows:

```

1 otbcli_ProfileReprocessing -ipf $inputprofile\
2 -opf $outputprofile\
3 -algo fit

```

2.1.3 The Python module

The Python module offers a wrapper around the applications to facilitate the implementation of a complete processing chain. This wrapper is not limited to the Python wrappers automatically generated for the applications. It also offers pre- and post-processing steps.

We start by defining a set of indices and names for the PROSPECT and Sail variables. The indices here have to be coherent with the order used in the bv file and the definition of the vars in `otbBVTypes.h`.

```
1 bvindex = {"MLAI": 0, "ALA": 1, "CrownCover": 2, "HsD": 3, "N": 4, "Cab": 5, \
2           "Car": 6, "Cdm": 7, "CwRel": 4, "Cbp": 9, "Bs": 10, "FAPAR": 11, \
3           "FCOVER": 12}
4 bv_val_names = {"MLAI": ['gai', 'lai-bvnet'], "FAPAR": ['fapar', 'fapar-bvnet'], \
5                "FCOVER": ['fcover', 'fcover-bvnet']}
```

The draws of the PROSPECT and Sail variables are generated using a simple wrapper for the `BVInputVariableGeneration` application (see section 2.1.2.1):

```
1 def generateInputBVDistribution(bvFile, nSamples):
2     app = otb.Registry.CreateApplication("BVInputVariableGeneration")
3     app.SetParameterInt("samples", nSamples)
4     app.SetParameterString("out", bvFile)
5     app.ExecuteAndWriteOutput()
```

The generation of the training data for the inversion uses a wrapper for the `ProSailSimulator` application and some post-processing of the generated files in order to add vegetation indices, viewing angles⁴, etc. as predictor variables. The code is also able to select FCOVER or FAPAR as dependent variables, but this option is not relevant to the current version of the Sentinel-2 Agriculture processing chains.

⁴Although sensor and solar angles can be used as predictors, we have made the choice of using a different regression model per viewing configuration and therefore the angles are not used as predictors. This implies that the reflectance simulations have to be performed for every viewing configuration setting the angles correspondingly. Since Sentinel-2 products will contain angular grids, it is possible to implement different models within a single scene. However, no sensitivity analysis has been performed allowing to assess the usefulness of this approach. Since this does not introduce much complexity into the system, a tiling scheme within scenes taking into account a fixed angular step should be implemented.

```

1 def generateTrainingData(bvFile, simuPars, trainingFile, bvidx,
2     add_angles=False, red_index=0, nir_index=0):
3     """
4     Generate a training file using the file of biophysical vars (bvFile) and
5     the simulation parameters dictionary (simuPars).
6     Write the result to trainingFile. The first column will be the biovar to
7     learn and the following columns will be the reflectances.
8     The add_angles parameter is used to store the viewing and solar angles as
9     features. If red_index and nir_index are set, the ndvi and the rvi are also
10    used as features. red_index=3 means that the red reflectance is the 3rd
11    column (starting at 1) in the reflectances file.
12    """
13    app = otb.Registry.CreateApplication("ProSailSimulator")
14    app.SetParameterString("bvfile", bvFile)
15    app.SetParameterString("soilfile", simuPars['soilFile'])
16    app.SetParameterString("rsrfile", simuPars['rsrFile'])
17    app.SetParameterString("out", simuPars['outputFile'])
18    app.SetParameterFloat("solarzenith", simuPars['solarZenithAngle'])
19    app.SetParameterFloat("sensorzenith", simuPars['sensorZenithAngle'])
20    app.SetParameterFloat("azimuth", simuPars['solarSensorAzimuth'])
21    app.SetParameterFloat("noisevar", simuPars['noisevar'])
22    app.ExecuteAndWriteOutput()
23    #combine the bv samples, the angles and the simulated reflectances for
24    #variable inversion and produce the training file
25    with open(trainingFile, 'w') as tf:
26        with open(bvFile, 'r') as bvf:
27            bvf.readline() #header line
28            with open(simuPars['outputFile'], 'r') as rf:
29                #the output line follows the format:
30                #outputvar inputvar1 inputvar2 ... inputvarN
31                for (refline, bvline) in zip(rf.readlines(), bvf.readlines()):
32                    outline = ""
33                    if bvidx == bvindex["FCOVER"] :
34                        outline = string.split(refline)[-1]
35                    elif bvidx == bvindex["FAPAR"] :
36                        outline = string.split(refline)[-2]
37                    else:
38                        outline = string.split(bvline)[bvidx]
39                    outline = outline+" "+\
40                        string.join(string.split(refline[:-1])[:-2], ' ')
41                    outline.rstrip()
42                    if add_angles:
43                        angles = 'simuPars['solarZenithAngle'] '+' +\
44                            'simuPars['sensorZenithAngle'] '+' +\
45                            'simuPars['solarSensorAzimuth'] '+'
46                        outline += " "+angles
47                    outline += "\n"
48                    tf.write(outline)
49    if red_index!=0 and nir_index!=0:
50        # we need to add 1 to the indices since the file already contains the
51        #variable in the first column
52        addVI(trainingFile, red_index+1, nir_index+1)

