LPI Lag Profile Inversion EISCAT3D, Work Package 11, Task 11.4

Ilkka Virtanen Department of Physics University of Oulu, Finland

> Version 0.3-5 June 7, 2016

License

LPI (Lag Profile Inversion)

Copyright 2010–2013 University of Oulu, Finland. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY OF OULU "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE UNIVERSITY OF OULU OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of University of Oulu.

Contents

1	Intr	roduction 5
	1.1	Purpose of this document
	1.2	Lag Profile Inversion
	1.3	Installation
	1.4	Help
2	Lag	profile inversion 7
	2.1	Transmitter and receiver signals
	2.2	Scattering from a target
	2.3	Target covariance functions
	2.4	Lag profiles
	2.5	Range ambiguity functions
	2.6	Radar measurement as a linear inverse problem
	2.7	Additional analysis steps
		2.7.1 Ground clutter suppression
		2.7.2 Voltage level decoding
3	$_{ m LPI}$	implementation 13
	3.1	Resampling and filtering
	3.2	Ground clutter suppression
	3.3	Voltage level decoding filters
	3.4	Data correlation
	3.5	Theory matrix
	3.6	Lag profile inversion solvers
	3.7	Input and output control
4	LPI	in practice 17
-	4.1	Examples with simulated data
	***	4.1.1 A coherent point target
		4.1.2 Ground clutter suppression

	4.2	Examp	oles with real data
		4.2.1	Autocovariance function measurement with a monos-
			tatic radar
		4.2.2	Bistatic measurements and crosscovariannee functions . 29
5	Doo	cument	ed source code 31
	5.1	Proces	s control
		5.1.1	LPI 31
		5.1.2	LPIexpand.input
		5.1.3	LPIinitCluster.R
		5.1.4	LPIinitRemoteNode.R
		5.1.5	LPIinitComputingSlaves.R
		5.1.6	currentTimes.R
		5.1.7	nextIntegrationPeriods.R
		5.1.8	LPIsolve.acf.R
		5.1.9	noUpdate.R
		5.1.10	LPIrun.remote.R
		5.1.11	LPIrun.R
		5.1.12	initLPIenv.R
		5.1.13	LPIsolve.R
	5.2		pre- and post-processing
		5.2.1	prepareLPIdata.R
		5.2.2	clutterSuppress.R
		5.2.3	decoFilter.R
		5.2.4	LPIaveragePower.R
		5.2.5	LPIsaveACF.R
		5.2.6	stripACF.R
	5.3	Correl	ation and inverse problem formulation 89
		5.3.1	laggedProducts.R
		5.3.2	lagprodVar.R
		5.3.3	rangeAmbiguity.R
		5.3.4	averageProfiles.R
		5.3.5	theoryRows.R
	5.4	Inverse	e problem solvers
		5.4.1	fishs.init.R
		5.4.2	fishs.add.R
		5.4.3	fishs.solve.R
		5.4.4	deco.init.R
		5.4.5	deco.add.R
		5.4.6	deco.solve.R
		5.4.7	dummy init. R

	5.4.8	dummy.add.R											106	
	5.4.9	dummy.solve.R .											107	
	5.4.10	ffts.init.R												
	5.4.11	ffts.add.R . . .											111	
	5.4.12	ffts.solve.R.											114	
	5.4.13	rlips.solve2.R											116	
5.5	C func	tions and headers											118	
	5.5.1	src/LPI.h											118	
	5.5.2	register.c											121	
	5.5.3	clutter $_$ meas.c											123	
	5.5.4	$clutter_subtract.c$											126	
	5.5.5	$dummy_add.c$											129	
	5.5.6	$deco_add.c \dots$											132	
	5.5.7	$fishs_add.c \dots$											135	
	5.5.8	$fishs_add_clutter.c$											138	
	5.5.9	$index_adjust.c$											140	
	5.5.10	average_power.c .											145	
	5.5.11	$lagged_products.c$											153	
	5.5.12	$average_profile.c\ .$											159	
	5.5.13	$mix_frequency.c$.											163	
	5.5.14	resample.c											167	
	5.5.15	$prepare_data.c.$											174	
	5.5.16	theory_rows.c											178	
	5.5.17	$range_ambiguity.c$											187	

Chapter 1

Introduction

1.1 Purpose of this document

Development and documentation of an incoherent scatter data analysis software based on the lag profile inversion paradigm was the Task 11.4 of the Work Pacakge 11 (Software theory and implementation) of the European Union's Framework 7 project EISCAT_3D. This document serves as part of the final report of the Task 11.4 as well as a tutorial for the package and lag profile inversion in general. A standard user manual of the R package is provided as a separate document.

1.2 Lag Profile Inversion

LPI is an R¹ package for suppressing range ambiguities from incoherent scatter radar² lag profiles. LPI solves gated autocovariance function or cross-covariance function estimates from voltage level transmitter and receiver signal samples. LPI replaces traditional decoding techniques with a statistical-inversion-based approach, which makes it applicable to radar experiments that use almost arbitrary transmitter modulations.

¹R is a free open source software environment for statistical computing and graphics. It is licensed under GPL and it is available for various platforms. See http://www.r-project.org for details.

²The package was developed for IS radars, but it is in principle applicable for all kinds of radars.

1.3 Installation

The package is distributed as a tarball (LPI_x.x.x.tar.gz) which can be installed via the standard

```
R CMD INSTALL LPI
```

procedure (preceding sudo is usually required in order to run the command as root). See R CMD INSTALL --help for installation options. R version >= 2.14.0 is required. In order to enable the rlips inverse problem solver also the rlips package must be installed³, whether rlips is installed before or after LPI is not important.

```
> 234 * 234
[1] 54756
```

1.4 Help

Standard R help pages are provided. They are collected together in the pdf file "LPI-manual.pdf". Both the manual and this document "LPI-tutorial.pdf" are contained in the distribution package as vignettes. The documents are stored in the distribution tarball (LPI_x.x.x.x.tar.gz) in /inst/doc/.

After installing the package the vignettes can be opened from R command line following the standard procedure

```
> require(LPI)
> vignette('LPI-manual')
> vignette('LPI-tutorial')
```

The same help messages can be shown on command line as well

```
> help(package='LPI')
> help(LPI)
```

etc.

³Latest version of rlips is currently available at http://www.sgo.fi/ m/pages/rlips.html

Chapter 2

Lag profile inversion

2.1 Transmitter and receiver signals

A radar transmitter emits a modulated radio signal that can be expressed as product of a continuous coherent carrier signal c(t) and a modulating transmitter envelope env(t). Because the carrier sigal contribution can be effectively removed by means of complex frequency mixing to baseband, we will neglect the carrier from this point on and consider only the transmitter envelope env(t).

The transmitted modulated signal is scattered from a target and the scattered signal s(t) enters a radar receiver. Because the receiver must have a finite impulse response p(t), the final detected signal is convolution of the scattered signal entering the receiver and the impulse response

$$z^{r}(t) = (s * p)(t).$$
 (2.1)

We will later need also the similar convolution of the transmitter envelope and the receiver impulse response,

$$z^{t}(t) = (\operatorname{env} * p)(t). \tag{2.2}$$

In reality, discrete signal samples will be recorded with a uniform sample interval Δt , which produces final recorded sample streams of discrete transmitter and receiver samples

$$z_i^t = z^t(t_i) \tag{2.3}$$

$$z_i^r = z^r(t_i) \tag{2.4}$$

where $t_i = i\Delta t$.

The receiver sample stream is not continuous in general, because monostatic radar systems cannot receive while transmitting. The transmitter sample stream will be effectively continuous because the transmitter envelope is known to be zero while the radar is not transmitting. The discontinuities in receiver sample stream have significant consequencies especially when detecting nearby targets with a monostatic high duty-cycle radar¹.

2.2 Scattering from a target

If the transmitter signal hits reflecting point target at distance R^t from the radar transmitter and R^r from the receiver the signal entering the receiver can be expressed as

$$s(t) = \xi \operatorname{env}(t - S) \tag{2.5}$$

where range S is signal travel from the transmitter, via the target, to the receiver and ξ is a complex coefficient. The signal is assumed to propagate at the speed of light c, allowing the range to be calculated as

$$S = \frac{R^t + R^r}{c}. (2.6)$$

In monostatic systems the range reduces to $S = 2R^t/c$. The target does not need to be stationary as Doppler shifts can be absorbed in the complex coefficient ξ .

If the taret is not point-like but covers a finite range of distances, $[S_1, S_2]$, the signal will be scattered from all parts of the target and the received signal can be written as

$$s(t) = \int_{S_1}^{S_2} \xi(S) \operatorname{env}(t - S) dS$$
 (2.7)

where $\xi(S)$ is a range-dependent complex coefficient. This kind of target is said to be spread in range or range-spread. Notice that ξ was defined as function of the total signal travel time S, and it will thus be different for two physically separeted receivers.

Finally, if amplitude or Doppler shift of the scattering changes as function of time, we must introduce a range and time dependent coefficient $\xi(S,t)$ and

 $^{^{1}}$ Monostatic incoherent scatter radars typically have duty-cycles from 5 to 25 %

write the received signal as

$$s(t) = \int_{S_1}^{S_2} \xi(S, t) * \text{env}(t - S) dS$$
 (2.8)

Likewise with range, the time-dependence of ξ was expressed as function of signal reception time instead of the time of scattering. A target is said to be spectrally overspread or Doppler-spread if power spectrum of any temporal variations is wider than inverse of signal travel time to and from the furthest part of the target.

F region of the ionosphere is spread in both range and spectrum, while the D region alone is spread only in range. Because the above layers cannot be generally neglected when probing the D region the ionosphere as whole must be considered as a spread target in both range and Doppler.

2.3 Target covariance functions

If scattering from any individual range S is modeled as a zero-mean random process, statistical properties of the target can be deduced from different covariance functions of received signals.

The basic data product of a radar is a set of signal autocovariance function estimates as function of range: $\sigma_a(S,\tau)$ where τ is time lag. This data product is measured by means of correlating a receiver sample stream with itself. Similarly, it is possible to correlate sample streams from two physically separate receivers in order to detect crosscorrelation function as function of range $\sigma_c(S,\tau)$. It is also possible that the crosscorrelation function is calculated in between sample streams recorded with the same device but at orthogonal polarizations. This kind of arrangement is used in orthogonal polarization coding and in Faraday rotation measurements.

2.4 Lag profiles

The autocovariance function as function of range $\sigma(S, \tau)$ was previously considered. The formulation essentially deals with fixing a range S and defining the autocovariance function of the scattering process at the given range.

Instead of fixing a range one can fix a time lag and deal with the resulting range profiles as well. These fixed time lags of covariance function as function of range are called lag profiles. Denoting the lag profile at lag τ_i with $\rho_i(S)$ we will have

$$\rho_i(S) = \sigma(S, \tau_i) \tag{2.9}$$

2.5 Range ambiguity functions

Expectation value of the product

$$m_{i,j} = z^r(t_i)\overline{z^r(t_j)} \tag{2.10}$$

can be written as

$$< m_{i,j} > = \int_{S_1}^{S_2} W(t_i, t_j, S) \sigma(S, t_i - t_j) dS$$
 (2.11)

where W(t,t',S) is the range ambiguity function,

$$W(t,t',S) = z^t(t-S)\overline{z^t(t'-S)}.$$
(2.12)

It is thus possible to write each product m(t,t') as

$$m_{i,j} = \int_{S_1}^{S_2} W(t_i, t_j, S) \sigma(S, t_i - t_j) dS + \varepsilon(t_i, t_j)$$

$$(2.13)$$

where the last term is random noise.

2.6 Radar measurement as a linear inverse problem

If the target is divided into discrete range gates the measurement (2.13) can be written as sum

$$m_{i,j} = \sum_{k=k_1}^{k_2} W_{i,j,k} \sigma_{k,i-j} + \varepsilon_{i,j}$$
 (2.14)

where the coefficients $W_{i,j,k}$ can be calculated from the continuous range ambiguity function.

In real measurements only discrete signal samples are available and the coefficients must be approximated from products of these samples. Oversampling or interpolation is generally needed in order to achieve sufficient accuracy.

When all measurements of a given lag are collected together one can form a linear inverse problem

$$\mathbf{m}_l = \mathbf{W}_l \boldsymbol{\rho}_l + \boldsymbol{\varepsilon}_l \tag{2.15}$$

where the measurement vector \mathbf{m}_l is a column vector of measurements $m_{i,i+l}$, i = 0, 1, 2, ..., the theory matrix \mathbf{W}_l contains the coefficients $W_{i,i+l,k}$, ρ_l is the unknown discrete lag profile and ε_l is random noise. If the noise is zero-mean and gaussian the Maximum Aposteriori (MAP) estimate of the lag profile is

$$\hat{\boldsymbol{\rho}}_l = \mathbf{Q}_l^{-1} \mathbf{W}_l^H \boldsymbol{\Sigma}_l^{-1} \mathbf{m} \tag{2.16}$$

$$\mathbf{Q}_l = \mathbf{W}_l^H \mathbf{\Sigma}_l^{-1} \mathbf{W}_l \tag{2.17}$$

where Σ_l is the measurement error covariance matrix and \mathbf{Q}_l is called Fisher information matrix. Posterior noise covariance matrix of the resolved lag profile is inverse matrix of the Fisher information matrix.

This formal solution is not practical for real life lag profile inversion. Instead of directly forming the theory matrix special solvers, such as rlips, are used which allow the theory matrix to be formed in smaller blocks.

2.7 Additional analysis steps

2.7.1 Ground clutter suppression

Although radar beams are nominally pointed towards the "empty" sky, there are always antenna sidelobes, some of which are pointed towards the terrain surrounding the radar transmitter. In monostatic radar systems this leaked signal may be reflected back towards the receiver antenna. The reflected signal may leak to the receiver through the same sidelobes, causing echoes called ground clutter. The ground clutter may be much stronger than the true ionospheric signal and should be suppressed in low altitude ionospheric measurements.

The ground clutter signal is possible to suppress because the scattering target is known to be stationary. It is thus possible to estimate an average clutter profile from voltage level data and to suppress it prior to correlation.

2.7.2 Voltage level decoding

Incoherent scatter spectrum in the ionospheric D region is rather narrow and it becomes possible to decode the received data at voltage level prior to lag profile inversion. After voltage level decoding the signal will correspond to a measurement with short pulses mathced to the measurement range resolution. It is thus possible to calculate only a rather small number of time lags and, furthermore, to replace the general lag profile inversion with simple averaging of lagged products. Voltage level decoding thus provides a possibility for very fast inversion of D region lag profiles. However, one should be very careful when measuring short time lags with this technique, because it may allow F region echoes to aliase on top of the true D region signal.

Chapter 3

LPI implementation

3.1 Resampling and filtering

Both transmitter and receiver samples are filtered and decimated to a common sample rate before lag profile inversion. Non-integer (but rational) fraction down sampling is supported. The resampling reduces to a boxcar filter if the filter length is a multiple of the original sample interval.

3.2 Ground clutter suppression

LPI contains an optional ground clutter suppression algorithm. It uses statistical inversion for estimating the average backscatter at voltage level and subtracts the convolution of the estimated signal and the transmitter samples from the receiver samples. This technique is statistically optimal in the sense that only one clutter profile per integration period is produced, which allows it to be estimated with high accuracy. Increase in noise power due to clutter suppression is thus minimized. The long coherent integration could make the technique inefficient when the clutter source is not exactly stationary. On the other hand, the technique does not set any requirements for the transmitter modulation, which is very advantageous when it is combined e.g. with multi-purpose modulations.

3.3 Voltage level decoding filters

Voltage level decoding with given filter coefficients, as well as both matched and inverse filters using measured transmission envelopes, are supported. With given coefficients both TX and RX data vectors are simply convolved with the given filter.

With the matched and inverse filters the filter coefficients are calculated from the TX data vectors and the decoding is performed one inter-pulse period at a time. The analysis should thus not continue above the range of the shortest IPP when the voltage level decoding is enabled. The further analysis is performed assuming that the filter would have completely removed range ambiguities from the filtered data. As a consequence, lag profiles calculated with voltage level matched filter will generally contain range ambiguities. These ambiguities will be severy in true power profiles, but, depending on the applied modulation, may be neglectable at longer lags.

3.4 Data correlation

Lag profile inversion requires two kinds of correlated data products to be produced: lagged products of the receiver samples and the range ambiguity functions.

The lagged products of the receiver samples can be trivially calculated from the filtered and decimated data. The range ambiguity functions can be calculated in a similar manner if the final sample interval is clearly shorter than modulation bit length, or if the applied modulation is a strong phase code.

Otherwise an approximation of the continuous transmission envelope is needed for calculating the range ambiguity function. LPI contains an option for interpolating the transmitter samples to higher sample rate before calculating the range ambiguity functions. When this option is enabled, the transmitter samples will be oversampled by factor of 11 by means of linear interpolation. The technique is not exactly optimal as the signal is first decimated and then imperfectly resampled, but it provides reasonably good range ambiguity function estimates when the transmitted bits have relatively sharp edges.

3.5 Theory matrix

The inversion theory matrix is constructed in blocks whose size is given as an input argument. In order to speed up the calculation process, only the first row of each block is calculated by means of summing the range ambiguity values within each range gate. The following rows are calculated by means of updating the preceding theory row via additions and subtractions of samples at edges of range gates. This procedure generates minor round-off errors to the theory matrix rows, but the error is negligible because 64-bit floats are used for storing the samples that are typically recorded with a 12-bit AD converter.

3.6 Lag profile inversion solvers

Altogether five different inverse problem solvers are supported.

- rlips R Linear Inverse Problem Solver¹. This is the only solver that makes use of GPUs.
- fishs A simple inverse problem solver based on direct calculation of Fisher information matrix.
- **deco** Matched filter decoding of lag profiles with variance estimation. Will lead to range ambiguities unless alternating codes or long cycles of random codes are used.
- ffts Lag profile inversion by means of FFT. Suitable for bistatic measurements, in which the limited beam intersection allows one to neglect problematic edge effects. Background noise suppression cannot be combined with ffts.
- dummy Dummy solver that calculates simple averages. Intended to be used together with voltage level decoding. Background noise suppression cannot be combined with dummy solver.

Only rlips, fishs, and deco require explicit theory matrix rows. When other solver are used the theory rows are not produced but the solvers operate directly on the correlated data vectors.

¹http://www.sgo.fi/ m/pages/rlips.html

3.7 Input and output control

In order to make LPI suitable for wide range of data formats, the package allows the user to define a set of functions used for data input and output. These functions can be collected in separate packages that can be maintained indepedently from LPI. Names of the I/O functions and packages including them are given as input arguments to the main solver function. It is usually practical to include also a simple wrapper function that generates a call to the main analysis loop of LPI. A raw data input function is mandatory, whereas a few other routines have defaults, see the LPI manual for details. Currently availabe I/O packages are LPI.gdf and LPI.KAIRA. See LPI-manual.pdf for detailed descriptions of the I/O functions.

Chapter 4

LPI in practice

This chapter contains example use cases of LPI. Before proceeding to the examples it may be worth having a look at the actual user manual. A pdf version can be opened from R command line after installing the package with

```
> library(LPI)
> vignette('LPI-manual')
```

It can also be found from within the distribution package as explained in Section 1.4. Standard R help pages are also available, please have a look at the package help page

```
> help(package=LPI)
```

and the help page of the main analysis function

> ?LPI

4.1 Examples with simulated data

Simplistic radar simulator can be easily combined with LPI by implementing the simulation in the data input functions. The following examples will run sequentially which allows us to simply define the functions in user workspace without collecting them in a separate package.

4.1.1 A coherent point target

We will begin the examples section with a simple detection of a stationary coherent target 200 km away from a monostatic radar. This is also a simple way to confirm that the package works properly. First the package needs to be loaded

> library(LPI)

We will then define a function for raw data input, the simple simulator will be build within this functions.

```
> datafun <- function( LPIparam , ... ){</pre>
+
+
    srate <- 1e4
    # First pre-allocate the output list
    outlist <- list( TX1=list() , TX2=list() , RX1=list() ,</pre>
                     RX2=list() , success=TRUE)
    # Data vector lengths, we can select TX1 because all
    # sample rates must be equal at this point
    nd <- round( LPIparam[["timeRes.s"]] * srate )</pre>
    # Let us use 1 ms pulses at random positions with 25 % duty-cycle
    # Pulse lengths counted as data samples
    plen <- floor( 1e-3 * srate )</pre>
    # Number of pulses in the whole data vectors
    np <- round( nd * .25 / plen )</pre>
    # Let us generate random pulse positions for TX1
    pstarts <- floor(runif( np ) * ( nd - plen - 1 ) ) + 1</pre>
    # Allocate the data and index vectors for TX1
+
    outlist[['TX1']][['cdata']] <- complex( nd , real=0 , imaginary=0 )</pre>
+
    # Then make random codes at each pulse position
    for( p in pstarts ){
      outlist[['TX1']][['cdata']][ p : ( p + plen - 1 ) ] <-
        runif( plen ) + 1i * runif( plen ) - (.5 + .5i)
```

```
}
+
    # Transmitter index vector can now be easily produced
    outlist[['TX1']][['idata']] <- abs( outlist[['TX1']][['cdata']] ) > 0
    # Add the ndata element
    outlist[['TX1']][['ndata']] <- as.integer(nd)</pre>
    # TX2 is identical with TX1
    outlist[['TX2']] <- outlist[['TX1']]</pre>
+
+
    # Our coherent target is assumed to be at 200 km range,
    # convert to sample intervals
    rtarg <- floor( 200e3 / 2.99792458e8 * 2 * srate )
    # The receiver samples are simply
    # a shifted copy of the transmitter samples
    outlist[['RX1']][['cdata']] <-
      c( rep( 0 , rtarg ) , outlist[['TX1']][['cdata']][1:(nd-rtarg)] )
+
+
    # Let us add some random noise on top of the receiver samples
    outlist[['RX1']][['cdata']] <-
      outlist[['RX1']][['cdata']] + (rnorm(nd) + 1i*rnorm(nd))*.3
+
    # Receiver index vector is negation of the transmitter index vector
    outlist[['RX1']][['idata']] <- !outlist[['TX1']][['idata']]</pre>
    # Add the ndata element
    outlist[['RX1']][['ndata']] <- as.integer(nd)</pre>
    # RX2 is identical with RX1
    outlist[['RX2']] <- outlist[['RX1']]</pre>
+
    return(outlist)
+ }
>
```

We will also define a new function for storing the results, it will simply copy them to the global workspace

```
> savefun <- function( LPIparam , intPeriod , ACF )
+ {
+ assign( paste('ACF',as.character(intPeriod),sep=''),ACF,.GlobalEnv)
+ }</pre>
```

We have now everything needed for the simulation run, let us call LPI. 'start-Time' and 'stopTime' are chosen arbitrarily, which is possible because our 'dataInputFunction' is actually a simulator and it will return samples for arbitrary time intervals. The function will print all parameters that may affect the inversion results.

```
> LPI(
+
   startTime = 1356998400,
   stopTime = 1356998410,
   lagLimits = seq(9),
                                     # all intra-pulse lags
   timeRes.s = 10,
                                      # 10 s integration time
   rangeLimits = seq(1,30)
                                 , # range gates
   resultDir = NA ,
                                      # we will not write results to files
   dataInputFunction = 'datafun' ,  # our data input function
   resultSaveFunction = 'savefun', # our function for saving results
+ )
         startTime: 1356998400.000000 (2013-01-01 00:00:00.000000 UT)
          stopTime: 1356998410.000000 (2013-01-01 00:00:10.000000 UT)
     inputPackages:
 dataInputFunction: datafun
dataEndTimeFunction: currentTimes
      clusterNodes:NA
               nup: RX1:1 RX2:1 TX1:1 TX2:1
      filterLength: RX1:1 RX2:1 TX1:1 TX2:1
    decodingFilter: none
          lagLimits: 1 2 3 4 5 6 7 8 9
       rangeLimits: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
                    20 21 22 23 24 25 26 27 28 29 30
         maxRanges: Inf
         timeRes.s: 10.000000
   maxClutterRange: RX1:0 RX2:0
   clutterFraction: RX1:1 RX2:1
backgroundEstimate: TRUE
         maxWait.s: -1.000000
        freqOffset: RX1:0.000000 RX2:0.000000 TX1:0.000000 TX2:0.000000
        indexShifts: RX1:0 0 RX2:0 0 TX1:0 0 TX2:0 0
```

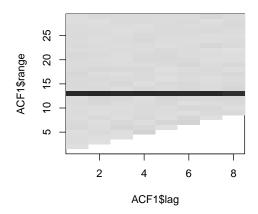


Figure 4.1: LPI detection of a simulated coherent point target.

solver: fishs nBuf: 10000 fullCovar: FALSE

rlips.options: type:c nbuf:1000 workgroup.size:128

remoteRX: FALSE normTX: FALSE nCode: NA ambInterp: FALSE

resultDir: NA

resultSaveFunction: savefun
paramUpdateFunction: noUpdate

useXDR: FALSE

.

Let us now have a look at the results stored in the variable 'ACF1'. The ACF matrix has one extra row for the background ACF because 'savefun' did not strip that off.

```
> image(ACF1$lag,ACF1$range,t(Re(ACF1$ACF[1:length(ACF1$range),])),
+ col=rev(gray(seq(1000)/1000)),zlim=c(-.2,1.2))
```

4.1.2 Ground clutter suppression

The LPI ground clutter suppression option is essentially a notch filter at zero Doppler frequency, this is a simple example of its operation.

We will replace the 'datafun' with a new version that simulates two point targets, one moving and another stationary. We will keep the original target at 200 km distance, but this time the target will have a small doppler shift. Another cluttering signal with zero Doppler will be added below the original one, and we will show how it can be suppressed from the final ACF.

Let us first re-define the data input function

```
> datafun <- function( LPIparam , ... ){</pre>
+
    srate <- 1e4
+
    # First pre-allocate the output list
    outlist <- list( TX1=list() , TX2=list() , RX1=list() ,</pre>
                     RX2=list() , success=TRUE)
+
    # Data vector lengths, we can select TX1 because all
    # sample rates must be equal at this point
    nd <- round( LPIparam[["timeRes.s"]] * srate )</pre>
    # Let us use 1 ms pulses at random positions with 25 % duty-cycle
    # Pulse lengths counted as data samples
+
    plen <- floor( 1e-3 * srate )</pre>
+
    # Number of pulses in the whole data vectors
    np <- round( nd * .25 / plen )</pre>
    # Let us generate random pulse positions for TX1
    pstarts <- floor(runif( np ) * ( nd - plen - 1 ) ) + 1</pre>
+
    # Allocate the data and index vectors for TX1
    outlist[['TX1']][['cdata']] <- complex( nd , real=0 , imaginary=0 )</pre>
    # Then make random codes at each pulse position
+
    for( p in pstarts ){
      outlist[['TX1']][['cdata']][ p : ( p + plen - 1 ) ] <-
```

```
runif( plen ) + 1i * runif( plen ) - (.5 + .5i)
   }
    # Transmitter index vector can now be easily produced
    outlist[['TX1']][['idata']] <- abs( outlist[['TX1']][['cdata']] ) > 0
    # Add the ndata element
    outlist[['TX1']][['ndata']] <- as.integer(nd)</pre>
   # TX2 is identical with TX1
    outlist[['TX2']] <- outlist[['TX1']]</pre>
    # Our coherent target is assumed to be at 200 km range,
   # convert to sample intervals
   rtarg <- floor( 200e3 / 2.99792458e8 * 2 * srate )
   # The cluttering is two range gates below the actual target
   rclut <- rtarg - 2
+
    # The receiver samples of the target will be now
    # multiplied with a complex sinusoid
    outlist[['RX1']][['cdata']] <-
      c( rep( 0 , rtarg ) , outlist[['TX1']][['cdata']][1:(nd-rtarg)] ) *
        exp(1i*seq(nd)*.01)
+
   # The cluttering target is stationary, simply add it
    outlist[['RX1']][['cdata']] <- outlist[['RX1']][['cdata']] +
      c( rep( 0 , rclut ) , outlist[['TX1']][['cdata']][1:(nd-rclut)] )
    # Let us add some random noise on top of the receiver samples
    outlist[['RX1']][['cdata']] <-
      outlist[['RX1']][['cdata']] + (rnorm(nd) + 1i*rnorm(nd))*.5
    # Receiver index vector is the negation of the transmitter index vector
    outlist[['RX1']][['idata']] <- !outlist[['TX1']][['idata']]</pre>
    # Add the ndata element
   outlist[['RX1']][['ndata']] <- as.integer(nd)</pre>
   # RX2 is identical with RX1
   outlist[['RX2']] <- outlist[['RX1']]</pre>
```

```
return(outlist)
+ }
>
We have now everything needed for the simulation run, let us call LPI, first
without clutter suppression
> LPI(
    startTime = 1356998400,
    stopTime = 1356998410,
                                     # all intra-pulse lags
    lagLimits = seq(9),
+
    timeRes.s = 10,
                                       # 10 s integration time
   rangeLimits = seq(1,30)
                                  , # range gates
   resultDir = NA ,
                                       # we will not write results to files
   resultDir = NA ,  # we will not write result dataInputFunction = 'datafun' ,  # our data input function
   resultSaveFunction = 'savefun', # our function for saving results
   maxClutterRange=0
+
+
          startTime: 1356998400.000000 (2013-01-01 00:00:00.000000 UT)
           stopTime: 1356998410.000000 (2013-01-01 00:00:10.000000 UT)
      inputPackages:
  dataInputFunction: datafun
dataEndTimeFunction: currentTimes
       clusterNodes:NA
                nup: RX1:1 RX2:1 TX1:1 TX2:1
       filterLength: RX1:1 RX2:1 TX1:1 TX2:1
     decodingFilter: none
          lagLimits: 1 2 3 4 5 6 7 8 9
        rangeLimits: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
                     20 21 22 23 24 25 26 27 28 29 30
          maxRanges: Inf
          timeRes.s: 10.000000
    maxClutterRange: RX1:0 RX2:0
    clutterFraction: RX1:1 RX2:1
 backgroundEstimate: TRUE
          maxWait.s: -1.000000
         freqOffset: RX1:0.000000 RX2:0.000000 TX1:0.000000 TX2:0.000000
        indexShifts: RX1:0 0 RX2:0 0 TX1:0 0 TX2:0 0
             solver: fishs
```

```
nBuf: 10000
         fullCovar: FALSE
     rlips.options: type:c nbuf:1000 workgroup.size:128
          remoteRX: FALSE
            normTX: FALSE
             nCode: NA
         ambInterp: FALSE
         resultDir: NA
 resultSaveFunction: savefun
paramUpdateFunction: noUpdate
            useXDR: FALSE
Let us copy the result to wait for later inspection.
> ACFclutter <- ACF1
In the second run we will apply clutter suppression all the way to 300 km
range.
> LPI(
   startTime = 1356998400,
   stopTime = 1356998410,
   lagLimits = seq(9),
                                    # all intra-pulse lags
   timeRes.s = 10,
                                    # 10 s integration time
   rangeLimits = seq(1,30)
                                 , # range gates
   resultDir = NA ,
                                     # we will not write results to files
   resultSaveFunction = 'savefun', # our function for saving results
   maxClutterRange=20
   )
         startTime: 1356998400.000000 (2013-01-01 00:00:00.000000 UT)
          stopTime: 1356998410.000000 (2013-01-01 00:00:10.000000 UT)
     inputPackages:
  dataInputFunction: datafun
dataEndTimeFunction: currentTimes
      clusterNodes:NA
               nup: RX1:1 RX2:1 TX1:1 TX2:1
      filterLength: RX1:1 RX2:1 TX1:1 TX2:1
    decodingFilter: none
```

rangeLimits: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

lagLimits: 1 2 3 4 5 6 7 8 9

```
20 21 22 23 24 25 26 27 28 29 30
          maxRanges: Inf
          timeRes.s: 10.000000
    maxClutterRange: RX1:20 RX2:20
    clutterFraction: RX1:1 RX2:1
 backgroundEstimate: TRUE
          maxWait.s: -1.000000
         freqOffset: RX1:0.000000 RX2:0.000000 TX1:0.000000 TX2:0.000000
        indexShifts: RX1:0 0 RX2:0 0 TX1:0 0 TX2:0 0
             solver: fishs
               nBuf: 10000
          fullCovar: FALSE
      rlips.options: type:c nbuf:1000 workgroup.size:128
           remoteRX: FALSE
             normTX: FALSE
              nCode: NA
          ambInterp: FALSE
          resultDir: NA
 resultSaveFunction: savefun
paramUpdateFunction: noUpdate
             useXDR: FALSE
Only one target is now detected, the lower one had zero doppler and was
subtracted at voltage level before lag profile inversion. Comparison of the
results with and without clutter suppression is given in Figure (4.2)
> image(ACFclutter$lag,ACFclutter$range,
        t(Re(ACFclutter$ACF[1:length(ACF1$range),])),
        col=rev(gray(seq(1000)/1000)), zlim=c(-.2,1.2))
```

> image(ACF1\$lag,ACF1\$range,t(Re(ACF1\$ACF[1:length(ACF1\$range),])),

col=rev(gray(seq(1000)/1000)), zlim=c(-.2,1.2))

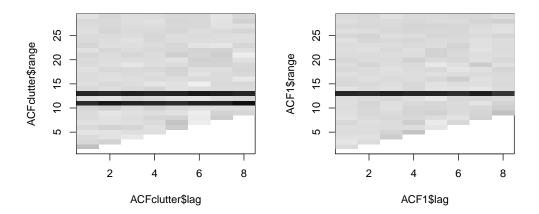


Figure 4.2: LPI detection of two simulated coherent point targets. The upper one has a small doppler shift whereas the lower one has zero doppler. When analysed without clutter suppression (left) both targets are detected. When the clutter suppression is applied (right) the lower one becomes subtracted at voltage level before the actual lag profile inversion.

