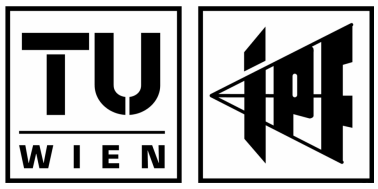
ASCAT Soil Moisture Report Series No. 6

Selection of Resampling Procedure

VIENNA UNIVERSITY OF TECHNOLOGY
INSTITUTE OF PHOTOGRAMMETRY AND REMOTE SENSING



2005 Sep 7

Status:	Issue 2.0
Authors:	IPF TU Wien (ZB)
Circulation:	IPF, EUMETSAT
Amendments:	

Issue	***Date***	***Details***	***Editor***
Issue 1.0	2005 July 12	Initial Document.	ZB
Issue 2.0	2005 Sep 7	After review notes by Hans Bonekamp	ZB

If further corrections are required please contact Zoltan Bartalis (zb@ipf.tuwien.ac.at).

Abstract

The present document constitutes the report of Working Package 2 of the project *Processor for ERS-SCAT-based Soil Moisture*. The proposal was submitted by the Institute of Photogrammetry and Remote Sensing (I.P.F.) at Vienna University of Technology (TU Wien) as a response to EUMETSAT's Request for Quotation 05/934. The objective of the project is to develop a demonstration software application for near real time (NRT) surface soil moisture retrieval from ERS-1/2 scatterometer (ESCAT) data, using version 4.0 of I.P.F.'s WARP (soil WAter Retrieval Package) processing software.

Working Package 2 deals with the spatial resampling of the global soil moisture retrieval parameter database existent at the I.P.F. to the satellite swath geometry of the ESCAT data. This processing step is required in order to calculate surface soil moisture from the backscatter measurements which are input in the system on a NRT basis. In this report we present the characteristics of the ESCAT Discrete Global Grid (DGG) used in WARP 4.0, the soil moisture parameter input files as well as the orbit geometry of the ESCAT instrument. We then develop an algorithm involving finding the spatial neighbourhood of the incoming ESCAT measurement location and resampling the nearest DGG parameters using a Hamming apodisation function. Suggestions for improvements of future versions of the WARP DGG are also given.

Contents

1 Introduction 1
2 The WARP 4.0 DGG and Parameter Database 3
2.1 The Grid 3
2.2 The Parameter Database 4
3 The ESCAT Data 5
3.1 Measurement Geometry 5
3.2 Data Format 5
4 The Resampling Procedure 7
4.1 Finding Grid Points Closest to a Node 7
4.2 Interpolating the Nearest Parameter Values 9
5 Future Work 16
Acknowledgements 18
Bibliography 19
Appendix A: C++ Program Code 20

1 Introduction

Since 1994 the Institute of Photogrammetry and Remote Sensing (I.P.F.) at Vienna University of Technology (TU Wien) is actively involved in deriving soil moisture data using scatterometer measurements from the scatterometers onboard the ERS-1 and ERS-2 satellites. For convenience, in the present document we will refer to the scatterometers as ESCAT. A result of this involvement is the development of the WARP (soil WAter Retrieval Package) processing chain, based on the soil moisture retrieval method developed by Wagner (1998). The method is based on long-term time series of ESCAT data using a change detection algorithm tailored to the sensor characteristics. The algorithm exploits the multiple viewing capabilities of the sensor in order to separate soil moisture and vegetation effects.

The TU Wien-model requires two basic steps. In the first step a set of parameters are retrieved, describing the specific backscatter properties (influence of vegetation, viewing geometry, noise, etc.) for each point of the Earth land surface. In the second step, these quasi-static (updated every several months/years only) parameters are used for the subsequent retrieval of soil moisture information. The retrieval of the backscatter properties involves long-term time series analysis. ESCAT σ^0 backscattering coefficient measurements have therefore been spatially aggregated into sets of regions (so-called grid areas) that partition the surface of the Earth in an approximately regular manner. Such regions form a Discrete Global Grid (DGG), as described in depth in Sahr et al. (2003). Each defined grid area is associated with time series of backscatter measurements and holds its own entry in the backscatter metadata database.

With the proposal *Processor for ERS-SCAT-based Soil Moisture* submitted to EUMETSAT, the I.P.F. has undertaken to develop a demonstration application software (called WARP^{NRT} 1.0) in which the TU Wien-method is applied in near real-time (NRT) mode to incoming ESCAT backscatter measurements, demonstrating the NRT generation of surface soil moisture data. In opposition to the long time series-based processing case of the latest version of the WARP software (v. 4.0), in the NRT case the incoming backscatter data are distributed spatially according to the geometry of the sensor swath, whereas the parameter database follows the geometry of the predefined DGG. This yields the problem of resampling the parameters in the closest neighbourhood of the incoming σ^0 location before the NRT processing can continue. This

report describes the solution chosen to solve this problem. Section 2 describes the DGG used in WARP 4.0 and the parameter database associated with it, while Section 3 presents the geometry of the ESCAT measurements within the sensor swath. In Section 4 the chosen resampling method is described. Additionally, Section 5 assesses aspects of the DGG that need to be improved ahead of future versions of the WARP software, which will also include data from the upcoming ASCAT scatterometer onboard the MetOp satellite series.

2 The WARP 4.0 DGG and Parameter Database

2.1 The Grid

The discrete global grid used in WARP 4.0 for ESCAT data aggregation is an adapted version of a sinusoidal global grid. First, a series of latitude small circles are created, equally spaced with a central angle of 0.25° in the south-north direction along any meridian, starting with the south pole (Fig. 2–1 a). A spherical earth with radius r = 6370 km is assumed, yielding a constant spacing between the latitude circles of $6370 \cdot 0.25 \cdot \pi / 180 \approx 27.79$ km, required by the processing of ESCAT data (Scipal 2002). Equation 2–1 gives the latitude of each such small circle.

$$\lambda_j = -90 + 0.25 \cdot (j - 0.5), \qquad 1 \le j < 4 \cdot 180, \quad \mathrm{j} \in \mathrm{N} \qquad (2\text{–}1)$$

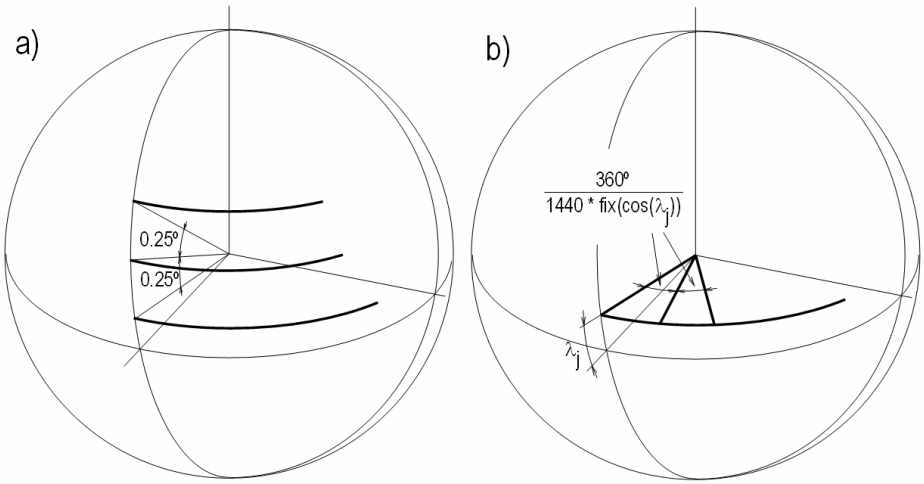


Figure 2–1.
Definition of the WARP 4.0 discrete global grid. The 'fix' function denotes decimal truncation.

In the longitudinal direction, the Equator is also divided into 0.25° longitude intervals, giving $4 \cdot 360 = 1440$ divisions. Each latitude circle is then subsequently divided into $1440 \cdot \cos \lambda_j$ divisions, ensuring the same 27.8 km spacing in the west-east direction as well, subject to slight variations due to decimal truncation (see Fig. 2–1 b). The number of grid points on each latitude circle decreases thus with increasing

latitude. Equation 2–2 gives the longitude of the grid points on each of the latitude circles:

$$\varphi_{i,j} = -180 + 0.25 \cdot \frac{(i - 0.5)}{\cos \lambda_j}, \quad 1 \le i < 4 \cdot 360 \cdot \cos \lambda_j, \quad \mathrm{i, j} \in \mathrm{N} \qquad (2\text{–}2)$$

The presented method covers the land surface of the Earth with more than 180000 single grid areas. Coastal zones and inland water bodies are excluded from the analysis.