```

The utility function to add the NDVI and the RVI to a reflectance ASCII file is like this:

```

1 def addVI(reflectances_file, red_index, nir_index):
2     rff = open(reflectances_file)
3     allfields = rff.readlines()
4     rff.close()
5     with open(reflectances_file, 'w') as rf:
6         for l in allfields:
7             rfls = string.split(l)
8             if len(rfls)>red_index:
9                 outline = string.join(string.split(l))
10                red = float(rfls[red_index-1])
11                pir = float(rfls[nir_index-1])
12                epsilon = 0.001
13                ndvi = (pir-red)/(pir+red+epsilon)
14                rvi = pir/(red+epsilon)
15                outline += " "+str(ndvi)+" "+str(rvi)+"\n"
16                rf.write(outline)

```

The learning the regression model uses the application wrapper:


```

1 def learnBVModel(trainingFile, outputFile, regressionType, normalizationFile, bestof=1):
2     app = otb.Registry.CreateApplication("InverseModelLearning")
3     app.SetParameterString("training", trainingFile)
4     app.SetParameterString("out", outputFile)
5     app.SetParameterString("errest", outputFile+"_errest")
6     app.SetParameterString("regression", regressionType)
7     app.SetParameterString("normalization", normalizationFile)
8     app.SetParameterInt("bestof", bestof)
9     app.ExecuteAndWriteOutput()

```

The LAI retrieval from ASCII files for the validation can be performed with the following function:

```

1 def invertBV(reflectanceFile, modelFile, normalizationFile, outputFile, \
2             removeFaparFcover=False, red_index=0, nir_index=0):
3     if removeFaparFcover:
4         #the reflectance file contains also the simulations of fapar and fcover
5         rff = open(reflectanceFile)
6         allfields = rff.readlines()
7         rff.close()
8         with open(reflectanceFile, 'w') as rf:
9             for l in allfields:
10                outline = string.join(string.split(l)[:2])+"\n"
11                rf.write(outline)
12
13     if red_index!=0 and nir_index!=0:
14         addVI(reflectanceFile, red_index, nir_index)
15
16
17     app = otb.Registry.CreateApplication("BVInversion")
18     app.SetParameterString("reflectances", reflectanceFile)
19     app.SetParameterString("model", modelFile)
20     app.SetParameterString("normalization", normalizationFile)
21     app.SetParameterString("out", outputFile)
22     app.ExecuteAndWriteOutput()

```

The use of the `BVImageInversion` application for images would be identical, but the removal of FAPAR and FCover and the addition of the vegetation indices should be implemented using the `BandMath` application which is standard in the ORFEO Toolbox distribution.

2.2 Phenological NDVI metrics

2.2.1 Overview of the subsystem

Figure 3 illustrates the processing chain for the production of the temporal NDVI metrics.

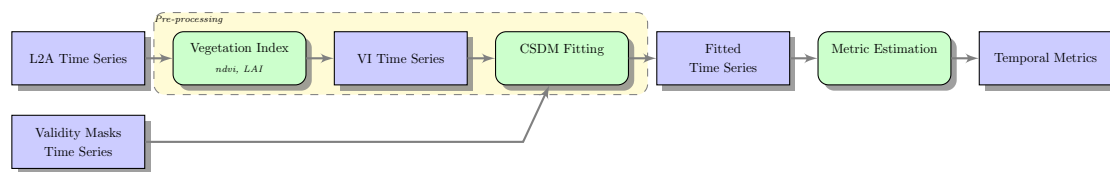


Figure 3: Block diagram for the estimation of the temporal NDVI metrics.

2.2.2 NDVI profile computation

The first step consists in computing the NDVI for each pixel of each date of the L2A image time series. For simplicity, no validity mask is used here. The NDVI computation will use the B8 band (not the B8a). Algorithm

1 describes de procedure.

Algorithm 1: NDVI time series computation

```

Data: tocr
Result: ndvi_ts
begin
  ndvi_ts  $\leftarrow \emptyset$ ;
  for image  $\in$  tocr do
    ndvi_image  $\leftarrow \emptyset$ ;
    for pixel  $\in$  image do
       $\text{ndvi\_image}[\text{positon}(\text{pixel})] \leftarrow \frac{\text{pixel}[\text{NIR}] - \text{pixel}[\text{R}]}{\text{pixel}[\text{NIR}] + \text{pixel}[\text{R}]}$ ;
    end
    ndvi_ts  $\leftarrow$  ndvi_ts + ndvi_image;
  end
end

```

2.2.3 Double logistic fitting

The second step consists in fitting a double logistic function (CSDM) to each pixel of the NDVI time series. The validity mask series (clouds, shadows, staurations, etc.) is used to discard the invalid NDVI values. Algorithm 2 describes de procedure.