4.2 Examples with real data

4.2.1 Autocovariance function measurement with a monostatic radar

Let us now continue with one second of real voltage level signal samples from EISCAT UHF beata experiment from March 13 2013 22:02:36 UT.

We will again define a data input function that loads the data from file,

```
> datafun <- function( LPIparam , intPeriod ){
+
+ # Load the sample data file
+ load('beata20130313.Rdata')
+
+ # Create the output list, we will always simply return
+ # all data in the file
+ odata <- list()
+ beata20130313$itx <- beata20130313$itx>0
+ beata20130313$irx <- beata20130313$irx>0
```

```
+ odata$TX1 <- list(cdata=beata20130313$cdata,idata=beata20130313$itx,ndata=be
+ odata$TX2 <- odata$TX1
+ odata$RX1 <- list(cdata=beata20130313$cdata,idata=beata20130313$irx,ndata=be
+ odata$RX2 <- odata$RX1
+ odata$success <- TRUE
+ return(odata)
+ }
Then we will again call LPI. Because our 'datafun' does not check sam-
pling times we can select arbitrary values for 'startTime', 'stopTime', and
'timeRes.s'.
> LPI(
    startTime = 1356998400,
    stopTime = 1356998401,
    lagLimits = seq(15),
                                        # all intra-pulse lags
                                      # 10 s integration time
    timeRes.s = 1,
    rangeLimits = c(seq(20,50), seq(55,150,by=5))
                                                           , # range gates
    resultDir = NA ,
                                      # we will not write results to files
+
   dataInputFunction = 'datafun' , # our data input function
    resultSaveFunction = 'savefun',  # our function for saving results
   maxClutterRange=20
+
    )
          startTime: 1356998400.000000 (2013-01-01 00:00:00.000000 UT)
           stopTime: 1356998401.000000 (2013-01-01 00:00:01.000000 UT)
      inputPackages:
  dataInputFunction: datafun
dataEndTimeFunction: currentTimes
       clusterNodes:NA
                nup: RX1:1 RX2:1 TX1:1 TX2:1
       filterLength: RX1:1 RX2:1 TX1:1 TX2:1
     decodingFilter: none
          lagLimits: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
        rangeLimits: 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
                     36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 55
                     60 65 70 75 80 85 90 95 100 105 110 115 120 125
                     130 135 140 145 150
          maxRanges: Inf
          timeRes.s: 1.000000
    maxClutterRange: RX1:20 RX2:20
```

```
clutterFraction: RX1:1 RX2:1
 backgroundEstimate: TRUE
          maxWait.s: -1.000000
         freqOffset: RX1:0.000000 RX2:0.000000 TX1:0.000000 TX2:0.000000
        indexShifts: RX1:0 0 RX2:0 0 TX1:0 0 TX2:0 0
             solver: fishs
               nBuf: 10000
          fullCovar: FALSE
      rlips.options: type:c nbuf:1000 workgroup.size:128
           remoteRX: FALSE
             normTX: FALSE
              nCode: NA
          ambInterp: FALSE
          resultDir: NA
 resultSaveFunction: savefun
paramUpdateFunction: noUpdate
             useXDR: FALSE
Let us plot the result again
> image(ACF1$lag,ACF1$range,
        t(Re(ACF1$ACF[1:length(ACF1$range),])),
        col=rev(gray(seq(1000)/1000)), zlim=c(-.2,1.2)*1e-5)
```

4.2.2 Bistatic measurements and crosscovariannce functions

Bistatic and crosscorrelation function measurements are not different from the monostatic analysis from LPI point-of-view. The differences are dealt with in the user-defined 'dataInputFunction', which must be dsigned to return appropriate TX / RX data combinations.

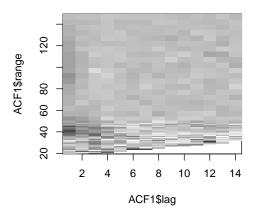


Figure 4.3: LPI analysis of one second of data from an EISCAT UHF beata experiment March 13 2013 22:02:36 UT. The results is very noisy because a very short period of data was used, but a clear E region is visible around range gate 40 and F region around range gate 100.

Chapter 5

Documented source code

5.1 Process control

5.1.1 LPI

The main analysis loop. All user control of LPI takes place via input arguments to LPI, it is the only function that needs to be manually called.

```
1 ## file:LPI.R
2 ## (c) 2010- University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
5
6 ##
7 ## The main analysis loop of LPI
8 ##
9 | ##
10 ##
11
12 LPI <- function(dataInputFunction,
                   inputPackages=c(),
13
                   startTime = 0, # 1st Jan 1970 00:00 UT
14
                   stopTime = 4000000000, # 2nd Oct 2096 07:00
15
16
                   nup = LPIexpand.input( 1 ),
                   filterLength = LPIexpand.input( 1 ),
17
18
                   decodingFilter = "none",
                   lagLimits = c(1,2),
19
20
                   rangeLimits = c(1,2),
                   maxRanges = Inf,
21
22
                   maxClutterRange = 0,
```

```
23
                    clutterFraction = 1,
24
                    timeRes.s = 10,
25
                    backgroundEstimate=TRUE,
26
                    clusterNodes = NA,
27
                    nodeMultip=1,
28
                    useXDR=FALSE,
29
                    maxWait.s = -1,
30
                    freqOffset = LPIexpand.input( 0 ),
31
                    indexShifts = LPIexpand.input( list(c(0,0)) )
                    solver = "fishs",
32
                    nBuf = 10000,
33
34
                    fullCovar = FALSE,
                    rlips.options = list( type="c" , nbuf=1000 ,
35
                        workgroup.size=128),
36
                    remoteRX = FALSE,
37
                    normTX = FALSE,
                    nCode = NA,
38
39
                    ambInterp = FALSE,
40
                    resultDir = paste(format(Sys.time(),"%Y-%m-%d
                        _%H:%M"),'LP',sep='_'),
                    dataEndTimeFunction="currentTimes",
41
42
                    resultSaveFunction = "LPIsaveACF",
43
                    paramUpdateFunction="noUpdate",
44
                    )
45
46
    {
47
       # Collect all input in a list that is handy to pass
48
          forwards
49
       par1 <- formals()</pre>
       par1['...'] <- NULL
50
      par2 <- list(...)</pre>
51
       par1names <- names(par1)</pre>
52
       par1 <- lapply( names( par1 ) , FUN=function(x){ eval( as
53
          .name(x))
       names(par1) <- par1names</pre>
54
55
       LPIparam <- c(par1,par2)
56
57
       # Expand parameters to LPI internal format and set
          storage modes as necessary
       LPIparam[["nup"]] <- LPIexpand.input( LPIparam[["nup"]] )</pre>
58
59
       storage.mode( LPIparam[["nup"]] ) <- "integer"</pre>
60
       LPIparam[["filterLength"]] <- LPIexpand.input( LPIparam[[</pre>
          "filterLength"]] )
       storage.mode( LPIparam[["filterLength"]] ) <- "integer"</pre>
61
       storage.mode( LPIparam[["lagLimits"]] ) <- "integer"</pre>
62
       storage.mode( LPIparam[["rangeLimits"]] ) <- "integer"</pre>
63
```

```
LPIparam[["maxClutterRange"]] <- LPIexpand.input(</pre>
64
          LPIparam[["maxClutterRange"]] )
65
       storage.mode( LPIparam[["maxClutterRange"]] ) <- "integer
      LPIparam[["clutterFraction"]] <- LPIexpand.input(</pre>
66
          LPIparam[["clutterFraction"]] )
       LPIparam[["freqOffset"]] <- LPIexpand.input( LPIparam[["</pre>
67
          freqOffset"]] )
68
       if( ! is.list( LPIparam[["indexShifts"]] ) ){
         LPIparam[["indexShifts"]] <- list(LPIparam[["</pre>
69
            indexShifts"]])
70
      LPIparam[["indexShifts"]] <- LPIexpand.input( LPIparam[["</pre>
71
          indexShifts"]] )
       for( dType in c("TX1","TX2","RX1","RX2")) storage.mode(
72
          LPIparam[["indexShifts"]][[dType]]) <- "integer"</pre>
73
       storage.mode( LPIparam[["nCode"]] ) <- "integer"</pre>
74
75
76
       # Print input arguments
77
       cat(sprintf("%20s %f (%s UT)\n","startTime:",startTime,
          format(as.POSIX1t(startTime, origin='1970-01-01',tz='ut
          '), "%Y-%m-%d %H:%M:%OS6")))
78
       cat(sprintf("%20s %f (%s UT)\n","stopTime:",stopTime,
          format(as.POSIX1t(stopTime,origin='1970-01-01',tz='ut'
          ), "%Y - %m - %d %H : %M : %OS6")))
79
       cat(sprintf("%20s"," inputPackages:"))
80
       for(n in inputPackages){cat(n,", ")}
81
       cat('\n')
       cat(sprintf("%20s %s\n","dataInputFunction:",
82
          dataInputFunction))
       cat(sprintf("%20s %s\n","dataEndTimeFunction:",
83
          dataEndTimeFunction))
84
       cat(sprintf("%20s"," clusterNodes:"))
85
       if( is.list(clusterNodes )){
86
         for(n in names(clusterNodes)){cat(sprintf("%s:",n));cat
            (clusterNodes[[n]],' ')};cat('\n')
87
       }else{
88
         cat( clusterNodes ); cat('\n')
89
       cat(sprintf("%20s","nup:"));for(dType in c("RX1","RX2","
90
          TX1","TX2")){cat(' ',dType,':',LPIparam[["nup"]][[
          dType]],sep='')};cat('\n')
91
       cat(sprintf("%20s","filterLength:"));for(dType in c("RX1"
          ,"RX2","TX1","TX2")){cat(' ',dType,':',LPIparam[["
          filterLength"]][[dType]], sep='')}; cat('\n')
       cat(sprintf("%20s %s\n","decodingFilter:",decodingFilter
92
          [1]))
```

```
cat(lagLimits,fill=70,labels=c(sprintf("%20s","lagLimits:
 93
           "),rep('
                                        ,1000)))
       cat(rangeLimits,fill=70,labels=c(sprintf("%20s","
 94
           rangeLimits:"),rep('
 95
       cat(maxRanges,fill=70,labels=c(sprintf("%20s","maxRanges:
           "),rep('
                                        ',1000)))
       cat(sprintf("%20s %f\n","timeRes.s:",timeRes.s))
 96
 97
        cat(sprintf("%20s RX1:%i RX2:%i \n", "maxClutterRange:",
           LPIparam$maxClutterRange["RX1"],LPIparam$
           maxClutterRange["RX2"]))
        cat(sprintf("%20s RX1:%i RX2:%i \n","clutterFraction:",
 98
           LPIparam$clutterFraction["RX1"],LPIparam$
           clutterFraction["RX2"]))
        cat(sprintf("%20s %s\n","backgroundEstimate:",
 99
           backgroundEstimate))
100
       cat(sprintf("%20s %f\n", "maxWait.s:", maxWait.s))
       cat(sprintf("%20s RX1:%f RX2:%f TX1:%f TX2:%f\n","
101
           freqOffset:",LPIparam$freqOffset["RX1"],LPIparam$
           freqOffset["RX2"],LPIparam$freqOffset["TX1"],LPIparam$
           freqOffset["TX2"]))
102
        cat(sprintf("%20s","indexShifts:"));for(dType in c("RX1",
           "RX2", "TX1", "TX2")){cat(' ', dType, ':', sep=''); cat(
           LPIparam$indexShifts[[dType]])};cat('\n')
103
        cat(sprintf("%20s %s\n","solver:",solver))
       cat(sprintf("%20s %i\n","nBuf:",nBuf))
104
105
       cat(sprintf("%20s %s\n","fullCovar:",fullCovar))
106
        cat(sprintf("%20s", "rlips.options:")); for(n in names(
           rlips.options)){cat(' ',n,':',rlips.options[[n]],sep='
           ')};cat('\n')
107
       cat(sprintf("%20s %s\n","remoteRX:",remoteRX))
       cat(sprintf("%20s %s\n","normTX:",normTX))
108
109
       cat(sprintf("%20s %i\n","nCode:",nCode))
       cat(sprintf("%20s %s\n", "ambInterp:", ambInterp))
110
       cat(sprintf("%20s %s\n","resultDir:",resultDir))
111
       cat(sprintf("%20s %s\n","resultSaveFunction:",
112
           resultSaveFunction))
       cat(sprintf("%20s %s\n","paramUpdateFunction:",
113
           paramUpdateFunction))
114
       cat(sprintf("%20s %s\n","useXDR:",useXDR))
115
       # Total number of integration periods requested
116
       LPIparam[["lastIntPeriod"]] <- round( ( stopTime -</pre>
117
           startTime ) / LPIparam[["timeRes.s"]] )
118
119
       # Create the result directory if a valid path was given
120
       if( is.character( resultDir ) ){
121
          if( nchar( resultDir ) > 0 ){
122
            dir.create( resultDir , recursive=TRUE , showWarnings
               =FALSE )
```

```
123
         }
124
125
126
        # Initialise the computer cluster for LPI.
127
        # Save the cluster definitions to the global workspace
128
        ctrlcl <<- LPIinitCluster( LPIparam[["clusterNodes"]] ,</pre>
           useXDR=useXDR )
129
        ncl <- length( ctrlcl )</pre>
130
131
        # A flag telling whether the analysis will be run in some
        # kind of cluster configuration or in a single process
132
133
        LPIparam[["iscluster"]] <- !all(is.na(ctrlcl))</pre>
134
135
        # Initialize a list for unsolved integration periods
        intPer.missing <- seq( LPIparam[["lastIntPeriod"]] )</pre>
136
137
138
        # Run analysis loop until end of data
139
        endOfData <- FALSE
140
        repeat{
141
142
          # Update the last available data samples
          LPIparam[["dataEndTimes"]] <- eval( as.name( LPIparam[[</pre>
143
             "dataEndTimeFunction"]] ))( LPIparam )
144
145
          # Latest integration period for which data is available
          LPIparam[["maxIntPeriod"]] <- floor( ( min(unlist(</pre>
146
             LPIparam[["dataEndTimes"]])) - LPIparam[["startTime"
             ]] ) / LPIparam[["timeRes.s"]] )
147
148
          # Select integration period numbers for the next
             analysis run
          # Latest periods will be analysed first in order to
149
             simplify real-time analysis
150
          waitSum <- 0
151
          while( is.null( intPer.current <-
             nextIntegrationPeriods( LPIparam , nodeMultip*ncl ,
             intPer.missing ))){
152
153
            # Break the loop after waiting
154
            # long enough for new data
            if( waitSum > LPIparam[["maxWait.s"]] ){
155
              endOfData <- TRUE
156
157
              break
            }
158
159
            # Wait 10 seconds
160
161
            Sys.sleep(10)
162
163
            # Increment the wait time counter
```

```
164
            waitSum <- waitSum + 10</pre>
165
166
            # Update the last available data samples
            LPIparam[["dataEndTimes"]] <- eval( as.name( LPIparam</pre>
167
                [["dataEndTimeFunction"]] ))( LPIparam )
168
169
            # Latest integration period for which data is
               available
170
            LPIparam[["maxIntPeriod"]] <- floor( ( min(unlist(</pre>
               LPIparam[["dataEndTimes"]])) - LPIparam[["
               startTime"]] ) / LPIparam[["timeRes.s"]] )
171
172
          }
173
          if ( endOfData ) break
174
175
176
          # Run analysis on each parallel node, or locally if
177
          # LPIparam[["iscluster"]]==FALSE
          if( LPIparam[["iscluster"]] ){
178
179
            clusterApplyLB( ctrlcl , intPer.current , fun=LPI:::
               LPIsolve.acf , LPIparam )
          }else{
180
181
              for( iper in intPer.current ){
182
                  LPI:::LPIsolve.acf( iper , LPIparam )
183
              }
          }
184
185
186
          # Print something to show that the analysis is running
187
          for( k in seq(length(intPer.current))) cat('.')
188
189
          # Remove the solved periods from the list of missing
          intPer.missing <- setdiff( intPer.missing , intPer.</pre>
190
             current )
191
192
          # Stop if all integration periods are solved
193
          if( length(intPer.missing)==0) break
194
195
        } # repeat
196
197
        # Shut down the cluster at end of analysis
        if(!all(is.na(LPIparam[["clusterNodes"]]))) stopCluster(
198
           ctrlcl )
199
200
        # This function does not return anything,
201
        # results are written to files.
202
        invisible()
203
204
     }
```

5.1.2 LPIexpand.input

```
1 ## file: LPIexpand.input.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
5
6 ##
7 ## Expand input argument list or vector
8 ## into the internally used format
9 | ##
10 ## Arguments:
11 ##
      parvec A vector (or list)
12 ##
13 ## Returns:
14 | ##
      outvec A named vector or list with elements
15 ## "RX1", "RX2", "TX1", and "TX2".
16 ##
17
18 LPIexpand.input <- function( parvec )
19
20
21
       # Names of the input list / vector
22
       namevec <- names(parvec)</pre>
23
       # If the input does not have names attributes, assume
24
25
       \# that the elements are in order RX1 , RX2 , TX1 , TX2
26
       # and repeat as necessary.
27
       if(is.null(namevec)){
28
         # Repeat the input
         outvec
                        <- rep(parvec,length.out=4)
29
30
         # Set names
         names(outvec) <- c( "RX1" , "RX2" , "TX1" , "TX2" )
31
32
         # Return the named vector / list
33
         return(outvec)
34
35
36
       # If the input had names(s), start inspecting them
37
38
       # A vector for the output
39
       outvec <- rep(NA,4)
       names(outvec) <- c( "RX1" , "RX2" , "TX1" , "TX2" )</pre>
40
41
42
       # First look if any of the internally used
43
       # names is used in the input
       if( any(namevec=="RX1")) outvec[1] <- parvec["RX1"]</pre>
44
       if( any(namevec == "RX2")) outvec[2] <- parvec["RX2"]</pre>
45
       if( any(namevec=="TX1")) outvec[3] <- parvec["TX1"]</pre>
46
       if( any(namevec == "TX2")) outvec[4] <- parvec["TX2"]</pre>
47
```

```
48
49
       # If the vector had elements "RX1" , "RX2" , "TX1" ,
50
       # and "TX2", return them in correct order
       if( !any(is.na(outvec))) return(outvec)
51
52
53
       # If there are still missing values,
       # look for elements "RX" and "TX"
54
55
       if( is.na(outvec[1])){
         if(any(namevec=="RX")) outvec[1] <- parvec["RX"]</pre>
56
57
58
       if( is.na(outvec[2])){
59
         if(any(namevec == "RX")) outvec[2] <- parvec["RX"]</pre>
60
61
       if( is.na(outvec[3])){
62
         if(any(namevec == "TX")) outvec[3] <- parvec["TX"]</pre>
63
64
       if( is.na(outvec[4])){
         if(any(namevec=="TX")) outvec[4] <- parvec["TX"]</pre>
65
66
67
68
       # If the vector is now properly filled, return it
69
       if( !any(is.na(outvec))) return(outvec)
70
71
       # Now look for elements "TR1" and "TR2"
72
       if( is.na(outvec[1])){
73
         if(any(namevec == "TR1")) outvec[1] <- parvec["TR1"]</pre>
74
75
       if( is.na(outvec[2])){
76
         if(any(namevec == "TR2")) outvec[2] <- parvec["TR2"]</pre>
77
78
       if( is.na(outvec[3])){
         if(any(namevec=="TR1")) outvec[3] <- parvec["TR1"]</pre>
79
       }
80
81
       if( is.na(outvec[4])){
82
         if(any(namevec == "TR2")) outvec[4] <- parvec["TR2"]</pre>
83
84
85
       # If the vector is now properly filled, return it
86
       if( !any(is.na(outvec))) return(outvec)
87
88
       # Finally remove the named elements from parvec and
89
       # try to fill the output vector
       parvec <- parvec[ nchar(namevec) == 0 ]</pre>
90
91
       if( length(parvec) > 0 ) outvec[is.na(outvec)] <- rep(</pre>
          parvec,length.out=sum(is.na(outvec)))
92
93
       # If the output is now full, return it
       if( !any(is.na(outvec))) return(outvec)
94
95
```

5.1.3 LPIinitCluster.R

```
1 ## file: LPIinitCluster.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 5
 6 ##
 7 ## Initialise the analysis cluster, which consists of:
 8 ##
        - a master process (which calls this function)
 9 | ##
        - length(nodes) control processes running on the same
10 ##
          computer with the master process (if the input list
11 | ##
          does not contain the entry localControl=FALSE)
12 ##
        - length(nodes) control processes running on any
13 ##
          computer on the cluster (names of these computers
14 ##
          are given in names(nodes) )
15 | ##
        - length(nodes[[i]]) computing slaves running on each
16 ##
          remote computer nodes[i] (if nodes[[i]] is a
17 ##
          character vector, these nodes may be also on several
18 ##
          different computers)
19 | ##
20 | ##
       The computing slaves do most of the actual work,
21 ##
       the other processs are mainly for data transfer
22 ##
        (the control processes closest to master do all
23 | ##
       disk I/O)
24 | ##
25 ## Each remote control process will be given one
26 ## integration period of raw data, whose ACF will be
27 ## calculated in parallel by the computing slaves of the
28 ## process. Thus, there are length(nodes) integration
29 ## periods in parallel, and each of them has
30 ## length(nodes[[i]]) parallel lag profiles inversions
31 ## running.
32 ##
33 ## Arguments:
34 | ##
      nodes A list of remote control machine names and
35 | ##
              definitions of computing slaves for each of them
36 ##
              e.g. nodes = list( tesla1=8, tesla2=8,
37 ##
                                  tesla3=8, tesla4=8, tesla5=8)
38 ##
                   or
39 ##
                   nodes = list(
40 | ##
                 tesla1=c( rep('tesla1',8), rep('tesla3',8) ),
41 ##
                 tesla4=c( rep('tesla4',8), rep('tesla5',8) )
42 ##
                                )
43 ##
44 ##
              The former example starts a control process
45 ##
              on each computer of the tesla cluster, and
46 | ##
              allocates one computing slave per core
47 ##
              (each of the computers has 8 cores).
```

```
48 | ##
              Thus, five integration periods are analysed
49 ##
              in parallel with 8 lag profiles in parallel
50 ##
              in each of them. The latter one runs only two
51 ##
              integration periods at a time, but each of
52 ##
              them has 16 lag profiles in parallel.
53 | ##
54 ##
              Notice that the latter option leads to
55 ##
              signicantly larger amount of network traffic,
56 ##
              as the remote control nodes transfer the data
57 ##
              to each computing slave separately
58 ##
59 ##
              Alternatively, one can give an integer number,
60 | ##
              which will start the given number of parallel
61 ##
              processes, running one integration period each
62 | ##
              on localhost. Any combination of the above
63 | ##
              inputs are also accepted.
64 | ##
65 ##
              The list nodes is treated as follows:
66 ##
                1. Put NAs to values <= 0
67 ##
                2. If only NA's were left from 1.,, do not
68 ##
                    start a cluster
69 ##
                    (analysis sequentially in the main process)
70 ##
                3. Unnamed entries are replaced with equal
71 | ##
                    number of entries localhost=1
72 ##
73 ##
              More examples:
74 | ##
                Start 5 parallel integration periods on
75 ##
76 ##
                localhost, and another 5 on "remotecomputer":
                  nodes=list(5,remotecomputer=1,
77 | ##
78 ##
                            remotecomputer=1, remotecomputer=1,
79 ##
                            remotecomputer=1, remotecomputer=1)
80 ##
81 ##
                Start 1 integration period with five parallel
                lags in localhost and another similar one on
82 ##
83 ##
                "remotecomputer":
84 ##
                  nodes=list(localhost=5,remotecomputer=5)
85 | ##
86 ##
                Start 4 parallel integration periods on both
87 ##
                remotecomputer1 and remotecomputer2, but do
88 ##
                not create the local control processes. This
89 ##
                requires that both computers have the input
90 | ##
                and output data directories mounted on same
91 | ##
                paths.
92 ##
                  nodes=list(remotecomputer1=4,
93 ##
                              remotecomputer2=4,
94 ##
                              localControl=FALSE)
95 ##
96 ##
                Do not use parallelism, solve everything
```

```
97 | ##
                  sequentally in the main process
98 ##
                    nodes=NA
99 ##
100 ##
101 | ##
102 | ##
103 ## Returns:
104 ##
         ctrlcl A list of class cluster of
105 ##
                 the local control nodes
106 ##
107 ##
                  The corresponding lists of remote control
108 ##
                  clusters and computing slaves
109 ##
                  are stored on the cluster nodes
110 ##
111
112 LPIinitCluster <- function( nodes , useXDR=FALSE )
113
114
        # Check if nodes has an entry "localControl",
115
116
        # if not, use default (TRUE)
        localControl <- TRUE</pre>
117
        if(is.list(nodes)){
118
          if(is.logical(nodes[["localControl"]])) localControl <-</pre>
119
               nodes[["localControl"]]
120
        }
121
122
        # Replace negative values with NAs
123
        for(k in seq(length(nodes))){
124
          if(is.numeric(nodes[[k]])){
            if(nodes[[k]] <= 0) nodes[[k]] <- NA
125
126
          }
127
128
129
        # If only NA values, we will run locally
130
        if(all(is.na(nodes))) return(NA)
131
132
        # Strip off all NAs (original NAs
133
        # and those from non-positive values)
134
        nodes[is.na(nodes)] <- NULL</pre>
135
136
        # Named nodes are left as such, unnamed are
        # assumed to denote the number of local
137
138
        # parallel integration periods
        nnames <- names(nodes)</pre>
139
140
        if(is.null(nnames)){
141
          ncnames <- rep(0,length(nodes))</pre>
142
        }else{
143
          ncnames <- nchar(names(nodes))</pre>
144
        }
```

```
145
       nodes2 <- c( nodes[ncnames>0] , rep( list( localhost=1 )
           , sum( unlist( nodes[ ncnames == 0 ] ) ) ) )
146
        # Remove the localControl entry
147
148
       nodes2[["localControl"]] <- NULL</pre>
149
        # Create the (optional) local control nodes
150
151
        if(localControl){
152
          # Create the local control nodes
          ctrlcl <- makeCluster( length( nodes2 ) , useXDR=useXDR</pre>
153
154
155
          # Load packaages LPI and parallel to each of the local
             control nodes
          clusterEvalQ( ctrlcl , library(LPI) )
156
157
158
          # Create the remote computer control processes
159
          for(k in 1:length(nodes2)){
160
            # Run initialisation at the local control process to
161
            # create the remote control process and its slaves
162
            \verb|clusterCall(|ctrlcl[k]|, LPIinitRemoteNode|, nodes2[k||
               ] , useXDR )
          }
163
164
165
        # Otherwise the remote nodes will
166
        # act as control nodes as well
167
       }else{
168
169
          # Create the remote control nodes directly
170
          ctrlcl <- makeCluster( names(nodes2) , useXDR=useXDR )</pre>
171
          # Load packaages LPI and parallel to
172
173
          # each of the remote control nodes
          clusterEvalQ( ctrlcl , library(LPI) )
174
175
176
          # Set remcl=NA on each node to notify that
177
          # the additional control step does not exist
178
          remcl <<- NA
179
          clusterExport( ctrlcl , 'remcl')
180
          # Initialise the computing slaves
181
182
          for(k in 1:length(nodes2)){
183
            clusterCall( ctrlcl[k] , LPI:::LPIinitComputingSlaves
                 , nodes2[[k]] , useXDR )
184
       }
185
186
187
       return(ctrlcl)
188
```

189 }

5.1.4 LPIinitRemoteNode.R

```
1 ## file: LPIinitRemoteNode.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 6 ##
 7 ## Establish a scoket connection between a local
 8 ## control process and a remote control process
 9 ## and initialise the computing slaves at the remote
10 | ##
11 ## Arguments:
12 ##
        remnode a list of the form
13 ##
                 remname = list(compslave1,compslave2,...)
14 ##
                 where remname is the remote analysis
                 computer and the list constains either
15 ##
16 ##
                 the number of computing slaves
17 ##
                 on that particular computer, or a vector
18 ##
                 of computer names at which to create the
19 | ##
                 computing slaves
20 ##
21
22 LPIinitRemoteNode <- function( remNode , useXDR )
23
24
25
       # Node name
       nodeName <- names(remNode)</pre>
26
27
28
       # If nodeName is localhost, do not start the remote
       # control process but make direct connections
29
30
       # to the slaves instead.
       if( nodeName == "localhost"){
31
32
         remcl <<- NA
33
         LPI:::LPIinitComputingSlaves( remNode[[1]] , useXDR )
34
         return (remcl)
35
       }
36
37
       # Establish the connection to the remote control nodes
38
       remcl <<- makeCluster( names(remNode) , useXDR=useXDR )</pre>
39
40
       # Load package LPI
       clusterEvalQ( remcl , library(LPI) )
41
42
       # Initialise the computing slaves
43
44
       clusterCall( remcl , LPIinitComputingSlaves , remNode
          [[1]] , useXDR )
45
       return(remcl)
46
```

48 }

5.1.5 LPIinitComputingSlaves.R

```
1 ## file: LPIinitComputingSlaves.R
2 ## (c) 2010- University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6 ##
7 ## Init the actual worker processes,
8 ## "computing slaves", of a remote control node.
9 | ##
10 ## Arguments:
11 | ##
      slaveNodes Cluster node definition, either an integer
12 ##
                  number of cluster nodes or a string vector
13 ##
                  of host names.
14 ##
15 ## Returns:
16 ##
      slavecl
                 An object of class SOCKcluster. The same
17 ##
                 object is also stored on the global workspace
18 ##
19
20 LPIinitComputingSlaves <- function( slaveNodes , useXDR )
21
22
23
      # If only one slave, do not allocate it but
24
      # run analysis in the control process
25
      if( slaveNodes == 1 ){
26
        return( slavecl <<- NA )
27
28
29
      # Create the cluster of computing slaves
30
      slavecl <<- makeCluster( slaveNodes , useXDR=useXDR )</pre>
31
32
      # Load LPI package to each of the nodes
33
       clusterEvalQ( slavecl , library(LPI) )
34
35
      return(slavecl)
36
37
    }
```

5.1.6 currentTimes.R

```
1 ## file:currentTimes.R
 2 ## (c) 2010- University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
6 ##
 7 ## Current unix time minus 5 seconds to be used for
 8 ## identifying the latest available data samples in
9 ## real time analysis.
10 ##
11 ## Arguments:
            An arbitrary list of arguments is accepted, but
      . . .
      none
13 ##
            of them will be used.
14 | ##
15 ## Returns:
16 ##
      curTimes A named vector ("TX1", "TX2", "RX1", "RX2") with
17 ##
                the current unix time -5 in each element.
18 ##
19 ##
20
21 currentTimes <- function( ... )
22
23
      return( LPIexpand.input( as.numeric(Sys.time()-5) ) )
24
    }
```

5.1.7 nextIntegrationPeriods.R

```
1 ## file:nextIntegrationPeriods.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
5
6 ##
7 ## Indices of n latest integration periods
8 ## that have not yet been analysed.
9 | ##
10 ## Arguments:
11 ##
      LPIparam
                     A LPI parameter list
12 | ##
                     Number of new periods to search for
13 | ##
      intPer.ready A list of solved period indices
14 ##
15 ## Returns:
16 ##
                    Indices of the integration periods to
      nextIpers
17 ##
                    be solved next.
18 ##
19
20 nextIntegrationPeriods <- function( LPIparam , n , intPer.
      missing )
21
22
23
24
       # Truly available periods
       intPer.available <- intPer.missing[ which( intPer.missing</pre>
25
           <= min( LPIparam[["maxIntPeriod"]] , LPIparam[["</pre>
          lastIntPeriod"]]) ) ]
26
27
       # We know that the integration periods are in order,
28
       # simply pick the n last ones
29
       nper <- length(intPer.available)</pre>
30
       if(nper==0) return(NULL)
31
       return(intPer.available[ max(1,( nper - n + 1 )) : nper
32
33
34 | ##
         # A vector for the integration period numbers
         nextIpers <- rep(0,n)</pre>
35 ##
36 ##
37 ##
         # Counter for identified new periods
38 | ##
        k <- 0
39 ##
40 ##
         # The period from which we will start seeking backwards
41 ##
         p <- min( LPIparam[["maxIntPeriod"]] , LPIparam[["</pre>
      lastIntPeriod"]] )
42 ##
```

```
43 ##
         # If the last data sample or analysis end time is
      before
         # beginning of analysis, there will be nothing to do
44 ##
         if( p < 0 ) return(NULL)</pre>
45 ##
46 | ##
         # Start looking backwards from the last period
47 ##
         while (k < n) {
48 | ##
           # Select periods that have not yet been analysed.
49 | ##
           if(!any(intPer.ready == p)){
50 ##
             k < - k+1
51 ##
             nextIpers[k] <- p</pre>
52 ##
           }
53 ##
           # Stop looking if we hit the analysis start time
54 ##
           if (p == 1) break
55 ##
           p <- p - 1
56 ##
57 ##
58 ##
         # Return NULL if nothing was found
59 ##
         if( k== 0 ) return(NULL)
60 ##
61 ##
         return( nextIpers[1:k] )
62
63 }
```