2.2 The Parameter Database

The structure of parameters required for subsequent retrieval of soil moisture and associated with each of the grid points is the following (for more details on the role of each parameter, see Wagner (1998)):

- *LON* : the longitude of the grid point
- *LAT* : the latitude of the grid point
- *ESD* : the σ^0 estimated standard deviation
- *SLO* : a structure containing:
  - *NAME* : the name of the slope periodic function
  - *C* : the first coefficient of the slope periodic function
  - *D* : the second coefficient of the slope periodic function
  - *PHASE* : the third coefficient of the slope periodic function
- *CURV* : a structure containing:
  - *NAME* : the name of the curvature periodic function
  - *C* : the first coefficient of the curvature periodic function
  - *D* : the second coefficient of the curvature periodic function
  - *PHASE* : the third coefficient of the curvature periodic function
- *NOISE_SLO* : the $\varepsilon_{\sigma'}$ noise
- *NOISE_S40* : the $\varepsilon_{\sigma^0_{40}}$ noise
- *DRY* : the dry reference backscatter
- *WET* : the wet reference backscatter

3 The ESCAT Data

3.1 Measurement Geometry

The ESCAT instrument consists of three antennae producing three beams looking 45° forward, sideways (90°), and 45° backward with respect to the satellite's motion direction along the orbit (see Fig. 3–1). The measurements from each beam consist of 19 so-called nodes spaced 25 km apart. As the satellite beams sweep along the earth surface yielding an approximately 500 km wide swath, each node produces its own σ^0 measurement, integrated over an area around 50 km in diameter. The three measurements originating in the three beams during the same satellite overpass are called triplets.

3.2 Data Format

At the time of writing the choice of the data format of the ESCAT data to be fed into the WARP^{NRT} 1.0 processor is not yet known. Since the software package has a demonstrative nature, we believe it is both reasonable and sufficient to perform the NRT simulation using ESA's Advanced Scatterometer Processing System (ASPS) product format at nominal (50 km) resolution. Data in this format is the result of a complete, state-of-the-art reprocessing of the entire ESCAT dataset with the scope of providing the large scientific communities active in novel scatterometer applications with high-quality and homogenous scatterometer data (Crapolicchio et al. 2004). Ultimately, the WARP^{NRT} 1.0 processor should be able to deal with individual inputs of backscatter triplets, the only requirement being that the geographical location of the triplet (node) and acquisition time is known.

ASPS data is organised in files containing Level 2 data for each satellite orbit, from ascending orbital node to ascending orbital node. Apart from the backscatter triplets for each swath node, the latitude/longitude location of the node and the acquisition time, the data comes with several useful flags and control bytes. These can be so-called Node Confidence Data (NCD, indicating incomplete or corrupt acquisitions, lacking Doppler compensation, so-called arcing and cali-

bration problems, etc.) or Geophysical Product Confidence Data (GPC, indicating whether a node falls onto land or ocean, etc.). Evidently, it is desirable to use this control data already in the grid-to-orbit resampling phase to find and eliminate nodes with undesired location or with erroneous or incomplete σ^0.

The complete structure of the ASPS data format can be found in Crapolicchio et al. (2004) or on the ASPS evaluation CD-ROMs.

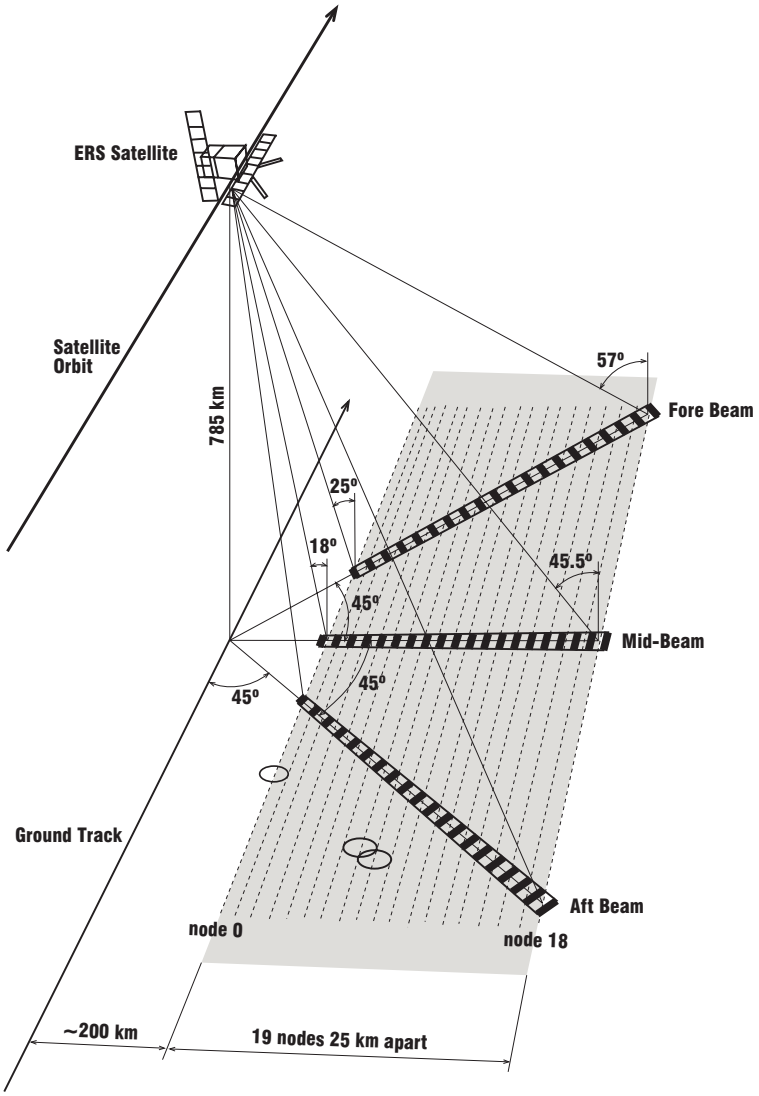


Figure 3-1.

Viewing geometry and sensor swath of the ESCAT scatterometer.

4 The Resampling Procedure

As mentioned earlier, the purpose of the present work is to develop a method so that for each incoming σ^0 triplet and its corresponding swath node 1) we find all the grid points of the parameter database within a certain search radius and 2) if there are enough such neighbours, their parameter values are interpolated for the location of the node in question.

4.1 Finding Grid Points Closest to a Node

Because the processing should run in NRT, and given the large number of grid points, finding the subset of the database structures that represent the close spatial neighbourhood of an orbit node location must be done in an efficient way. We propose a two-step approach for finding the close neighbours of a swath node, tailored to the DGG described in Section 2.1:

- Use indexing of the database according to latitude and longitude for finding the n closest latitude circles on both sides of the node in question. Then, going east and west on each of these latitude circles, find the n closest grid points in each direction, thus extracting a rectangular-shaped subset of the parameter database.
- Calculate the metric distances (spherical arc lengths) between these up to $2n \times 2n$ points and the node in question. Make a final selection of those grid points for which the distance is less than a certain search radius $r_{\max}$.

The latitude–longitude indexing of the parameter database could be done by reordering it to a two-dimensional array of structures, according to which latitude circle and longitude position the corresponding grid points are situated at. As discussed previously, at higher latitudes there are significantly less parameter structures to store than at the Equator. Because neither the C++ or IDL programming languages support two-dimensional arrays with variable row/column length, a good

way to avoid storing a lot of void structures (a fixed array would have 720×1440 elements) is to keep the parameter structures in a one-dimensional array and only store the corresponding indices in a 720×1440 array (see Fig. 4–1). When filling this array with indices, one can refer to all grid points without valid parameters (over oceans, etc.) with a not-a-number (NaN) index value or an index that is larger than the number of elements in the parameter array.