Algorithm 2: Double logistic fitting

```

Data: tocr, mask
Result: csdm_ts
begin
  csdm_ts  $\leftarrow \emptyset$ ;
  for pixel  $\in$  tocr, mask_pixel  $\in$  mask do
    v_pixel  $\leftarrow$  remove_invalid_dates(pixel, mask_pixel);
    out_pixel  $\leftarrow$  csdm_fit(v_pixel);
    csdm_ts[positon(pixel)]  $\leftarrow$  out_pixel;
  end
end

```

This is done using the `SigmoFitting` application for which the `DoExecute()` method is implemented as follows:

```

1 void DoExecute()
2 {
3     // prepare the vector of dates
4     auto dates = pheno::parse_date_file(this->GetParameterString("dates"));
5     // pipeline
6     FloatVectorImageType::Pointer inputImage = this->GetParameterImage("in");
7     FloatVectorImageType::Pointer maskImage;
8     bool use_mask = true;
9     if(IsParameterEnabled("mask"))
10         maskImage = this->GetParameterImage("mask");
11     else
12     {
13         maskImage = inputImage;
14         use_mask = false;
15         otbAppLogINFO("No mask will be used.\n");
16     }
17     inputImage->UpdateOutputInformation();
18     maskImage->UpdateOutputInformation();
19     bool fit_mode = true;
20     unsigned int nb_out_bands = inputImage->GetNumberOfComponentsPerPixel();
21     if(IsParameterEnabled("mode") && GetParameterString("mode") == "params")
22     {
23         fit_mode = false;
24         nb_out_bands = 12;
25         otbAppLogINFO("Parameter estimation mode.\n");
26     }
27     using FunctorType =
28         pheno::TwoCycleSigmoidFittingFunctor<FloatVectorImageType::PixelType>;
29     filter = FilterType::New();
30     filter->SetInput(0, inputImage);
31     filter->SetInput(1, maskImage);
32     filter->GetFunctor().SetDates(dates);
33     filter->GetFunctor().SetUseMask(use_mask);
34     filter->GetFunctor().SetReturnFit(fit_mode);
35     filter->SetNumberOfOutputBands(nb_out_bands);
36     filter->UpdateOutputInformation();
37     SetParameterOutputImage("out", filter->GetOutput());
38 }

```

The core of the processing (the `csdm_fit` procedure of algorithm 2), is implemented in the `pheno::TwoCycleSigmoidFittingFunctor` class. The following type definitions are used:

```

1 using MinMaxType = std::pair<double, double>;
2 using CoefficientType = VectorType;
3 using ApproximationErrorType = double;
4 using ApproximationResultType = std::tuple<CoefficientType, MinMaxType,
5                                           VectorType, VectorType,
6                                           ApproximationErrorType>;

```

The interface of the `pheno::TwoCycleSigmoidFittingFunctor` is defined as follows:

```

1  template <typename PixelType>
2  class TwoCycleSigmoidFittingFunctor
3  {
4  protected:
5      std::vector<tm> dates;
6      VectorType dv;
7      bool return_fit;
8      bool fit_only_invalid;
9      bool use_mask;
10
11 public:
12     struct DifferentSizes {};
13     TwoCycleSigmoidFittingFunctor() : return_fit{true}, fit_only_invalid{true},
14                                     use_mask{true} {};
15
16     void SetDates(const std::vector<tm>& d) {
17         // ....
18     }
19     // ....
20     PixelType operator()(PixelType pix, PixelType mask)
21     {
22         // ....
23     }
24     // ....
25
26 };

```

The core of the processing is the `operator()` of the functor:

```

1 PixelType operator()(PixelType pix, PixelType mask)
2 {
3     auto nbDates = pix.GetSize();
4     auto tmp_mask = mask;
5     if(!use_mask)
6         tmp_mask.Fill(0);
7     if(dates.size()!=nbDates) throw DifferentSizes{};
8     PixelType tmppix{nbDates};
9     tmppix.Fill(typename PixelType::ValueType{0});
10    // If the mask says all dates are valid, keep the original value
11    if(tmp_mask == tmppix && fit_only_invalid && use_mask) return pix;
12    VectorType vec(nbDates);
13    VectorType mv(nbDates);
14    for(auto i=0; i<nbDates; i++)
15    {
16        vec[i] = pix[i];
17        mv[i] = tmp_mask[i];
18    }
19    // A date is valid if it is not NaN and the mask value == 0.
20    auto pred = [=](int e) { return !(std::isnan(vec[e])) &&
21        (mv[e]==(typename PixelType::ValueType{0})); };
22    auto f_profiles = filter_profile(vec, dates, pred);
23    decltype(vec) profile=f_profiles.first;
24    decltype(vec) t=f_profiles.second;
25    // If there are not enough valid dates, keep the original value
26    if(profile.size() < 4)
27    {
28        return pix;
29    }
30    auto approx = normalized_sigmoid::TwoCycleApproximation(profile, t);
31    auto x_1 = std::get<0>(std::get<1>(approx));
32    auto mm1 = std::get<1>(std::get<1>(approx));
33    auto x_2 = std::get<0>(std::get<2>(approx));
34    auto mm2 = std::get<1>(std::get<2>(approx));
35    if(return_fit)
36    {
37        // ...
38    }
39    else
40    {
41        PixelType result(12);
42        result[0] = (mm1.second-mm1.first);
43        result[1] = mm1.first;
44        for(auto i=0; i<4; ++i)
45            result[i+2] = x_1[i];
46
47        result[6] = (mm2.second-mm2.first);
48        result[7] = mm2.first;
49        for(auto i=0; i<4; ++i)
50            result[i+8] = x_2[i];
51
52        return result;
53    }
54 }