5.1.8 LPIsolve.acf.R.

```
1 ## file: LPIsolve.acf.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
5 ##
6
7 ## This function is run in local control nodes.
8 ## Read data for one integration period, send it for
9 ## analysis in a remote computer, and write the returned
10 ## ACF to file
11 | ##
12 ## Arguments:
13 | ##
      intPeriod
                  Integration period number, counted from
14 | ##
                  LPIparam[["firstTime"]] in steps of
                  LPIparam [["timeRes.s"]]
15 ##
16 ##
17 ## Returns:
18 ##
      intPeriod
                 The integration period number.
19 | ##
20
21 LPIsolve.acf <- function( intPeriod , LPIparam )
22
23
       # Load packages that are needed for reading the data
24
      for( pn in LPIparam[["inputPackages"]] ){
25
         require( pn , character.only=TRUE )
26
27
28
       # Parameter list update
      LPIparam <- eval( as.name( LPIparam[["paramUpdateFunction
29
          "]] ))( LPIparam , intPeriod )
30
31
       if( !is.null(LPIparam)){
32
           # Read raw data, name of the data input function
33
           # should be stored in a character string
34
           LPIdatalist.raw
                             <- eval(as.name(LPIparam[["</pre>
              dataInputFunction"]]))( LPIparam , intPeriod )
35
36
           # If data reading was successfull
37
           if(LPIdatalist.raw[["success"]]){
38
             # require that there are at least some TX and RX
39
                 samples
             if( (sum(LPIdatalist.raw[["RX1"]][["idata"]]) > 0)
40
                 (sum(LPIdatalist.raw[["RX2"]][["idata"]]) > 0)
41
                    &
```

```
(sum(LPIdatalist.raw[["TX1"]][["idata"]]) > 0)
42
                 (sum(LPIdatalist.raw[["TX2"]][["idata"]]) > 0))
43
44
45
               # Frequency mixing, filtering, etc., the output
               # collected in a list and stored on the user
46
                  workspace
47
               LPIdatalist.final <<- prepareLPIdata( LPIparam ,
                  LPIdatalist.raw )
48
49
               # Call the function that will send the data to
50
               # proper place and run the actual analysis
               ACF <- LPI:::LPIrun.remote( substitute(
51
                  LPIdatalist.final) )
52
53
               # Store the results
               eval( as.name( LPIparam[["resultSaveFunction"]])
54
                  )( LPIparam , intPeriod , ACF )
55
56
            }
           }
57
58
59
60
      # Return the integration period
61
      # number to the main process
62
      return(intPeriod)
63
64
    }
```

5.1.9 noUpdate.R

```
1 ## file:noUpdate.R
 2 ## (c) 2010- University of Oulu, Finland
 3 ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 6
 7 ##
 8 ## LPI parameter list update function
9 ## Return the list itself in first call,
10 ## NULL in the second call with the same list
11 ##
12 ## Arguments:
13 | ##
      LPIparam A LPI parameter list
14 ##
      intPeriod Integration period number
15 ##
16 ## Returns:
17 ##
      LPIparam An exact copy of the input LPIparam in
                first call, NULL in second call with the
18 ##
19 ##
                same list
20 ##
21
22 noUpdate <- function( LPIparam , intPeriod )
23
24
25
           if(is.null(LPIparam[["callN"]])){
               LPIparam[["callN"]] <- 1</pre>
26
27
               return(LPIparam)
28
29
30
           return(NULL)
31
32
      }
```

5.1.10 LPIrun.remote.R

```
1 ## file: LPIrun.remote.R
2 ## (c) 2010- University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
5 ##
6
7 ##
8 ## Send data to a remote computer and run analysis in there
9 | ##
10 ## Arguments:
11 | ##
      LPIenv.name Name of the LPI environment to use, the
12 ##
                   environment was copied on global workspace
13 ##
                   by LPIsolve.ACF
14 | ##
15 ## Returns:
16 ##
      ACFlist
                   A list that contains the solved ACF, its
17 ##
                   covariance matrices, lags, etc.
18 ##
19
20 LPIrun.remote <- function( LPIenv.name )
21
    {
22
23
       # Check if we are running in a cluster or not
24
       if( eval(LPIenv.name)[["iscluster"]]
25
26
         # Check that this is a local control node
27
         if(!is.na(remcl)){
28
29
           # Send the data environment to the remote node
30
           clusterExport( remcl , paste(LPIenv.name)
31
32
           # Run the remote analysis
33
           ACF <- clusterCall( remcl , LPI:::LPIrun , LPIenv.
              name )
34
35
           # Return the ACF
           return( ACF[[1]] )
36
         }
37
38
39
      }
40
41
       # If the analysis is run in a single process,
       # or if this is a remote control node,
42
43
       # just run the analysis in this process
       return( LPI:::LPIrun( LPIenv.name ) )
44
45
46
     }
```

5.1.11 LPIrun.R

```
1 ## file: LPIrun.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Copy the raw data to computing slaves and solve lag
9 ## profiles in there. Combine the solved profiles into
10 ## a full ACF and return it.
11 | ##
12 ## Arguments:
13 | ##
      LPIenv.name Name of the LPI environment to use
14 | ##
                    for the analysis.
15 ##
16 ## Returns:
17 ## ACFlist
                    A list that contains the solved ACF, its
18 ##
                    covariance matrices, lags, etc.
19 | ##
20 ## The LPI environment, which may be a named list
21 ## as well, must be stored on the global workspace.
22 ##
23
24 LPIrun <- function( LPIenv.name)
25
    {
26
27
       # Number of lags, each full lag
28
       # will get its own call of LPIsolve
       nlags <- eval(LPIenv.name)[["nLags"]]</pre>
29
30
       x <- seq( nlags )
31
32
       # Number of range gates
33
       ngates <- eval(LPIenv.name)[['nGates']]</pre>
34
       maxgates <- max(ngates)</pre>
35
36
       # Are we going to calculate a full covariance matrix?
37
       fullcovar <- eval(LPIenv.name)[['fullCovar']]</pre>
38
39
       # Range-gate centre points
40
       r <- eval(LPIenv.name)[['rangeLimits']]</pre>
41
       rgates <- ( r[1:maxgates] + r[2:(maxgates+1)] -1 ) / 2
42
43
       # Lag-gate centre points
44
       1 <- eval(LPIenv.name)[["lagLimits"]]</pre>
45
       lgates <- ( l[1:nlags] + l[2:(nlags+1)] -1 ) / 2
46
47
       # If the computing slaves do not exist, set
```

```
48
       # slavecl=NA and run the analysis on this process
       if(!exists('slavecl')) slavecl <- NA</pre>
49
50
51
       # If we are running on a cluster
52
       if( eval(LPIenv.name)[["iscluster"]] & ( !is.na( slavecl
           ) ) ) {
53
54
         # Copy the data to all computing slaves
         clusterExport( slavecl , paste( LPIenv.name ) )
55
56
57
         # Allocate necessary vectors on each slave
         clusterCall( slavecl , initLPIenv , LPIenv.name )
58
59
60
         # Run the analysis processes on the slaves
         ACFlist <- clusterApplyLB( slavecl , x , fun=LPI:::
61
            LPIsolve , LPIenv.name=LPIenv.name )
62
63
       # If not running on cluster, solve the lag profiles
64
65
       # sequentially. Mimic the output list of cluster
       # calls in order to simplify further processing.
66
67
      }else{
68
69
         # Create a list for the lag profiles
70
         ACFlist <- vector(mode='list',length=nlags)
71
72
         # Allocate vectors etc.
73
         initLPIenv( LPIenv.name )
74
75
         # Run the actual analysis sequentially
76
         for( k in 1:nlags ){
77
           ACFlist[[k]] <- LPI:::LPIsolve( lag=x[k] , LPIenv.
              name=LPIenv.name )
78
         }
79
80
       }
81
82
       # Collect the lag numbers from ACF list
83
       lagnums <- x
84
       for(k in 1:nlags ){
         lagnums[k] <- ACFlist[[k]][['lagnum']]</pre>
85
86
87
88
       # Find correct order for the lag profiles
89
       lagorder <- x[order(lagnums)]</pre>
90
91
       # Order the ACF list
       ACFlist <- ACFlist[lagorder]
92
93
```

```
94
        # Make ACF and variance matrices
 95
        ACFmat <- matrix(NA,ncol=nlags,nrow=(maxgates+1))
 96
 97
        # Collect the lag profiles to the ACF matrix
        for( k in 1:nlags){
 98
99
          if(ngates[k]>0){
100
            # Copy the solved lag profile
101
            ACFmat[1:ngates[k],k] <- ACFlist[[k]][['lagprof']][1:
               ngates[k]]
102
            # Copy the background ACF estimate
103
                                   <- ACFlist[[k]][['lagprof']][
            ACFmat[maxgates+1,k]
               ngates[k]+1]
104
         }
105
       }
106
107
       # If full covariance matrices were solved
108
        if(fullcovar){
109
          # allocate matrix for variances and a cube for the
             covariance matrices
110
                   <- matrix(NA,ncol=nlags,nrow=(maxgates+1))
111
          COVARmat <- array(NA,dim=c((maxgates+1),(maxgates+1),</pre>
             nlags))
112
          for( k in 1:nlags){
113
            if(ngates[k]>0){
              # Copy variances
114
115
              VARmat[1:ngates[k],k]
                                                        <- Re(diag(
                  ACFlist[[k]][['covariance']]))[1:ngates[k]]
116
              VARmat[maxgates+1,k]
                                                        <- Re(diag(
                 ACFlist[[k]][['covariance']]))[ngates[k]+1]
117
              # Copy covariance matrices
118
              COVARmat[1:ngates[k],1:ngates[k],k]
                                                        <- ACFlist[[
                 k]][['covariance']][1:ngates[k],1:ngates[k]]
119
              COVARmat[(maxgates+1),1:ngates[k],k]
                                                       <- ACFlist[[
                 k]][['covariance']][(ngates[k]+1),1:ngates[k]]
120
              COVARmat[1:ngates[k],(maxgates+1),k]
                                                        <- ACFlist[[
                 k]][['covariance']][1:ngates[k],(ngates[k]+1)]
              COVARmat[(maxgates+1),(maxgates+1),k]
121
                                                       <- ACFlist[[
                 k]][['covariance']][(ngates[k]+1),(ngates[k]+1)]
122
            }
          }
123
        # If only variances were solved
124
125
       }else{
126
          # Allocate a matrix for the variances,
127
          # set COVARmat to NULL
128
          VARmat
                   <- matrix(NA, ncol=nlags, nrow=(maxgates+1))</pre>
129
          COVARmat <- NULL
130
          for( k in 1:nlags){
131
            if ( ngates[k] > 0 ){
132
              # Copy the variances
```

```
VARmat[1:ngates[k],k] <- Re(ACFlist[[k]][['</pre>
133
                  covariance']])[1:ngates[k]]
              VARmat[(maxgates+1),k] <- Re(ACFlist[[k]][[')</pre>
134
                  covariance']])[ngates[k]+1]
            }
135
          }
136
       }
137
138
139
140
        # Collect the results in a list and return it.
        # A list is used because an environment
141
       # is much slover to transfer
142
143
       ACFreturn <- list()
                                    <- ACFmat
144
       ACFreturn[["ACF"]]
       ACFreturn[["var"]]
145
                                    <- VARmat
        ACFreturn[["covariance"]] <- COVARmat
146
147
        ACFreturn[["lag"]]
                                    <- lgates
        ACFreturn[["range"]]
148
                                    <- rgates
        ACFreturn[["nGates"]]
                                    <- ngates
149
150
151
       return(ACFreturn)
152
153
    }
```

5.1.12 initLPIenv.R.

```
1 ## file:initLPIenv.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Allocate and initialise necessary vectors and variables
9 ## for the actual lag profile inversion. This function is
10 ## called once per integration period in each computing slave
11 | ##
12 ## Arguments:
13 | ##
      LPIenv.name Name of the LPI environment used for
14 ##
                   the analysis.
15 ##
16 ## Returns:
17 ##
        Nothing, the udpated environment is stored on
18 ##
        the global workspace.
19 | ##
20
21 initLPIenv <- function( LPIenv.name )
22
23
24
       # Get the LPI environment (transferred as a list,
       # convert into an environment first)
25
      LPIenv <- as.environment( eval( LPIenv.name ) )</pre>
26
27
      # Allocate vector for the range ambiguity function
28
       assign('camb', vector(mode='complex',length=(LPIenv[["
29
          nData"]]*LPIenv[["nDecimTX"]]))
                                               , LPIenv )
30
31
       # Range ambiguity indices
32
       assign( 'iamb', vector(mode='logical',length=(LPIenv[["
          nData"]]*LPIenv[["nDecimTX"]]))
                                              , LPIenv )
33
34
       # Laged products
       assign( 'cprod', vector(mode='complex',length=LPIenv[["
35
          nData"]])
                                                 , LPIenv )
36
37
       # Lagged product indices
       assign('iprod', vector(mode='logical',length=LPIenv[["
38
         nData"]])
                                                 , LPIenv )
39
40
      # Lagged product variances
41
       assign( 'var' , vector(mode='numeric',length=LPIenv[["
          nData"]])
                                                 , LPIenv )
42
```

```
43
       # Theory matrix rows, one extra row because
       # theory_rows needs a temp vector
44
45
       assign( 'arows', vector(mode='complex',length=((max(
          LPIenv[["nGates"]])+1)*(LPIenv[["nBuf"]]+1))), LPIenv
46
47
       # Indices for theory matrix rows, one extra row because
48
       # theory_rows needs a temp vector
       assign( 'irows', vector(mode='logical',length=((max(
49
          LPIenv[["nGates"]])+1)*(LPIenv[["nBuf"]]+1))), LPIenv
50
51
       # Measurement vector
52
       assign( 'meas', vector(mode='complex',length=LPIenv[["
          nBuf"]])
                                                 , LPIenv )
53
      # Measurement variances
54
55
       assign( 'mvar' , vector(mode='numeric',length=LPIenv[["
          nBuf"]])
                                                 , LPIenv )
56
57
      # Buffer row counter
       assign( 'nrows', as.integer(0)
58
          , LPIenv )
59
60
      # Copy the modified environment back
61
       # to the user workspace
62
       assign( paste(LPIenv.name) , LPIenv , envir=.GlobalEnv)
63
64
      return()
65
66
    }
```

5.1.13 LPIsolve.R

```
1 ## file: LPIsolve.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 6
 7 ##
 8 ## Solve the MAP estimate of a lag profile,
 9 ## starting from raw voltage samples
10 ##
11 ## Arguments:
12 ##
        LPIenv
                   A lag profile inversion environment
13 ##
                   Lag number, all fractional lags from
        lag
14 ##
                   LPIenv[["lagLimits"]][lag] to
15 ##
                   LPIenv[["lagLimits"]][lag+1]-1
16 ##
                   are integrated in the same profile
17 ##
18 ## Returns:
19 ##
      lagprof
                   A named list containing the MAP estimate
20 | ##
                   of the lag profile together with
21 ##
                   its (co)variance.
22 ##
24 LPIsolve <- function( lag , LPIenv.name )
25
    {
26
27
       # Get the LPI environment from the global workspace
28
       LPIenv <- eval(LPIenv.name)</pre>
29
30
       # Return immediately if number of gates is <= 0</pre>
       if( LPIenv[["nGates"]][lag] <= 0 ) return(list(lagnum=lag</pre>
31
          ))
32
33
       # If rlisp is used, make sure it has been loaded.
34
       # rlips is not required in startup in order to
35
       # allow analysis without installing it. Other
36
       # solvers are included in the LPI package.
37
       # Switch quietly to fishs if rlips is not available.
       if(LPIenv$solver=="rlips"){
38
39
         require(rlips) -> rres
40
         if( !rres ) assign( 'solver' , 'fishs' , LPIenv )
       }
41
42
43
       # Initialise the inverse problem solver
       if(LPIenv$solver=="rlips"){
44
         solver.env <- rlips.init( ncols = LPIenv$nGates[lag] +</pre>
45
            1 , nrhs = 1 , type = LPIenv$rlips.options[["type"]]
```

```
, nbuf = LPIenv$rlips.options[["nbuf"]] , workgroup
            .size = LPIenv$rlips.options[["workgroup.size"]] )
46
       }else if ( LPIenv$solver=="fishs" ){
47
         solver.env <- fishs.init( LPIenv[["nGates"]][lag] + 1 )</pre>
48
       }else if ( LPIenv[["solver"]]=="deco" ){
49
         solver.env <- deco.init( LPIenv[["nGates"]][lag] + 1 )</pre>
50
       }else if ( LPIenv[["solver"]]=="dummy" ){
51
         solver.env <- dummy.init( range( LPIenv[["rangeLimits"</pre>
            ]][ 1 : (LPIenv[["nGates"]][lag]+1) ]) )
       }else if ( LPIenv[["solver"]]=="ffts" ){
52
         solver.env <- ffts.init( LPIenv[["nGates"]][lag]</pre>
53
            LPIenv[["TX1"]][["idata"]][1:LPIenv[["nData"]]])
54
       }
55
       # Copy of LPIenv[["nData"]]
56
       ndcpy <- LPIenv[["nData"]]</pre>
57
58
59
       # Walk through all fractional time-lags
       for( l in seq( LPIenv[["lagLimits"]][lag] , ( LPIenv[["
60
          lagLimits"]][lag+1] - 1 ) )){
61
62
         # If the lag is longer than the data vector
63
         # it cannot be calculated
64
         if( l >= LPIenv[["nData"]]) break
65
66
         # Current position in data vector, we will skip the
            first nGates samples
67
         assign( "nCur" , as.integer(LPIenv[["rangeLimits"]][
            LPIenv[["nGates"]][lag]+1]+1) , LPIenv)
68
69
         # Calculate the lagged products
70
         laggedProducts( LPIenv , 1 )
71
72
         # Variances of lagged products
73
         lagprodVar( LPIenv , 1 )
74
75
         # Calculate range ambiguity function
76
         rangeAmbiguity( LPIenv , 1 )
77
78
         # Optional pre-averaging of lag-profiles
79
         if( !is.null( LPIenv[["nCode"]] )){
           if( !is.na( LPIenv[["nCode"]] )){
80
81
             if( LPIenv[["nCode"]] > 0 ){
82
               averageProfiles( LPIenv , 1 )
83
               nd <- min( LPIenv[["nData"]] , which( diff(</pre>
                   LPIenv[["TX1"]][["idata"]] ) == 1 )[ LPIenv[["
                   nCode"]] + 1 ] )
84
               LPIenv[["nData"]] <- ifelse( is.na(nd) , LPIenv[[</pre>
                   "nData"]] , nd )
```

```
85
                # Approximate the variance.
 86
                # This is not exactly accurate!
                if(!is.na(nd)) LPIenv[["var"]] <- LPIenv[["var"</pre>
 87
                    ]] / ( sum(diff(LPIenv[["TX1"]][["idata"]])
                    ==1) / LPIenv[["nCode"]] )
 88
              }
 89
            }
          }
 90
 91
          # Solvers "dummy" and "ffts" operate
 92
          # directly with the product vectors
 93
          if( LPIenv[["solver"]] == "dummy" ){
 94
 95
 96
            dummy.add( e
                               = solver.env
97
                      M.data = LPIenv[["cprod"]] ,
98
                      M.ambig = LPIenv[["camb"]]
                       I.ambig = LPIenv[["iamb"]]
99
100
                       I.prod = LPIenv[["iprod"]]
                      E.data = LPIenv[["var"]] , nData = as.
101
                          integer( LPIenv[["nData"]] - 1 ) )
102
103
          }else if( LPIenv[["solver"]]=="ffts"){
104
105
            ffts.add( e
                               = solver.env
106
                      M.data = LPIenv[["cprod"]]
                      M.ambig = LPIenv[["camb"]]
107
                       I.ambig = LPIenv[["iamb"]]
108
109
                       I.prod = LPIenv[["iprod"]]
110
                      E.data = LPIenv[["var"]]
                      nData
                               = as.integer(LPIenv[["nData"]] - 1)
111
112
                       )
113
114
          # Other solvers need theory matrix rows
115
          }else{
116
117
            # Produce theory matrix rows in
            # (small) sets and add them to the solver
118
119
            while ( newrows <- theory Rows ( LPIenv , lag ) ) {
120
121
              # If new rows were produced
              if( LPIenv[["nrows"]]>0){
122
123
124
                # select the correct solver
125
                if(LPIenv$solver=="rlips"){
126
127
                  rlips.add( e = solver.env
                             A.data = LPIenv[["arows"]][1:(LPIenv
128
                                 [["nrows"]] * (LPIenv [["nGates"]][
                                lag]+1))] ,
```

```
129
                             M.data = LPIenv[["meas"]][1:LPIenv[["
                                 nrows"]]] ,
130
                             E.data = LPIenv[["mvar"]][1:LPIenv[["
                                 nrows"]]]
                             )
131
132
133
                }else if(LPIenv$solver=='fishs'){
134
135
                  fishs.add( e = solver.env ,
                             A.data = LPIenv[["arows"]][1:(LPIenv
136
                                 [["nrows"]] * (LPIenv[["nGates"]][
                                 lag]+1))] ,
137
                             M.data = LPIenv[["meas"]][1:LPIenv[["
                                 nrows"]]] ,
                             E.data = LPIenv[["mvar"]][1:LPIenv[["
138
                                 nrows"]]]
139
140
                }else if(LPIenv[["solver"]] == "deco" ){
141
142
                  deco.add( e = solver.env ,
143
                            A.data = LPIenv[["arows"]][1:(LPIenv[[
144
                                "nrows"]] * (LPIenv [["nGates"]] [lag
                            M.data = LPIenv[["meas"]][1:LPIenv[["
145
                                nrows"]]] ,
                            E.data = LPIenv[["mvar"]][1:LPIenv[["
146
                               nrows"]]]
                            )
147
148
149
                }
             }
150
            }
151
152
          }
153
154
          # Make sure that the original value is
          # stored in LPIenv[["nDataa"]]
155
156
          LPIenv[["nData"]] <- as.integer(ndcpy)</pre>
157
158
       }
159
160
        # Solve the inverse problem
161
162
        if(LPIenv$solver=="rlips"){
163
          rlips.solve2( e = solver.env ,full.covariance = LPIenv
             [["fullCovar"]])
164
        }else if(LPIenv$solver=="fishs"){
165
          fishs.solve( e = solver.env , full.covariance = LPIenv
             [["fullCovar"]] )
```

```
}else if(LPIenv[["solver"]]=="deco"){
166
167
         deco.solve( e = solver.env )
       }else if(LPIenv[["solver"]] == "dummy"){
168
169
          dummy.solve( e = solver.env , LPIenv[["rangeLimits"
             [1:(LPIenv[["nGates"]][lag]+1)])
       }else if( LPIenv[["solver"]]=="ffts"){
170
171
          ffts.solve( e = solver.env , LPIenv[["rangeLimits"
             ]][1:(LPIenv[["nGates"]][lag]+1)])
172
173
174
       # Create the return environment
175
       lagprof <- new.env()</pre>
176
177
       # Assign the solution to the new environment
178
       assign( "lagprof" , solver.env[["solution"]] , lagprof )
       assign( "covariance" , solver.env[["covariance"]] ,
179
           lagprof )
180
       assign( "lagnum" , lag , lagprof )
181
182
       # Kill the solver object
183
       if(LPIenv$solver=="rlips") rlips.dispose(solver.env)
184
185
       # Conversion to list because it is faster to transfer
186
       return(as.list(lagprof))
187
188
     }
```