With this index array, given the latitude and longitude of a swath node, finding the neighbouring latitude circles on both the northern and southern side of the node is straightforward, by solving Equation 2–1 for the index j. Similarly, for finding the nearest neighbours on a latitude circle, Equation 2–2 has to be solved for the index i. Care has to be taken though when trying to find the rectangular neighbourhood of locations at the edge or outside the extent of the parameter grid. In north-south direction, the lowest and highest latitude present in the parameter database is -54.875 and 83.625 respectively. Allowing for the buffer zones a selection of $n = 3$ and $r_{\max} = 36$ km (see Section 4.2) would create, we can reasonably set the useful node latitude range to $[-54, 83]$. In order to avoid the introduction of an unusable buffer zone on both sides of the 180° meridian (affecting the Tchuktchen Peninsula in the Russian far east), a simple but efficient workaround is to copy the first few values in each row of the index file and insert them towards the end of the row, right after the position corresponding to +180° longitude. Similarly, the last few values before the +180° position have to be inserted before the beginning of the row.

Fig. 4–2 shows an example of ASPS ESCAT single orbit data overlaid onto the extent of the soil moisture parameter database. Fig. 4–3 is a magnification of the previous figure in the region west of Burundi, in central Africa. The descending orbit swath runs through the centre of the image, while the parameter grid features 'voids' in place of lakes Tanganyika and Kivu. As an example, a first (most near-range) node is highlighted (red asterisk) at the edge of the swath, as well as the rectangular neighbourhood of parameters. Within the rectangular subset, a circle with 36 km radius is plotted. The grid points inside the circle are selected and constitute the input for the interpolating process discussed in the next section.

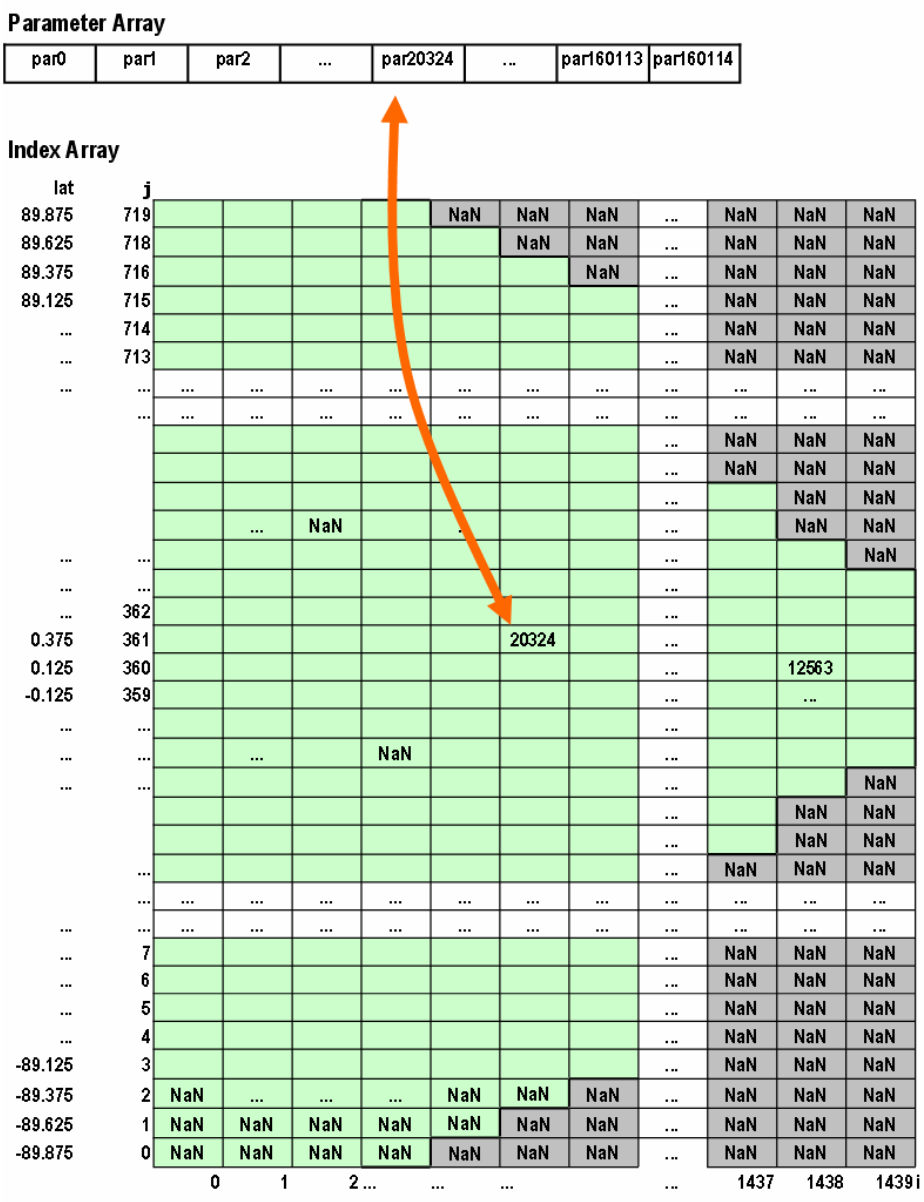


Figure 4-1.

Indexing of the parameter database.

4.2 Interpolating the Nearest Parameter Values

As soon as the grid points within the search radius are returned and their number is greater than or equal to a number k chosen beforehand, we can proceed by interpolating the various parameter values to the swath node location in question.

However convenient the latitude–longitude indexing of the parameter grid might seem, the DGG used in WARP 4.0 is not at all regularly spaced or oriented. The aforementioned decimal truncation phenomenon (see Section 2.1) creates 'pattern discontinuities' between horizontal regions with more homogenous structure. This means that the se-

lected grid points within the search radius have not only variable distance to the considered swath node, but their distribution around the node can also be uneven. A common way of resampling such irregularly distributed data points is the convolution with a so-called apodisation (tampering) function. An apodisation function is in general a weighting function that creates an average value by assigning more weight to points closer to its centre value and less to those further away. A common apodisation function used in radar remote sensing is the Hamming function (window), described by:

$$H(x) = 0.54 + 0.46\cos\left(\frac{\pi r}{a}\right), \tag{4–1}$$

where r is the distance to the centre of the window and a is the radius of the window (Weisstein 2005). For $a = 36$ km the shape of $H(x)$ is shown in Fig. 4–4. Once the grid points within the search radius are known, one should obviously choose $a \le r_{\max}$. The choice of $a = r_{\max} = 36$ km is justified by the fact that the same size for the Hamming window radius is used when resampling the σ^0 measurements to the grid points, yielding a spatial resolution of approximately 50 km, which matches the original ESCAT data resolution. Additionally, it seems to be a good compromise between not decreasing the spatial resolution too much while keeping a reliable number of grid points to average from: the number of grid points within the search radius varies from 0 to 7, with an average of 5.23. Setting the minimum number of grid points required for the Hamming filtering to take place to $k = 3$ is thus reasonable.

Some of the regions over which Hamming filtering will have to take place are “convex” or “concave” coastlines and lakeshores, where it sure that some points within the search radius have no defined values for all parameters. In order to avoid averaging over such regions, a second condition is introduced: for the Hamming filtering to take place, the number of points with an invalid (NaN, not-a-number) value within the search radius should not exceed the number of points with valid values.

Thus, in case of m valid points within the search radius ($m \ge k$) *and* the number of invalid points less than the number of valid points, any Hamming filtered parameter $\overline{P}$ is given by:

$$\overline{P} = \frac{\sum_{i=1}^{m} P_i \cdot H(r_i)}{\sum_{i=1}^{m} H(r_i)}. \tag{4–2}$$

In the contrary case, the value for parameter in question for the present node will be assigned a NaN value.

It is important to notice that of the input parameters presented in Section 2.2 only *ESD*, *NOISE_SLO* and *NOISE_S40* can be subjected to the Hamming function directly, since they are static (independent of acquisition time) and they are used as is in the subsequent processing. The weighted and averaged values after the Hamming filtering will be denoted $\overline{ESD}$, $\overline{\varepsilon_{\sigma'}}$ and $\overline{\varepsilon_{\sigma^0_{40}}}$ respectively.