```

The approximation is done in the `normalized_sigmoid::TwoCycleApproximation(profile, t)` call:

```

1 template <ContainerC V, DateContainerC W>
2 std::tuple<VectorType, ApproximationResultType, ApproximationResultType>
3 TwoCycleApproximation(const V& profile, const W& t)
4 {
5     auto pca = PrincipalCycleApproximation(profile, t);
6     // Approximate the residuals
7     auto residual_approx = Approximation(std::get<3>(pca), t);
8     // The approximated profile is the sum of the approximations
9     auto yHat = std::get<2>(pca)+std::get<2>(residual_approx);
10    // Return {x, minmax, yHat, residuals, err};
11    return std::make_tuple(yHat, pca, residual_approx);
12 }

```

The `PrincipalCycleApproximation` can be implemented as follows:

```

1  template <ContainerC V, DateContainerC W>
2  ApproximationResultType PrincipalCycleApproximation(const V& profile,
3                                                     const W& t)
4  {
5      // Profile approximation
6      auto approx_result = Approximation(profile, t);
7      auto mm = std::get<1>(approx_result);
8      auto residuals = std::get<3>(approx_result)+mm.first;
9
10     // Residual approximation
11     approx_result = Approximation(residuals, t);
12
13
14     // Subtract the residual approximation from the original profile
15     auto yHat = std::get<2>(approx_result);
16     residuals = profile-yHat;
17
18     // Approx the original minus the residuals
19     approx_result = Approximation(residuals, t);
20
21     // Get the data to be returned
22     auto x = std::get<0>(approx_result);
23     auto minmax = std::get<1>(approx_result);
24     yHat = std::get<2>(approx_result);
25     residuals = profile-yHat;
26     auto err = std::get<4>(approx_result);
27
28     return ApproximationResultType{x, minmax, yHat, residuals, err};
29 }

```

And the approximation is obtained by using the Approximation function:

```

1  template <ContainerC V>
2  ApproximationResultType Approximation(const V& profile, const V& t)
3  {
4      auto minmax = std::minmax_element(std::begin(profile), std::end(profile));
5      auto t_max = std::begin(t);
6      std::advance(t_max, std::distance(std::begin(profile), minmax.second));
7      auto vmax = *(minmax.second);
8      auto vmin = *(minmax.first);
9      auto prof = (profile-vmin)/(vmax-vmin);
10     auto x(guesstimater(prof, t));
11     auto fprofile = gaussianWeighting(prof, t, *t_max, 75.0);
12     auto phFs(F<V>);
13     ParameterCostFunction fs{4, t.size(), fprofile, t, phFs};
14     auto err(optimize(x, fs));
15     auto yHat(phFs(t,x));
16     auto residuals = prof - yHat;
17     yHat = yHat*(vmax-vmin)+vmin;
18     residuals = residuals*(vmax-vmin);
19     auto mm = std::make_pair(vmin, vmax);
20     return ApproximationResultType{x, mm, yHat, residuals, err};
21 }

```

The guesstimater() function initializes the values of the parameters:

```

1  template <ContainerC V>
2  inline
3  V guesstimater(V p, V t)
4  {
5      // Find the max position
6      auto pmax = std::max_element(std::begin(p), std::end(p)) - std::begin(p);
7      V x0(4);
8      x0[0] = t[pmax] - 25;
9      x0[1] = 10.0;
10     x0[2] = t[pmax] + 25;
11     x0[3] = 10.0;
12     return x0;
13 }

```

The gaussianWeighting() function reduces the values far from the principal peak of the profile:

```

1  template <ContainerC V>
2  inline
3  V gaussianWeighting(const V& p, const V& t, double m, double s)
4  {
5      auto fp = p;
6      for(auto i=0; i<t.size(); i++)
7      {
8          fp[i] *= gaussianFunction(t[i],m,s);
9      }
10
11     return fp;
12 }

```

And the `gaussianFunction()` is just the well known Gaussian:

```

1  double gaussianFunction(double x, double m, double s)
2  {
3      return exp(-vnl_math_sqr((x-m)/(s)/2.0));
4  }

```

The parameters are optimized using a Levenberg Marquardt optimizer available in the Orfeo Toolbox via the VNL libraries:

```

1  double optimize(VectorType& x, ParameterCostFunction f)
2  {
3      vnl_levenberg_marquardt levmarq(f);
4      levmarq.set_f_tolerance(1e-10);
5      levmarq.set_x_tolerance(1e-100);
6      levmarq.set_g_tolerance(1e-100);
7      levmarq.minimize(x);
8      return levmarq.get_end_error();
9  }

```

The `ParameterCostFunction` is

```

1  class ParameterCostFunction : public vnl_least_squares_function
2  {
3  public:
4      ParameterCostFunction(unsigned int nbPars, unsigned int nbD,
5                          const VectorType& yy, const VectorType& tt,
6                          FunctionType func) :
7          vnl_least_squares_function(nbPars, nbD, no_gradient), nbDates(nbD), y(yy),
8          t(tt), phenoFunction(std::move(func)) {}
9
10     inline
11     void f(const VectorType& x, VectorType& fx) override
12     {
13         auto yy = phenoFunction(t, x);
14         for(auto i=0; i<nbDates; ++i)
15             fx[i] = (y[i] - yy[i]);
16     }
17
18     private:
19         VectorType y;
20         VectorType t;
21         unsigned int nbDates;
22         FunctionType phenoFunction;
23 };

```

And in our case, the function to be optimized is

```

1 namespace normalized_sigmoid{
2 template <typename T, ContainerC V>
3 inline
4 T double_sigmoid(T t, const V& x)
5 {
6     return 1.0/(1.0+exp((x[0]-t)/x[1]))-1.0/(1.0+exp((x[2]-t)/x[3]));
7 }
8
9 template <ContainerC V>
10 inline
11 V F(const V& t, const V& x)
12 {
13     auto tsize(t.size());
14     V y(tsize);
15     for(auto i=0; i<tsize; ++i)
16         y[i] = double_sigmoid(t[i], x);
17     return y;
18 }
19 }

```

2.2.4 Metric estimations

The following phenological parameters were selected as meaningful to describe the vegetation status: emergence date, date of the maximum growth, length of the maturity plateau and senescence date.