5.2 Signal pre- and post-processing

5.2.1 prepareLPIdata.R

```
1 ## file:prepareLPIdata.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 5 ##
 6
 7 ##
 8 ## Create the LPI environment that is passed from local
 9 | ## control nodes to remote nodes. A list is created instead
10 ## of the final environment because it is faster to transfer.
11 | ##
12 ## Arguments:
13 | ##
      LPIparam
                       An LPI parameter list
14 ##
      LPIdatalist.raw A raw data list returned by a data input
15 ##
                       function. (See e.g. readLPIdata.gdf)
16 ##
17 ## Returns:
18 | ##
      LPIdatalist.final The final data list that is transferred
19 ##
                           to the solver nodes.
20 | ##
21
22 prepareLPIdata <- function( LPIparam , LPIdatalist.raw )
23
24
       # Internally used data vectors
       dTypes <- c( "RX1" , "RX2" , "TX1" , "TX2" )
25
26
27
       # An empty list for the output data
28
      LPIdatalist.final <- vector(mode="list",length=4)
29
      names(LPIdatalist.final) <- dTypes</pre>
30
31
32
       # A list for TX1 pulse start positions in all data
       # vectors (these will be different if sample rates
33
34
       # are different).
35
       # Initialise with zeros to handle data vectors without
36
       # pulses (they will also go through the whole system
37
       # and NA results will be written). The pulseStarts will
       # be passed to c-routines as such, and 0 is thus the
38
39
       # firs index.
       pulseStarts <- list( TX1 = c(0) , TX2 = c(0) , RX1 = c(0)
40
           , RX2 = c(0)
41
42
       # A list for first sample to use in decimation
43
       # in each data vector
```

```
44
       firstSample \leftarrow c(TX1 = 0, TX2 = 0, RX1 = 0, RX2 = 0)
45
46
       # Pulse start positions in TX1 ( >0 used because
47
       # c-routines may have put values larger than one
48
       # to the idata vector)
       pulseStarts[["TX1"]] <- which( diff( LPIdatalist.raw[["</pre>
49
          TX1"]][["idata"]][1:LPIdatalist.raw[["TX1"]][["ndata"
          ]]] > 0 ) == 1 )
50
51
       # Calculate the corresponding pulse
52
       # start positions in other data vectors
53
       for( XXN in dTypes ){
54
         pulseStarts[[XXN]] <- round( as.numeric(pulseStarts[["</pre>
            TX1"]]) / LPIparam[["filterLength"]][["TX1"]] *
            LPIparam[["nup"]][["TX1"]] * LPIparam[["filterLength
            "]][[XXN]] / LPIparam[["nup"]][[XXN]] )
55
         firstSample[[XXN]] <- pulseStarts[[XXN]][1]</pre>
56
      }
57
58
       # The below fix does not work if 'nup' are not common for
           all data vectors.
       # Disable in this case.
59
60
61
       if(all(LPIparam[["nup"]] == LPIparam[["nup"]]["TX1"])){
62
         # Strip off samples to make each
         # IPP a multiple of filter length
63
64
         for( XXN in dTypes ){
65
           # New pulse start positions that
66
           # are even multiples of the filter length
           pstarts2 <- pulseStarts[[XXN]] - round( ( pulseStarts</pre>
67
              [[XXN]] - firstSample[[XXN]] ) %% ( LPIparam[["
              filterLength"]][[XXN]] / LPIparam[["nup"]][[XXN]]
              ) )
68
69
           # Do something only if the pulse positions
70
           # really need to be modified
71
           if( any( pstarts2 != pulseStarts[[XXN]] ) ){
72
73
             # Amount of shift needed in original data
74
             ncut <- pulseStarts[[XXN]] - pstarts2</pre>
75
             ntx <- length(ncut)</pre>
76
77
             # Because we are cutting off data samples,
78
             # the start point k-1 will already be adjusted
79
             # when handling point k. We will thus need to
80
             # subtract the number of points cut in point
             # k-1 from the original ncut[k]. Then take
81
82
             # modulus to make sure that no points will be
83
             # cut unless really necessary and that number
```

```
# of points to cut is not negative
 84
 85
              ncut[2:ntx] <- ncut[2:ntx] - ncut[1:(ntx-1)]</pre>
86
              ncut <- ncut %% round( LPIparam[["filterLength"]][[</pre>
                  XXN]] / LPIparam[["nup"]][[XXN]] )
              ind <- rep( TRUE , LPIdatalist.raw[[XXN]][["ndata"</pre>
 87
                  ]])
              for( k in seq(length(pstarts2)) ){
 88
                if( ( ncut[k] > 0 ) & (pulseStarts[[XXN]][k]<</pre>
 89
                    LPIdatalist.raw[[XXN]][["ndata"]]) ) ind[(
                    pulseStarts[[XXN]][k]-ncut[k]+1):pulseStarts[[
                    XXN]][k]] <- FALSE
 90
 91
              # Number of data points must have changed
 92
              # as samples were cut off, update the values
              LPIdatalist.raw[[XXN]][["ndata"]] <- min(</pre>
 93
                  LPIdatalist.raw[[XXN]][["ndata"]] , sum(ind) )
 94
              LPIdatalist.raw[[XXN]][["cdata"]] <- LPIdatalist.</pre>
                  raw[[XXN]][["cdata"]][ind][1:LPIdatalist.raw[[
                  XXN]][["ndata"]]]
 95
              LPIdatalist.raw[[XXN]][["idata"]] <- LPIdatalist.</pre>
                  raw[[XXN]][["idata"]][ind][1:LPIdatalist.raw[[
                  XXN]][["ndata"]]]
 96
            }
 97
          }
 98
        }
 99
100
101
        # The idata vectors will be modified according
102
        # to LPIparam$indexShift before decimation.
        # Take this into account in firstSamples.
103
104
        # Again keep 0 as the first index, because
105
        # the indices will be passed to c-routines as such
        firstSample[["TX1"]] <- firstSample[["TX1"]] + LPIparam[[</pre>
106
           "indexShifts"]][["TX1"]][1]
107 #
         while( firstSample[["TX1"]] < 0 ){</pre>
108 #
           firstSample[["TX1"]]
                                 <- firstSample[["TX1"]] -
       LPIparam[["filterLength"]][["TX1"]] / LPIparam[["nup"]][["
       TX1"]]
109
        while( firstSample[["TX1"]] < 0 ){</pre>
110
          firstSample[["TX1"]] <- firstSample[["TX1"]] +</pre>
             LPIparam[["filterLength"]][["TX1"]] / LPIparam[["nup
             "]][["TX1"]]
111
112
        firstFraction <- c( TX1 = 0 , TX2 = 0 , RX1 = 0 , RX2 = 0
113
            )
        for( XXN in dTypes ){
114
115
          firstSampleF <- firstSample[["TX1"]] * LPIparam[["</pre>
             filterLength"]][[XXN]] / LPIparam[["filterLength"
```

```
]][["TX1"]] / LPIparam[["nup"]][[XXN]] * LPIparam[["
              nup"]][["TX1"]]
116
          firstSample[[XXN]] <- round( firstSampleF )</pre>
          firstFraction[[XXN]] <- round( ( firstSample[[XXN]] -</pre>
117
              firstSampleF ) * LPIparam[["nup"]][[XXN]] )
        }
118
119
120
121
        # Conversions to integer mode
        storage.mode( firstSample ) <- "integer"</pre>
122
        storage.mode( LPIparam[["filterLength"]] ) <- "integer"</pre>
123
124
        storage.mode( firstFraction ) <- "integer"</pre>
125
126
        # Index corrections, frequency mixing,
127
        # and filtering in C routines
128
        for( XXN in dTypes ){
129
130
          storage.mode( LPIparam[["indexShifts"]][[XXN]] ) <- "</pre>
              integer"
131
132
          LPIdatalist.final[[XXN]] <-</pre>
            .Call( "prepare_data"
133
134
                   LPIdatalist.raw[[XXN]][["cdata"]]
135
                   LPIdatalist.raw[[XXN]][["idata"]]
136
                   LPIdatalist.raw[[XXN]][["ndata"]]
                   LPIparam[["freqOffset"]][XXN]
137
                   LPIparam[["indexShifts"]][[XXN]]
138
139
                   LPIparam[["nup"]][XXN]
140
                   LPIparam[["filterLength"]][XXN]
                   firstSample[[XXN]]
141
142
                   firstFraction[[XXN]]
                   TRUE
143
144
                   )
145
        }
146
147
148
        # Use length of the shortest data vector
149
        LPIdatalist.final[["nData"]] <-</pre>
150
          min(
              LPIdatalist.final[["RX1"]][["ndata"]],
151
              LPIdatalist.final[["RX2"]][["ndata"]],
152
              LPIdatalist.final[["TX1"]][["ndata"]],
153
154
              LPIdatalist.final[["TX2"]][["ndata"]]
155
              )
156
157
158
        # Optional TX amplitude normalisation
159
        if( LPIparam[["normTX"]] ){
```

```
160
          itx1 <- which(LPIdatalist.final[["TX1"]][["idata"]][1:</pre>
             LPIdatalist.final[["nData"]]])
161
          itx2 <- which(LPIdatalist.final[["TX2"]][["idata"]][1:</pre>
             LPIdatalist.final[["nData"]]])
          txamp1 <- mean(abs(LPIdatalist.final[["TX1"]][["cdata"</pre>
162
             ]][itx1]))
          txamp2 <- mean(abs(LPIdatalist.final[["TX2"]][["cdata"</pre>
163
             ]][itx2]))
          LPIdatalist.final[["TX1"]][["cdata"]][itx1] <- exp(1i*
164
             Arg(LPIdatalist.final[["TX1"]][["cdata"]][itx1])) *
             txamp1
165
          LPIdatalist.final[["TX2"]][["cdata"]][itx2] <- exp(1i*
             Arg(LPIdatalist.final[["TX2"]][["cdata"]][itx2])) *
             txamp2
       }
166
167
168
       # Optional ground clutter suppression
       if( ( LPIparam[["maxClutterRange"]]["RX1"] > 0 ) & (
169
           LPIparam[["clutterFraction"]][["RX1"]] > 0 )){
170
          clutterSuppress( LPIdatalist.final[["TX1"]]
             LPIdatalist.final[["RX1"]] , LPIparam[["rangeLimits"
             ]][1] , LPIparam[["maxClutterRange"]]["RX1"] ,
             LPIdatalist.final[["nData"]] , LPIparam[["
             clutterFraction"]][["RX1"]] )
171
172
       if( ( LPIparam[["maxClutterRange"]]["RX2"] > 0 ) & (
           LPIparam[["clutterFraction"]][["RX2"]] > 0 )){
173
          clutterSuppress( LPIdatalist.final[["TX2"]] ,
              LPI datalist.final \hbox{\tt [["RX2"]]} \ , \ LPI param \hbox{\tt [["rangeLimits"]} 
             ]][1] , LPIparam[["maxClutterRange"]]["RX2"] ,
             LPIdatalist.final[["nData"]] , LPIparam[["
             clutterFraction"]][["RX2"]] )
174
       }
175
176
177
       # Optional voltage level decoding
       if( is.numeric( LPIparam[["decodingFilter"]] ) ){
178
179
180
            LPIdatalist.final[["RX1"]][["cdata"]][!LPIdatalist.
               final[["RX1"]][["idata"]]] <- 0+0i
            LPIdatalist.final[["RX2"]][["cdata"]][!LPIdatalist.
181
               final[["RX2"]][["idata"]]] <- 0+0i
            LPIdatalist.final[["TX1"]][["cdata"]][!LPIdatalist.
182
               final[["TX1"]][["idata"]]] <- 0+0i
            LPIdatalist.final[["TX2"]][["cdata"]][!LPIdatalist.
183
               final[["TX2"]][["idata"]]] <- 0+0i
184
185
            nd <- LPIdatalist.final[["nData"]]</pre>
186
```

```
187
            LPIdatalist.final[["RX1"]][["cdata"]] <- LPI:::</pre>
188
               decoFilter.cdata( LPIdatalist.final[["RX1"]][["
               cdata"]][1:nd] , LPIdatalist.final[["TX1"]][["
               cdata"]][1:nd] , LPIdatalist.final[["TX1"]][["
               idata"]][1:nd] , LPIparam[["decodingFilter"]][1] )
189
190
            LPIdatalist.final[["TX1"]][["cdata"]] <- LPI:::</pre>
               decoFilter.cdata( LPIdatalist.final[["TX1"]][["
               cdata"]][1:nd] , LPIdatalist.final[["TX1"]][["
               cdata"]][1:nd] , LPIdatalist.final[["TX1"]][["
               idata"]][1:nd] , LPIparam[["decodingFilter"]][1] )
191
192
            LPIdatalist.final[["RX2"]][["cdata"]] <- LPI:::</pre>
               decoFilter.cdata( LPIdatalist.final[["RX2"]][["
               cdata"]][1:nd] , LPIdatalist.final[["TX2"]][["
               cdata"]][1:nd] , LPIdatalist.final[["TX2"]][["
               idata"]][1:nd] , LPIparam[["decodingFilter"]][1] )
193
194
            LPIdatalist.final[["TX2"]][["cdata"]] <- LPI:::</pre>
               decoFilter.cdata( LPIdatalist.final[["TX2"]][["
               cdata"]][1:nd] , LPIdatalist.final[["TX2"]][["
               cdata"]][1:nd] , LPIdatalist.final[["TX2"]][["
               idata"]][1:nd] , LPIparam[["decodingFilter"]][1] )
195
       }else if( is.character( LPIparam[["decodingFilter"]] )){
196
197
198
            if( any( LPIparam[["decodingFilter"]][1] == c("
               matched","inverse") ) ){
199
200
                LPIdatalist.final[["RX1"]][["cdata"]][!
                   LPIdatalist.final[["RX1"]][["idata"]]] <- 0+0i
                LPIdatalist.final[["RX2"]][["cdata"]][!
201
                   LPIdatalist.final[["RX2"]][["idata"]]] <- 0+0i
202
                LPIdatalist.final[["TX1"]][["cdata"]][!
                   LPIdatalist.final[["TX1"]][["idata"]]] <- 0+0i
                LPIdatalist.final[["TX2"]][["cdata"]][!
203
                   LPIdatalist.final[["TX2"]][["idata"]]] <- 0+0i
204
205
                nd <- LPIdatalist.final[["nData"]]</pre>
206
207
208
                LPIdatalist.final[["RX1"]][["cdata"]] <- LPI:::</pre>
                   decoFilter.cdata( LPIdatalist.final[["RX1"]][[
                   "cdata"]][1:nd] , LPIdatalist.final[["TX1"]][[
                   "cdata"]][1:nd] , LPIdatalist.final[["TX1"]][[
                   "idata"]][1:nd] , LPIparam[["decodingFilter"
                   ]][1])
209
```

```
210
                LPIdatalist.final[["TX1"]][["cdata"]] <- LPI:::</pre>
                    decoFilter.cdata( LPIdatalist.final[["TX1"]][[
                    "cdata"]][1:nd] , LPIdatalist.final[["TX1"]][[
                    "cdata"]][1:nd] , LPIdatalist.final[["TX1"]][[
                    "idata"]][1:nd] , LPIparam[["decodingFilter"
                   ]][1])
211
                LPIdatalist.final[["RX2"]][["cdata"]] <- LPI:::</pre>
212
                    decoFilter.cdata( LPIdatalist.final[["RX2"]][[
                    "cdata"]][1:nd] , LPIdatalist.final[["TX2"]][[
                    "cdata"]][1:nd] , LPIdatalist.final[["TX2"]][[
                   "idata"]][1:nd] , LPIparam[["decodingFilter"
213
                LPIdatalist.final[["TX2"]][["cdata"]] <- LPI:::</pre>
214
                    decoFilter.cdata( LPIdatalist.final[["TX2"]][[
                    "cdata"]][1:nd] , LPIdatalist.final[["TX2"]][[
                    "cdata"]][1:nd] , LPIdatalist.final[["TX2"]][[
                    "idata"]][1:nd] , LPIparam[["decodingFilter"
                    ]][1])
215
                LPIdatalist.final[["TX1"]][["idata"]] <- LPI:::</pre>
216
                    decoFilter.idata( LPIdatalist.final[["TX1"]][[
                    "idata"]][1:nd] )
217
                LPIdatalist.final[["TX2"]][["idata"]] <- LPI:::</pre>
218
                    decoFilter.idata( LPIdatalist.final[["TX2"]][[
                    "idata"]][1:nd] )
219
            }
220
221
222
223
       # Largest range in rangeLimits
224
       maxr <- as.integer(max(LPIparam[["rangeLimits"]]))</pre>
225
226
        # Average signal powers, loop three times in order to
           make simple noise spike detection as well
227
        for(niter in seq(3)){
228
229
          # Average power in signal vector RX1
          LPIdatalist.final[["RX1"]][["power"]] <-</pre>
230
             LPIaveragePower( LPIdatalist.final[["RX1"]][["cdata"
             ]] , LPIdatalist.final[["TX1"]][["idata"]]
             LPIdatalist.final[["RX1"]][["idata"]] , LPIdatalist.
             final[["nData"]] , maxr )
231
232
          # Average power in signal vector RX2
          LPIdatalist.final[["RX2"]][["power"]] <-</pre>
233
             LPIaveragePower( LPIdatalist.final[["RX2"]][["cdata"
```

```
]] , LPIdatalist.final[["TX2"]][["idata"]] ,
             LPIdatalist.final[["RX2"]][["idata"]] , LPIdatalist.
             final[["nData"]] , maxr )
234
235
         # Flag data points whose power is more than four times
             the average at a given height,
         # but only if there were reasonably many samples in the
236
             averages
         if(LPIdatalist.final[["RX1"]][["power"]][1] < .05 ){</pre>
237
238
             itx1 <- which( abs(LPIdatalist.final[["RX1"]][["</pre>
                 cdata"]][1:LPIdatalist.final[["nData"]]]) > (
                 sqrt(LPIdatalist.final[["RX1"]][["power"]])*4) )
239
             LPIdatalist.final[["RX1"]][["idata"]][itx1] <-</pre>
                 FALSE
240
241
         if(LPIdatalist.final["RX2"]][["power"]][1] < .05){
242
             itx2 <- which( abs(LPIdatalist.final[["RX2"]][["</pre>
                 cdata"]][1:LPIdatalist.final[["nData"]]]) > (
                 sqrt(LPIdatalist.final[["RX2"]][["power"]])*4) )
243
             LPIdatalist.final[["RX2"]][["idata"]][itx1] <-</pre>
                 FALSE
244
         }
245
246
247
       # maxr points in the beginning will not have
       # a reasonable power estimate, flag these points as well
248
       LPIdatalist.final[["RX1"]][["idata"]][1:maxr] <- FALSE</pre>
249
250
       LPIdatalist.final[["RX2"]][["idata"]][1:maxr] <- FALSE</pre>
251
252
       253
       ## Copy parameters from LPIparam to ##
       ## the final data list as necessary ##
254
255
       256
       # Lag values
257
258
       LPIdatalist.final[["lagLimits"]] <- LPIparam[["lagLimits"
          ]]
259
       LPIdatalist.final[["nLags"]]
                                         <- length(LPIdatalist.
           final[["lagLimits"]]) - 1
260
261
262
       # Maximum ranges, repeat the last value as necessary
263
       LPIdatalist.final[["maxRanges"]] <- LPIparam[["maxRanges</pre>
           "]]
264
       nmaxr <- length(LPIdatalist.final[["maxRanges"]])</pre>
265
       if( nmaxr < LPIdatalist.final[["nLags"]] ){</pre>
         LPIdatalist.final[["maxRanges"]] <- c( LPIdatalist.
266
             final[["maxRanges"]] , rep(LPIdatalist.final[["
             maxRanges"]][nmaxr],(LPIdatalist.final[["nLags"]]-
```

```
nmaxr)))
        }
267
268
269
270
271
272
        # Range gate limits
        LPIdatalist.final[["rangeLimits"]] <- LPIparam[["</pre>
273
           rangeLimits"]]
274
        LPIdatalist.final[["nGates"]] <- rep( length(LPIparam[["</pre>
           rangeLimits"]]) - 1 , LPIdatalist.final[["nLags"]] )
        for( k in seq(LPIdatalist.final[["nLags"]]) ){
275
276
          LPIdatalist.final[["nGates"]][k] <- length( LPIparam[["
             rangeLimits"]][ LPIparam[["rangeLimits"]] <</pre>
             LPIdatalist.final[["maxRanges"]][k] ] ) - 1
277
        }
278
279
        # The TX vectors are always decimated
        # in the present version
280
281
        LPIdatalist.final[["nDecimTX"]] <- 1</pre>
282
283
        # Number of theory matrix rows to buffer
284
        LPIdatalist.final[["nBuf"]] <- LPIparam[["nBuf"]]</pre>
285
286
        # Inverse problem solver
287
        LPIdatalist.final[["solver"]] <- LPIparam[["solver"]]</pre>
288
289
        # Options to rlips
290
        LPIdatalist.final[["rlips.options"]] <- LPIparam[["rlips.</pre>
           options"]]
291
292
        # Do we calculate background ACF estimates
293
        LPIdatalist.final[["backgroundEstimate"]] <- LPIparam[["</pre>
           backgroundEstimate"]]
294
295
        # Should full covariance matrix or only its
296
        # diagonal be calculated
297
        LPIdatalist.final[["fullCovar"]] <- LPIparam[["fullCovar"
           ]]
298
299
        # Are we running in a cluster or locally
        LPIdatalist.final[["iscluster"]] <- LPIparam[["iscluster"</pre>
300
301
302
        # Is the rx data from a remote site?
303
        LPIdatalist.final[["remoteRX"]] <- LPIparam[["remoteRX"]]</pre>
304
305
        # Number of codes if pre-averaging is being used
        LPIdatalist.final[["nCode"]] <- LPIparam[["nCode"]]</pre>
306
```

```
307
308
       # Should interpolation be used when calculating
309
       # the range ambiguity functions
       LPIdatalist.final[["ambInterp"]] <- LPIparam[["ambInterp"
310
           ]]
311
312
       # Make sure that the storage modes are correct
313
        storage.mode(LPIdatalist.final[["TX1"]][["cdata"]])
           complex"
        storage.mode(LPIdatalist.final[["TX2"]][["cdata"]])
314
           complex"
        storage.mode(LPIdatalist.final[["TX1"]][["idata"]])
315
           logical"
        storage.mode(LPIdatalist.final[["TX2"]][["idata"]])
                                                               <- "
316
           logical"
        storage.mode(LPIdatalist.final[["RX1"]][["cdata"]])
317
           complex"
       storage.mode(LPIdatalist.final[["RX2"]][["cdata"]])
318
           complex"
        storage.mode(LPIdatalist.final[["RX1"]][["idata"]])
319
           logical"
        storage.mode(LPIdatalist.final[["RX2"]][["idata"]])
320
                                                               <- "
           logical"
321
        storage.mode(LPIdatalist.final[["RX1"]][["power"]])
           double"
        storage.mode(LPIdatalist.final[["RX2"]][["power"]])
322
           double"
323
       storage.mode(LPIdatalist.final[["lagLimits"]])
           integer"
        storage.mode(LPIdatalist.final[["rangeLimits"]])
324
           integer"
        storage.mode(LPIdatalist.final[["nDecimTx"]])
325
           integer"
326
       storage.mode(LPIdatalist.final[["nBuf"]])
           integer"
        storage.mode(LPIdatalist.final[["nData"]])
327
           integer"
328
        storage.mode(LPIdatalist.final[["nGates"]])
           integer"
        storage.mode(LPIdatalist.final[["nLags"]])
329
           integer"
        storage.mode(LPIdatalist.final[["nCode"]])
330
           integer"
331
        storage.mode(LPIdatalist.final[["ambInterp"]])
                                                               <- "
           logical"
        storage.mode(LPIdatalist.final[["backgroundEstimate"]])
332
           <- "logical"
333
334
       return( LPIdatalist.final )
```

336 }

5.2.2 clutterSuppress.R

```
1 ## file:clutterSuppress.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 6
 7 ##
 8 ## Ground clutter suppression as follows:
10 ## 1. Scattered signal in ranges between rmin and rmax is
         solved by means of voltage-level inversion.
12 ## 2. The solved profile is convolved with the transmission
13 | ##
         envelope and the convolution is subtracted from the
14 ##
         receiver samples.
15 ##
16 ## Arguments:
17 ##
      txdata A transmitter data list that contains named
18 ##
               vectors 'cdata' and 'idata'
19 ##
      rxdata A receiver data list that cntains named
20 ##
               vectors 'cdata' and 'idata'
21 ##
               Smallest range from which clutter should
      rmin
22 ##
               be suppressed
23 | ##
      rmax
               Largest range from which clutter should
24 ##
               be suppressed
25 ##
      ndata
               Number of points in data vectors
       clutterFraction Fraction of the full integration
26 | ##
27 ##
               period used for the clutter profile estimation
28 ##
               A float from the interval (0,1]
29 ##
30 ## Returns:
31 | ##
      solution The solved clutter profile
32 ##
33 ## Clutter-suppressed receiver data is written to the
34 ## vector rxdata[["cdata"]]
35 | ##
36
37 clutterSuppress <- function( txdata , rxdata , rmin , rmax ,
      ndata , clutterFraction )
38
39
40
       # If rmin > rmax there will be nothing to subtract
41
       if( rmin > rmax ) return()
42
43
       # No reason to continue if ndata is not positive
       if( ndata <= 0 ) return()</pre>
44
45
46
       # Set negative ranges to zero
```

```
47
       rmin <- max( rmin , 0 )</pre>
48
       rmax <- max( rmax , 0 )</pre>
49
50
51
       # Number of range gates to solve
52
       nr \leftarrow rmax - rmin + 1
53
54
       # Initialize a fishs object
       e <- fishs.init( ncols = nr )
55
56
       # number of points used in clutter profile estimation
57
       nclutter <- round( ndata * min( clutterFraction , 1 ) )</pre>
58
59
60
       # Set correct storage modes
61
       storage.mode( ndata ) <- "integer"</pre>
62
       storage.mode( nclutter ) <- "integer"
       storage.mode( rmin ) <- "integer"</pre>
63
64
       storage.mode( rmax ) <- "integer"</pre>
65
66
       # Add data to the inverse problem
67
       nrow <- .Call( "clutter_meas",</pre>
68
                       txdata[["cdata"]],
                       txdata[["idata"]],
69
70
                       rxdata[["cdata"]],
                       rxdata[["idata"]],
71
72
                       ndata,
73
                       rmin,
74
                       rmax,
75
                       e[["Qvec"]],
76
                       e[["y"]]
77
                       )
78
79
       # Do not subtract if the number of measurement rows
80
       # is smaller than number of unknowns
       if( nrow < nr ){</pre>
81
         warning("Not enough data points for clutter suppression
82
             .")
83
         invisible( NULL )
84
85
86
       # Otherwise solve the inverse problem
87
       fishs.solve(e)
88
       # The unmeasured points should be zero instead of NA
89
90
       e[["solution"]][is.na(e[["solution"]])] <- 0+0i
91
92
       # Do the actual subtraction
       ncor <- .Call( "clutter_subtract",</pre>
93
                       txdata[["cdata"]],
94
```

```
txdata[["idata"]],
rxdata[["cdata"]],
95
96
                          rxdata[["idata"]],
97
98
                          ndata,
99
                          rmin,
                          rmax,
100
101
                          e[["solution"]]
102
103
104
         invisible(e$solution)
105
106
     }
```

5.2.3 decoFilter.R.

```
1 ## file:decoFilter.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Voltage level decoding, either matched or inverse
9 ## filtering, using measured transmitter samples.
10 | ##
11 ## Arguments:
12 | ##
                   A complex receiver data vector
      cdata
13 ##
      cenv
                   A complex transmitter data vector
14 ##
                   A logical vector of transmitter data indices
      idata
15 ## filterType Decoding filter. Either a complex vector of
      filter taps, 'matched' or 'inverse'
16 ##
17 ## Returns:
18 ##
      cdata The complex receiver data vector after decoding
19 ##
20
21 decoFilter.cdata <- function( cdata , cenv , idata ,
      filterType='inverse')
22
      # Pulse start positions and number of pulses
23
      txstarts <- which( diff(idata>0) == 1 )
24
25
       if(idata[1]) txstarts <- c(0,txstarts)</pre>
26
      ntx <- length(txstarts)</pre>
27
       txstarts <- c(txstarts,length(cdata))</pre>
28
29
30
       # If there are no transmission pulses, then simply return
31
       if(ntx<1) return(cdata)</pre>
32
33
       # Set the data points before the first pulse to zero
34
       if( txstarts[1] > 0 ) cdata[1:txstarts[1]] <- 0+0i</pre>
35
36
37
       # Set transmitter data to zero at points that are not
          transmitter samples
       cenv[!idata] <- 0+0i
38
39
       # Filtering with user-defined coefficients
40
41
       if( is.numeric( filterType ) ){
42
           nfilter <- length(filterType)</pre>
43
           for( k in seq( ntx ) ){
               cenv[ (txstarts[k]+1) : (txstarts[k+1]) ] <- 0+0i</pre>
44
```

```
45
               cenv[ (txstarts[k]+1) : (txstarts[k]+nfilter)]
                   <- filterType
46
               cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] <-</pre>
47
                   fft(
                        fft( cdata[ (txstarts[k]+1) : (txstarts[k
48
                           +1]) ] ) /
                        fft( cenv[ (txstarts[k]+1) : (txstarts[k
49
                           +1]) ])
50
                         inverse=TRUE ) /
51
                            (txstarts[k+1]-txstarts[k]) * sqrt(
                               sum(abs(cenv[ (txstarts[k]+1) : (
                               txstarts[k+1]) ])**2))
52
           }
       }else if( is.character( filterType ) ){
53
           # Inverse filtering
54
           if(filterType == "inverse"){
55
56
               for( k in seq( ntx ) ){
                    cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] <-</pre>
57
58
                        fft(
59
                            fft( cdata[ (txstarts[k]+1) : (
                                txstarts[k+1]) ] ) /
60
                            fft( cenv[ (txstarts[k]+1) : (
                               txstarts[k+1]) ] )
61
                             inverse=TRUE ) /
62
                                (txstarts[k+1]-txstarts[k]) *
                                    sqrt(sum(abs(cenv[ (txstarts[k
                                    ]+1) : (txstarts[k+1]) ])**2))
63
               }
64
           # Matched filtering
           }else if(filterType=="matched"){
65
66
               for( k in seq( ntx ) ){
                    cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] <-</pre>
67
                        fft(
68
69
                            fft( cdata[ (txstarts[k]+1) : (
                               txstarts[k+1]) ] ) *
                            Conj( fft( cenv[ (txstarts[k]+1) : (
70
                               txstarts[k+1]) ] ) )
71
                            , inverse=TRUE ) /
72
                                (txstarts[k+1]-txstarts[k]) /
                                    sqrt(sum(abs(cenv[ (txstarts[k
                                    ]+1) : (txstarts[k+1]) ])**2))
73
74
           # Other filters are not supported at the moment
75
           }else{
76
               stop("Unknown decoding filter")
77
           }
78
       }else{
79
           stop("Unknown decoding filter")
80
       }
```

```
81
82
     return(cdata)
83
84 }
85
86 ##
87 ## Index corrections for decoded receiver data
88 ##
89 ##
90 ## Arguments:
91 ##
       idata A logical vector of transmitter data indices
92 ##
93 ## Returns:
94 ##
       idata A corrected index vector with only first index
95 ##
             of each pulse set.
96 ##
97
98 decoFilter.idata <- function( idata )
99
100
101
        # Pulse start positions
102
       txstarts <- which( diff(idata>0) == 1 )
103
        if(idata[1]) txstarts <- c(0,txstarts)</pre>
104
       ntx <- length(txstarts)</pre>
105
       txstarts <- c(txstarts,length(idata))</pre>
106
        # Each pulse should have been compressed into
107
108
        # a single sample in the decoding
109
       for( k in seq( ntx ) ){
110
          idata[(txstarts[k]+2):txstarts[k+1]] <- FALSE</pre>
111
       }
112
113
       return(idata)
114
115 }
```

5.2.4 LPIaveragePower.R

```
1 ## file: LPIaveragePower.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 6
 7 ##
 8 ## Average power profiles
 9 | ##
10 ## Arguments:
11 ##
      cdata
                A complex data vector
12 | ##
       idatatx A logical vector of transmitter pulse positions
       idatarx A logical vector of usable receiver samples
13 | ##
14 | ##
                Number points in data vectors
       ndata
       maxrange Largest range from which the power is needed
15 | ##
16 ##
17 ## Returns:
18 ##
     pdata
             Average power profile vector
19 | ##
20
21 LPIaveragePower <- function( cdata , idatatx , idatarx ,
      ndata , maxrange )
22
23
       # Call the C function
       pow <- .Call( "average_power" , cdata , idatatx , idatarx</pre>
24
           , ndata , maxrange )
25
26
       \# Check the first element, .01 means that number of
       # summed power values is 10 in average.
27
28
       # The first element will be NA if no pulses were found,
29
       # then it does not really matter what we do..
30
       if( is.na( pow[1] ) ){
         pow[] <- mean( abs( cdata[idatarx])**2 )</pre>
31
32
       else if(pow[1] > .1){
33
         pow[] <- mean( abs( cdata[idatarx])**2 )</pre>
34
35
36
       return(pow)
37
```

5.2.5 LPIsaveACF.R

```
1 ## file: LPIsaveACF.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Save resolved ACF to file
9 | ##
10 ## Arguments:
11 ##
      LPIparam A LPI parameter list
12 | ##
       intPeriod Integration period number
13 | ##
       ACF
                 An ACF list returned by LPIsolve
14 ##
15 ## Returns:
16 ##
      resFile
                 Result file name
17 ##
18
19 LPIsaveACF <- function( LPIparam , intPeriod , ACF )
20
21
       # Number of range gates
22
       ngates <- length(ACF[["range"]])</pre>
23
       # Number of lags
24
25
       nlags <- length(ACF[["lag"]])</pre>
26
27
       # Seconds since 1970
       ACF[["time.s"]] <- LPIparam[["startTime"]] + intPeriod*
28
          LPIparam[["timeRes.s"]]
29
30
       # The same time as a string, useful for debugging
       # time conversions and for plotting
31
32
       ACF[["timeString"]] <-
33
         format( as.POSIXct( ACF[["time.s"]] , origin='
            1970-01-01', tz='UTC'), "%Y-%m-%d %H:%M:%OS3 UT")
34
35
       # Result file name
       resFile <- gsub(' ','0',file.path( LPIparam[["resultDir"</pre>
36
          ]] , paste( sprintf( '%13.0f' , trunc( ACF[["time.s"]]
            * 1000 ) ) , "LP.Rdata" , sep='') ))
37
38
       # Range
       names(ACF[["range"]]) <- paste('gate', seq(ngates), sep='')</pre>
39
40
41
       names(ACF[["lag"]]) <- paste('lag', seq(nlags), sep='')</pre>
42
43
```

```
44
       # Background ACF
45
       ACF[["backgroundACF"]] <- ACF[["ACF"]][(ngates+1),]
46
       ACF[["backgroundvar"]] <- ACF[["var"]][(ngates+1),]
47
       names(ACF[["backgroundACF"]]) <- paste('lag',seq(nlags),</pre>
          sep=',')
       names(ACF[["backgroundvar"]]) <- paste('lag', seq(nlags),</pre>
48
          sep=',')
49
50
       # ACF and variance without the background samples
       ACF[["ACF"]] <- matrix(ACF[["ACF"]][1:ngates,],ncol=nlags
51
       ACF[["var"]] <- matrix(ACF[["var"]][1:ngates,],ncol=nlags
52
       dimnames(ACF[["ACF"]]) <- list(paste('gate', seq(ngates),</pre>
53
          sep=''), paste('lag', seq(nlags), sep=''))
       dimnames(ACF[["var"]]) <- list(paste('gate', seq(ngates),</pre>
54
          sep=''), paste('lag', seq(nlags), sep=''))
55
       # Dimnames for the optional full covariance matrix
56
57
       if(LPIparam[["fullCovar"]]) dimnames(ACF[["covariance"]])
           <- list( c(paste('gate',seq(ngates),sep=''),'
          background') , c(paste('gate',seq(ngates),sep=''),'
          background') , paste('lag', seq(nlags), sep=''))
58
59
       # Strip off skipped time lags
        laginds <- apply( ACF[["ACF"]] , FUN=function(x){ any( !</pre>
60 #
      is.na(x))}, MARGIN = 2)
       laginds <- which( c( LPIparam[["maxRanges"]] , rep(</pre>
61
          LPIparam [["maxRanges"]] [length (LPIparam [["maxRanges"
          ]])] , nlags ))[1:nlags] >= LPIparam[["rangeLimits"
          ]][1] )
62
       ACF <- stripACF( ACF , rgates = seq( ngates ) , lags=
          laginds , fullCovar=LPIparam[["fullCovar"]])
63
64
       # Range gate limits
       ACF[["rangeLimits"]] <- LPIparam[["rangeLimits"]]
65
       names(ACF[["rangeLimits"]]) <- ""</pre>
66
67
68
       # Lag integration limits
       ACF[["lagLimits"]] <- LPIparam[["lagLimits"]]
69
70
       names(ACF[["lagLimits"]]) <- ""</pre>
71
72
       # Maximum ranges
73
       ACF[["maxRanges"]] <- LPIparam[["maxRanges"]]
74
       names(ACF[["maxRanges"]]) <- ""</pre>
75
76
       # Write the output list to the file
77
       save( ACF=ACF , file=resFile )
78
```

```
79  # Return the file name invisibly
80  invisible( resFile )
81
82 }
```

5.2.6 stripACF.R

```
1 ## file:stripACF.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Return an ACF list with only selected ranges and lags
9 | ##
10 ## Arguments:
                   An ACF list returned by runLPI or stored by
11 | ##
        ACFlist
      LPIsaveACF
12 ##
                   Range gate indices
        rgates
13 ##
        lags
                   Lag indices
                   TRUE if the ACFlist contains the full
14 | ##
        fulCovar
      covariance matrices
15 ##
16 ## Returns:
17 ##
        ACFlist
                   A modified ACF list
18 ##
19
20 stripACF <- function( ACFlist , rgates , lags , fullCovar=
      FALSE)
21
22
23
       # An empty list for the output
24
       ACFlist2 <- list()
25
26
       # If rgates and lags are logical vectors
27
       # convert them into indices
28
       if(is.logical(rgates)) rgates <- which(rgates)</pre>
29
       if(is.logical(lags)) lags <- which(lags)</pre>
30
31
       # Pick the ACF and variance values
32
       ACFlist2[["ACF"]] <- ACFlist[["ACF"]][rgates,lags]</pre>
       ACFlist2[["var"]] <- ACFlist[["var"]][rgates,lags]</pre>
33
34
       # Make sure that ACF, var, and covariance are still
35
          arrays
36
       dim(ACFlist2[["ACF"]]) <- c( length(rgates) , length(lags</pre>
37
       dim(ACFlist2[["var"]]) <- c( length(rgates) , length(lags</pre>
38
       if(fullCovar){
39
         covdims <- dim(ACFlist[["covariance"]])</pre>
         ACFlist2[["covariance"]] <- ACFlist[["covariance"]][c(
40
            rgates, covdims[1]),c(rgates,covdims[2]),lags]
```

```
41
         dim(ACFlist2[["covariance"]]) <- c( (length(rgates)+1)</pre>
             , (length(rgates)+1) , length(lags) )
       }
42
43
44
45
       ACFlist2[["lag"]] <- ACFlist[["lag"]][lags]
46
       ACFlist2[["range"]] <- ACFlist[["range"]][rgates]
47
       ACFlist2[["nGates"]] <- pmin(rep(length(rgates),length(
          lags)), ACFlist[["nGates"]][lags])
       ACFlist2[["backgroundACF"]] <- ACFlist[["backgroundACF"
48
          ]][lags]
49
       ACFlist2[["backgroundvar"]] <- ACFlist[["backgroundvar"
          ]][lags]
       ACFlist2[["timeString"]] <- ACFlist[["timeString"]]</pre>
50
       ACFlist2[["time.s"]] <- ACFlist[["time.s"]]
51
52
53
       # Udpate names to match with the new indexing
       nlags <- length(lags)</pre>
54
55
       ngates <- length(rgates)</pre>
56
57
       names(ACFlist2[["range"]]) <- paste('gate',seq(ngates),</pre>
          sep=',')
58
       names(ACFlist2[["lag"]]) <- paste('lag',seq(nlags),sep='')</pre>
59
       names(ACFlist2[["backgroundACF"]]) <- paste('lag',seq(</pre>
          nlags),sep='')
60
       names(ACFlist2[["backgroundvar"]]) <- paste('lag',seq(</pre>
          nlags),sep='')
       dimnames(ACFlist2[["ACF"]]) <- list(paste('gate',seq(</pre>
61
          ngates), sep=''), paste('lag', seq(nlags), sep=''))
62
       dimnames(ACFlist2[["var"]]) <- list(paste('gate',seq(</pre>
          ngates),sep=''),paste('lag',seq(nlags),sep=''))
63
64
65
       return(ACFlist2)
66
67
     }
```

5.3 Correlation and inverse problem formulation

5.3.1 laggedProducts.R

```
1 ## file:laggedProducts.R
2 ## (c) 2010 - University of Oulu, Finland
3 ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
5 ##
6
7 | ##
8 ## Calculation of lagged products
9 | ##
10 ## Arguments:
       LPIenv An LPI environment
11 | ##
12 | ##
        lag
              Lag number
13 ##
14 ##
15 ## Returns:
16 ##
       success TRUE if at least one lagged product was
17 ##
                 successfully calculated, FALSE otherwise.
18 ##
19 ## The lagged products are (over)written to
20 ## the vector LPIenv[["cprod."]]
21 | ##
22
23 laggedProducts <- function( LPIenv , lag )
24
25
26
      # Make sure that the lag number is an integer
27
       storage.mode(lag) <- "integer"</pre>
28
29
       # Call the c function
30
       return( .Call( "lagged_products" ,
31
                      LPIenv[["RX1"]][["cdata"]] ,
                      LPIenv[["RX2"]][["cdata"]]
32
                      LPIenv[["RX1"]][["idata"]]
33
                      LPIenv[["RX2"]][["idata"]]
34
                      LPIenv[["cprod"]]
35
                      LPIenv[["iprod"]]
36
37
                      LPIenv[["nData"]]
                      LPIenv[["nData"]]
38
39
                      lag
40
                      )
41
              )
42
    }
```