The *SLO.C*, *SLO.D*, *SLO.PHASE* are intermediate input parameters for various sine-based empirical periodic functions $\Psi'(t)$. These functions characterise the yearly evolution of the slope σ' of the backscatter–incidence angle relationship $\sigma^0(\theta)$. There are approximately one hundred such functions defined in WARP 4.0 and they all accept the same number of coefficients (three coefficients and the acquisition time) but their type can vary from grid point to grid point (Scipal 2002). The type of function is given by the field *SLO.NAME*. Since the three coefficients cannot be Hamming windowed directly (they depend on the type of $\Psi'(t)$), σ' slope values have first to be calculated by substitution in the periodic function in question for all grid points within the search radius, one by one. The formula is:

$$\sigma'(40,t) = C' + D' \cdot \Psi'(t), \tag{4–3}$$

where C' and D' are *SLO.C* and *SLO.D*, respectively and t is the time of acquisition measured in year fractions.

The resulting slope values (as many as grid points within the search radius) can then be subjected to Hamming filtering, yielding the output $\overline{\sigma'}$.

Similarly, the *CURV.C*, *CURV.D* and *CURV.PHASE* parameters are input coefficients to sets of empirical periodic functions $\Psi''(t)$ describing the curvature σ'' of the backscatter–incidence angle relationship $\sigma^0(\theta)$. The same procedure applies as for the *SLO* parameters, with the substitution formula:

$$\sigma''(40,t) = C'' + D'' \cdot \Psi''(t), \tag{4–4}$$

where C'' and D'' are *CURV.C* and *CURV.D*, respectively. The resulting filtered curvature is $\overline{\sigma''}$.

As far as *DRY* and *WET* are concerned, these are annual minimum and maximum backscatter values which are used together with the $\Psi'(t)$ and $\Psi''(t)$ sets of functions to calculate the backscatter under dry and wet conditions, σ^0_{DRY} and σ^0_{WET}, respectively. Therefore, σ^0_{DRY} and σ^0_{WET} (which are also among the output parameters of the resampling procedure) must also be computed separately for each grid point within the search radius, using the formulae:

$$\sigma^0_{DRY}(40,t) = C^0_{DRY} - D' \cdot \Psi'(t) \cdot (\theta_{DRY} - 40^o) - \\ - \frac{1}{2} D'' \cdot \Psi''(t) \cdot (\theta_{DRY} - 40^o)^2 \quad (4\text{–}5)$$

and

$$\sigma^0_{WET}(40,t) = C^0_{WET} - D' \cdot \Psi'(t) \cdot (\theta_{WET} - 40^o) - \\ - \frac{1}{2} D'' \cdot \Psi''(t) \cdot (\theta_{WET} - 40^o)^2 \quad (4\text{–}6)$$

where C^0_{DRY} and C^0_{WET} are *DRY* and *WET*, respectively. The θ_{DRY} and θ_{WET} angles are so-called cross-over angles and in the current implementation they are set to 25° and 40° respectively. After applying the Hamming function, the resulting averaged reference backscatter values will be $\overline{\sigma^0_{DRY}}$ and $\overline{\sigma^0_{WET}}$.

The final output of the reprocessing algorithm will thus be $\overline{ESD}$, $\overline{\varepsilon_{\sigma'}}$, $\overline{\varepsilon_{\sigma^0_{40}}}$, $\overline{\sigma'}$, $\overline{\sigma''}$, $\overline{\sigma^0_{DRY}}$ and $\overline{\sigma^0_{WET}}$. Some of these variables are measured in dB. Since the soil moisture retrieval algorithm is entirely based on backscatter measured in dB (as opposed to linear values), it is useful to remark that the Hamming function will also be applied directly to the dB values. The diagram in Fig. 4–5 gives an overview of the reprocessing algorithm, using the variable names from the C++ program code shown in Appendix A. The program uses the standard C++ `float`, `math` and `getopt` libraries as well as `ph`, a proprietary library of I.P.F., making handling of arrays and strings more convenient. Resampling the parameters corresponding to one orbit of ESCAT data requires a time period in the order of seconds, well in compliance with the NRT character of the application.

Figure 4–6 compares the *ESD* parameter over the Burundi region, as it is contained in the original parameter database and after resampling to the orbit geometry.

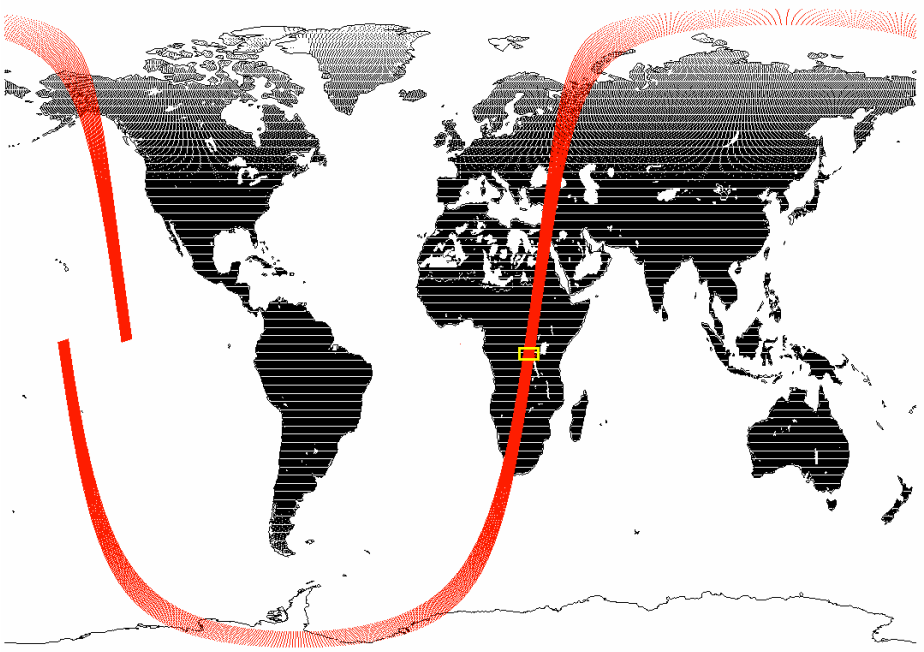


Figure 4–2.

ESCAT orbit swath and the discrete global parameter grid locations.

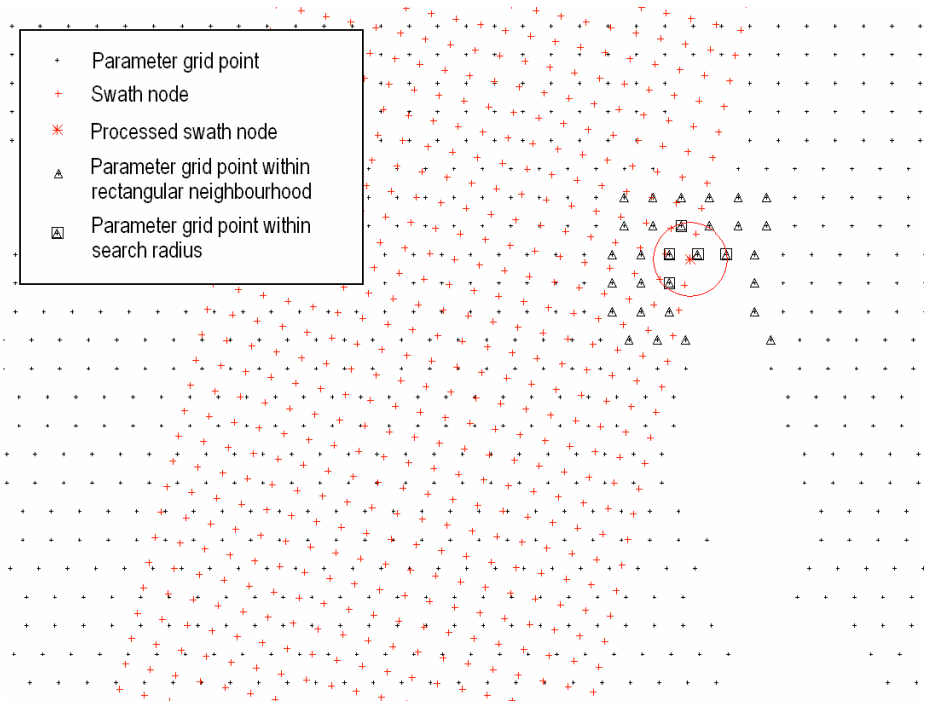


Figure 4–3.