The logistic function has the form:

$$f(x) = \frac{1}{1 + e^{\frac{x_0 - x}{x_1}}} \quad (1)$$

The double logistic is:

$$g(x) = A(f_1(x) - f_2(x)) + B = A \left(\frac{1}{1 + e^{\frac{x_0 - x}{x_1}}} - \frac{1}{1 + e^{\frac{x_2 - x}{x_3}}} \right) + B, \quad (2)$$

where $A + B$ is the maximum value and B is the minimum.

Since

$$\frac{df(x)}{dx} = \frac{e^{\frac{x_0 - x}{x_1}}}{x_1 \left(1 + e^{\frac{x_0 - x}{x_1}} \right)^2} \quad (3)$$

we have

$$g'(x) = \frac{dg(x)}{dx} = A \left(\frac{e^{\frac{x_0 - x}{x_1}}}{x_1 \left(1 + e^{\frac{x_0 - x}{x_1}} \right)^2} - \frac{e^{\frac{x_2 - x}{x_3}}}{x_3 \left(1 + e^{\frac{x_2 - x}{x_3}} \right)^2} \right) \quad (4)$$

2.2.4.1 Date of the maximum positive gradient By definition, this is x_0 .

2.2.4.2 Starting date The date for which the straight line with the slope of x_0 intercepts the horizontal axis. If

$$g'(x_0) = m \quad (5)$$

this line's equation is

$$y = mx + b \quad (6)$$

and verifies that

$$\begin{aligned} 0 &= mx_0 + b \\ g(x_0) &= mx_0 + b \end{aligned} \quad (7)$$

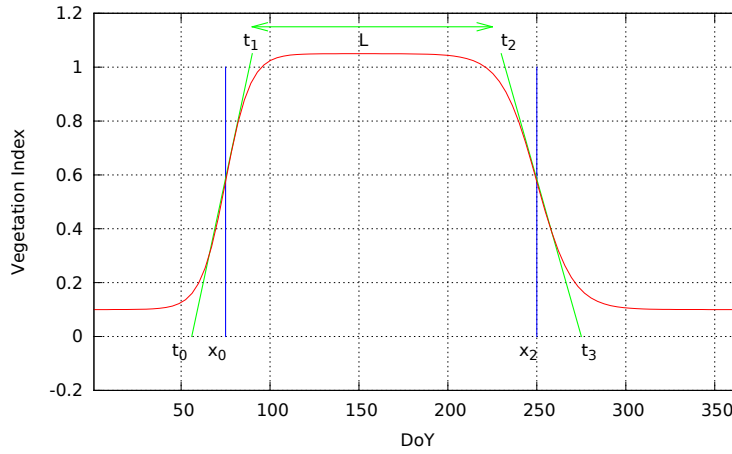


Figure 4: Double logistic function and associated parameters

which gives

$$y = g'(x_0)x + (g(x_0) - g'(x_0)x_0) \quad (8)$$

and therefore,

$$t_0 = \frac{mx_0 - g(x_0)}{m} = x_0 - \frac{g(x_0)}{g'(x_0)} \quad (9)$$

2.2.4.3 Length of the plateau We define t_1 as the date for which the previous straight line reaches the maximum value.

$$t_1 = \frac{A + B - (g(x_0) - g'(x_0)x_0)}{g'(x_0)} \quad (10)$$

Similarly, we can use the straight line associated to the descending slope:

$$y = g'(x_2)x + (g(x_2) - g'(x_2)x_2) \quad (11)$$

and define

$$t_2 = \frac{A + B - (g(x_2) - g'(x_2)x_2)}{g'(x_2)} \quad (12)$$

And the length of the plateau is:

$$L = t_2 - t_1 \quad (13)$$

2.2.4.4 Senescence date In a similar way than for t_0 , we obtain:

$$t_3 = x_2 - \frac{g(x_2)}{g'(x_2)} \quad (14)$$

2.2.4.5 Metrics estimation implementation Using the definitions above, algorithm 3 describes the metric estimation loop.

Algorithm 3: NDVI time metrics computation

Data: *csdm_ts*
Result: *metrics_image*
begin
 metrics_image $\leftarrow \emptyset$;
 for *pixel* \in *csdm_ts* **do**
 out_pixel \leftarrow *pheno_metrics(pixel)*;
 metrics_image[*position(pixel)*] \leftarrow *out_pixel*;
 end
end