5.3.2 lagprodVar.R

```
1 ## file:lagprodVar.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Variances of lagged products. Calculated
9 ## as lagged products of average power values.
10 | ##
11 ## Arguments:
12 ##
       LPIenv A LPI environment
13 ##
        lag
              Lag number
14 ##
15 ## Returns:
16 ##
                 TRUE if a variance estimate was successfully
        success
                 calculated for at least one data point,
17 ##
18 ##
                 FALSE otherwise.
19 ## The variances are (over)written to LPIenv[["var"]]
20 ##
21
22 lagprodVar <- function( LPIenv , lag )
23
24
25
       # Make sure that lag is an integer
       storage.mode(lag) <- "integer"</pre>
26
27
       # Call the C function
28
29
       return( .Call( "lagged_products_r"
                      LPIenv[["RX1"]][["power"]]
30
31
                      LPIenv[["RX2"]][["power"]]
32
                      LPIenv[["var"]]
33
                      LPIenv[["nData"]]
                      LPIenv[["nData"]]
34
35
                      lag
36
                      )
              )
37
38
    }
```

5.3.3 rangeAmbiguity.R

```
1 ## file:rangeAmbiguity.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 6
 7 ##
 8 ## Calculation of range ambiguity functions.
 9 | ##
10 ## Arguments:
11 | ##
      LPIenv A LPI environment
12 ##
       lag
               Lag number
13 ##
14 ##
15 ## Returns:
16 ##
               TRUE if at least one point was successfully
      success
17 ##
                calculated, FALSE otherwise.
18 ##
                The range ambiguity function is
19 ##
                (over) written to LPIenv$camb.
20 | ##
21 ##
22 ##
23 ##
24 ##
25
26 range Ambiguity <- function ( LPIenv , lag )
27
     {
28
29
       # True oversampling is not supported.
       if( LPIenv[['nDecimTX']] != 1) stop("True transmitter
30
          signal oversampling is not supported.")
31
32
       # Make sure that lag is an integer
33
       storage.mode(lag) <- "integer"</pre>
34
35
       # Simulate oversampling by means of interpolation.
       # This works well if the pulses have
36
37
       # sharp edges and constant amplitude.
       if( LPIenv[["ambInterp"]] ){
38
39
         return( .Call( "range_ambiguity"
                        LPIenv[["TX1"]][["cdata"]]
40
41
                        LPIenv [["TX2"]] [["cdata"]]
42
                        LPIenv[["TX1"]][["idata"]]
                        LPIenv[["TX2"]][["idata"]]
43
                        LPIenv[["camb"]]
44
                        LPIenv[["iamb"]]
45
                        LPIenv[["nData"]]
46
```

```
LPIenv[["nData"]]
47
48
                        lag
                        )
49
50
                )
51
       }
52
53
       # Simple lagged products of decimated data,
54
       # works with strong codes.
55
       return( .Call( "lagged_products"
56
                      LPIenv[["TX1"]][["cdata"]] ,
57
                      LPIenv[["TX2"]][["cdata"]] ,
                      LPIenv[["TX1"]][["idata"]]
58
59
                      LPIenv[["TX2"]][["idata"]]
60
                      LPIenv[["camb"]]
                      LPIenv[["iamb"]]
61
                      LPIenv[["nData"]]
62
63
                      LPIenv[["nData"]]
64
                      lag
65
                      )
              )
66
67
68
    }
```

5.3.4 averageProfiles.R

```
1 ## file:averageProfiles.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Lag-profile pre-averaging before the actual inversion.
9 ## Provides significant speed-up but may lead to somewhat
10 ## reduced estimation accuracy
11 | ##
12 ## This routine is intended to be used in real-time
13 ## analysis with limited computing resources when speed
14 ## gain with reduced accuaracy and flexibility is accepable.
15 ##
16 ##
17 ## Arguments:
     LPIenv A LPI environment
19 ##
      1
              Lag number
20 ##
21 ## Returns:
22 ##
      success TRUE if both lagged products and range ambiguity
23 ##
                functions were successfully averaged.
24 | ##
25 ## The averaged profiles are overwritten to
26 ## LPIenv[["cprod"]] and LPIenv[["camb"]]
27 ##
28
29 averageProfiles <- function( LPIenv , 1 )
30
31
32
       s1 <- .Call( "average_profile" , LPIenv[["cprod"]]</pre>
          LPIenv[["TX1"]][["idata"]] , as.integer( LPIenv[["
          nData"]] - 1 ) , as.integer( LPIenv[["nCode"]] ) )
33
34
      s2 <- .Call( "average_profile" , LPIenv[["camb"]]</pre>
          LPIenv[["TX1"]][["idata"]] , as.integer( LPIenv[["
          nData"]] - 1 ) , as.integer( LPIenv[["nCode"]] ) )
35
36
       invisible( ( s1 & s2 ) )
37
38
    }
```

5.3.5 theoryRows.R

```
1 ## file: theoryRows.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Form theory matrix rows for lag profile inversion
9 | ##
10 ## Arguments:
                  A LPI environment
11 | ##
       LPIenv
12 ##
        lag
                  Lag number
13 ##
14 ## Returns:
                 TRUE if at least one theory matrix row
15 ##
        success
16 ##
                 was successfully produces, FALSE otherwise.
17 ##
18 ## The rows are written to LPIenv[["arows"]],
19 ## the correspoding measurements to LPIenv[["meas"]],
20 ## variance to LPIen[["mvar"]], and number of rows
21 ## generated to LPIenv[["nrows"]]
22 ##
23 | ##
24
25 theoryRows <- function( LPIenv , lag )
26
27
       # Call the C routine
28
       return( .Call( "theory_rows"
29
                      LPIenv[['camb']] ,
30
                      LPIenv[['iamb']]
31
32
                     LPIenv[['cprod']],
33
                     LPIenv[['iprod']],
34
                     LPIenv[['var']]
35
                     LPIenv[['nData']]
36
                     LPIenv[['nCur']]
                      as.integer(LPIenv[['nCur']]+LPIenv[['nBuf'
37
                         ]])
38
                     LPIenv[['rangeLimits']] ,
39
                     LPIenv[['nGates']][lag] ,
40
                     LPIenv[['arows']] ,
41
                     LPIenv[['irows']] ,
                     LPIenv[['meas']] ,
42
                     LPIenv[['mvar']],
43
                     LPIenv[['nrows']],
44
                     LPIenv[["backgroundEstimate"]],
45
                     LPIenv[["remoteRX"]]
46
```

47) 48) 49 }

5.4 Inverse problem solvers

5.4.1 fishs.init.R.

```
1 ## file:fishs.init.R
2 ## (c) 2010 - University of Oulu, Finland
3 ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 | ##
8 ## Linear inverse problem solution by means of direct
      calculation
9 | ## of Fisher information matrix. Initialization function.
10 | ##
11 ## Arguments:
12 ## ncols Number of unknowns (theory matrix columns)
13 | ##
14 ## Returns:
15 ## s
            A fishs solver environment
16 ##
17
18 fishs.init <- function( ncols , ... )
19
      # New environment for the solver
20
      s <- new.env()
21
22
23
      # Number of columns in the theory matrix
24
      assign( 'ncol' , ncols , s )
25
26
      # A vector for upper triangular part of
27
      # the Fisher information matrix
28
      assign('Qvec', rep(0+0i,(ncols*(ncols+1)/2)), s)
29
30
      # A vector for weighted measurements
31
      assign( 'y'
                      , rep(0+0i,ncols) , s )
32
33
      # Make sure that the storage modes are
      # correct for later c function calls
34
       storage.mode(s$Qvec) <- storage.mode(s$y) <- "complex"
35
       storage.mode(s$ncol) <- "integer"</pre>
36
37
38
      return(s)
39
40
    }
```

5.4.2 fishs.add.R

```
1 ## file:fishs.add.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 5 ##
 6
 7 ##
 8 ## Linear inverse problem solution by means of direct
 9 ## calculation of Fisher information matrix.
10 ## Data accumulation function.
11 | ##
12 ## Arguments:
13 | ##
               A fishs solver environemnt
      е
       A.data Theory matrix rows as a vector (row-by-row)
14 ##
15 ## M.data Measurement vector
16 ##
      E.data Measurement variance vector
17 ##
18 ## Returns:
19 ##
      success TRUE if the rows were successfully added.
20 ##
21
22 fishs.add <- function( e , A.data , M.data , E.data=1 )
23
24
25
       # Number of theory rows to add
26
       nrow <- as.integer(length(M.data))</pre>
27
28
       # Variance vector
29
       E.data <- rep(E.data,length.out=nrow)</pre>
30
31
       # Check storage modes before calling the c function
32
       storage.mode(A.data) <- "complex"</pre>
33
       storage.mode(M.data) <- "complex"</pre>
       storage.mode(E.data) <- "double"</pre>
34
35
       storage.mode(nrow)
                              <- "integer"
36
37
       # Call the c function
       return( .Call( "fishs_add" , e[["Qvec"]] , e[["y"]] , A.
    data , M.data , E.data , e[["ncol"]] , nrow ))
38
39
40
     }
```

5.4.3 fishs.solve.R.

```
1 ## file:fishs.solve.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Linear inverse problem solution by means of direct
9 ## calculation of Fisher information matrix.
10 ## Final solver function.
11 | ##
12 ## Arguments:
13 | ##
                       A fishs solver environment
      full.covariance Logical, full covariance matrix is
14 | ##
      calculated
15 ##
                       if TRUE, otherwise only variances are
      returned.
16 ##
17 ## Returns:
      Nothing, the solution is assigned to
19 ##
      the solver environment
20 ##
22 fishs.solve <- function( e , full.covariance = TRUE , ... )
23
24
25
      # Allocate a matrix for the full
26
       # Fisher information matrix
27
      Q <- matrix( 0 , ncol=e[["ncol"]] , nrow=e[["ncol"]] )</pre>
28
29
      # Copy the upper triangular part form e$Qvec
30
      i <- 1
31
       for( k in seq( e$ncol ) ){
         Q[k,k:e[["ncol"]]] <-e[["Qvec"]][i:(i+(e
32
            [["ncol"]] - k ) ) ]
33
        i <- i + e[["ncol"]] - k + 1
34
35
      # The lower triangular part is
36
37
       # complex conjugate of the upper one
                 <- Q + Conj( t( Q ) )
38
39
      # The above row multiplies the diagonal
40
41
       # with 2, divide accordingly
42
      diag( Q) <- diag( Q ) / 2
43
      # Select points at which the diagonal of Q is zero,
44
```

```
45
       # these points have not been measured at all and
46
       # need to be regularized before inverting the matrix
47
               <- Re( diag( Q ) ) == 0
       nainds
48
       \# Set unit values on the diagonal at unmeasured points.
49
50
       # This will not affect the other unknowns because
51
       # they cannot correlate with this one
52
       diag( Q )[ nainds ] <- 1
53
54
       # Covariance matrix is inverse matrix of
       # the Fisher information matrix
55
       # Even if there were measurements the matrix might not be
56
           invertible
       # return NA matrix in this case
57
58
                           <- tryCatch( solve( Q ) , error=
       covariance
          function(e){Q*NA})
59
60
       # Multiply the covariance matrix with e$y from right.
       # For some reason the direct matrix multiplication
61
62
       # with %*% does not work properly in some machines.
       solution <- rep(0+0i,e[["ncol"]])</pre>
63
64
       for(k in seq(e[["ncol"]])) solution[k] <- sum( covariance</pre>
          [k,] * e[["y"]] )
65
66
       # Set NAs to points that were not actually measured
67
       solution[ nainds ] <- NA</pre>
68
       \mbox{\#} Assign the solution to the solver environment \mbox{e}
69
70
       assign('solution', solution, e)
71
72
       # The full covariance matrix was already calculated, pick
73
       # the diagonal if that is enough.
74
       # Put NA to unmeasured points.
75
       if( full.covariance ){
         covariance[ nainds ,
76
                                      ] <- NA
77
         covariance[
                            , nainds ] <- NA
78
      }else{
79
        covariance
                                        <- diag( covariance )
80
         covariance[ nainds ]
                                        < - NA
81
82
83
       # Assign the covariance to the solver environment e
84
       assign( 'covariance', covariance, e)
85
86
       invisible()
87
88
```

5.4.4 deco.init.R

```
1 ## file:deco.init.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
5 ##
6
7 ##
8 ## Matched filter decoder. Initialization function.
9 | ##
10 ## Arguments:
      ncols Number of unknowns (theory matrix columns)
11 ##
12 ##
             Additional arguments are allowed by not used
13 | ##
             in order to make the solver more compatible
14 ##
             with others.
15 | ##
16 ## Returns:
17 ## e
           A deco solver environment
18 ##
19
20 deco.init <- function( ncols , ...)
21
22
23
      # A new environment for the solver
24
      s <- new.env()
25
      # Number of columns in theory matrix
26
27
       assign('ncol', ncols, s)
28
29
      # Diagonal of the Fisher information matrix
30
       assign( 'Qvec' , rep(0,ncols) , s )
31
32
      # Scaled measurements
33
      assign('y'
                      , rep(0,ncols) , s )
34
35
      # Make sure that the storage modes are
36
      # correct for later c function calls
       storage.mode(s$Qvec) <- storage.mode(s$y) <- "complex"
37
38
       storage.mode(s$ncol) <- "integer"</pre>
39
40
      # return the environment
41
      return(s)
42
43
```

5.4.5 deco.add.R

```
1 ## file:deco.add.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Matched filter decoder. Data accumulation function.
9 | ##
10 ## Arguments:
11 ##
              A deco solver environemnt
      A.data Theory matrix rows as a vector (row-by-row)
12 | ##
13 ##
      M.data Measurement vector
      E.data Measurement variance vector
14 | ##
15 | ##
16 ## Returns:
17 ##
      success TRUE if the rows were successfully added.
18 ##
19
20
21 deco.add <- function( e , A.data , M.data , E.data=1 )
22
23
       # Number of theory rows
24
       nrow <- as.integer(length(M.data))</pre>
25
26
       # Measurement variance vector
27
       E.data <- rep(E.data,length.out=nrow)</pre>
28
29
       # Set storage modes
30
       storage.mode(A.data) <- "complex"</pre>
31
       storage.mode(M.data) <- "complex"</pre>
32
       storage.mode(E.data) <- "double"</pre>
33
                             <- "integer"
       storage.mode(nrow)
34
35
       # Call the c routine
       return( .Call( "deco_add" , e$Qvec , e$y , A.data , M.
36
          data , E.data , e$ncol , nrow ))
37
38
    }
```

5.4.6 deco.solve.R

```
1 ## file:deco.solve.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Matched filter decoder. Final solver function.
9 | ##
10 ## Arguments:
                       A deco solver environment
11 ##
12 ##
      full.covariance Logical, full covariance matrix is
                       calculated if TRUE, otherwise only
13 | ##
14 ##
                       variances are returned.
15 ##
16 ## Returns:
17 ##
      Nothing, the solution is assigned to
18 ##
      the solver environment
19 | ##
20
21 deco.solve <- function( e , ...)
22
23
      # Diagonal of the Fisher information matrix
24
      # (Matched filter decoding is equivalent with assuming
25
      # that the nondiagonal elements are zeros)
26
      Qdiag <- e[["Qvec"]]
27
28
      # The points at which Qdiag is zero were not measured
29
       # at all, flag these points
30
      nainds <- Qdiag == 0
31
32
      # Put unit values to the unmeasured points. This does
33
      # not affect the other points as they cannot
34
      # correlated with the unmeasured ones.
35
      Qdiag[nainds] <- 1
36
37
      # Variance is simply the inverse of the diagonal
38
      # of the Fisher information
      variance <- 1 / Qdiag
39
40
      # Assign the solution to the solver environment
41
42
       assign( 'solution' , variance * e[["y"]] , e )
43
44
      # Set NAs to the unmeasured points
      e[["solution"]][nainds] <- NA
45
46
      # Same for the variances
47
```

```
assign('covariance', variance, e)
e[["covariance"]][nainds] <- NA

invisible()

invisible()
```

5.4.7 dummy.init.R

```
1 ## file:dummy.init.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Dummy inverse problem solver that calculates
9 ## simple averages.
10 ## Initialization function.
11 | ##
12 ## Arguments:
13 ##
      rrange extreme ranges to be solved c(rmin, rmax)
14 | ##
15 ## Returns:
16 ##
             A dummy solver environment
17 ##
18
19 dummy.init <- function( rrange )
20
21
       # A new environment for the solver
22
23
       s <- new.env()
24
25
       # Number of ranges (this is different
26
       # from number of final range gates)
27
       nr <- abs(diff(rrange))</pre>
28
29
       # A vector for sum of weighted measurements
30
       msum <- rep(0+0i,nr)
31
32
       # A vector for sum of information
33
       vsum <- rep(0,nr)</pre>
34
35
       # Minimum range
36
       rmin <- min(rrange)</pre>
37
38
       # Maximum range
39
       rmax <- max(rrange)</pre>
40
41
       # Make sure that storage modes are correct
42
       storage.mode(msum) <- "complex"</pre>
       storage.mode(vsum) <- "double"</pre>
43
       storage.mode(rmin) <- "integer"</pre>
44
       storage.mode(rmax) <- "integer"</pre>
45
46
47
       # Assign the variables to the environment
```

5.4.8 dummy.add.R

```
1 ## file:dummy.add.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 6
 7 ##
 8 ## Dummy inverse problem solver that
 9 ## calculates simple averages.
10 ## Data accumulation function.
11 | ##
12 ## Arguments:
13 | ##
               A dummy solver environemnt
      M.data Measurement vector
14 | ##
15 | ##
      M.ambig Range ambiguity function
       I.ambig Indices of non-zero ambiguity values
16 ##
17 ##
      I.prod Indices of usable lagged products
18 ##
      E.data Measurement variance vector
19 | ##
      nData
               Number of points in data vectors
20 | ##
21 ## Returns:
22 ##
      success TRUE if the data was successfully added
23 | ##
24
25 dummy.add <- function( e , M.data , M.ambig , I.ambig , I.
      prod , E.data , nData )
26
27
       # Call the C routine
28
29
       return( .Call( "dummy_add" ,
                      e[["msum"]] ,
30
31
                      e[["vsum"]]
32
                      e[["rmin"]]
33
                      e[["rmax"]]
34
                      M.data ,
                      M.ambig ,
35
36
                      I.ambig ,
37
                      I.prod ,
                      E.data ,
38
39
                      nData)
              )
40
41
42
```

5.4.9 dummy.solve.R

```
1 ## file:dummy.solve.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## Dummy inverse problem solver that
9 ## calculates simple averages.
10 ## Final solver function.
11 | ##
12 ## Arguments:
13 | ##
              A dummy solver environment
      е
14 ##
      rlims Range gate limits
15 | ##
16 ## Returns:
17 ##
      Nothing, the solution is assigned to
18 ##
      the solver environment
19 | ##
20
21 dummy.solve <- function( e , rlims )
22
23
24
25
       # Final solver function.
26
27
       # I. Virtanen 2012
28
29
30
       # Number of range gates
31
       nr <- length(rlims) - 1</pre>
32
33
       # Vectors for the solution and variance
34
       solution <- rep(0+0i,nr)
35
       covariance <- rep(0,nr)</pre>
36
37
       # Range integration for the data points that have
       # the best possible resolution at this point.
38
39
       for( r in seq(nr) ){
40
41
         # Lower limit of this range gates
         r1 <- rlims[r] - rlims[1] + 1
42
43
44
         # Upper limit of this range gate
         r2 <- rlims[r+1] - rlims[1]
45
46
47
         # The vector e$msum contains variance weighted sum,
```

```
48
         # we can simply sum its elements.
49
         solution[r] <- sum(e[["msum"]][r1:r2])</pre>
50
51
         # The vector e$vsum contains informations, sum them.
52
         covariance[r] <- sum(e[["vsum"]][r1:r2])</pre>
53
54
       }
55
56
       # Variance is inverse of the information
57
       covariance <- c( 1/covariance , NA )
58
       # Multiply the solution with the final variances
59
       solution <- c( solution , NA ) * covariance</pre>
60
61
62
       # Vectors solution and covariance will now contain
63
       # variance-weighted averages of the lag profiles
       # and their variances. Assign to the solver environment
64
       assign( 'solution' , solution , e )
assign( 'covariance' , covariance , e )
65
66
67
68
       invisible()
69
70 }
```

5.4.10 ffts.init.R

```
1 ## file:ffts.init.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## FFT deconvolution.
9 ## Initialization function.
10 | ##
11 ## Arguments:
      rrange Extreme ranges to be solved c(rmin,rmax)
13 ##
      itx
              A logical vector of transmitter pulse positions.
14 ##
15 ## Returns:
16 ##
            A ffts solver environment
17 ##
18
19 ffts.init <- function( rrange , itx )
20
21
       # Minimum range
22
                  <- min( rrange )
      rmin
23
24
       # Maximum range
25
                   <- max( rrange )</pre>
      rmax
26
27
       # longest inter-pulse period
       ippmax \leftarrow max(diff(which(diff(itx > 0) == 1)),
28
          showWarnings=FALSE )
29
30
       # Select the FFT length
31
      n <- max( nextn( ippmax ) , nextn( rmax*2 ) )</pre>
32
33
       # Allocate vectors
                   <- rep( 0+0i , n )
34
      fу
35
                   <- famb.tmp
                                         <- rep( 0+0i , n )
       amb.tmp
                   <- rep( 0+0i , n )
36
      meas.tmp
37
       sqfamb
                   <- rep( 0 , n )
38
       varsum
                   <- 0
                   <- 0
39
      nmeas
40
41
      # Set storage modes
42
       storage.mode( rmin )
                                 <- "integer"
                                 <- "integer"
43
       storage.mode( rmax )
       storage.mode( n )
                                 <- "integer"
44
                                 <- "integer"
45
       storage.mode(nmeas)
                                  <- "complex"
46
       storage.mode(fy)
```

```
storage.mode( amb.tmp ) <- "complex"</pre>
47
48
        storage.mode( famb.tmp ) <- "complex"</pre>
        storage.mode( meas.tmp ) <- "complex"</pre>
49
50
        storage.mode( sqfamb ) <- "double"</pre>
                                       <- "double"
51
        storage.mode(varsum)
52
53
        # Create a new environment and assign everything to it
54
        s <- new.env()
                              , n
, rmin
                                                , s )
55
        assign('n'
       assign('rmin'
assign('rmax'
assign('fy'
assign('sqfamb'
56
                                               , s )
                                , rmax
57
                                                , s )
58
                               , fy
        assign('sqfamb', sqfamb', s)
assign('amb.tmp', amb.tmp, s)
assign('famb.tmp', famb.tmp, s)
59
60
61
        assign( 'meas.tmp'
                                 , meas.tmp , s )
62
        assign('nmeas', nmeas, s)
assign('varsum', varsum, s)
63
64
65
        # return the environment
66
67
        return(s)
68
69 }
```

5.4.11 ffts.add.R

```
1 ## file:ffts.add.R
 2 ## (c) 2010 - University of Oulu, Finland
 3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
 4 ## Licensed under FreeBSD license.
 6
 7 ##
 8 ## FFT deconvolution.
 9 ## Data accumulation function.
10 | ##
11 ## Arguments:
12 | ##
               An ffts solver environemnt
13 ##
      M.data Measurement vector
      M.ambig Range ambiguity function
14 ##
      I.ambig Indices of non-zero ambiguity values
15 | ##
      I.prod Indices of usable lagged products
16 | ##
17 ##
      E.data Measurement variance vector
18 ##
      nData Number of points in data vectors
19 | ##
20 ## Returns:
21 ##
      success TRUE if the data was successfully added
22 ##
24 ffts.add <- function( e , M.data , M.ambig , I.ambig , I.prod
       , E.data , nData )
25
26
27
       # FFT deconvolution. Data accumulation function.
28
29
       # I. Virtanen 2012
30
31
32
       # Return immediately if the ambiguity
33
       # function is zero at all points
34
       if( ! any( I.ambig[1:nData] ) ) return()
35
36
       # Remove possibly remaining non-zero values
37
       # from points with unset index vector
      M.data[ which(!I.prod)
                                ] <- 0+0i
38
39
      E.data[ which(!I.prod)
                                ] <- 0
      M.ambig[ which(!I.ambig) ] <- 0+0i</pre>
40
41
       # Locate pulse start positions
42
43
      ps <- which( diff( I.ambig[1:nData] > 0 ) == 1 )
44
45
       # The first point should be adjusted to pulse start,
       # so it is safe to use if the index is set
46
```

```
47
       if( I.ambig[1] ) ps <- c( 1 , ps )
       npulse <- length( ps )</pre>
48
49
50
       # Locate pulse end positions
51
      pe <- which( diff(I.ambig[1:nData] > 0) == -1)
52
53
       # pe and ps should be of the same length,
54
       # but check anyway...
       npulse <- min( length(pe) , length(ps) )</pre>
55
56
       # Add data from one IPP at a time
57
58
       for( k in seq( npulse ) ){
59
60
         # Set temporary vectors to zero
61
         e[["amb.tmp"]][] <- e[["meas.tmp"]][] <- 0.+0.i
62
63
         # Pulse end or data end (should always be pulse end,
         # but check anyway)
64
                           <- min( nData , pe[k] )</pre>
65
66
67
         # max range or data end
                           <- min( nData , ( ps[k] + e[["n"]] - 1
68
         pe2
             ) )
69
70
         # Copy one pulse
         e[["amb.tmp"]][ 1 : ( pe1 - ps[k] + 1 ) ] <- M.ambig[
71
             ps[k] : pe1 ]
72
73
         # Take fft
74
         e[["famb.tmp"]][] <- fft( e[["amb.tmp"]] )</pre>
75
76
         # Copy data
77
         e[["meas.tmp"]][ 1 : ( pe2 - ps[k] + 1 ) ] <- M.data[
            ps[k] : pe2 ]
78
         # Actual addition to the solver
79
                          <- e[["fy"]]
80
         e[["fy"]][]
                                            + Conj( e[["famb.tmp"
            ]] ) * fft( e[["meas.tmp"]] )
81
         e[["sqfamb"]][] <- e[["sqfamb"]] + abs( e[["famb.tmp"]]
             )**2
82
      }
83
84
85
       # Variances
       e[["varsum"]] <- e[["varsum"]] + sum( E.data[ 1 : nData ]
86
87
       e[["nmeas"]] <- e[["nmeas"]] + sum( ( I.prod[ 1 : nData
           ] > 0 ) )
88
```

```
89 invisible()
90 91 }
```

5.4.12 ffts.solve.R

```
1 ## file:ffts.solve.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
6
7 ##
8 ## FFT deconvolution.
9 ## Final solver function.
10 | ##
11 ## Arguments:
              A ffts solver environment
13 ## rlims Range gate limits
14 ##
15 ## Returns:
      Nothing, the solution is assigned to the solver
      environment
17 ##
18 ffts.solve <- function(e, rlims)
19
20
       # FFT deconvolution. Final solver function.
21
22
23
       # I. Virtanen 2012
24
25
26
       # Solve the lag profile by means of FFT
       sol <- fft( e[["fy"]]</pre>
27
                               / e[["sqfamb"]] , inverse=TRUE )
          / e[["n"]]
28
29
       # Variance, the same value will be repeated at all ranges
30
       var <- e[["varsum"]] / as.double(e[["nmeas"]]) * mean( 1</pre>
          / e[["sqfamb"]] )
31
32
       # Number of range gates
       nr <- length(rlims) - 1</pre>
33
34
35
       # Final solution and variance vectors
36
       solution <- rep(0+0i,nr)</pre>
37
       covariance <- rep(0,nr)</pre>
38
39
       for( r in seq(nr) ){
40
41
         # Lower limit of range gate
42
                        <- rlims[r] + 1
         r1
43
         # Upper limit of range gate
44
```

```
<- rlims[r+1]
45
         r2
46
47
          # All points have equal variances, calculate simple
              average
          solution[r]
                          <- mean( sol[r1:r2] , na.rm=TRUE )</pre>
48
49
50
          # Scale the variance
51
          covariance[r] <- var/(r2-r1+1)</pre>
52
53
       # The background ACF cannot be measured with this
54
           technique, set it to NA.
        covariance <- c( covariance , NA ) \,
55
56
        solution <- c( solution , NA )
57
       \mbox{\tt\#} Assign the results to the solver environment.
58
       assign( 'solution' , solution , e )
assign( 'covariance' , covariance , e )
59
60
61
       invisible()
62
63
64
    }
```

5.4.13 rlips.solve2.R

```
1 ## file:rlips.solve2.R
2 ## (c) 2010 - University of Oulu, Finland
3 | ## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 ## Licensed under FreeBSD license.
5 ##
6
7 ## Call rlips.solve after regularization for
8 ## unknowns that were not measured at all
9 ## Set the corresponding values to NA before returning
10 | ##
11 ## Arguments:
12 | ##
                       An rlips solver environment
13 ##
      fullCovariance Logical, if TRUE full covariance matrix
14 ##
                       is calculated, otherwise only the
15 | ##
                       variances.
16 ## Returns:
17 ##
      Nothing, the solution is assigned to the
18 ##
      solver environment.
19 | ##
20
21 rlips.solve2 <- function( e , full.covariance = TRUE )
22
23
       # Read data from gpu memory
24
      rlips.get.data( e )
25
26
       # Select non-measured points
27
       nainds <- which( Re( diag( e$R.mat ) ) == 0 )</pre>
28
29
       # Add regularizing imaginary measurements
30
       regrow <- rep(0+0i,e$ncols)</pre>
31
       for( n in nainds ){
         regrow[] <- 0+0i
32
33
         regrow[n] <- 1+0i
34
         rlips.add( e , A.data = regrow , M.data = 1.0+0.0i )
35
36
       # Solve the problem
37
38
       rlips.solve( e , calculate.covariance = TRUE , full.
          covariance = full.covariance )
39
40
       # Set NAs to appropriate points in the solution
41
       sol <- e$solution
       sol[nainds] <- NA
42
43
       assign('solution', sol, e)
44
45
       # Set the unmeasured points to NA
       # in the covariance matrix as well.
46
```

```
47
        covar <- e$covariance
48
        if( full.covariance ){
          covar[ , nainds ] <- NA
covar[ nainds , ] <- NA</pre>
49
50
51
        }else{
52
          covar[nainds] <- NA
53
54
55
        \mbox{\tt\#} Assign the covariance matrix to the solver environment
56
        assign( 'covariance' , covar , e )
57
58
        invisible()
59
60
    }
```