Enlargement of the region west of Burundi in Fig. 4–2. Notice that one of the grid points within the search radius holds a NaN-value and is thus not displayed.

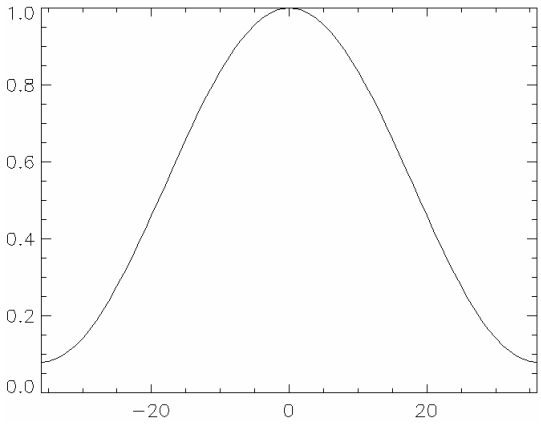


Figure 4–4.

Example of a one-dimensional Hamming window with a = 36 km.

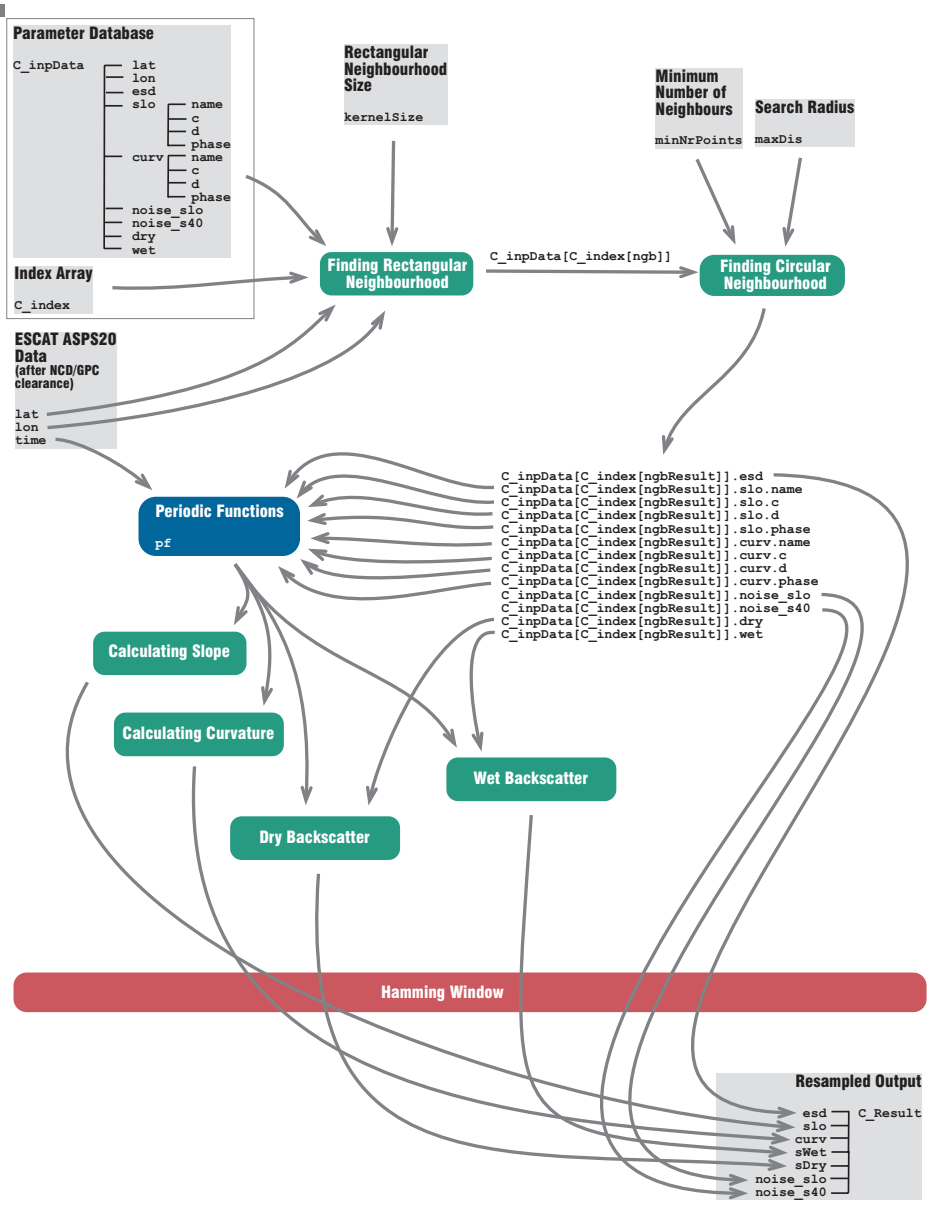


Figure 4–5.

Diagram of the resampling procedure.

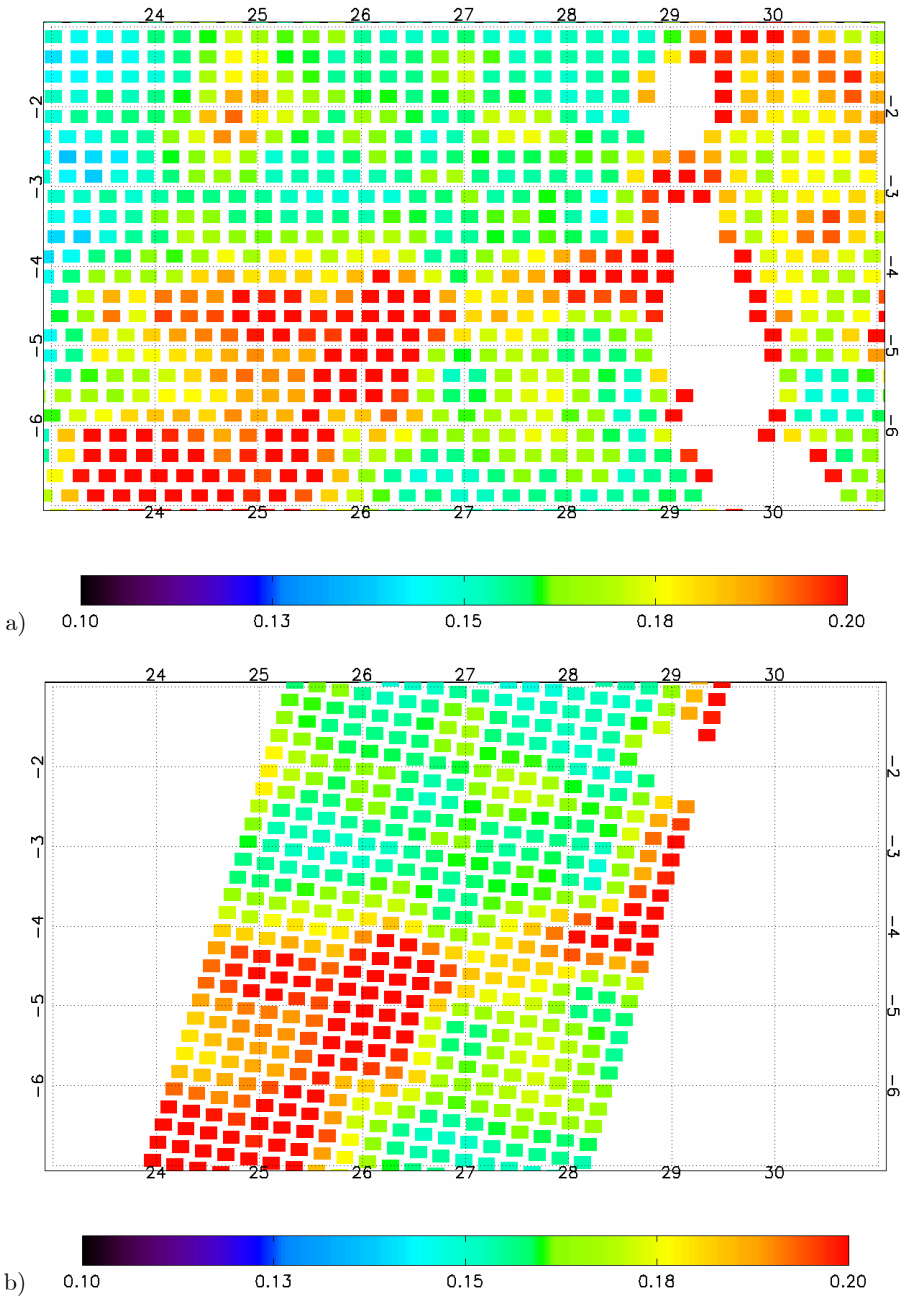


Figure 4-6.