This algorithm is implemented in terms of several functions. The derivative of the double logistic is:

```

1  template <typename T>
2  inline
3  T diff_sigmoid(T t, T x0, T x1)
4  {
5      auto b = exp((x0-t)/x1);
6      return 1.0/(x1*(1+1/b)*(1+b));
7  }
8
9  template <typename T, ContainerC V>
10 inline
11 T diff_double_sigmoid(T t, const V& x)
12 {
13     return diff_sigmoid(t, x[0], x[1])-diff_sigmoid(t, x[2], x[3]);
14 }
```

And, finally the metrics estimation is just a straightforward FORMula TRANslation, but in C++:

```

1  template <typename T, ContainerC V>
2  inline
3  std::tuple<T, T, T, T, T, T> pheno_metrics(const V& x, T maxvalue=1.0,
4                                             T minvalue=0.0)
5  {
6      auto A = maxvalue-minvalue;
7      auto B = minvalue;
8      auto gx0 = A*double_sigmoid(x[0], x)+B;
9      auto dgx0 = A*diff_double_sigmoid(x[0], x);
10     auto gx2 = A*double_sigmoid(x[2], x)+B;
11     auto dgx2 = A*diff_double_sigmoid(x[2], x);
12
13     auto t0 = x[0] - gx0/dgx0;
14     auto t1 = (A+B-(gx0-dgx0*x[0]))/dgx0;
15     auto t2 = (A+B-(gx2-dgx2*x[2]))/dgx2;
16     auto t3 = x[2] - gx2/dgx2;
17
18     return std::make_tuple(dgx0, t0, t1, t2, t3, dgx2);
19 }
```

3 Appendix

3.1 The OTB-BV library

This section describes the C++ library which implements the low level details of the LAI retrieval. This library makes extensive use of the ORFEO Toolbox.

3.1.1 Type definitions

A definition of a set of types is given in `otbBVTypes.h`:

```

1 namespace otb
2 {
3     enum class IVNames {MLAI, ALA, CrownCover, HsD, N, Cab, Car, Cdm, CwRel, Cbp,
4         Bs, IVNamesEnd};
5     enum AcquisitionParameters {TTS, TTO, PSI, TTS_FAPAR, AcquisitionParametersEnd};
6
7     using AcquisitionParsType = std::map< AcquisitionParameters, double >;
8     using PrecisionType = double;
9     using BVType = std::map< IVNames, PrecisionType >;
10
11     using NormalizationVectorType =
12         std::vector<std::pair<PrecisionType, PrecisionType>>;
13 }

```

Solar irradiance values for the FAPAR computation in the simulations (not needed for LAI) are given in `otbSolarIrradianceFAPAR.h` :

```

1 namespace otb{
2 static
3 std::vector<std::pair<PrecisionType, PrecisionType>> solar_irradiance_fapar = {
4     {0.4000, 1614.0900},
5     {0.4025, 1631.0900},
6     {0.4050, 1648.0900},
7     // etc.
8     {0.7000, 1408.2400}
9 };
10 }
11 #endif

```

3.1.2 Utilities

A utility function for counting columns in an ASCII file is given in `otbBVUtil.h`. It is used to automatically get the number of predictor variables in the non-linear regression procedures.

```

1 namespace otb
2 {
3     unsigned short int countColumns(std::string fileName)
4     {
5         std::ifstream ifile(fileName.c_str());
6         std::string line;
7         auto nbSpaces = 0;
8         if (ifile.is_open())
9         {
10             getline(ifile,line);
11             ifile.close();
12             boost::trim(line);
13             auto found = line.find(' ');
14             while(found!=std::string::npos)
15             {
16                 ++nbSpaces;
17                 found = line.find(' ', found+1);
18             }
19             return nbSpaces+1;
20         }
21         else
22         {
23             itkGenericExceptionMacro(<< "Could not open file " << fileName);
24         }
25     }
26 }
27

```

3.1.3 Simulation

The file `otbProSailSimulatorFunctor.h` contains the definition of a functor which performs the simulation of the reflectances for a pixel. The simulation is delegated to the PROSPECT and Sail implementations available in the ORFEO Toolbox.

The interface of the class is as follows:

```

1 namespace otb
2 {
3     namespace Functor
4     {
5
6         template <class TSatRSR, unsigned int SimNbBands = 2000>
7         class ProSailSimulator
8         {
9         public:
10             /** Standard class typedefs */
11             typedef TSatRSR SatRSRType;
12             typedef typename SatRSRType::Pointer SatRSRPointerType;
13             typedef typename otb::ProspectModel ProspectType;
14             typedef typename otb::LeafParameters LeafParametersType;
15             typedef typename LeafParametersType::Pointer LeafParametersPointerType;
16             typedef typename otb::SailModel SailType;
17
18             typedef typename SatRSRType::PrecisionType PrecisionType;
19             typedef std::pair<PrecisionType,PrecisionType> PairType;
20             typedef typename std::vector<PairType> VectorPairType;
21             typedef otb::SpectralResponse< PrecisionType, PrecisionType> ResponseType;
22             typedef otb::ReduceSpectralResponse < ResponseType,SatRSRType>
23                                     ReduceResponseType;
24             typedef typename std::vector<PrecisionType> OutputType;
25
26             inline
27             OutputType operator ()()
28             protected:
29
30                 double ComputeFAPAR(SailType::SpectralResponseType* absorptance);
31                 /** Satellite Relative spectral response*/
32                 SatRSRPointerType m_SatRSR;
33                 LeafParametersPointerType m_LP;
34                 double m_LAI; //leaf area index
35                 double m_Angl; //average leaf angle
36                 double m_PSoil; //soil coefficient
37                 double m_Skyl; //diffuse/direct radiation
38                 double m_HSpot; //hot spot
39                 double m_TTS; //solar zenith angle
40                 double m_TTS_FAPAR; //solar zenith angle for fapar computation
41                 double m_TTO; //observer zenith angle
42                 double m_PSI; //azimuth
43                 BVType m_BV;
44             };
45
46         }
47     }