5.5 C functions and headers

$5.5.1 \quad src/LPI.h$

```
1 // file:LPI.h
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 // Data types and function prototypes
8 #include <R.h>
9 | #include <math.h>
10 #include <stdint.h>
11 #include <Rinternals.h>
12 #include < Rdefines.h>
13 #include <R_ext/Rdynload.h>
14
15 static const double pi=3.1415926535;
16 #define AMB_N_INTERP
17
18
19 // gdf file input
20 SEXP read_gdf_data_R( SEXP ndata , SEXP nfiles , SEXP
     filepaths , SEXP istart , SEXP iend , SEXP bigendian);
21 SEXP read_gdf_data( SEXP cata , SEXP idatar , SEXP idatai ,
     SEXP ndata , SEXP nfiles, SEXP filepaths , SEXP istart ,
     SEXP iend , SEXP bigendian);
22
23 // Frequency mixing
24 SEXP mix_frequency_R( SEXP cdata , SEXP ndata , SEXP
     frequency);
25 SEXP mix_frequency( SEXP cdata , SEXP ndata , SEXP frequency)
26
27 // Index adjustments
28 SEXP index_adjust_R( SEXP idata , SEXP ndata , SEXP shifts );
29 SEXP index_adjust( SEXP idata , SEXP ndata , SEXP shifts );
30
31 // Lagged products
32 SEXP lagged_products_alloc( SEXP cdata1 , SEXP cdata2 , SEXP
     idata1 , SEXP idata2 , SEXP ndata1 , SEXP ndata2 , SEXP
     lag);
33 SEXP lagged_products( SEXP cdata1 , SEXP cdata2 , SEXP idata1
      , SEXP idata2 , SEXP cdatap , SEXP idatap , SEXP ndata1 ,
      SEXP ndata2 , SEXP lag );
34 SEXP lagged_products_r( SEXP rdata1 , SEXP rdata2 , SEXP
     prdata , SEXP ndata1 , SEXP ndata2 , SEXP lag );
```

```
36 // Theory matrix construction
37 SEXP theory_rows_alloc( SEXP camb , SEXP iamb , SEXP cprod ,
      SEXP iprod , SEXP rvar , SEXP ndata , SEXP ncur , SEXP
      nend , SEXP rlims , SEXP nranges , SEXP fitsize , SEXP
      background, SEXP remoterx );
38 SEXP theory_rows( SEXP camb , SEXP iamb , SEXP cprod , SEXP
      iprod , SEXP rvar , SEXP ndata , SEXP ncur , SEXP nend ,
      SEXP rlims , SEXP nranges , SEXP arows , SEXP irows , SEXP
       mvec , SEXP mvar , SEXP nrows , SEXP background, SEXP
      remoterx );
39
40 // Inverse problem solvers
41 SEXP fishs_add( const SEXP Qvec, const SEXP yvec , const SEXP
       arows , const SEXP meas , const SEXP var , const SEXP nx
      , const SEXP nrow );
42 SEXP deco_add( const SEXP Qvec , const SEXP yvec , const
      SEXP arows , const SEXP meas , const SEXP var , const SEXP
       nx , const SEXP nrow );
43 SEXP dummy_add( SEXP msum , SEXP vsum , SEXP rmin , SEXP rmax
       , SEXP mdata , SEXP mambig , SEXP iamb , SEXP iprod ,
      SEXP edata , SEXP ndata );
44
45 // All data preparations collected together
46 SEXP prepare_data( SEXP cdata , SEXP idata , SEXP ndata ,
      SEXP frequency, SEXP shifts , SEXP nup , SEXP nfilter ,
      SEXP nfirst , SEXP nfirstfrac , SEXP ipartial );
47
48 // Average signal power in points withe identical IPPs and
      pulse lengths
49 SEXP average_power( SEXP cdata , SEXP idatatx , SEXP idatarx
      , SEXP ndata , SEXP maxrange);
50
51 // Average lag profile
52 SEXP average_profile( SEXP cdata , SEXP idata , SEXP ndata ,
      SEXP N_CODE);
53
54 // Resampling
55 SEXP resample (SEXP cdata , SEXP idata , SEXP ndata , SEXP
      \operatorname{\mathsf{nup}} , SEXP \operatorname{\mathsf{nfilter}} , SEXP \operatorname{\mathsf{nfirst}} , SEXP \operatorname{\mathsf{nfirstfrac}} , SEXP
      ipartial);
56 SEXP resample_R( SEXP cdata , SEXP idata , SEXP ndata , SEXP
      \operatorname{nup} , SEXP \operatorname{nfilter} , SEXP \operatorname{nfirst} , SEXP \operatorname{nfirstfrac} , SEXP
      ipartial);
57
58 // Range ambiguity function calculation with optional
      interpolation
59 SEXP range_ambiguity( SEXP cdata1 ,SEXP cdata2 , SEXP idata1
      , SEXP idata2 , SEXP cdatap , SEXP idatap , SEXP ndata1 ,
```

5.5.2 register.c

```
1 // file:register.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 // R registration of C functions
8 #include "LPI.h"
9 static const R_CallMethodDef callMethods[20] = {
    { "read_gdf_data_R"
                               , (DL_FUNC) & read_gdf_data_R
              , 6 } ,
                                , (DL_FUNC) & mix_frequency_R
11
    { "mix_frequency_R"
              , 3 } ,
      "index_adjust_R"
                                , (DL_FUNC) & index_adjust_R
12
               , 3 } ,
    { "lagged_products_alloc" , (DL_FUNC) &  
13
        lagged_products_alloc , 7 } ,
    { "lagged_products"
14
                                , (DL_FUNC) & lagged_products
              , 9 } ,
                                , (DL_FUNC) & lagged_products_r
15
    { "lagged_products_r"
            , 6 }
    { "fishs_add"
                                , (DL_FUNC) & fishs_add
16
                     , 7 }
17
    { "theory_rows_alloc"
                                , (DL_FUNC) & theory_rows_alloc
            , 13} ,
    { "theory_rows"
                                , (DL_FUNC) & theory_rows
18
                  , 17} ,
    { "prepare_data"
                                , (DL_FUNC) & prepare_data
19
                 , 10} ,
20
    { "average_power"
                                , (DL_FUNC) & average_power
    { "deco_add"
                                , (DL_FUNC) & deco_add
21
                     , 7 } ,
22
    { "average_profile"
                                , (DL_FUNC) & average_profile
              , 4 } ,
    { "dummy_add"
                                , (DL_FUNC) & dummy_add
23
                    , 10} ,
    { "resample"
24
                                , (DL_FUNC) & resample
                     , 8 } ,
25
    { "resample_R"
                                , (DL_FUNC) & resample_R
                   , 8 } ,
      "range_ambiguity"
                                , (DL_FUNC) & range_ambiguity
              , 9 } ,
    { "clutter_meas"
                                , (DL_FUNC) & clutter_meas
27
                 , 9 }
    { "clutter_subtract"
28
                                , (DL_FUNC) & clutter_subtract
             , 8 } ,
```

5.5.3 clutter_meas.c

```
1 // file:clutter_meas.c
2|// (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8 /*
9
10
     Ground clutter suppression. This function adds clutter
11
     signal measurements to an inverse problem. The function
     clutter_subtract subtracts clutter contribution from a
12
13
     signal.
14
15
     Arguments:
      tcdata Complex transmitter samples
16
17
              Transmitter sample indices
      tidata
18
      rcdata Complex receiver samples
19
      ridata Receiver sample indices
               Data vector length
20
      ndata
      rmin
21
               Minimum range
22
      rmax
               Maximum range
23
      Qvec
               Upper triangular part of Fisher information matrix
24
      yvec
               Modified measurement vector
25
26
     Returns:
27
      nrow
               Number of measurement rows in the inverse problem
28
29 */
30 SEXP clutter_meas( const SEXP tcdata , const SEXP tidata ,
      \verb|const| SEXP| \verb|rcdata| , \verb|const| SEXP| \verb|ridata| , \verb|const| SEXP| \verb|ndata| , \\
       const SEXP \operatorname{rmin} , \operatorname{const} SEXP \operatorname{rmax} , \operatorname{const} SEXP \operatorname{Qvec} ,
      const SEXP yvec )
31 | {
32
     Rcomplex *tcd = COMPLEX( tcdata );
33
     int *tid = LOGICAL( tidata );
     Rcomplex * rcd = COMPLEX( rcdata );
34
35
     int *rid = LOGICAL( ridata );
     const int nd = *INTEGER( ndata );
36
37
     const int r0 = *INTEGER( rmin );
     const int r1 = *INTEGER( rmax );
38
39
     int i;
40
     int j;
41
42
     int k;
43
     int r;
44
     int isum;
```

```
45
    int nx;
    SEXP nrow;
46
47
     int nr;
48
49
    // Output
50
    PROTECT( nrow = allocVector( INTSXP , 1 ) );
51
52
    // Make sure that the data vectors contain non-zero
    // values only at points in which the logical vectors
53
     // are not set
54
    for( i = 0; i < nd; ++i){
55
       if( tid[i]==0 ){
56
57
         tcd[i].r = 0.0;
58
         tcd[i].i = 0.0;
      }
59
60
       if( rid[i] == 0 ){
61
         rcd[i].r = 0.0;
62
         rcd[i].i = 0.0;
63
    }
64
65
    // Initialization
66
67
    nr = 0;
68
    nx = r1 - r0 + 1;
    r = 0;
69
70
    isum = 0;
     // Sum tx indices and set r
71
72
    for( i = 0 ; i \le r1 ; ++i ){
73
      // The largest range is corresponds to index 0,
74
       // after nx samples we will be below rmin.
75
       if( i < nx ) isum += tid[i];</pre>
       // Increment r
76
77
       ++r;
78
       // Set r to zero if a transmitter sample is meat
       if( tid[i] ) r = 0;
79
80
       // increment the rx data pointer
81
      ++rcd;
    }
82
83
    // Go through all data points
84
    for( i = r1 ; i < nd ; ++i ){
85
       // Set r = 0 if a transmitter sample is meat
86
87
       if( tid[i] ) r = 0;
88
       // Are we below rmax?
89
       if( r <= r1 ){
         // Are we above rmin?
90
91
         if(r >= r0){
       // Are the pulses within the clutter ranges?
92
       if( isum ){
93
```

```
// Is this receiver sample usable?
94
95
          if( rid[i] ){
96
            // Add a measurement
            fishs_add_clutter( Qvec , yvec , tcd , rcd , nx );
97
            // Increment measurement row counter
98
99
            ++nr;
100
          }
       }
101
102
         }
103
       }
        // Update counters if this was not the last sample
104
105
       if( i < nd ){
         isum -= tid[ i - r1 ];
106
107
         isum += tid[ i - r0 + 1 ];
108
         ++r;
109
          ++rcd;
110
          ++tcd;
111
       }
     }
112
113
114
     // Copy the number of rows to output
     *INTEGER( nrow ) = nr;
115
116
117
     UNPROTECT(1);
118
119
     // Return number of measured rows
     return( nrow );
120
121
122 }
```

5.5.4 clutter_subtract.c

```
1 // file:clutter_subtract.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
    Ground clutter suppression. This function subtracts clutter
10
    signal from data.
11
12
    Arguments:
13
     tcdata
             Complex transmitter samples
             Transmitter sample indices
14
     tidata
     rcdata
             Complex receiver samples
15
     ridata Receiver sample indices
16
17
              Data vector length
     ndata
18
     rmin
             Minimum range
19
             Maximum range
     cldata Measured clutter signal profile
20
21
22
    Returns:
23
     nrow
              Number of points at which clutter
24
              signal was suppressed
25
26 */
27
28 SEXP clutter_subtract( const SEXP tcdata , const SEXP tidata
      , const SEXP rcdata , const SEXP ridata , const SEXP ndata
       , const SEXP rmin , const SEXP rmax , const SEXP cldata )
29 {
30
    Rcomplex *tcd = COMPLEX( tcdata );
    int *tid = LOGICAL( tidata );
31
32
    Rcomplex * rcd = COMPLEX( rcdata );
33
    int *rid = LOGICAL( ridata );
34
    Rcomplex *cld = COMPLEX( cldata );
    const int nd = *INTEGER( ndata );
35
    const int r0 = *INTEGER( rmin );
36
    const int r1 = *INTEGER( rmax );
37
38
39
    int i;
40
    int j;
    int k;
41
42
    int r;
43
    int isum;
44
    int nx;
    SEXP nrow;
45
```

```
46
    int nr;
47
    Rcomplex clsum;
48
    Rcomplex * tcd2;
49
    Rcomplex * cld2;
50
51
    // Output
52
    PROTECT( nrow = allocVector( INTSXP , 1 ) );
53
    // Initialization
54
    nr = 0;
55
    nx = r1 - r0 + 1;
56
    r = 0;
57
58
    isum = 0;
     // Sum tx indices and set r
59
    for( i = 0 ; i <= r1 ; ++i ){
60
       // The largest range is corresponds to index 0,
61
62
       // after nx samples we will be below rmin.
63
       if( i < nx ) isum += tid[i];</pre>
64
       // Increment r
65
       ++r;
66
       // Set {\tt r} to zero if a transmitter sample is meat
67
       if(tid[i])r = 0;
68
      // increment the rx data pointer
69
      ++rcd;
70
71
72
    // Go through all data points
73
    for( i = r1 ; i < ( nd - nx )
                                     ; ++i ){
74
       // Set r = 0 if a transmitter sample is meat
75
       if(tid[i])r = 0;
76
       // Are we below rmax?
77
       if( r <= r1 ){
78
         // Are we above rmin?
79
         if(r >= r0){
80
       // Are the pulses within the clutter ranges?
       if( isum ){
81
82
         // Is this receiver sample usable?
83
         if( rid[i] ){
84
           // Calculate clutter contribution and subtract it
85
           clsum.r = 0.;
           clsum.i = 0.;
86
           tcd2 = tcd;
87
88
           cld2 = cld;
89
           for( j = 0; j < nx; ++j){
             clsum.r += tcd2->r * cld2->r - tcd2->i * cld2->i;
90
             clsum.i += tcd2 -> r * cld2 -> i + tcd2 -> i * cld2 -> r;
91
92
             ++tcd2;
93
             ++cld2;
           }
94
```

```
rcd->r -= clsum.r;
95
96
            rcd->i -= clsum.i;
97
            // Increment measurement row counter
98
            ++nr;
          }
99
100
       }
101
          }
102
103
        // Update counters if this was not the last sample \,
104
       if( i < nd ) \{
         isum -= tid[ i - r1 ];
105
          isum += tid[ i - r0 + 1 ];
106
107
          ++r;
108
          ++rcd;
109
          ++tcd;
       }
110
111
112
     // Copy the number of rows to output
113
     *INTEGER( nrow ) = nr;
114
115
     UNPROTECT(1);
116
117
118
     // Return number of measured rows
119
     return( nrow );
120
121 }
```

5.5.5 dummy_add.c

```
1 // file:dummy_add.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
     Simple variance- and power-weighted average lag profile.
10
     Works only below one IPP range.
11
12
      Arguments:
13
      msum
             Sum of normalised measurements
             sum of normalised inverse variances
14
      vsum
            Lower edge of the measurement
15
      rmin
            Upper edge
16
      rmax
17
      mdata Complex measurement vector (lag profile)
      mamb Complex range ambiguity function
18
19
      iamb Range ambiguity function indices
      iprod Lagged product indices
20
21
       edata Measurement variances
      ndata Data vector length
22
23
24
     Returns:
25
       success 1 if the processing was succesful, 0 otherwise
26
27 */
28
29 SEXP dummy_add( SEXP msum , SEXP vsum , SEXP rmin , SEXP rmax
       , SEXP mdata , SEXP mamb , SEXP iamb , SEXP iprod , SEXP
      edata , SEXP ndata )
30 {
31
    Rcomplex *ms = COMPLEX(msum);
    double *vs = REAL(vsum);
32
33
    int r1 = *INTEGER(rmin);
34
    int r2 = *INTEGER(rmax);
    Rcomplex *cd = COMPLEX(mdata);
35
36
    Rcomplex *ad = COMPLEX(mamb);
    int *ia = LOGICAL(iamb);
37
38
    int *ip = LOGICAL(iprod);
39
    double *vd = REAL(edata);
40
    int nd = *INTEGER(ndata);
41
42
    int i, j, r, r0;
43
44
    SEXP
                          success;
              * restrict i_success;
45
     int
```

```
46
47
    // success output
48
    PROTECT( success = allocVector( LGLSXP , 1 ) );
49
50
    // local pointer to the success output
51
    i_success = LOGICAL( success );
52
53
    // set the success output
    *i_success = 1;
54
55
    // Skip first r2 points, their range ambiguity function
56
    // is not known
57
58
    r = r2+1;
59
    r0 = 0;
60
61
    // Walk through the data vector
    for( i = 0; i < nd; ++i){
62
      // Check that we are above r1
63
       if(r >= r1){
64
         // Check that we are below r2
65
         if(r < r2){
66
67
       // Check that the point is flagged as usable
68
       if(ip[i]){
69
         // The average vector starts from range r1
70
         j = r-r1;
         // Divide the lagged product with its variance and
71
         // multiply with TX power
72
73
         ms[j].r += cd[i].r / vd[i] * ad[r0].r;
74
         ms[j].i += cd[i].i / vd[i] * ad[r0].r;
75
         // Inverse of variance scaled accordingly
76
         vs[j] += ad[r0].r * ad[r0].r / vd[i];
77
78
         }
       }
79
80
81
       // If a new pulse is transmitted set range to zero,
82
       // otherwise increment the range counter.
83
       if( ia[i] ){
84
         r = 0;
         r0 = i;
85
86
      }else{
87
         ++r;
88
    }
89
90
91
    UNPROTECT(1);
92
93
    return(success);
94
```

95|}

5.5.6 deco_add.c

```
1 // file:deco_add.c
2|// (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
     Matched filter decoding, modified from fishs_add.
10
11
     Arguments:
            Diagonal of the Fisher information matrix
12
      Qvec
13
             Modified measurement vector
      yvec
      arows Theory matrix rows
14
15
      meas
             Measurements
16
             Measurement variances
      var
17
             Number of unknowns
      nх
18
      nrow Number of theory rows in arows
19
20
     Returns:
      success 1 if the processing was successful, 0 otherwise
21
22
23 */
24
25 SEXP deco_add( const SEXP Qvec , const SEXP yvec , const
      SEXP arows , const SEXP meas , const SEXP var
      SEXP nx , const SEXP nrow )
26 {
    Rcomplex *q = COMPLEX(Qvec);
27
    Rcomplex *y = COMPLEX(yvec);
28
29
    int n = *INTEGER(nx);
    int nr = *INTEGER(nrow);
30
    int i = 0;
31
32
    int j = 0;
33
    int l = 0;
34
35
    Rcomplex * restrict qtmp;
    Rcomplex * restrict acpy = COMPLEX(arows);
36
37
    Rcomplex * restrict atmp;
38
    Rcomplex * restrict ytmp;
39
    Rcomplex * restrict mcpy = COMPLEX(meas);
40
    double
             * restrict vcpy = REAL(var);
41
42
    SEXP
                         success;
43
              * restrict i_success;
    int
44
45
    // Success output
```

```
PROTECT( success = allocVector( LGLSXP , 1 ) );
46
47
48
    // Local pointer to the success output
    i_success = LOGICAL( success );
49
50
51
    // Set the success output
    *i_success = 1;
52
53
    // Go through all theory matrix rows
54
55
    for( 1 = 0 ; 1 < nr ; ++1 ){
56
       // Pointers to y-vector and Fisher information matrix
57
          diagonal
      ytmp = y;
58
59
       qtmp = q;
60
      // Go through all range gates
61
62
      for( i = 0 ; i < n ; ++i ){
63
         // Second pointer to the theory matrix
64
65
         // (Strictly speaking not needed...)
66
         atmp = acpy;
67
68
         // Add information (only diaonal)
69
         qtmp->r += (acpy->r * atmp->r + acpy->i * atmp->i) /
            *vcpy;
70
         qtmp->i += ( acpy->r * atmp->i - acpy->i * atmp->r ) /
            *vcpy;
71
         // Increment the second theory matrix counter
72
73
         ++atmp;
74
75
         // Increment information matrix counter (only diagonal)
76
         ++qtmp;
77
78
         // Add the corresponding measurement to the y-vector
79
         ytmp->r += (mcpy->r * acpy->r + mcpy->i * acpy->i) /
80
         ytmp->i += ( mcpy->i * acpy->r - mcpy->r * acpy->i ) /
            *vcpy;
81
82
         // Increment the y-vector counter
83
         ++ytmp;
84
85
         // Increment the theory matrix counter
         ++acpy;
86
87
88
      }
89
```

```
// Increment the variance and measurement vector counters
++mcpy;
++vcpy;

3
y4
}

UNPROTECT(1);

return(success);

// Increment the variance and measurement vector counters
// Universe success
// Increment the variance and measurement vector counters
// Universe success
// Increment the variance and measurement vector counters
// Universe success
// Increment the variance and measurement vector counters
// Universe success
// Universe s
```

5.5.7 fishs_add.c

```
1 // file:fishs_add.c
   2 // (c) 2010- University of Oulu, Finland
   3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
   4 // Licensed under FreeBSD license.
   6 #include "LPI.h"
   7
  8 /*
   9
                         Inverse problem solver using direct calculation of the
10
                         Fisher information matrix. Data accumulation.
11
12
                          Arguments:
13
                              Qvec
                                                         Upper triangular part of the Fisher
14
                                                          information matrix as a vector
                                                         Modified measurement vector
15
                             yvec
                              arows Theory matrix rows
16
17
                                                     Measurements
                             meas
18
                             var
                                                         Measurement variances
19
                                                       Number of unknowns
                             nrow Number of theory rows in arows
20
21
22
                         Returns:
23
                              success 1 if the processing was successful, 0 otherwise
24
25 */
26
27 SEXP fishs_add( const SEXP Qvec , const SEXP yvec , const
                          SEXP arows , const SEXP meas % \left( 1\right) =\left( 1\right) +\left( 1\right
                          SEXP nx , const SEXP nrow )
28 {
29
                    Rcomplex *q = COMPLEX(Qvec);
                    Rcomplex *y = COMPLEX(yvec);
30
                    int n = *INTEGER(nx);
31
32
                    int nr = *INTEGER(nrow);
33
                    int i = 0;
34
                    int j = 0;
                    int 1 = 0;
35
36
37
                    Rcomplex * restrict qtmp;
38
                    Rcomplex * restrict acpy = COMPLEX(arows);
39
                    Rcomplex * restrict atmp;
40
                    Rcomplex * restrict ytmp;
                    Rcomplex * restrict mcpy = COMPLEX(meas);
41
42
                                                              * restrict vcpy = REAL(var);
                    double
43
44
                    SEXP
                                                                                                                   success;
45
                     int
                                                              * restrict i_success;
```

```
46
47
    // success output
48
    PROTECT( success = allocVector( LGLSXP , 1 ) );
49
50
    // local pointer to the success output
    i_success = LOGICAL( success );
51
52
53
    // set the success output
    *i_success = 1;
54
55
    // Go through all theory matrix rows
56
    for( 1 = 0 ; 1 < nr ; ++1 ){
57
58
59
      // Pointers to y-vector and Fisher information matrix
60
      ytmp = y;
61
      qtmp = q;
62
63
      // Go through all range gates
      for( i = 0 ; i < n ; ++i ){
64
65
         // Second pointer to the theory matrix
66
67
         atmp = acpy;
68
69
         // Go through all columns in the upper triangular part
70
         for(j = 0; j < (n - i); ++j){
71
72
       // Add information
73
       qtmp->r += (acpy->r * atmp->r + acpy->i * atmp->i) / *
74
       qtmp->i += ( acpy->r * atmp->i - acpy->i * atmp->r ) / *
          vcpy;
75
76
       // Increment the second theory matrix counter
77
       ++atmp;
78
79
       // Increment the information matrix counter
80
      ++qtmp;
81
82
        }
83
84
         // Add the corresponding measurement to the y-vector
         ytmp->r += (mcpy->r * acpy->r + mcpy->i * acpy->i) /
85
            *vcpy;
         ytmp->i += ( mcpy->i * acpy->r - mcpy->r * acpy->i ) /
86
            *vcpy;
87
88
         // Increment the y-vector counter
89
         ++ytmp;
90
```

```
// Increment the theory matrix counter
91
92
         ++acpy;
93
       }
94
 95
       // Increment the variance and measurement vector counters
96
97
       ++mcpy;
98
       ++vcpy;
99
100
     }
101
     UNPROTECT(1);
102
103
     return(success);
104
105
106 }
```

5.5.8 fishs_add_clutter.c

```
1 // file:fishs_add_clutter.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
     A special version of fisher solver for ground clutter
10
     estimation. Assumes unit variance and adds only one
11
     row at a time.
12
13
     Arguments:
            Upper triangular part of Fisher information matrix
14
      Qvec
             Modified measurement vector
15
            One row of theory matrix
16
      arow
17
      meas
            Measurement
18
            Number of unknowns
      nx
19
20 */
21
22 void fishs_add_clutter( const SEXP Qvec , const SEXP yvec ,
      Rcomplex * arow , Rcomplex * meas , const int nx )
23 {
    Rcomplex *q = COMPLEX(Qvec);
24
25
    Rcomplex *y = COMPLEX(yvec);
26
    int n = nx;
    int i = 0;
27
    int j = 0;
28
29
30
    Rcomplex * restrict qtmp;
31
    Rcomplex * restrict acpy = arow;
32
    Rcomplex * restrict atmp;
33
    Rcomplex * restrict ytmp;
34
    Rcomplex * restrict mcpy = meas;
35
36
    // Pointers to y-vector and Fisher information matrix
37
    ytmp = y;
38
    qtmp = q;
39
    // Go through all range gates
40
41
    for( i = 0 ; i < n ; ++i ){
42
43
      // Second pointer to the theory matrix
44
      atmp = acpy;
45
      // Go through all columns in the upper triangular part
46
```

```
for( j = 0 ; j < (n - i) ; ++j){
47
48
49
           // Add information
           qtmp->r += ( acpy->r * atmp->r + acpy->i * atmp->i );
50
51
           qtmp \rightarrow i += (acpy \rightarrow r * atmp \rightarrow i - acpy \rightarrow i * atmp \rightarrow r);
52
53
           // Increment the second theory matrix counter
54
           ++atmp;
55
56
           // Increment the information matrix counter
57
           ++qtmp;
58
59
60
        // Add the corresponding measurement to the y-vector
61
62
        ytmp \rightarrow r += (mcpy \rightarrow r * acpy \rightarrow r + mcpy \rightarrow i * acpy \rightarrow i);
63
        ytmp \rightarrow i += (mcpy \rightarrow i * acpy \rightarrow r - mcpy \rightarrow r * acpy \rightarrow i);
64
        // Increment the y-vector counter
65
66
        ++ytmp;
67
68
        // Increment the theory matrix counter
69
        ++acpy;
70
71
      }
72
73 }
```

5.5.9 index_adjust.c

```
1 // file:index_adjust.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8 /*
9
    Adjust tx / rx indices. The rising edges are shifted
10
    shifts[0] samples and the falling edges shifts[1]
    samples towards larger indices. Also negative
    shifts are allowed.
12
13
14
    This function allocates new data vectors
15
16
    Arguments:
17
     idata
             ndata integer vector of TX pulse / RX positions
     ndata
              Number of data points in idata
18
19
     shifts 2-vector of shifts
              (shifts at rising and falling edges)
20
21
22
    Returns:
23
                A list with elements
24
                 idata
                         Index vector after adjustements
25
             success Logical, set if all processing
26
                     was successful
27 */
28
29 SEXP index_adjust_R( SEXP idata , SEXP ndata , SEXP shifts )
30 {
31
    SEXP ans;
32
    SEXP idata_new;
    SEXP s;
33
    SEXP names;
34
35
    char *cnames[2] = {"idata", "success"};
36
    int *inew;
37
    int *iold;
38
    register uint64_t k;
39
40
41
    // Output list ans[[1]] = idata , ans[[2]] = success
42
    PROTECT( ans = allocVector( VECSXP , 2 ) );
43
44
    // Allocate the new logical vector
45
    PROTECT( idata_new = allocVector( LGLSXP , *(INTEGER(ndata)
        )));
46
```

```
47
                      // A pointer to the new data vector
                       inew = LOGICAL( idata_new );
 48
 49
                      // A pointer to the old data vector % \left( 1\right) =\left( 1\right) \left( 
50
51
                       iold = LOGICAL( idata );
52
53
                      // Copy data from old to new
 54
                      for (k = 0; k < *(INTEGER(ndata)); ++k){
                                inew[k] = iold[k];
 55
 56
 57
58
                      // The success logical
 59
                      PROTECT( s = allocVector( LGLSXP , 1 ) );
 60
                      // The actual work
 61
 62
                      s = index_adjust( idata_new , ndata , shifts );
 63
 64
                      // Collect the data into the return list
                      SET_VECTOR_ELT( ans , 0 , idata_new );
 65
                       SET_VECTOR_ELT( ans , 1 , s );
 66
 67
 68
                      // Set the name attributes
                      PROTECT( names = allocVector( STRSXP , 2 ));
 69
 70
                      SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
                      SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
 71
                      setAttrib( ans , R_NamesSymbol , names);
 72
 73
 74
                      UNPROTECT (4);
75
76
                      return(ans);
77
78 }
79
80 /*
                      Adjust TX / RX indices. The rising edges are shifted
81
                      shifts[0] samples and the falling edges shifts[1]
 82
                       samples towards larger indices.
83
 84
                      Also negative shifts are allowed.
 85
                      This function overwrites the idata vector
 86
 87
 88
                       Arguments:
 89
                           idata
                                                                 ndata integer vector of TX pulse / RX positions
                                                                 Number of data points in idata
 90
                           ndata
                                                                 2-vector of shifts
 91
 92
                                                                  (shifts at rising and falling edges)
 93
 94
                      Returns:
                           success 1 if all processing was successful, 0 otherwise
 95
```

```
96
97 */
98
99 SEXP index_adjust( SEXP idata , SEXP ndata , SEXP shifts)
100 {
     int *id = INTEGER(idata);
101
102
     int *nd = INTEGER(ndata);
103
     int *sh = INTEGER(shifts);
104
     // temporary variables
105
     int sh1;
106
     register int64_t k;
107
     int lasttrue;
108
     int ncut;
109
     int nadd;
     // for the return value
110
111
     SEXP success;
112
     int *isuccess;
113
     // Allocate the return value and initialise it
114
115
     PROTECT(success = allocVector(LGLSXP,1));
116
     isuccess = LOGICAL(success);
     *isuccess = 1;
117
118
119
     // The shift on rising edges is done by
120
     //shifting the whole index vector
121
122
     // Find the last true index in the whole vector,
     // it will be needed later
123
124
     lasttrue = 0;
     for (k = (*nd - 1); k \ge 0; --k)
125
126
       if( id[k] ){
127
         lasttrue = k;
128
         break;
129
     }
130
131
     // If sh[0] < 0, shift towards smaller indices
132
     if(sh[0] < 0){
133
134
       for (k = 0; k < (*nd + sh[0]); ++k){
         id[k] = id[ k - sh[0] ];
135
       }
136
       // The last value is repeated in the remaining points
137
138
       for (k = (*nd + sh[0]); k < *nd; ++k){
139
         id[k] = id[(*nd - 1)];
140
       }
     }
141
142
143
     // If sh[0] > 0, shift towards larger indices
     if(sh[0] > 0){
144
```

```
145
        for(k = (*nd - 1); k >= sh[0]; --k){
146
          id[k] = id[ k - sh[0] ];
147
       // The first value is repeated in the first {\rm sh}\left[0\right] points
148
149
        for(k = (sh[0] - 1); k > 0; --k){
150
         id[k] = id[0];
151
     }
152
153
154
     // Add the shift that was already done to sh[1]
     sh1 = sh[1] - sh[0];
155
156
     // If {\rm sh1} < 0 we are supposed to shift
157
158
     // the falling edges towards smaller indices
159
     if(sh1 < 0){
160
       ncut = 0;
161
       for(k = (*nd - 1); k \ge 0; --k){
162
          if(id[k] == 0){
163
            ncut = 0;
164
          }else{
165
            --ncut;
166
167
         if ( ncut >= sh1 ) id[k] = 0;
168
169
     // If sh1 > 0 we are supposed to shift
170
     \ensuremath{//} the falling edges towards larger indices
171
172
     if( sh1 > 0 ){
173
       nadd = 0;
174
        for (k = 0; k < *nd; ++k){
175
          if( id[ k ] == 0 ){
176
            ++nadd;
177
          }else{
178
            nadd = 0;
179
180
         if ( nadd <= sh1 ) id[k] = 1;
181
182
183
     }
184
185
     // Now there may be errors in the very end of the index
     // vector, correct using the stored index lasttrue
186
     for( k = ( lasttrue + sh[1] + 1 ) ; k < *nd ; ++k ){
187
       id[k] = 0;
188
189
190
191
     // Remove protection from the return value
     UNPROTECT(1);
192
193
```

```
194  // Return the variable success only, the data is stored
195  // in the R vectors 'cdata', 'idatar', and 'idatai'
196  return(success);
197
198 }
```