The ESD parameter for the Burundi area in a) the original DGG representation; b) in the sensor swath geometry, after resampling with the Hamming window.

5 Future Work

After WARP^{NRT} 1.0, which will serve mainly for demonstrative purposes, the ultimate goal of the present project is to develop WARP^{NRT} 2.0, which will be based on WARP 5.0 and will use scatterometer data from the ASCAT instrument onboard the upcoming MetOp satellites. With this in mind, at this point it is useful to give some considerations about the advantages and drawbacks of the *DGG* used in the present report, paving way for several improvements in the coming versions:

Advantages

- Simple to implement
- Fast/efficient retrieving of desired neighbourhoods by latitude–longitude indexing

Drawbacks

- Inter-point spacing is not constant. Other than the spacing in west-east direction, this problem could only be solved partially, even if a completely different DGG using e.g. base regular polyhedrons and spatial partitioning was adopted – see Sahr et al. (2003). The variable inter-point distance along the latitude circles (due to the mentioned decimal truncation, see Section 2.1) is a side-effect of the fact that grid points are continuously distributed around the circumference of the latitude circles. It would be desirable to 'sacrifice' this continuity in favour of equal east-west inter-point spacing for all latitude circles. This would imply that the grid point distribution across a certain meridian would not necessarily be continuous. It seems reasonable to choose the 180° meridian for this discontinuity, since most of it falls over the ocean.
- Associating σ^0 measurements within a search radius (36 km) to each grid point actually means oversampling the data (there will always be an overlap of circular sampling areas around the grid points). To our understanding, this is not a major problem when deriving or interpreting the soil moisture data, since processing is based on time series rather than images.

- The largest drawback in the present implementation is considered to be the modelling of an ellipsoid by a sphere. ESCAT data use the GEM6 (Goddard Earth Model 6) reference ellipsoid (ESA 1992), with an equatorial Earth radius of 6378.144 km and a polar Earth radius of 6356.759 km. The difference between the ellipsoid-based so-called geodetic latitude and the geocentric (geographic) latitude of the spherical model (see Fig. 5–1) is shown in Fig. 5–2 for geographic latitudes ranging from 0° to 90°. The largest difference occurs at 45° latitude and its value (~0.192°) is comparable to the spacing of the latitude circles of the GCC (0.250°). This hints at the necessity to replace the sphere with the GEM6 reference ellipsoid for future WARP versions.

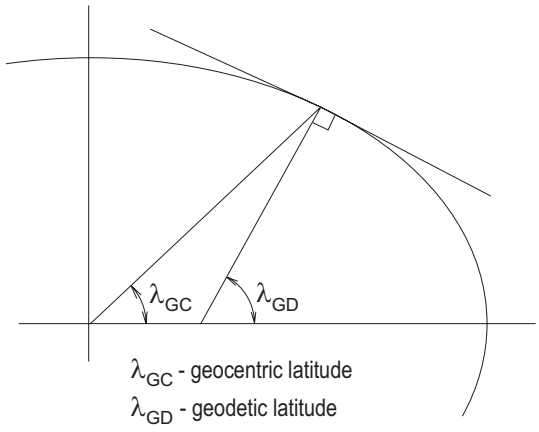


Figure 5–1.

Different latitude definitions.

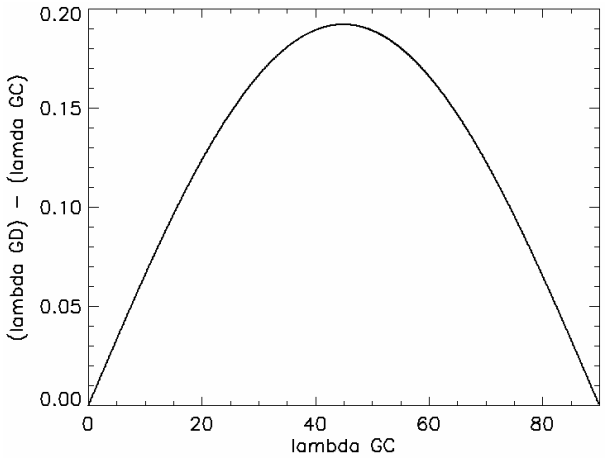


Figure 5–2.

Difference between geodetic and geocentric latitudes vs. geocentric latitude (in degrees).

Acknowledgements

The present work was funded by the Austrian Science Fund (FWF) through the GLOBSCAT (L148-N10) project and by the Austrian Academy of Sciences through the HOE-31 project. ERS data has been provided by ESA through the Cat-1 2728 project.

Bibliography

Crapolicchio, R., P. Lecomte and X. Neyt (2004). The Advanced Scatterometer Processing System for ERS Data: Design, Products and Performances. ENVISAT & ERS Symposium, Salzburg, Austria, 6-10 September 2004.

ESA (1992). *ESA ERS-1 Product Specification.* Technical Report SP-1149, Issue 3.0. Frascati, Italy, European Space Agency.

Sahr, K., D. White and A. J. Kimerling (2003). *Geodesic discrete global grid systems.* Cartography and Geographic Information Science **30**(2): 121-134.

Scipal, K., (2002) Global Soil Moisture Retrieval from ERS Scatterometer Data, dissertation, Institute of Photogrammetry and Remote Sensing, Vienna University of Technology, Vienna, Austria

Wagner, W., (1998) Soil Moisture Retrieval from ERS Scatterometer Data, dissertation, Vienna University of Technology, Vienna, Austria

Weisstein, E. W., *Hamming Function,* http://mathworld.wolfram.com/HammingFunction.html, accessed on 2005 Jul 17