```

The processing is done in the `operator()` member which is implemented as follows:

```

1  inline
2  OutputType operator ()()
3  {
4      OutputType pix;
5      for(auto i=0;i<m_SatRSR->GetNbBands();i++)
6          pix.push_back(0.0);
7      // ....
8      auto prospect = ProspectType::New();
9      // ....
10     auto refl = prospect->GetReflectance()->GetResponse();
11     auto trans = prospect->GetTransmittance()->GetResponse();
12     auto sail = SailType::New();
13     // ....
14     auto sailSim = sail->GetViewingReflectance()->GetResponse();
15     auto fCover = sail->GetFCoverView();
16     auto sail_fapar = SailType::New();
17     // ....
18     auto fAPAR = this->ComputeFAPAR(sail_fapar->GetViewingAbsorptance());
19     VectorPairType hxSpectrum;
20     for(auto i=0;i<SimNbBands;i++)
21     {
22         PairType resp;
23         resp.first = static_cast<PrecisionType>((400.0+i)/1000);
24         resp.second = sailSim[i].second;
25         hxSpectrum.push_back(resp);
26     }
27     auto aResponse = ResponseType::New();
28     aResponse->SetResponse( hxSpectrum );
29     auto reduceResponse = ReduceResponseType::New();
30     reduceResponse->SetInputSatRSR(m_SatRSR);
31     reduceResponse->SetInputSpectralResponse( aResponse );
32     reduceResponse->SetReflectanceMode(true);
33     reduceResponse->CalculateResponse();
34     for(auto i=0;i<m_SatRSR->GetNbBands();i++)
35         pix[i] = (*reduceResponse)(i);
36     pix.push_back(fCover);
37     pix.push_back(fAPAR);
38     return pix;
39 }

```

The commented (`//`) sections above correspond to the parameter settings for PROSPECT and Sail. For the leaf parameters (PROSPECT) we have:

```

1  auto m_LP = LeafParametersType::New();
2  m_LP->SetCab(m_BV[IVNames::Cab]);
3  m_LP->SetCar(m_BV[IVNames::Car]);
4  m_LP->SetCBrown(m_BV[IVNames::Cbp]);
5  double Cw = m_BV[IVNames::Cdm]/(1.-m_BV[IVNames::CwRel]);
6  if(Cw<0) Cw = 0.0;
7  m_LP->SetCw(Cw);
8  m_LP->SetCm(m_BV[IVNames::Cdm]);
9  m_LP->SetN(m_BV[IVNames::N]);

```

And the Sail parameters are set as follows:

```

1  m_LAI = m_BV[IVNames::MLAI];
2  m_Angl = m_BV[IVNames::ALA];
3  m_PSoil = m_BV[IVNames::Bs];
4  m_Skyl = 0.3;
5  m_HSpot = m_BV[IVNames::HsD];
6  auto sail = SailType::New();
7  sail->SetLAI(m_LAI);
8  sail->SetAngl(m_Angl);
9  sail->SetPSoil(m_PSoil);
10 sail->SetSkyl(m_Skyl);
11 sail->SetHSpot(m_HSpot);
12 sail->SetTTS(m_TTS);
13 sail->SetTTO(m_TTO);
14 sail->SetPSI(m_PSI);
15 sail->SetReflectance(prospect->GetReflectance());
16 sail->SetTransmittance(prospect->GetTransmittance());
17 sail->Update();

```

3.1.4 Regression

The non-linear regression is implemented using Neural Networks or Support Vector Machines. In both cases, the OpenCV implementation available through the ORFEO Toolbox is used. However, the ORFEO Toolbox wrapper for the `NeuralNetworkMachineLearningModel` does not allow to perform regression (only classification is provided), and therefore, a `NeuralNetworkRegressionMachineLearningModel` has been implemented.

It boils down to inheriting and overloading the `Predict` method in order to do regression:

```

1 namespace otb
2 {
3 template <class TInputValue, class TTargetValue>
4 class ITK_EXPORT NeuralNetworkRegressionMachineLearningModel
5 : public NeuralNetworkMachineLearningModel <TInputValue, TTargetValue>
6 {
7 // ....
8 };
9 }

1 template<class TInputValue, class TOutputValue>
2 typename
3 NeuralNetworkRegressionMachineLearningModel<TInputValue,
4 TOutputValue>::TargetSampleType
5 NeuralNetworkRegressionMachineLearningModel<TInputValue,
6 TOutputValue>::Predict(
7 const
8 InputSampleType & input) const
9 {
10 //convert listsample to Mat
11 cv::Mat sample;
12 otb::SampleToMat<InputSampleType>(input, sample);
13 cv::Mat response; //(1, 1, CV_32FC1);
14 m_ANNModel->predict(sample, response);
15 TargetSampleType target;
16 target[0] = response.at<float>(0, 0);
17 return target;
18 }

```

3.1.5 Reprocessing

Reprocessing functions working on time profiles are defined in `otbProfileReprocessing.h`.