5.5.10 average_power.c

```
1 // file:average_power.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
    Average power vector for variance estsimation
10
    The algorithm proceeds as follows
11
12
13
    1. Locate falling edges of pulses from idatatx
    2.\ \mbox{locate} the first falling edge at least maxrange samples
14
       from the beginning, give this pulse the pulse index 0
15
    3. Pick maxrange samples from idatatx from immediately
16
17
       *before* the first falling edge
18
    4. At all other falling edges, compare the maxrange points
19
       before the edge with the samples picked in (3)
    5. If the vectors compared in (4) are identical, also this
20
       pulse is given pulse index 0, repeat for all pulses
21
    6. If there pulses are left without an index, select the
22
23
       first of them and repeat steps (4) and (5) to give
24
       these pulses the index 1.
    7. Continue with indices 2, 3, ...
25
       until all pulses have an index
26
27
    8. When all pulses have indices, calculate average
28
       power profiles from pulses with identical indices
29
30
31
    Arguments:
32
     cdata
               Complex receiver samples
33
     idatatx Transmitter sample index vector
34
     idatarx Receiver sample index vector
35
              Number of points in data vectors
36
     maxrange Maximum range for power profile estimation
37
38
    Returns:
39
     pdata
               Average power vector. The first element contains
         the
               ratio largest pulse index / number of pulses.
40
41
42
43
44 SEXP average_power( SEXP cdata , SEXP idatatx , SEXP idatarx
      , SEXP ndata , SEXP maxrange)
45 {
```

```
46
     Rcomplex * cd = COMPLEX( cdata );
     int * idtx = LOGICAL( idatatx );
47
     int * idrx = LOGICAL( idatarx );
48
     int nd = *INTEGER( ndata );
49
50
     int maxr = *INTEGER( maxrange );
51
52
     SEXP pdata;
53
     double *pd;
     double *ptmp;
54
55
     int *pedges;
     int nedges;
56
57
     int *pinds;
58
     int *nsamp;
     int k, i, j;
59
     int pindcur;
60
     int pindmax;
61
     int npulse;
62
     int indprev;
63
64
     int p1;
65
     int sameamb;
66
     int r;
67
     int ippend;
68
     int ntot;
69
     double ptot;
70
71
     ntot = 0;
72
     ptot = .0;
73
74
     // Inspect the TX index vector
75
     // to make sure that 1 is exactly 1
76
     for( k = 0 ; k < nd ; ++k ) idtx[ k ] = idtx[ k ] ? 1 : 0 ;
77
78
     // Allocate the power vector
79
     PROTECT( pdata = allocVector( REALSXP , nd ) );
80
81
     // A pointer to the power vector
82
     pd = REAL( pdata );
83
84
     // Initialise to zero
     for( k = 0 ; k < nd ; ++k ) pd[ k ] = 0.;
85
86
87
     // Allocate a temporary vector for
88
     // power profile calculation
     ptmp = R_Calloc( nd , double );
89
90
91
     // Initialise to zero
92
     for( k = 0 ; k < nd ; ++k ) ptmp[ k ] = 0.;
93
94
     // Allocate a vector for sample counter
```

```
95
     nsamp = R_Calloc( nd , int );
96
97
     // Initialise to zero
     for(k = 0; k < nd; ++k) nsamp[k] = 0;
98
99
100
     // Allocate a vector for pulse edge positions
101
     // (this could be shorter if needed)
102
     pedges = R_Calloc( nd , int );
103
104
     // Initialise to zero
     for( k = 0 ; k < nd ; ++k ) pedges[ k ] = 0;</pre>
105
106
107
     // Allocate a vector for pulse indices
108
     pinds = R_Calloc( nd , int );
109
110
     // Initialise to -1
     for( k = 0; k < nd; ++k) pinds[ k] = -1;
111
112
113
114
     // Locate all falling edges of pulses
115
     nedges = 0;
     for(k = 0; k < (nd - 1); ++k)
116
117
118
         if( idtx[ k ] )
119
         if( !(idtx[ k + 1] ) )
120
121
             pedges[ nedges++ ] = k;
122
123
124
       }
125
       }
126
127
     // The first falling pulse edge at least
     // maxr samples from the beginning
128
     for(k = 0; k < nedges; ++k)
129
130
         if( pedges[ k ] > maxr )
131
132
133
         p1 = k;
134
         break;
       }
135
136
       }
137
138
     // Inspect the tx indices and give a unique index for
139
     // each unique 0-lag range-ambiguity function
140
     pindcur = 0;
141
     for(k = p1; k < nedges; ++k)
142
143
         // pinds < 0 for pulses that do not yet have an index
```

```
144
          if( pinds[ k ] < 0 )</pre>
145
146
          // Go through all the pulses
          for( i = k ; i < nedges ; ++i )
147
148
            {
149
               \ensuremath{//} Compare only with pulses that
150
               // do not yet have an index
151
               if( pinds[ i ] < 0 )</pre>
152
153
               // Inspect the points just before this pulse
154
               sameamb = 1;
155
               for( j = 0 ; j < maxr ; ++ j )
156
                 {
157
                   if( (idtx[ pedges[ k ] - j ]) != (idtx[ pedges[
                        i ] - j ]) )
158
                 {
159
                   sameamb = 0;
160
                   break;
                 }
161
162
                 }
163
               // If the ambiguities were identical,
164
               // assign the pulse with the index pindcur
165
               if( sameamb ) pinds[ i ] = pindcur;
166
            }
167
            }
          // Increment pindcur
168
169
          ++pindcur;
170
        }
171
        }
172
      // There may be a pulse / pulses without an index
173
174
      // in the begin of data vector.
175
      // Give them an index if possible
176
      if (p1 > 0)
177
        {
          for( i = p1; i < nedges; ++i)
178
179
180
          sameamb = 1;
181
          for( j = 0 ; j < pedges[ p1 - 1 ] ; ++j )
182
               if( idtx[ pedges[ p1 - 1 ] - j ] != idtx[ pedges[ i
183
                   ] - j ] )
184
            {
185
               sameamb = 0;
186
               break;
            }
187
188
            }
189
          if ( sameamb )
190
```

```
191
              pinds[ p1 - 1 ] = pinds[ i ];
192
              break;
193
           }
194
195
         // Give a new index for the pulse p1-1 if it did not
             match
196
          // with any of the exisiting ones. Pulses before p1-1
         // not be used and they do not need an index.
197
         if( pinds[ p1 - 1 ] < 0 ) pinds[ p1 - 1 ] = pindcur;
198
199
200
201
     // Store the largest pind
202
     pindmax = pindcur;
203
204
     // We have now an index for each pulse that needs one.
         Pulses
     // with equal indices have similar power profile range
205
         ambiguity
     // functions and their signal powers can be averaged.
206
     // Now we will walk through all different pulse indices,
207
208
     // calculate the correspondign power-profiles, and
209
     // store the results in appropriate places in the average
210
     // power vector
211
212
     // Start from the first falling edge, or
213
     // one point before if necessary
214
     if(p1 > 0) --p1;
215
     // Go through all pulses
216
217
     for (k = p1 ; k < nedges ; ++k)
218
219
220
          // The indices will be set to -1 after processing,
          // an index >= indicates that the point has not
221
222
         // yet been processed
223
         if( pinds[ k ] >= 0 )
224
225
226
          // Initialise the temporary power vector to zero
         for( i = 0 ; i < nd ; ++i ) ptmp[ i ] = 0.;
227
228
229
          // Initialise the sample counter to zero
230
         for( i = 0 ; i < nd ; ++i ) nsamp[ i ] = 0 ;
231
232
         // Check remaining pulses and try to find
233
         // the same index
234
         for (j = k; j < nedges; ++j)
235
            {
```

```
236
              // If a matching index is found, add power from the
237
              // ipp to the temporary profile and increment
                 sample
              // counter accordingly
238
              if( pinds[j] == pinds[k])
239
240
241
242
              // Find distance to the next pulse end (must not
243
              // stop at pulse start in order to facilitate
              // bistatic operation)
244
              if( (j + 1) >= nedges)
245
246
247
                  ippend = nd - pedges[ j ];
248
                }
249
              else
250
                {
251
                  ippend = pedges[ j + 1 ] - pedges[ j ];
252
              for( i = 0 ; i < ippend ; ++i )
253
254
                {
255
                  r = pedges[j] + i;
                  // This cuts off points that are too close to
256
257
                  // the beginning of the data vector
258
                  if(r >= maxr)
                {
259
260
                  if( idrx[ r ] )
261
262
                      ptmp[ i ] += cd[ r ].r * cd[ r ].r + cd[ r
                          ].i * cd[ r ].i;
                      nsamp[ i ] += 1;
263
264
                      ptot += cd[r].r * cd[r].r + cd[r].i
                         * cd[ r ].i;
                      ++ntot;
265
                    }
266
                }
267
                }
268
269
            }
270
            }
271
          // Divide the summed powers by
272
          // the number of summed samples
273
274
         for(i = 0; i < nd; ++i)
275
276
              if( nsamp[ i ] > 100 ){
277
            ptmp[ i ] /= (double) nsamp[ i ];
278
              }else{
            ptmp[ i ] = -1.;
279
280
              }
            }
281
```

```
282
283
          // Go through the indices again and copy the power
284
          // values to appropriate places Set pinds to -1 at
          // points that have already been visited
285
286
          pindcur = pinds[ k ];
287
          for(j = k; j < nedges; ++j)
288
            {
289
              if( pinds[ j ] == pindcur )
            {
290
              if( (j + 1) >= nedges)
291
292
293
                  ippend = nd - pedges[ j ];
294
295
              else
296
                {
297
                  ippend = pedges[ j + 1 ] - pedges[ j ];
                }
298
              for( i = 0 ; i < ippend ; ++i )
299
300
301
                  r = pedges[j] + i;
302
                  pd[ r ] = ptmp[ i ];
303
304
              pinds[j] = -1;
305
            }
306
307
308
       }
309
310
       }
311
312
     // Put the grand average power to points that did not have
     // enough averaged samples (they are set to -1 at this
313
         point)
     ptot /= (float)ntot;
314
315
     for( i = 0; i < nd; ++i){
316
        if( pd[ i ] < 0.) pd[ i ] = ptot;
317
318
319
     // Store the ratio pindmax / nedges to the first data point
     \ensuremath{//} If the ratio is large the power estimation will not
320
         perform
321
     // very well.
322
     // The power value in this point cannot ever be needed in
323
     pd[0] = (float)pindmax / (float)nedges;
324
325
     // Free the temporary allocations
326
     Free(ptmp);
```

```
327 Free(nsamp);
328 Free(pinds);
329 Free(pedges);
330
331 UNPROTECT(1);
332 return(pdata);
333
334 }
```

5.5.11 lagged_products.c

```
1 // file:lagged_products.c
2 // (c) 2010 - University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
    Calculate lagged products of a signal
9
    and its complex conjugate.
10
11
    This function allocates new data vectors
12
13
    Arguments:
             ndata1 vector of complex signal samples
14
     cdata1
              ndata2 vector of complex signal samples
15
     cdata2
              ndata1 integer vector of usable
16
17
              RX sample positions
18
     idata2
              ndata2 integer vector of usable
19
              RX sample positions
              Number of samples in cdata1 and idata1
20
     ndata1
     ndata2
              Number of samples in cdata2 and idata2
21
22
     lag
              Lag
23
24
    Returns:
25
                A list with elements
     ans
26
                         Complex vector of lagged products
                 cdata
27
                 idata
                         Index vector for cdata
28
                 ndata
                         Data vector length
                 success Logical, set if all processing
29
30
                          was successful
31 */
32
33 SEXP lagged_products_alloc( SEXP cdata1 , SEXP cdata2 , SEXP
      idata1 , SEXP idata2 , SEXP ndata1 , SEXP ndata2 , SEXP
      lag)
34 {
    Rcomplex *cd1 = COMPLEX(cdata1);
35
36
    Rcomplex *cd2 = COMPLEX(cdata2);
    int *id1 = LOGICAL(idata1);
37
    int *id2 = LOGICAL(idata2);
38
39
    int *nd1 = INTEGER(ndata1);
40
    int *nd2 = INTEGER(ndata2);
    int *1 = INTEGER(lag);
41
42
43
    SEXP ans;
    SEXP lcdata;
44
45
    Rcomplex *lcd;
```

```
46
    SEXP lidata;
47
    int *lid;
48
    SEXP success;
49
    int *isuccess;
50
    SEXP ndata;
51
    int *nd;
52
    SEXP names;
    char *cnames[4] = {"cdata","idata","ndata","success"};
53
54
    int k=0;
55
    // Allocate the return value list
56
    PROTECT( ans = allocVector( VECSXP , 4 ) );
57
58
59
    // Allocate the ndata output
    PROTECT( ndata = allocVector( INTSXP , 1 ) );
60
61
62
    // A local pointer to ndata
63
    nd = INTEGER( ndata );
64
65
    // Output data length will be minimum of the two
66
    // input data lengths, minus the time-lag
    *nd = *nd1 - *1;
67
68
    if (*nd1 > *nd2) *nd = *nd2 - *1;
69
70
    // Allocate the lagged product vector
71
    PROTECT( lcdata = allocVector( CPLXSXP , *nd ) );
72
73
    // A local pointer to the lagged product vector
74
    lcd = COMPLEX( lcdata );
75
76
    // Allocate an index vector for the lagged products
77
    PROTECT( lidata = allocVector( LGLSXP , *nd ) );
78
79
    // A local pointer to the lagged product vector
    lid = LOGICAL( lidata );
80
81
82
    // Allocate the success return value
83
    PROTECT( success = allocVector( LGLSXP , 1 ) );
84
85
    // A local pointer to the success value
    isuccess = LOGICAL( success );
86
87
    *isuccess = 1;
88
89
    // The actual lagged product calculation
90
    for(k = 0; k < *nd; ++k){
91
92
       // Calculate the index vector point
      lid[k] = (id1[k] * id2[k+ *l]);
93
94
```

```
// Calculate the actual data product only if the index
                                            vector is set
                               if(lid[k]){
   96
                                       lcd[k].r = cd1[k].r * cd2[k+*l].r + cd1[k].i * cd2[k+*l].r + cd1[k].r + cd1[k].i * cd2[k+*l].r + cd1[k].r + cd2[k+*l].r + cd1[k].r + cd2[k+*l].r + cd2[
   97
                                       lcd[k].i = -cd1[k].r * cd2[k+*1].i + cd1[k].i * cd2[k+*1].i + cd2[k+*
   98
                                                        *1].r;
   99
                      }
100
101
102
                      // Collect the return values under the list "ans"
103
104
                      SET_VECTOR_ELT( ans , 0 , lcdata );
105
                      SET_VECTOR_ELT( ans , 1 , lidata );
106
                      SET_VECTOR_ELT( ans , 2 , ndata );
107
                      SET_VECTOR_ELT( ans , 3 , success );
108
109
                      // Set the name attributes
                      PROTECT( names = allocVector( STRSXP , 4 ) );
110
                      {\tt SET\_STRING\_ELT(\ names\ ,\ 0\ ,\ mkChar(\ cnames[0]\ )\ );}
111
112
                      SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
113
                      SET_STRING_ELT( names , 2 , mkChar( cnames[2] ) );
114
                      SET_STRING_ELT( names , 3 , mkChar( cnames[3] ) );
115
                      setAttrib( ans , R_NamesSymbol , names );
116
117
                      UNPROTECT (6);
118
119
                      return(ans);
120
121 }
122
123
124
125
126 /*
127
                      Calculate lagged products of a signal
128
                      and its complex conjugate.
129
130
                      This function overwrites existing data vectors
131
132
                      Arguments:
                           cdata1 ndata1 vector of complex signal samples
133
134
                                                           ndata2 vector of complex signal samples
                           cdata2
135
                          idata1
                                                          ndata1 integer vector of usable
136
                                                           RX sample positions
137
                           idata2
                                                           ndata2 integer vector of usable
138
                                                            RX sample positions
139
                                                            complex vector for the lagged products
                           cdatap
                                                          integer vector for the lagged product indices
140
                           idatap
```

```
141
               Number of samples in cdata1 and idata1
               Number of samples in cdata2 and idata2
142
      ndata2
143
      lag
               Lag
144
145
     Returns:
146
      success 1 if processing was succesful, 0 otherwise
147
148 */
149
150 SEXP lagged_products( SEXP cdata1 , SEXP cdata2 , SEXP idata1
        , SEXP idata2 , SEXP cdatap ,\
                  SEXP idatap , SEXP ndata1 , SEXP ndata2 , SEXP
151
                     lag )
152 {
153
                            COMPLEX(cdata1);
     Rcomplex *cd1
                         =
154
     Rcomplex *cd2
                            COMPLEX (cdata2);
               *id1
155
     int
                            LOGICAL(idata1);
156
                         = LOGICAL(idata2);
     int
               *id2
157
     Rcomplex *cdp
                            COMPLEX(cdatap);
                         = LOGICAL(idatap);
158
     int
               *idp
159
     int
               nd1
                         = *INTEGER(ndata1);
160
     int
               nd2
                         = *INTEGER(ndata2);
161
     int
               1
                         = *INTEGER(lag)
162
     SEXP
               success
163
     int
               *isuccess
                         = 0
164
                k
     int
165
     int
                npr
166
167
     // Output data length will be minimum of the
168
     // two input data lengths, minus the time-lag
169
     npr = nd1 - 1;
170
     if ( nd1 > nd2 ) npr = nd2 - 1;
171
172
     // Allocate the success return value
     PROTECT( success = allocVector( LGLSXP , 1 ) );
173
174
175
     // A local pointer to the success value
176
     isuccess = LOGICAL( success );
177
     *isuccess = 1;
178
179
     // The actual lagged product calculation
     for(k = 0; k < npr; ++k){
180
181
182
        // The logical vector
183
        idp[k] = (id1[k] * id2[k+1]);
184
185
       // Multiply the actual data points only
       // if the logical vector is set
186
187
       if(idp[k]){
```

```
188
          cdp[k].r = cd1[k].r * cd2[k+1].r + cd1[k].i * cd2[k+1]
          cdp[k].i = cd1[k].r * cd2[k+1].i - cd1[k].i * cd2[k+1]
189
             ].r;
190
       }
191
     }
192
193
     // Set the logical vector to false at
     // points where it cannot be calculated
194
     for(k = 0; k < 1; ++k){
195
       idp[npr+k] = 0;
196
197
198
199
     UNPROTECT(1);
200
201
     return(success);
202
203 }
204
205
206
207 /*
208
     Real-valued lagged products for variance estimation.
209
210
     No Index vectors, because they are carried with
211
     the complex vectors.
212
213
     This function overwrites existing data vectors
214
215
     Arguments:
216
      rdata1 ndata1 vector of real signal samples
      rdata2 ndata2 vector of real signal samples
217
              real vector for the lagged products
218
      prdata
               Number of samples in rdata1
219
      ndata1
               Number of samples in rdata2
220
      ndata2
221
      lag
               Lag
222
223
     Returns:
224
      success 1 if processing was successful, 0 otherwise
225
226 */
227 SEXP lagged_products_r( SEXP rdata1 , SEXP rdata2 , SEXP
      prdata , SEXP ndata1 ,\
228
                SEXP ndata2 , SEXP lag )
229 {
230
     double *rd1
                          REAL(rdata1)
231
     double *rd2
                         REAL(rdata2)
232
     double *prd
                          REAL (prdata)
233
     int
             nd1
                       = *INTEGER(ndata1);
```

```
234
     int
            nd2
                      = *INTEGER(ndata2);
235
                       = *INTEGER(lag)
     int
            1
236
     SEXP
             success
237
     int
             *isuccess
238
     int
             k
239
     int
             npr
240
     // Output data length will be minimum of the two input
241
242
     // data lengths, minus the time-lag
243
     npr = nd1 - 1;
     if ( nd1 > nd2 ) npr = nd2 - 1;
244
245
246
     // Allocate the success return value
247
     PROTECT( success = allocVector( LGLSXP , 1 ) );
248
249
     // A local pointer to the success value
250
     isuccess = LOGICAL( success );
251
     *isuccess = 1;
252
     \ensuremath{//} The actual lagged product calculation
253
254
     for( k = 0; k < npr; ++k){
255
       prd[k] = rd1[k] * rd2[k+ 1];
256
257
258
     UNPROTECT(1);
259
260
     return(success);
261
262 }
```

5.5.12 average_profile.c

```
1 // file:average_profile.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
    Average lag-profile vector for speeding up
10
    the inversion process. Each average is
11
    calculated over samples from the same point in
    the repeated code cycle
12
13
14
    The complicated structure is used because
    measuremnts may contain additional sync
15
16
    times which need to be skipped.
17
18
19
    Arguments:
     cdata Complex lagged product vector
20
     idata Index vector for cdata
21
     ndata Data vector length
22
23
     N_CODE Code cycle length
24
25
    Returns:
     success 1 if the processing was successful, 0 otherwise
26
27
28 */
29
30 SEXP average_profile( SEXP cdata , SEXP idata , SEXP ndata ,
      SEXP N_CODE)
31 {
32
    Rcomplex * cd = COMPLEX( cdata );
33
    int * id = LOGICAL( idata );
34
    int nd = *INTEGER( ndata );
35
    int ncode = *INTEGER( N_CODE );
36
37
    double *aver;
    double *avei;
38
39
    Rcomplex *ad;
    R_len_t *nave;
40
41
    R_len_t k;
42
    R_len_t ind1 , ind2, ipp_count;
    SEXP success;
43
44
    int *isuccess;
45
46
    // Allocate the return value and initialise it
```

```
47
    PROTECT(success = allocVector(LGLSXP,1));
    isuccess = LOGICAL(success);
48
49
    *isuccess = 1;
50
51
    // Allocate the average vectors,
52
    // real and imaginary parts separately
53
    aver = (double*) R_Calloc( nd , double );
54
    avei = (double*) R_Calloc( nd , double );
55
56
    // Initialise to zero
    for( k = 0; k < nd; ++k){
57
       aver[ k ] = 0.;
58
59
       avei[ k ] = 0.;
60
    }
61
62
    // Allocate vector for data sample counter
63
    nave = R_Calloc( nd , R_len_t );
64
    // Initialise to zero
65
66
    for( k = 0 ; k < nd ; ++k ) nave[ k ] = 0;
67
68
    // Start from begniing of the data vctor
69
    ind1 = 0;
70
    ind2 = 0;
71
72
    // Search for the start of the first pulse
    while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;</pre>
73
74
    while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;
75
    ipp_count = 0;
76
77
    // Repeat until end of data
78
    while( ind2 < nd ){
79
80
       // At this point we should be at pulse starts, loop until
       // we hit a point at which both pulses have ended.
81
82
       while( id[ind1] | id[ind2]){
83
         aver[ind1] += cd[ind2].r;
84
         avei[ind1] += cd[ind2].i;
85
         ++nave[ind1];
86
         ++ind1;
87
         ++ind2;
         if(ind2==nd) break;
88
89
90
91
       if(ind2==nd) break;
92
93
       // Add power values until either of the indices
       // hits the next pulse
94
       while( (id[ind1] == 0) & (id[ind2] == 0)){
95
```

```
96
          aver[ind1] += cd[ind2].r;
          avei[ind1] += cd[ind2].i;
97
98
          ++nave[ind1];
99
          ++ind1;
100
          ++ind2;
101
          if(ind2==nd) break;
102
103
104
       if(ind2==nd) break;
105
       // Make sure that both indices point to a pulse start,
106
        // increment if necessary (This takes possible sync
107
108
        // times into account)
109
        while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;
       while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;</pre>
110
111
112
       if(ind2==nd) break;
113
114
       // Increment the ipp counter
115
       ++ipp_count;
116
       if( ipp_count == ncode ){
117
          ipp_count = 0;
118
          ind1 = 0;
119
          while( id[ind1] == 0 ) ++ind1;
120
       }
     }
121
122
123
     // Divide the summed values with number of summed pulses
124
     for( k = 0; k < nd; ++k){
125
       if( nave[ k ] ){
126
          aver[k] /= (double)nave[k];
127
          avei[k] /= (double)nave[k];
128
       }
     }
129
130
131
132
     // Now there are averaged values available for one code
133
     // cycle, copy the valeus to make further analysis
134
     // simpler. Start from beginning of the data vector.
135
     ind1 = 0;
     ind2 = 0;
136
137
138
     // Search for the start of the first pulse
     while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;</pre>
139
     while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ ind2;
140
141
     ipp_count = 0;
142
143
     // Repeat until end of data
     while( ind2 < nd ){
144
```

```
145
        // At this point we should be at pulse starts,
        // loop until both pulses have ended
146
147
        while( id[ind1] | id[ind2]){
          cd[ind2].r = aver[ind1];
148
149
          cd[ind2].i = avei[ind1];
150
          ++ind1;
151
          ++ind2;
152
          if(ind2==nd) break;
153
154
        if(ind2==nd) break;
155
156
157
        // Add power values until either of
158
        // the indices hits the next pulse
        while( (id[ind1] == 0) & (id[ind2] == 0)){
159
160
          cd[ind2].r = aver[ind1];
          cd[ind2].i = avei[ind1];
161
162
          ++ind1;
163
          ++ind2;
164
          if(ind2==nd) break;
165
166
167
        if(ind2==nd) break;
168
169
        // Make sure that both indices point to a pulse start
        while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;</pre>
170
        while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;
171
172
173
        if(ind2==nd) break;
174
175
        // Increment the ipp counter
176
        ++ipp_count;
177
        if( ipp_count == ncode ){
178
          ipp_count = 0;
179
          ind1 = 0;
180
          while( id[ind1] == 0 ) ++ind1;
181
        }
182
     }
183
184
     // Free the temporary vectors
185
     Free(nave);
     Free(aver);
186
187
     Free(avei);
188
189
     UNPROTECT(1);
190
191
     return( success );
192
193 }
```

5.5.13 mix_frequency.c

```
1 // file:mix_frequency.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
    Frequency mixing for IQ data
10
    This function allocates new vectors
11
12
13
    Argumnets:
                 ndata complex vector of data samples
     cdata
14
                 Number of samples in cdata
15
     ndata
     frequency The mixing frequency
16
17
18
    Returns:
19
     ans
                A list with elements
                         Complex data samples after
20
                 cdata
21
                          frequency mixing
             success Logical, set if all processing
22
23
                          was successful
24 */
25
26
27 SEXP mix_frequency_R( SEXP cdata , SEXP ndata , SEXP
      frequency )
28 {
    SEXP ans;
29
30
    SEXP cdata_new;
31
    SEXP s;
32
    SEXP names;
    char *cnames[2] = {"cdata", "success"};
33
34
    Rcomplex *cnew;
35
    Rcomplex *cold;
    register uint64_t k;
36
37
38
39
    // Output list ans[[1]] = cdata , ans[[2]] = success
    PROTECT( ans = allocVector( VECSXP , 2 ) );
40
41
    // Allocate the new complex vector
42
43
    PROTECT( cdata_new = allocVector( CPLXSXP , *(INTEGER(ndata
        ))));
44
45
    // A pointer to the new data vector
```

```
46
    cnew = COMPLEX( cdata_new );
47
48
    // A pointer to the old data vector
     cold = COMPLEX( cdata );
49
50
51
    // Copy data from old to new
52
    for(k = 0; k < *(INTEGER(ndata)); ++k){
53
       cnew[k].r = cold[k].r;
       cnew[k].i = cold[k].i;
54
55
56
57
    // The success logical
58
    PROTECT( s = allocVector( LGLSXP , 1 ) );
59
    // The actual frequency mixing
60
    s = mix_frequency( cdata_new , ndata , frequency );
61
62
    // Collect the data into the return list
63
    {\tt SET\_VECTOR\_ELT(\ ans\ ,\ 0\ ,\ cdata\_new\ );}
64
65
    SET_VECTOR_ELT( ans , 1 , s );
66
67
    // Set the name attributes
    PROTECT( names = allocVector( STRSXP , 2 ));
68
69
    SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
    SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
70
71
    setAttrib( ans , R_NamesSymbol , names);
72
73
    UNPROTECT (4);
74
75
    return(ans);
76
77 }
78
79 /*
    Frequency mixing for IQ data
80
81
82
    This function overwrites the cdata vector
83
84
    Argumnets:
85
     cdata
                 ndata complex vector of data samples
                 Number of samples in cdata
86
     ndata
                 The mixing frequency
87
     frequency
88
89
    Returns:
90
                 1 if all processing was successful, 0 otherwise
     success
92 SEXP mix_frequency( SEXP cdata , SEXP ndata , SEXP frequency)
93 {
94 // Pointers to the R variables
```

```
95
     Rcomplex *cd = COMPLEX(cdata);
     int *nd = INTEGER(ndata);
96
97
     double *fr = REAL(frequency);
98
     register uint64_t k, nc;
99
     double arg;
100
     Rcomplex ctmp;
101
     // Temporary variables
102
     int ncycle;
103
     double tmpprod;
     double idiff;
104
105
     double *coefr;
106
     double *coefi;
107
     // For the return value
108
     SEXP success;
109
     int *isuccess;
110
111
     // Allocate the return value and initialise it
112
     PROTECT(success = allocVector(LGLSXP,1));
113
     isuccess = LOGICAL(success);
114
     *isuccess = 1;
115
116
     // The multiplicand will be cyclic, find the cycle length
117
     ncycle = *nd;
118
     for(k = 1; k < *nd; ++k){
119
        tmpprod = *fr * (double)(k);
        idiff = tmpprod - (double)((int)(tmpprod));
120
121
       if( fabs(idiff) <= FLT_MIN ){</pre>
122
          ncycle = k;
123
          break;
124
       }
     }
125
126
     // If the cycle length is one, the mixing would not change
127
         anything
128
     if ( ncycle == 1 ) {
129
       UNPROTECT (1);
130
        return(success);
131
132
133
     // Tabulate the cyclic coefficients.
     \ensuremath{//} This usually saves time as radar engineers tend to
134
     // select nice numerical values for the frequencies
135
136
     coefr = (double*) R_Calloc( ncycle , double );
137
     coefi = (double*) R_Calloc( ncycle , double );
138
     for(k = 0; k < ncycle; ++k){
                 = 2.0 * pi * *fr * (double)(k);
139
140
        coefr[k] = cos(arg);
        coefi[k] = sin(arg);
141
     }
142
```

```
143
144
     // Actual mixing
     nc = 0;
145
     for(k = 0; k < *nd; ++k){
146
147
       ctmp.r = cd[k].r;
148
       ctmp.i = cd[k].i;
       cd[k].r = ctmp.r * coefr[nc] - ctmp.i * coefi[nc];
149
150
       cd[k].i = ctmp.i * coefr[nc] + ctmp.r * coefi[nc];
151
       ++nc;
152
      if( nc == ncycle ) nc = 0;
153
154
155
     // Free the memory allocated for the coefficient tables
156
     Free(coefr);
157
     Free(coefi);
158
159
     // Remove protection from the return value
160
     UNPROTECT(1);
161
     \ensuremath{//} Return the variable success only, the data is stored in
162
163
     // the R vectors 'cdata', 'idatar', and 'idatai'
164
     return(success);
165
166 }
```