Appendix A: C++ Program Code

```
//***************************************************************************

#include <float.h>
#include <math.h>
#include <getopt.h>
#include <ph_strng.hpp>
#include <ph_vect.hpp>
#include <ph_opsys.hpp>

#include "pf.hpp"

#ifdef USE_STD_HEADERS
  #include <strstream>
  #include <fstream>
  #include <cassert>
using namespace std;
#else
  #include <assert.h>
  #ifndef _MSC_VER
    #include <strstream.h>
  #else
    #include <strstrea.h>
  #endif
#endif


// definition of constants
const float R =           6370;
const int   kernelSize =  7;
const float maxDis =     36.0;
const float alpha =       0.54f;
const int   minNrPoints = 3;
const int NaN = 200000;
const nbLon = 1452;
const nbLat = 720;
const float cross_dry = 25.0;
const float cross_wet = 40.0;

//***************************************************************************

// structure for periodic function coefficients
struct C_pfStruc
{
  char name[ 8 ];
  float c,
        d,
        phase;
};

//***************************************************************************

// structure for parameter sets
struct C_parStruc
{
  float lon;
  float lat;
  float esd;
  C_pfStruc slo;
  C_pfStruc curv;
  float noise_slo;
  float noise_s40;
  float dry;
  float wet;
```

```
};

//*****************************************************************************

// structure of resampled output
struct C_resampledStruc
{
  float esd;
  float slo;
  float curv;
  float sWet;
  float sDry;
  float noise_slo;
  float noise_s40;
};

//*****************************************************************************

// return the distance on the sphere with radius R
// between two positions given in spherical coordinates
float arcLength( float lon1, float lat1, float lon2, float lat2, float R )
{
  double temp=  cos( ( lon1 - lon2 ) * pi / 180 );
  double cosalpha = temp * cos( lat1*pi/180 ) * cos( lat2*pi/180 )
                    + sin( lat1*pi/180 ) * sin( lat2*pi/180 );
  double alpha = acos( cosalpha );
  return (float) (alpha * R);
}

//*****************************************************************************

// return the sum of the first i_nbItems of vector f_kernel
float total( float f_kernel[], int i_nbItems )
{
  float sum = 0;
  for( int k = 0; k < i_nbItems; k++ )
    sum += (f_kernel)[k];
  return sum;
}

//*****************************************************************************

// workaround for missing standard-C++ round function
int round( float arg )
{
  if( arg >= 0 )
    return (int) (arg+0.5);
  else
    return (int) (arg-0.5);
}

//*****************************************************************************

int round( double arg )
{
  return round( (float) (arg) );
}

//*****************************************************************************

// read in the file containing the parameter index array
int i_readIndexFile( PH_string C_dbIndFileName,
                     PH_vector< PH_vector< long > > &C_index )
{
  ifstream C_inpStream( C_dbIndFileName, ios::binary );
  int lat = 0;
  long val = 0;
  while( C_inpStream )
  {
    for( int lon = 0; lon < nbLon; lon++ )
    {
      C_inpStream.read( (char*) &val, sizeof( long ) );
      if( !C_inpStream )
        return 1;
      C_index[lat][lon] = val;
    }
    lat++;
  }
  assert( lat == nbLat );
  return 1;
}

//*****************************************************************************

// read in the file containing the data base of parameter structures
int i_readDataBase( PH_string C_dbFileName,
                    PH_vector< C_parStruc > &C_inpData )
```

```
{
  ifstream C_inpStream( C_dbFileName, ios::binary );
  int i = 0,
      sLon =   sizeof( C_inpData[0].lon ),
      sLat =   sizeof( C_inpData[0].lat ),
      sEsd =   sizeof( C_inpData[0].esd ),
      sSloN =  sizeof( C_inpData[0].slo.name ),
      sSloC =  sizeof( C_inpData[0].slo.c ),
      sSloD =  sizeof( C_inpData[0].slo.d ),
      sSloP =  sizeof( C_inpData[0].slo.phase ),
      sCurvN = sizeof( C_inpData[0].curv.name ),
      sCurvC = sizeof( C_inpData[0].curv.c ),
      sCurvD = sizeof( C_inpData[0].curv.d ),
      sCurvP = sizeof( C_inpData[0].curv.phase ),
      sNoiseS = sizeof( C_inpData[0].noise_slo ),
      sNoise4 = sizeof( C_inpData[0].noise_s40 ),
      sDry =   sizeof( C_inpData[0].dry ),
      sWet =   sizeof( C_inpData[0].wet );

  float lon = 0.0;
  while( C_inpStream )
  {
    if( !( int( C_inpData.u_length() ) >= i) )
      C_inpData.u_resize( C_inpData.u_length() + 1 );

    C_inpStream.read((char *) &lon,                 sLon );
    if( C_inpStream )
      C_inpData[i].lon = lon;
    else
      return 0;
    C_inpStream.read((char *) &C_inpData[i].lat,        sLat );
    C_inpStream.read((char *) &C_inpData[i].esd,        sEsd );
    C_inpStream.read(          C_inpData[i].slo.name,   sSloN-1 );
      C_inpData[i].slo.name[sSloN]='\0';
    C_inpStream.read((char *) &C_inpData[i].slo.c,      sSloC );
    C_inpStream.read((char *) &C_inpData[i].slo.d,      sSloD );
    C_inpStream.read((char *) &C_inpData[i].slo.phase,  sSloP );
    C_inpStream.read(          C_inpData[i].curv.name,  sCurvN-1 );
      C_inpData[i].curv.name[sCurvN]='\0';
    C_inpStream.read((char *) &C_inpData[i].curv.c,     sCurvC );
    C_inpStream.read((char *) &C_inpData[i].curv.d,     sCurvD );
    C_inpStream.read((char *) &C_inpData[i].curv.phase, sCurvP );
    C_inpStream.read((char *) &C_inpData[i].noise_slo,  sNoiseS );
    C_inpStream.read((char *) &C_inpData[i].noise_s40,  sNoise4 );
    C_inpStream.read((char *) &C_inpData[i].dry,        sDry );
    C_inpStream.read((char *) &C_inpData[i].wet,        sWet );

    ++i;
  }
  return 0;
}

//*****************************************************************************

// select the right periodic function, according to the input name string
float pf( PH_string name, float t, float c, float d, float phase )
{
  int fnNb = atoi( name.C_tail( 4 ) );
  float retVal = 0;

  switch( fnNb )
  {
    case 2003: retVal = pf_2003( t, c, d, phase ); break;
    case 2002: retVal = pf_2002( t, c, d, phase ); break;
    case 2001: retVal = pf_2001( t, c, d, phase ); break;
    case 2000: retVal = pf_2000( t, c, d, phase ); break;
    case 1709: retVal = pf_1709( t, c, d, phase ); break;
    case 1708: retVal = pf_1708( t, c, d, phase ); break;
    case 1707: retVal = pf_1707( t, c, d, phase ); break;
    case 1706: retVal = pf_1706( t, c, d, phase ); break;
    case 1705: retVal = pf_1705( t, c, d, phase ); break;
    case 1704: retVal = pf_1704( t, c, d, phase ); break;
    case 1703: retVal = pf_1703( t, c, d, phase ); break;
    case 1702: retVal = pf_1702( t, c, d, phase ); break;
    case 1701: retVal = pf_1701( t, c, d, phase ); break;
    case 1700: retVal = pf_1700( t, c, d, phase ); break;
    case 1603: retVal = pf_1603( t, c, d, phase ); break;
    case 1602: retVal = pf_1602( t, c, d, phase ); break;
    case 1601: retVal = pf_1601( t, c, d, phase ); break;
    case 1600: retVal = pf_1600( t, c, d, phase ); break;
    case 1533: retVal = pf_1533( t, c, d, phase ); break;
    case 1532: retVal = pf_1532( t, c, d, phase ); break;
    case 1531: retVal = pf_1531( t, c, d, phase ); break;
    case 1530: retVal = pf_1530( t, c, d, phase ); break;
    case 1523: retVal = pf_1523( t, c, d, phase ); break;
    case 1522: retVal = pf_1522( t, c, d, phase ); break;
    case 1521: retVal = pf_1521( t, c, d, phase ); break;
```

```
    case 1520: retVal = pf_1520( t, c, d, phase ); break;
    case 1516: retVal = pf_1516( t, c, d, phase ); break;
    case 1515: retVal = pf_1515( t, c, d, phase ); break;
    case 1514: retVal = pf_1514( t, c, d, phase ); break;
    case 1513: retVal = pf_1513( t, c, d, phase ); break;
    case 1512: retVal = pf_1512( t, c, d, phase ); break;
    case 1511: retVal = pf_1511( t, c, d, phase ); break;