3.1.5.1 On-line reprocessing The on-line reprocessing uses a local window defined by its backward radius `bwd_radius` and its forward radius `fwd_radius`. For example, if $(bwd_radius, fwd_radius) = (1, 1)$ the window contains 3 dates and is centered on the middle one.

The `smooth.time.series.local.window.with.error` takes as input a date vector `dts`, the vector with the measurement values (i.e. LAI) `ts`, and the vector of the estimated errors `ets`. For each reprocessed date, the new value is computed using a weighted linear combination of the values inside the window.

```

1  std::pair<VectorType, VectorType>
2  smooth_time_series_local_window_with_error(VectorType dts,
3                                             VectorType ts,
4                                             VectorType ets,
5                                             size_t bwd_radius = 1,
6                                             size_t fwd_radius = 1)
7  {
8
9      /**
10         -----
11         /           /           /
12         win_first  current  win_last
13     */
14     assert(ts.size()==ets.size() && ts.size()==dts.size());
15     auto result = ts;
16     auto result_flag = ts;
17     auto ot = result.begin();
18     auto otf = result_flag.begin();
19     auto eit = ets.begin();
20     auto last = ts.end();
21     auto win_first = ts.begin();
22     auto win_last = ts.begin();
23     auto e_win_first = ets.begin();
24     auto e_win_last = ets.begin();
25     auto dti = dts.begin();
26     auto d_win_first = dts.begin();
27     auto d_win_last = dts.begin();
28     *otf = not_processed_value;
29     //advance iterators
30     std::advance(ot, bwd_radius);
31     std::advance(otf, bwd_radius);
32     std::advance(eit, bwd_radius);
33     std::advance(dti, bwd_radius);
34     std::advance(win_last, bwd_radius+fwd_radius);
35     std::advance(e_win_last, bwd_radius+fwd_radius);
36     std::advance(d_win_last, bwd_radius+fwd_radius);
37     while(win_last!=last)
38     {
39         auto current_d = d_win_first;
40         auto current_e = e_win_first;
41         auto current_v = win_first;
42         auto past_it = d_win_last; ++past_it;
43
44         PrecisionType sum_weights{0.0};
45         PrecisionType weighted_value{0.0};
46         while(current_d != past_it)
47         {
48             auto cw = compute_weight(fabs(*current_d*dti),fabs(*current_e));
49             sum_weights += cw;
50             weighted_value += (*current_v)*cw;
51             ++current_d;
52             ++current_e;
53             ++current_v;
54         }
55         *ot = weighted_value/sum_weights;
56         *otf = processed_value;
57         ++win_first;
58         ++win_last;
59         ++e_win_first;
60         ++e_win_last;
61         ++d_win_first;
62         ++d_win_last;
63         ++ot;
64         ++otf;
65         ++eit;
66         ++dti;
67     }
68     *otf = not_processed_value;
69     return std::make_pair(result,result_flag);
70 }

```

The compute weight function is:

```

1  template <typename T>
2  inline
3  T compute_weight(T delta, T err)
4  {
5      T one{1};
6      return (one/(one+delta)+one/(one+err));
7  }

```

3.1.5.2 Fit CSDM The double logistic fitting is performed using functions provided by the **pheno** library which is described in section 2.2. These functions are used by the **fit_csdm** function below, which takes as input a date vector **dts**, the vector with the measurement values (i.e. LAI) **ts**. A third parameter **ets** corresponding to the vector of the estimated errors is not used in the current version of the function.

The function returns a **std::pair** of vectors containing the fitted profile and a vector of flags for each date indicating whether or not each date has been processed.

```

1  std::pair<VectorType, VectorType>
2  fit_csdm(VectorType dts, VectorType ts, VectorType ets)
3  {
4      assert(ts.size()==ets.size() && ts.size()==dts.size());
5      auto result = ts;
6      auto result_flag = ts;
7      // std::vector to vnl_vector
8      pheno::VectorType profile_vec(ts.size());
9      pheno::VectorType date_vec(dts.size());
10
11     for(auto i=0; i<ts.size(); i++)
12     {
13         Profile_vec[i] = ts[i];
14         date_vec[i] = dts[i];
15     }
16
17     // fit
18     auto approximation_result =
19         pheno::normalized_sigmoid::TwoCycleApproximation(profile_vec, date_vec);
20     auto princ_cycle = std::get<1>(approximation_result);
21     auto x_hat = std::get<0>(princ_cycle);
22     auto min_max = std::get<1>(princ_cycle);
23     auto A_hat = min_max.second - min_max.first;
24     auto B_hat = min_max.first;
25     auto p = pheno::normalized_sigmoid::F(date_vec, x_hat);
26     //fill the result vectors
27     for(auto i=0; i<ts.size(); i++)
28     {
29         result[i] = p[i]*A_hat+B_hat;
30         result_flag[i] = processed_value;
31     }
32
33     return std::make_pair(result,result_flag);
34 }

```