5.5.14 resample.c

```
1 // file:resample.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
    Resampling with linear interpolation. Reduces to a simple
    boxcar filter when the filter length is an integer
10
11
    multiple of the original sample interval.
12
13
    Final sample rate must be smaller than or
14
    equal to the original one.
15
16
    This function overwrites existing data vectors
17
18
    Arguments:
19
     cdata
              Complex data samples
              Index vector for cdata
20
     idata
     ndata
              Data vector length
21
22
               Upsamling factor
     nup
23
     nfilter
              Filter length on upsampled data
               (final length is nfilter / nup)
24
25
     nfirst
               Decimation start index
     nfirstfrac start point within the boxcar filter in
26
         upsampled units
27
     ipartial 0 if partial matched with filter
28
               should not be accepted in idata vector
29
30
    Returns:
31
      success 1 if resampling was successful, 0 otherwise
32
33 */
34
35 SEXP resample ( SEXP cdata , SEXP idata , SEXP ndata , SEXP
      nup , SEXP nfilter , SEXP nfirst , SEXP nfirstfrac , SEXP
      ipartial )
36 {
37
38
    Rcomplex * restrict cd = COMPLEX(cdata);
39
    int * restrict id = LOGICAL(idata);
    int nd = *INTEGER(ndata);
40
41
    const int nu = *INTEGER(nup);
    const int nf = *INTEGER(nfilter);
42
43
    const int ns = *INTEGER(nfirst);
    const int nsf = *INTEGER(nfirstfrac);
```

```
const int ipar = *LOGICAL(ipartial);
46
    uint64_t i, j, k, l, m, n;
47
    double frac;
48
    Rcomplex tmpsum;
49
    int tmpi[2];
50
51
    // For the return value
52
    SEXP success;
53
    int * restrict isuccess;
54
    // Allocate the return value and initialise it
55
    PROTECT(success = allocVector(LGLSXP,1));
56
57
    isuccess = LOGICAL(success);
58
    *isuccess = 1;
59
60
61
      i the current filter start point in upsampled data
62
      j the current point inside the (upsampled) boxcar filter
      k the current point within the original data vector
63
64
      1 the current point within the resampled data vector
65
66
67
    i = ns * nu ;
                    // Starting point in upsampled units
                        // We are originally at the
    // j = nu-1;
                        // beginning of the boxcar filter
69
    //
                    // increment with nu-1, we will use the full
70
    j = nsf + nu - 1;
         sample
                    // cd[k] in any case. The first resampled
71
                        one will
                    // be wrong if nsf/=0, but we could not help
72
                         this if
73
                    // nsf < 0 in any case.</pre>
                    // Starting point in original sampling
74
    k = ns;
                    // Current point in the final filtered and
75
    1 = 0;
                    // decimated data vector, start filling
76
                    // from beginning
77
    tmpsum.r = 0.; // Initialise the temp filter sum to zero
78
79
    tmpsum.i = 0.;
80
    tmpi[0] = 1;
    tmpi[1] = 0;
81
82
83
84
    while( ( ( i + nf ) / nu ) <= nd ){ // Current filter start
         + filter length <= data length
85
       while( j < nf ){</pre>
                                // One filter length of data
         tmpsum.r += cd[ k ].r; // Add the current point to the
86
            filter sum
        tmpsum.i += cd[ k ].i;
87
         tmpi[0] *= id[k];
88
```

```
89
          tmpi[1] += id[k];
 90
          j += nu;
                                  // Jump to the next point that
             actually needs to be calculated
                                  // Increment the sample counter
 91
             of the original data vector
 92
 93
       //
              // Fraction of the k'th sample in the original data
 94
              // vector that will go to 1+1'th resampled point
              frac = ((double)(j - nf + 1)) / (double)nu;
 95
       //
       // not like this, it will create effectively two filters
 96
97
       // this should be better
98
       frac = 0.;
99
       if( ( j - nf + 1 ) == nu ) frac = 1.;
100
101
           //
102
           // the whole fraction thing could be removed, but the
              above lines will fix
           // this for the time being. IV 2016-02-16.
103
104
           //
105
           // ... on the other hand, this will be rather easy to
              convert into upsamling, if that
106
           // would ever be needed?
107
           //
108
109
110
111
112
       // Now k could be beyond the data vector length,
113
114
       // check that it is not
       if( k < nd ){
115
          // Add the fraction that belongs to the k'th point
116
117
          tmpsum.r += ( 1. - frac )*cd[k].r;
          tmpsum.i += ( 1. - frac )*cd[k].i;
118
          if ( frac < .99999 ) tmpi[0] *= id[k];
119
120
          if ( frac < .99999 ) tmpi[1] += id[k];
121
          // Now tmpsum is ready, copy its contents to
122
          // the l'th element of the data vector
          cd[1].r = tmpsum.r;
123
          cd[1].i = tmpsum.i;
124
125
          id[1] = ipar ? tmpi[1] : tmpi[0];
126
          // Put the remaining fraction of
127
          // k'th sample to the tmpsum
128
          tmpsum.r = frac*cd[k].r;
129
          tmpsum.i = frac*cd[k].i;
130
          tmpi[0] = ( frac < .00001 ) ? 1 : id[k];</pre>
131
          tmpi[1] = ( frac < .00001 ) ? 0 : id[k];
132
          // One filter length backwards
```

```
j -= nf;
133
          // The sample where we ended in the previous step was
134
135
          // already added to tmpsum, jump to the next one
          j += nu;
136
137
          // Move one filter length forwards
138
          /*
139
          i += nf;
140
          ++k;
          */
141
142
          ++1;
143
144
145
       // i and k must be incremented also at end of data to get
           us out of the loop
        i += nf;
146
       ++k;
147
148
149
     // If we were exactly at end of data frac is unity, we will
150
          still get one more sample
     // k was incremented after hitting the end of data
151
152
     if( k == (nd + 1)){
153
       if( frac > .9999999 ){
154
          cd[1].r = tmpsum.r;
          cd[1].i = tmpsum.i;
155
          id[1] = ipar ? tmpi[1] : tmpi[0];
156
157
          ++1;
158
       }
159
     }
160
161
     *(INTEGER(ndata)) = 1;
162
163
     // remove protection from the return value
164
     UNPROTECT(1);
165
     \ensuremath{//} return the variable success only, the data is now stored
166
     // in the R vectors 'cdata', 'idatar', and 'idatai'
167
168
     return(success);
169
170 }
171
172
173 /*
     Resampling with linear interpolation. Reduces to a simple
174
175
     boxcar filter when the filter length is an integer
176
     multiple of the original sample interval.
177
178
     Final sample rate must be smaller than
179
     or equal to the original one.
```

```
180
181
     This function allocates new data vectors
182
183
     Arguments:
184
                Complex data samples
      cdata
185
                Index vector for cdata
      idata
186
      ndata
                Data vector length
187
                Upsamling factor
      nup
188
      nfilter Filter length on upsampled data (final length
                is nfilter / nup)
189
                Decimation start index
190
      nfirst
191
      ipartial O if partial matched with filter should not be
192
                accepted in idata vector
193
194
     Returns:
195
                A list with components:
      ans
196
                cdata
                         Resampled complex data vector
                         Index vector for cdata
197
                idata
                         Data vector length
198
                ndata
199
                success 1 if resampling was successful,
200
                          0 otherwise
201
202 */
203
204
205 SEXP resample_R( SEXP cdata , SEXP idata , SEXP ndata , SEXP
       nup , SEXP nfilter , SEXP nfirst , SEXP nfirstfrac , SEXP
       ipartial)
206 {
207
     SEXP ans;
208
     SEXP cdata_new;
     SEXP idata_new;
209
     SEXP ndata_new;
210
     SEXP s;
211
     SEXP names;
212
     char *cnames[4] = {"cdata","idata","ndata","success"};
213
214
     Rcomplex * restrict cnew;
215
     Rcomplex * restrict cold;
216
     int * restrict inew;
     int * restrict iold;
217
218
     uint64_t k;
     PROTECT_INDEX cpind=0;
219
220
     PROTECT_INDEX ipind=0;
221
222
223
     // Output list ans[[1]] = cdata , ans[[2]] = idata ,
224
     // ans[[3]] = ndata , ans[[4]] = success
     PROTECT( ans = allocVector( VECSXP , 4 ) );
225
226
```

```
227
     // Allocate the new complex vector
     PROTECT_WITH_INDEX( cdata_new = allocVector( CPLXSXP , *(
228
         INTEGER(ndata)) ) , &cpind );
229
230
     // Allocate the new logical vector
231
     PROTECT_WITH_INDEX( idata_new = allocVector( LGLSXP , *(
         INTEGER(ndata)) ) , &ipind );
232
233
     // Allocate the new ndata variable
234
     PROTECT( ndata_new = allocVector( INTSXP , 1 ) );
235
236
     // A pointer to the new cdata vector
237
     cnew = COMPLEX( cdata_new );
238
239
     // A pointer to the old cdata vector
240
     cold = COMPLEX( cdata );
241
242
     // A pointer to the new idata vector
     inew = LOGICAL( idata_new );
243
244
245
     // A pointer to the old idata vector
     iold = LOGICAL( idata );
246
247
248
     // Copy data from old cdata to new cdata
249
     for(k = 0; k < *(INTEGER(ndata)); ++k){
250
       cnew[k].r = cold[k].r;
       cnew[k].i = cold[k].i;
251
252
253
254
     // Copy data from old idata to new idata
     for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){</pre>
255
256
       inew[k] = iold[k];
257
     }
258
     // Use the same pointers to copy old ndata to new ndata
259
     inew = INTEGER( ndata_new);
260
261
     iold = INTEGER( ndata );
262
     *inew = *iold;
263
264
     // The success logical
     PROTECT( s = allocVector( LGLSXP , 1 ) );
265
266
267
     // The actual resampling
268
     s = resample( cdata_new , idata_new , ndata_new , nup ,
        nfilter , nfirst , nfirstfrac , ipartial );
269
270
     // Reallocate the vectors to match with the new data length
     SET_LENGTH( cdata_new , *INTEGER(ndata_new) );
271
     REPROTECT( cdata_new , cpind );
272
```

```
SET_LENGTH( idata_new , *INTEGER(ndata_new) );
273
274
     REPROTECT( idata_new , ipind );
275
276
     // Collect the data into the return list
     SET_VECTOR_ELT( ans , 0 , cdata_new );
277
     SET_VECTOR_ELT( ans , 1 , idata_new );
278
     SET_VECTOR_ELT( ans , 2 , ndata_new );
279
     SET_VECTOR_ELT( ans , 3 , s );
280
281
282
     // Set the name attributes
     PROTECT( names = allocVector( STRSXP , 4 ));
283
     SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
284
285
     {\tt SET\_STRING\_ELT(\ names\ ,\ 1\ ,\ mkChar(\ cnames[1]\ )\ );}
286
     {\tt SET\_STRING\_ELT(\ names\ ,\ 2\ ,\ mkChar(\ cnames[2]\ )\ );}
     SET_STRING_ELT( names , 3 , mkChar( cnames[3] ) );
287
     setAttrib( ans , R_NamesSymbol , names);
288
289
290
     UNPROTECT(6);
291
292
     return(ans);
293
294 }
```

5.5.15 prepare_data.c

```
1 // file:prepare_data.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
    Frequency mixing, index adjustments,
10
    and filtering in a single function
11
12
    Arguments:
13
     cdata
                Complex voltage data vector
                Integer vector of usable data indices
     idata
14
15
     ndata
                Data vector length
     frequency Frequency offset
16
17
     shifts
                Corrections to idata
                Upsaling factor in resampling
18
     nup
19
     nfilter
                Filter length (for upsampled data, final
                filter length is nfilter / nup)
20
                Decimation start index
21
     nfirst
22
      ipartial Logical, are partial matches of
23
                idata with the filter accepted?
24
25
    Returns:
26
                A list with elements
     ans
27
                         Final complex data vector
                         Final index vector
28
                 idata
                        Final data vector length
29
                 ndata
             success Logical, set if all processing
30
31
                          was successfull
32
33
   */
34
35
36 SEXP prepare_data( SEXP cdata , SEXP idata , SEXP ndata ,
      SEXP frequency, SEXP shifts , SEXP nup , SEXP nfilter ,
      SEXP nfirst , SEXP nfirstfrac , SEXP ipartial )
37 {
38
    SEXP ans;
39
    SEXP cdata_new;
40
    SEXP idata_new;
    SEXP ndata_new;
41
42
    SEXP s;
43
    SEXP names;
    char *cnames[4] = {"cdata","idata","ndata","success"};
44
    Rcomplex * restrict cnew;
```

```
46
    Rcomplex * restrict cold;
47
    int * restrict inew;
48
    int * restrict iold;
49
    uint64_t k;
50
    PROTECT_INDEX cpind=0;
51
    PROTECT_INDEX ipind=0;
52
53
54
55
    // Output list ans[[1]] = cdata ans[[2]] = pdata
     // ans[[3]] = idata , ans[[4]] = ndata , ans[[5]] = success
56
    PROTECT( ans = allocVector( VECSXP , 5 ) );
57
58
59
     // Allocate the new complex vector
60
    PROTECT_WITH_INDEX( cdata_new = allocVector( CPLXSXP , *(
        INTEGER(ndata)) ) , &cpind );
61
62
     // Allocate the new logical vector
    PROTECT_WITH_INDEX( idata_new = allocVector( LGLSXP , *(
63
        INTEGER(ndata)) ) , &ipind );
64
65
     // Allocate the new ndata variable
66
    PROTECT( ndata_new = allocVector( INTSXP , 1 ) );
67
68
    // A pointer to the new cdata vector
    cnew = COMPLEX( cdata_new );
69
70
71
    // A pointer to the old cdata vector
72
    cold = COMPLEX( cdata );
73
74
    // A pointer to the new idata vector
    inew = LOGICAL( idata_new );
75
76
77
    // A pointer to the old idata vector
     iold = LOGICAL( idata );
78
79
    // Copy data from old cdata to new cdata
80
81
    for (k = 0; k < *(INTEGER(ndata)); ++k){
82
       cnew[k].r = cold[k].r;
      cnew[k].i = cold[k].i;
83
84
85
86
    // Copy data from old idata to new idata
    for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){</pre>
87
88
       inew[k] = iold[k];
    }
89
90
    // Use the same pointers to copy old ndata to new ndata
91
    inew = INTEGER( ndata_new);
92
```

```
93
     iold = INTEGER( ndata );
94
     *inew = *iold;
95
96
     // The success logical
97
     PROTECT( s = allocVector( LGLSXP , 1 ) );
98
99
     // Frequency mixing
100
     s = mix_frequency( cdata_new , ndata_new , frequency );
101
102
     // Index adjustments
     s = index_adjust( idata_new , ndata_new , shifts );
103
104
105
     // Filtering
106
     s = resample( cdata_new , idata_new , ndata_new , nup ,
        nfilter , nfirst , nfirstfrac , ipartial );
107
108
     // Set cdata_new to zero at all points where idata_new == 0
109
     inew = LOGICAL( idata_new );
     for( k = 0 ; k < *INTEGER(ndata_new) ; ++k ){</pre>
110
111
       if(inew[k] == 0){
112
         cnew[k].r = .0;
113
         cnew[k].i = .0;
114
       }
115
     }
116
     // Reallocate the vectors to match with the new data length
117
118
     SET_LENGTH( cdata_new , *INTEGER(ndata_new) );
119
     REPROTECT( cdata_new , cpind );
120
     SET_LENGTH( idata_new , *INTEGER(ndata_new) );
     REPROTECT( idata_new , ipind );
121
122
     // Collect the data into the return list
123
     SET_VECTOR_ELT( ans , 0 , cdata_new );
124
     SET_VECTOR_ELT( ans , 1 , idata_new );
125
     SET_VECTOR_ELT( ans , 2 , ndata_new );
126
     SET_VECTOR_ELT( ans , 3 , s );
127
128
129
     // Set the name attributes
130
     PROTECT( names = allocVector( STRSXP , 4 ));
     SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
131
     SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
132
     SET_STRING_ELT( names , 2 , mkChar( cnames[2] ));
133
134
     SET_STRING_ELT( names , 3 , mkChar( cnames[3] ) );
135
     setAttrib( ans , R_NamesSymbol , names);
136
     UNPROTECT(6);
137
138
139
     return(ans);
140
```

141|}

5.5.16 theory_rows.c

```
1 // file:theory_rows.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6
7 #include "LPI.h"
8
9
    Make theory matrix rows and measurement vectors.
10
11
    This function allocates new data vectors.
12
13
    Arguments:
     camb
                  Complex range ambiguity functions
14
                  Index vector of range ambiguity functions
15
     iamb
                  Complex lagged product vector
16
     cprod
17
                  Index vector of lagged products
     iprod
18
     rvar
                  Measurement variance vector
19
     ndata
                  Data vector length
20
     ncur
                  Current sample index
                  Last sample index to use (in this call)
21
     nend
                  Range gate limits
22
     rlims
23
     nranges
                  Number of range gates
                  O if the vectors should not be reallocated to
24
     fitsize
25
                  match the final data size.
26
                  O if additional background term is not used
     background
27
     remoterx
                  O if measurements TX times should not be used
28
29
30
    Returns:
31
                A list with elements
     ans
32
                         Theory matrix rows
                 arows
33
                         Theory row indices
                 irows
34
                         Inversion measurement vector
35
                         Measurement variances
                 var
36
                         Number of theory rows produced
                 nrows
                 success Logical, set if all processing
37
38
                         was successful
39
40
   */
41
42 SEXP theory_rows_alloc( SEXP camb , SEXP iamb , SEXP cprod ,
     SEXP iprod , SEXP rvar , SEXP ndata , SEXP ncur , SEXP
      nend , SEXP rlims , SEXP nranges , SEXP fitsize , SEXP
      background , SEXP remoterx )
43 {
44 const int n_cur = *INTEGER(ncur);
```

```
const int n_end = *INTEGER(nend);
46
    const int n_ranges = *INTEGER(nranges);
47
    const int fit_size = *LOGICAL(fitsize);
48
    SEXP ans;
49
    SEXP arows;
50
    SEXP irows;
51
    SEXP mvec;
52
    SEXP mvar;
53
    SEXP success;
    SEXP nrows;
54
    SEXP names;
55
56
    int n_rows;
    const char * c_names[6] = {"arows","irows","m","var","
57
        nrows","success"};
    PROTECT_INDEX arind = 0;
58
    PROTECT_INDEX irind = 0;
59
    PROTECT_INDEX mind = 0;
60
61
    PROTECT_INDEX vind = 0;
62
63
64
    // Output list
    PROTECT( ans = allocVector( VECSXP , 5 ) );
65
66
67
    // A vector for the theory matrix rows
    PROTECT_WITH_INDEX( arows = allocVector( CPLXSXP , ( (
68
        n_end - n_cur + 1 ) * ( n_ranges + 1) ) ) , & arind );
69
70
    // A vector for the theory matrix indices
    PROTECT_WITH_INDEX( irows = allocVector( LGLSXP , ( ( n_end
71
         - n_cur + 1 ) * ( n_ranges + 1) ) ) , & irind );
72
73
    // A vector for the measurements
74
    PROTECT_WITH_INDEX( mvec = allocVector( CPLXSXP , ( n_end -
         n_cur + 1 ) ) , & mind );
75
76
    // A vector for the measurement errors
    PROTECT_WITH_INDEX( mvar = allocVector( REALSXP , ( n_end -
77
         n_cur + 1 ) ) , & vind );
78
79
    // Number of rows for the R output
    PROTECT( nrows = allocVector( INTSXP , 1 ) );
80
81
82
    // Success output
83
    PROTECT( success = allocVector( LGLSXP , 1 ) );
84
85
    // Call the theory_rows function to actually make the rows
    success = theory_rows( camb , iamb , cprod , iprod , rvar ,
         ndata , ncur , nend , rlims ,\
```

```
87
                              nranges , arows , irows , mvec ,
                                 mvar , nrows , background ,
                                 remoterx );
 88
     // Read the row count
 89
 90
     n_rows = *(INTEGER(nrows));
 91
 92
     // Reallocate the vectors to match with the data lengths
 93
     if(fit_size){
       SET_LENGTH( arows , ( n_rows * ( n_ranges + 1 ) ) );
 94
95
       REPROTECT( arows , arind );
96
       SET_LENGTH( irows , ( n_rows * ( n_ranges + 1 ) ) );
97
       REPROTECT( irows , irind );
       SET_LENGTH( mvec , n_rows );
98
       REPROTECT( mvec , mind );
99
100
       SET_LENGTH( mvar , n_rows );
101
       REPROTECT( mvar , vind );
102
     }
103
104
     // Collect the data into the return list
105
     {\tt SET\_VECTOR\_ELT(\ ans\ ,\ 0\ ,\ arows}
                                          );
106
     SET_VECTOR_ELT( ans , 1 , irows
                                          );
107
     SET_VECTOR_ELT( ans , 2 , mvec
                                          );
108
     SET_VECTOR_ELT ( ans , 3 , mvar
                                          );
                                          );
109
     SET_VECTOR_ELT( ans , 4 , nrows
     SET_VECTOR_ELT( ans , 5 , success );
110
111
112
     // Set the names attributes
     PROTECT( names = allocVector( STRSXP , 5 ) );
113
     SET_STRING_ELT( names , 0 , mkChar( c_names[0] ) );
114
115
     SET_STRING_ELT( names , 1 , mkChar( c_names[1] ) );
     SET_STRING_ELT( names , 2 , mkChar( c_names[2] ) );
116
     SET_STRING_ELT( names , 3 , mkChar( c_names[3] ) );
117
     SET_STRING_ELT( names , 4 , mkChar( c_names[4] ) );
118
119
     SET_STRING_ELT( names , 5 , mkChar( c_names[5] ) );
120
     setAttrib( ans , R_NamesSymbol , names);
121
122
123
     UNPROTECT (7);
124
125
     return(ans);
126
127 }
128
129
130
131
132
133 /*
```

```
134
     Make theory matrix rows and measurement vectors.
135
136
     This function overwrites existing data vectors
137
138
     Arguments:
139
      camb
                   Complex range ambiguity functions
140
      iamb
                   Index vector of range ambiguity functions
141
                   Complex lagged product vector
      cprod
142
      iprod
                  Index vector of lagged products
                  Measurement variance vector
143
      rvar
                  Data vector length
144
      ndata
145
      ncur
                  Current sample index
146
      nend
                  Last sample index to use (in this call)
147
      rlims
                  Range gate limits
148
                  Number of range gates
      nranges
149
                  Complex theory rows
      arows
150
      irows
                  Theory row indices
151
                   Inversion measurement vector
      mvec
152
      mvar
                   Inversion measurement variances
153
      nrows
                   Number of theory rows produced during
154
                   this call
155
      background 0 if additional background term is not used
156
      remoterx
                   O if measurements TX times should not be used
157
158
     Returns:
                   O if no theory rows were produced _and_ end of
159
      success
                   data was reached, 1 otherwise
160
161
    */
162
163
164 SEXP theory_rows( SEXP camb , SEXP iamb , SEXP cprod , SEXP
      iprod , SEXP rvar , SEXP ndata , SEXP ncur , SEXP nend ,
      SEXP rlims , SEXP nranges , SEXP arows , SEXP irows , SEXP
       mvec , SEXP mvar , SEXP nrows , SEXP background , SEXP
      remoterx )
165 {
166
     const Rcomplex * restrict amb = COMPLEX(camb);
167
     const int * restrict amb_i = LOGICAL(iamb);
168
     const Rcomplex * restrict prod = COMPLEX(cprod);
169
     const int * restrict prod_i = LOGICAL(iprod);
     const double * restrict var =
                                    REAL(rvar);
170
171
     int n_cur = *INTEGER(ncur);
172
     int n_end = *INTEGER(nend);
173
     const int * restrict r_lims = INTEGER(rlims);
174
     const int n_ranges = *INTEGER(nranges);
175
     const int n_data = *INTEGER(ndata);
176
     const int bg = *LOGICAL(background);
177
     const int remrx = *LOGICAL(remoterx);
178
     Rcomplex * restrict a_rows = COMPLEX(arows);
```

```
179
     int * restrict i_rows = LOGICAL(irows);
180
     Rcomplex * restrict m_vec = COMPLEX(mvec);
181
     double * restrict m_var = REAL(mvar);
182
     SEXP success;
183
     int * restrict i_success;
184
     int n_rows;
185
     R_len_t k;
186
     R_len_t n_start;
187
     R_len_t i;
188
     R_len_t j;
     R_len_t subi;
189
190
     R_len_t addi;
191
     R_len_t gati;
192
     int r_min;
193
     int r_lim;
194
     int r_max;
195
     int r_cur;
196
197
198
     // Check that n_end <= n_data</pre>
199
     n_{end} = (n_{data} > n_{end} ? n_{end} : n_{data});
200
201
     // Check that n_cur <= n_data</pre>
202
     n_cur = ( n_data > n_cur ? n_cur : n_data );
203
204
     // Success output
205
     PROTECT( success = allocVector( LGLSXP , 1 ) );
206
207
     // Local pointer to the success output
208
     i_success = LOGICAL( success );
209
210
     // Set the success output
211
     *i_success = 1;
212
213
     // The lowest range gate limnit - 1
214
     r_min = r_lims[0] - 2;
215
216
     // Samples with non-zero range ambiguity
217
     // function at heights below r_lim
     // will not be used in the theory matrix
218
     // Initialize r\_min for monostatic reception
219
220
     r_lim = r_min;
221
     // -1 (all samples accpected) for remote reception
222
     if ( remrx ) r_{lim} = -1;
223
224
     // The highest range gate limit
225
     r_max = r_lims[n_ranges] + 1;
226
227
     // Make the first theory row.
```

```
228
     n_start = n_cur;
229
     // If we are too close to start of data
230
     // skip points as necessary
     if( n_start < r_lims[ n_ranges ] ) n_start = r_lims[</pre>
231
         n_ranges ];
232
233
     // Make sure that we did not yet pass the end point
234
     if( n_start < n_end ){</pre>
235
       // Go through all range-gates
       for( i = 0 ; i < n_ranges ; ++i ){
236
          // Initialize the theory matrix to zero
237
          a_rows[i].r = .0;
238
          a_rows[i].i = .0;
239
240
          i_rows[i] = 0;
241
242
          // Add contribution from all ranges
243
          // integrated to this gate
244
          for( j = r_lims[i] ; j < r_lims[ i + 1 ] ; ++j ){
245
246
            // In amb_i == 0 points there might be erroneous
247
       // values from previously calculated lags,
248
            // it is thus extremely important to check
249
       // amb_i before addition / subtraction!
250
            if(amb_i[ n_start - j ]){
251
              a_rows[i].r += amb[ n_start - j ].r;
              a_rows[i].i += amb[ n_start - j ].i;
252
253
              i_rows[i] += amb_i[ n_start - j ];
254
255
         }
       }
256
257
       // The last gate will be 1 or 0, depending on whether
258
259
       // the background ACF will be suppressed or not.
260
       a_rows[n_ranges].r = (bg == 0 ? 0.0 : 1.0);
261
       a_rows[n_ranges].i = 0.0;
262
       i_rows[ n_ranges ]
                           = ( bg == 0 ? 0 : 1 );
263
264
     // If the first row could not be formed
265
     // set success to false and return
266
     }else{
267
       *i_success = 0;
268
269
270
     // From this point on all possible theory rows will be
271
     // formed but only those with indprod set are stored,
272
     // others are immediately overwritten
273
     // Number of stored rows
274
275
     n_rows = 0;
```

```
276
277
     // Range from the latest pulse
278
     r_cur = r_max;
     for(k = (n_start - r_min); k < n_start; ++k){
279
280
       if(k >= 0){
281
          if(amb_i[k]){
282
            r_cur = 0;
283
          }else{
284
            ++r_cur;
285
       }
286
287
288
289
     // Use all data points from n_start to n_end
290
     for( k = n_start ; k < n_end ; ++k ){
291
292
       // If this data point will be used (!=0 for clarity,
293
       // the prod_i vector may contains values larger than 1)
       if( (prod_i[k] != 0) & (r_cur > r_lim) & (r_cur < r_max))
294
           {
295
296
          // Copy data to the measurement vector
297
          m_vec[n_rows].r = prod[k].r;
298
         m_vec[n_rows].i = prod[k].i;
299
                          = var[k];
         m_var[n_rows]
300
301
          // Copy the current theory vectors to the next one.
302
          for( i = 0 ; i < ( n_ranges + 1 ) ; ++i ){
303
            i_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ]
               i_rows[ n_rows * ( n_ranges + 1 ) + i ];
304
            // Set the theory rows exactly to zero at points
305
       // where the index vector is zero. This makes
306
       // identification of blind ranges much easier.
            if(i_rows[ n_rows * ( n_ranges + 1 ) + i ]==0){
307
308
              a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].r =
                  0.0;
              a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].i =
309
                  0.0;
310
              a_rows[ n_rows * ( n_ranges + 1 ) + i ].r = 0.0;
311
              a_rows[ n_rows * ( n_ranges + 1 ) + i ].i = 0.0;
312
            // Otherwise copy the theory matrix row
313
            }else{
314
              a_rows[(n_rows + 1) * (n_ranges + 1) + i].r =
                  a_rows[ n_rows * ( n_ranges + 1 ) + i ].r;
              a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].i =
315
                  a_rows[ n_rows * ( n_ranges + 1 ) + i ].i;
316
           }
         }
317
318
```

```
319
          // Increment the theory row counter
320
          ++n_rows;
321
       }
322
323
324
       // Now form the next theory row using the previous
325
        // one and the range limit indices
326
        for( i = 0 ; i < n_ranges ; ++i ){
327
          // Index in the theory matrix
328
          // (that is stored as a vector)
          gati = n_rows * ( n_ranges + 1 ) + i;
329
          // Index of the data point that
330
331
          // will be added to this gate
332
          addi = k - r_lims[i] + 1;
333
          // Index of the data point that
          // will be subtracted from this gate
334
335
          subi = k - r_lims[i+1] + 1;
336
          // Do additions / subtractions only if the point
337
338
          // contains a non-zero ambiguity value
          if( amb_i[ addi ] ){
339
340
            a_rows[ gati ].r += amb[ addi ].r;
341
            a_rows[ gati ].i += amb[ addi ].i;
342
            i_rows[ gati ]
                            += amb_i[ addi ];
343
344
          if( amb_i[ subi ] ){
345
            a_rows[ gati ].r -= amb[ subi ].r;
346
            a_rows[ gati ].i -= amb[ subi ].i;
347
            i_rows[ gati ] -= amb_i[ subi ];
348
349
350
351
352
       // Count samples to exclude everything that contains
        // echoes from below the first gate
353
354
        if( amb_i[ k ] ){
355
          r_cur = 0;
356
       }else{
357
         ++r_cur;
358
359
360
     }
361
362
     // Write the row count to the output variable
363
     *( INTEGER( nrows ) ) = n_rows;
364
     // Update the current position in the data vector
365
     *( INTEGER( ncur ) ) = n_end;
366
367
```

```
368 UNPROTECT(1);
369
370 return(success);
371
372 }
```

5.5.17 range_ambiguity.c

```
1 // file:range_ambiguity.c
2 // (c) 2010- University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
6 #include "LPI.h"
7
8
9
    Range ambiguity function with linear
10
    interpolation of TX data
11
12
    Arguments:
13
     cdata1
             First complex transmitter samples
             Second complex transmitter samples
14
     cdata2
     idata1 First transmitter sample indices
15
             Seconds transmitter sample indices
16
     idata2
     cdatap Complex range ambiguity function
17
18
     idatap Range ambiguity index vector
19
     ndata1 Length of vectors cdata1 and idata1
     ndata2 Length of vectors cdata2 and idata2
20
21
     lag
             Lag
22
23
    Returns:
24
     success 1 if all processing was successful, 0 otherwise
25
26 */
27
28 SEXP range_ambiguity( SEXP cdata1 , SEXP cdata2 , SEXP idata1
       , SEXP idata2 , SEXP cdatap , SEXP idatap , SEXP ndata1 ,
       SEXP ndata2 , SEXP lag )
29 {
30
    Rcomplex *cd1 = COMPLEX(cdata1);
    Rcomplex *cd2 = COMPLEX(cdata2);
31
32
    int *id1 = LOGICAL(idata1);
33
    int *id2 = LOGICAL(idata2);
34
    Rcomplex *cdp = COMPLEX(cdatap);
    int *idp = LOGICAL(idatap);
35
36
    int nd1 = *INTEGER(ndata1);
37
    int nd2 = *INTEGER(ndata2);
38
    int 1 = *INTEGER(lag);
    SEXP success;
39
40
    int *isuccess;
    int k = 0;
41
42
    int npr;
43
    int ninterp = AMB_N_INTERP;
44
    int i;
45
    double * tmpr1;
```

```
double * tmpi1;
46
47
    double * tmpr2;
    double * tmpi2;
48
49
50
    // Allocate temporary vectors for interpolated data
51
    tmpr1 = (double*) R_Calloc( 2*ninterp , double );
52
    tmpi1 = (double*) R_Calloc( 2*ninterp , double );
53
    tmpr2 = (double*) R_Calloc( 2*ninterp , double );
    tmpi2 = (double*) R_Calloc( 2*ninterp , double );
54
55
56
    // Output data length will be minimum of the
57
    // two input data lengths, minus the lag
58
    npr = nd1 - 1;
59
    if ( nd1 > nd2 ) npr = nd2 - 1;
60
    // Allocate the success return value
61
62
    PROTECT( success = allocVector( LGLSXP , 1 ) );
63
    // A local pointer to the success value
64
65
    isuccess = LOGICAL( success );
66
    *isuccess = 1;
67
68
    // The actual lagged product calculation
69
    for(k = 0; k < npr; ++k){
70
      // The index vector
71
      idp[k] = (id1[k] * id2[k+1]);
72
      // Multiply data values only if the index vector was set
73
      if(idp[k]){
74
        // Initialize the temporary vectors to zero
        for( i = 0; i < (2 * ninterp ); ++i){}
75
76
      tmpr1[i] = .0;
77
      tmpi1[i] = .0;
78
      tmpr2[i] = .0;
79
      tmpi2[i] = .0;
80
        // Linear interpolation towards the previous data point
81
82
        if(k > 1){
83
      for( i = 0 ; i < ninterp ; ++i ){</pre>
84
        tmpr1[i] = cd1[k-1].r + (cd1[k].r - cd1[k-1].r) * (
            1. - (double)i / (double)( 2 * ninterp ) );
        tmpi1[i] = cd1[k-1].i + (cd1[k].i - cd1[k-1].i) * (
85
            1. - (double)i / (double)( 2 * ninterp ) );
        tmpr2[i] = cd2[k-1+1].r + (cd2[k+1].r - cd2[k-1+1].r)
86
             * ( 1. - (double)i / (double)( 2 * ninterp ) );
        tmpi2[i] = cd2[k-1+1].i + (cd2[k+1].i - cd2[k-1+1].i)
87
             * ( 1. - (double)i / (double)( 2 * ninterp ) );
88
89
90
        // Linear interpolation towards the next data point
```

```
91
          if( k < npr ){
            for( i = 0 ; i < ninterp ; ++i ){</pre>
 92
 93
              tmpr1[i+ninterp] = cd1[k].r + (cd1[k+1].r - cd1[k])
                 ].r ) * ( (double)i / (double)( 2 * ninterp ) );
              tmpi1[i+ninterp] = cd1[k].i + (cd1[k+1].i - cd1[k])
 94
                 ].i ) * ( (double)i / (double)( 2 * ninterp ) );
              tmpr2[i+ninterp] = cd2[k+1].r + (cd2[k+1+1].r -
 95
                 cd2[k+1].r ) * ( (double)i / (double)( 2 *
                 ninterp ) );
              tmpi2[i+ninterp] = cd2[k+1].i + ( cd2[k+1+1].i -
 96
                 cd2[k+1].i ) * ( (double)i / (double)( 2 *
                 ninterp ) );
 97
            }
 98
          }
99
          // Initialize the final data value to zero
100
          cdp[k].r = .0;
          cdp[k].i = .0;
101
102
          // Add products of the interpolated data
          for( i = 0 ; i < ( 2 * ninterp ) ; ++i ){
103
104
            cdp[k].r += tmpr1[i] * tmpr2[i] + tmpi1[i] * tmpi2[i
105
            cdp[k].i += tmpr1[i] * tmpi2[i] - tmpi1[i] * tmpr2[i
               ];
106
107
          // Divide with number of summed values
108
          cdp[k].r /= (double)(2*ninterp);
109
          cdp[k].i /= (double)(2*ninterp);
110
     }
111
112
113
     // Set 1 index values from the beginning to false
114
     for(k = 0; k < 1; ++k){
115
        idp[npr+k] = 0;
116
117
118
     // Free the temporary vectors
119
     Free(tmpr1);
120
     Free(tmpi1);
121
     Free(tmpr2);
122
     Free(tmpi2);
123
124
     UNPROTECT(1);
125
126
     return(success);
127
128 }
```