    case 1510: retVal = pf_1510( t, c, d, phase ); break;
    case 1503: retVal = pf_1503( t, c, d, phase ); break;
    case 1502: retVal = pf_1502( t, c, d, phase ); break;
    case 1501: retVal = pf_1501( t, c, d, phase ); break;
    case 1500: retVal = pf_1500( t, c, d, phase ); break;
    case 1404: retVal = pf_1404( t, c, d, phase ); break;
    case 1403: retVal = pf_1403( t, c, d, phase ); break;
    case 1402: retVal = pf_1402( t, c, d, phase ); break;
    case 1401: retVal = pf_1401( t, c, d, phase ); break;
    case 1400: retVal = pf_1400( t, c, d, phase ); break;
    case 1312: retVal = pf_1312( t, c, d, phase ); break;
    case 1311: retVal = pf_1311( t, c, d, phase ); break;
    case 1310: retVal = pf_1310( t, c, d, phase ); break;
    case 1309: retVal = pf_1309( t, c, d, phase ); break;
    case 1308: retVal = pf_1308( t, c, d, phase ); break;
    case 1307: retVal = pf_1307( t, c, d, phase ); break;
    case 1306: retVal = pf_1306( t, c, d, phase ); break;
    case 1305: retVal = pf_1305( t, c, d, phase ); break;
    case 1304: retVal = pf_1304( t, c, d, phase ); break;
    case 1303: retVal = pf_1303( t, c, d, phase ); break;
    case 1302: retVal = pf_1302( t, c, d, phase ); break;
    case 1301: retVal = pf_1301( t, c, d, phase ); break;
    case 1300: retVal = pf_1300( t, c, d, phase ); break;
    case 1200: retVal = pf_1200( t, c, d, phase ); break;
    case 1112: retVal = pf_1112( t, c, d, phase ); break;
    case 1110: retVal = pf_1110( t, c, d, phase ); break;
    case 1109: retVal = pf_1109( t, c, d, phase ); break;
    case 1108: retVal = pf_1108( t, c, d, phase ); break;
    case 1107: retVal = pf_1107( t, c, d, phase ); break;
    case 1106: retVal = pf_1106( t, c, d, phase ); break;
    case 1105: retVal = pf_1105( t, c, d, phase ); break;
    case 1104: retVal = pf_1104( t, c, d, phase ); break;
    case 1103: retVal = pf_1103( t, c, d, phase ); break;
    case 1102: retVal = pf_1102( t, c, d, phase ); break;
    case 1100: retVal = pf_1100( t, c, d, phase ); break;
    case 1101: retVal = pf_1101( t, c, d, phase ); break;
    case 1000: retVal = pf_1000( t, c, d, phase ); break;
    case 2004: retVal = pf_2004( t, c, d, phase ); break;
    case 8102: retVal = pf_8102( t, c, d, phase ); break;
    case 8101: retVal = pf_8101( t, c, d, phase ); break;
    case 8100: retVal = pf_8100( t, c, d, phase ); break;
    case 8000: retVal = pf_8000( t, c, d, phase ); break;
    case 8109: retVal = pf_8109( t, c, d, phase ); break;
    case 8103: retVal = pf_8103( t, c, d, phase ); break;
    case 8104: retVal = pf_8104( t, c, d, phase ); break;
    case 8105: retVal = pf_8105( t, c, d, phase ); break;
    case 8106: retVal = pf_8106( t, c, d, phase ); break;
    case 8107: retVal = pf_8107( t, c, d, phase ); break;
    case 8108: retVal = pf_8108( t, c, d, phase ); break;
    default: assert( 0 ); break;
  }
  return retVal;
}

//*****************************************************************************

// resampling proper
int i_resample( const PH_vector< C_parStruc > &C_inpData,
                const PH_vector< PH_vector< long > > &C_index,
                float lat,
                float lon,
                float time,
                C_resampledStruc &C_result )
{


  if( lat < -54.0
      || lat > 83.0
      || lon < -180.0
      || lon > 180.0 )
    return 1;


  long ngb[ kernelSize ][ kernelSize ];

  // fill the kernel
  for( int m = -kernelSize / 2; m <= kernelSize / 2; m++ )
  {
    float nLat = float( lat + m * 0.25 );
    int latInd = round( ( nLat + 90. ) / 0.25 );
```

```
  int giMax = int( nbLon * cos( nLat * pi / 180 ) );
  int lonInd = round( giMax * ( lon + 180.) / 360.);

  lonInd += 6; // allow for buffer of 180° +-1.5° in longitude;
  // index file contains 6 more columns at the left and right hand side

  for( int j = int( - kernelSize / 2 ); j <= int( kernelSize / 2 ); j++ )
    ngb[ m + int( kernelSize / 2 ) ] [ j + int( kernelSize / 2) ]
      = C_index[ latInd ][ j + lonInd ];
}

int i_nbResultItems = 0;
float disResult[ kernelSize*kernelSize ];
long ngbResult  [ kernelSize*kernelSize ];
float dis      [ kernelSize ][ kernelSize ];

for( int k = 0; k < kernelSize; k++ )
  for( int j = 0; j < kernelSize; j++ )
  {
    if( ngb[k][j] != NaN )
      dis[k][j] = arcLength( C_inpData[ ngb[ k ][ j ] ].lon, C_inpData[ ngb[ k ][ j ] ].lat,
                             float( lon ), float( lat ), R );
    else
      dis[k][j] = 0;
    if( dis[k][j] >= maxDis )
    {
      disResult[ i_nbResultItems ] = dis[k][j];
      ngbResult[ i_nbResultItems ] = ngb[k][j];
      i_nbResultItems++;
    }
  }


if( i_nbResultItems < minNrPoints )
{
  cerr << "Too few result items at latitude: " << lat
       << "° longitude :" << lon << "°" << endl;
  return 1;
}


float weight[ kernelSize*kernelSize ]; // constant value needed for memory allocation
for( int k = 0; k < i_nbResultItems; k++ )
  weight[ k ] = (float) (alpha+(1-alpha) * cos( pi / maxDis * disResult[ k ] ) );

float totalWeight = total( weight, i_nbResultItems );

// initialize the structure to return
C_result.esd        = 0.0;
C_result.slo        = 0.0;
C_result.curv       = 0.0;
C_result.noise_slo  = 0.0;
C_result.noise_s40  = 0.0;
C_result.sDry       = 0.0;
C_result.sWet       = 0.0;

for( int k = 0; k < i_nbResultItems; k++ )
{
  float slo = pf( PH_string( C_inpData[ ngbResult[ k ] ].slo.name ),
                  time,
                  C_inpData[ ngbResult[ k ] ].slo.c,
                  C_inpData[ ngbResult[ k ] ].slo.d,
                  C_inpData[ ngbResult[ k ] ].slo.phase );

  float curv = pf( PH_string( C_inpData[ ngbResult[ k ] ].curv.name ),
                  time,
                  C_inpData[ ngbResult[ k ] ].curv.c,
                  C_inpData[ ngbResult[ k ] ].curv.d,
                  C_inpData[ ngbResult[ k ] ].curv.phase );

  float sigma_dry =
    C_inpData[ ngbResult[ k ] ].dry
      - pf( PH_string( C_inpData[ ngbResult[ k ] ].slo.name ),
                  time,
                  0.0,
                  C_inpData[ ngbResult[ k ] ].slo.d,
                  C_inpData[ ngbResult[ k ] ].slo.phase ) * ( cross_dry - 40 )
      - 0.5 * pf( PH_string( C_inpData[ ngbResult[ k ] ].curv.name ),
                  time,
                  0.0,
                  C_inpData[ ngbResult[ k ] ].curv.d,
                  C_inpData[ ngbResult[ k ] ].curv.phase ) * pow( cross_dry - 40, 2 );

  float sigma_wet =
    C_inpData[ ngbResult[ k ] ].wet
      - pf( PH_string( C_inpData[ ngbResult[ k ] ].slo.name ),
                  time,
```

```
                    0.0,
                    C_inpData[ ngbResult[ k ] ].slo.d,
                    C_inpData[ ngbResult[ k ] ].slo.phase ) * ( cross_wet - 40 )
        - 0.5 * pf( PH_string( C_inpData[ ngbResult[ k ] ].curv.name ),
                    time,
                    0.0,
                    C_inpData[ ngbResult[ k ] ].curv.d,
                    C_inpData[ ngbResult[ k ] ].curv.phase ) * pow( cross_wet - 40, 2 );

    // do the Hamming-weighting
    C_result.esd        += C_inpData[ ngbResult[ k ] ].esd        * weight[ k ];
    C_result.slo        += slo                                    * weight[ k ];
    C_result.curv       += curv                                   * weight[ k ];
    C_result.sDry       += sigma_dry                              * weight[ k ];
    C_result.sWet       += sigma_wet                              * weight[ k ];
    C_result.noise_slo  += C_inpData[ ngbResult[ k ] ].noise_slo  * weight[ k ];
    C_result.noise_s40  += C_inpData[ ngbResult[ k ] ].noise_s40  * weight[ k ];
  }
  C_result.esd        /= totalWeight;
  C_result.slo        /= totalWeight;
  C_result.curv       /= totalWeight;
  C_result.sWet       /= totalWeight;
  C_result.sDry       /= totalWeight;
  C_result.noise_slo  /= totalWeight;
  C_result.noise_s40  /= totalWeight;

  return 1;
}
```