

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	Introduction	5
1.1	Purpose of this document	5
1.2	Lag Profile Inversion	5
1.3	Installation	6
1.4	Help	6
2	Lag profile inversion	7
2.1	Transmitter and receiver signals	7
2.2	Scattering from a target	8
2.3	Target covariance functions	9
2.4	Lag profiles	9
2.5	Range ambiguity functions	10
2.6	Radar measurement as a linear inverse problem	10
2.7	Additional analysis steps	11
2.7.1	Ground clutter suppression	11
2.7.2	Voltage level decoding	12
3	LPI implementation	13
3.1	Resampling and filtering	13
3.2	Ground clutter suppression	13
3.3	Voltage level decoding filters	14
3.4	Data correlation	14
3.5	Theory matrix	15
3.6	Lag profile inversion solvers	15
3.7	Input and output control	16
4	LPI in practice	17
4.1	Examples with simulated data	17
4.1.1	A coherent point target	18
4.1.2	Ground clutter suppression	22

4.2	Examples with real data	27
4.2.1	Autocovariance function measurement with a monos- tatic radar	27
4.2.2	Bistatic measurements and crosscovariannce functions .	29
5	Documented source code	31
5.1	Process control	31
5.1.1	LPI	31
5.1.2	LPIexpand.input	37
5.1.3	LPIinitCluster.R	40
5.1.4	LPIinitRemoteNode.R	45
5.1.5	LPIinitComputingSlaves.R	47
5.1.6	currentTimes.R	48
5.1.7	nextIntegrationPeriods.R	49
5.1.8	LPIsolve.acf.R	51
5.1.9	noUpdate.R	53
5.1.10	LPIrun.remote.R	54
5.1.11	LPIrun.R	55
5.1.12	initLPIenv.R	59
5.1.13	LPIsolve.R	61
5.2	Signal pre- and post-processing	66
5.2.1	prepareLPIdata.R	66
5.2.2	clutterSuppress.R	77
5.2.3	decoFilter.R	80
5.2.4	LPIaveragePower.R	83
5.2.5	LPIsaveACF.R	84
5.2.6	stripACF.R	87
5.3	Correlation and inverse problem formulation	89
5.3.1	laggedProducts.R	89
5.3.2	lagprodVar.R	90
5.3.3	rangeAmbiguity.R	91
5.3.4	averageProfiles.R	93
5.3.5	theoryRows.R	94
5.4	Inverse problem solvers	96
5.4.1	fishs.init.R	96
5.4.2	fishs.add.R	97
5.4.3	fishs.solve.R	98
5.4.4	deco.init.R	100
5.4.5	deco.add.R	101
5.4.6	deco.solve.R	102
5.4.7	dummy.init.R	104

5.4.8	dummy.add.R	106
5.4.9	dummy.solve.R	107
5.4.10	ffts.init.R	109
5.4.11	ffts.add.R	111
5.4.12	ffts.solve.R	114
5.4.13	rlips.solve2.R	116
5.5	C functions 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	132
5.5.7	fishs_add.c	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 Package 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.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 $\text{env}(t)$. Because the carrier signal 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 $\text{env}(t)$.

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

$$z^r(t) = (s * p)(t). \quad (2.1)$$

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

$$z^t(t) = (\text{env} * p)(t). \quad (2.2)$$

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

$$z_i^t = z^t(t_i) \quad (2.3)$$

$$z_i^r = z^r(t_i) \quad (2.4)$$

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 consequences 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

$$s(t) = \xi \text{env}(t - S) \quad (2.5)$$

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

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

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 target 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

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

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 separated 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

¹Monostatic incoherent scatter radars typically have duty-cycles from 5 to 25 %

write the received signal as

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

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

$$\rho_i(S) = \sigma(S, \tau_i) \quad (2.9)$$

2.5 Range ambiguity functions

Expectation value of the product

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

can be written as

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

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

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

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

$$m_{i,j} = \int_{S_1}^{S_2} W(t_i, t_j, S) \sigma(S, t_i - t_j) dS + \varepsilon(t_i, t_j) \quad (2.13)$$

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

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

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

$$\mathbf{m}_l = \mathbf{W}_l \boldsymbol{\rho}_l + \boldsymbol{\varepsilon}_l \quad (2.15)$$

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

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

$$\mathbf{Q}_l = \mathbf{W}_l^H \boldsymbol{\Sigma}_l^{-1} \mathbf{W}_l \quad (2.17)$$

where $\boldsymbol{\Sigma}_l$ 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 matched 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 alias 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 severe 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.

¹<http://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 independently 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 available 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 , ... ){
+
+   srate <- 1e4
+
+   # First pre-allocate the output list
+   outlist <- list( TX1=list() , TX2=list() , RX1=list() ,
+                   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 )
+
+   # Let us use 1 ms pulses at random positions with 25 % duty-cycle
+
+   # Pulse lengths counted as data samples
+   plen <- floor( 1e-3 * srate )
+
+   # Number of pulses in the whole data vectors
+   np <- round( nd * .25 / plen )
+
+   # Let us generate random pulse positions for TX1
+   pstarts <- floor(runif( np ) * ( nd - plen - 1 ) ) + 1
+
+   # Allocate the data and index vectors for TX1
+   outlist[['TX1']][['cdata']] <- complex( nd , real=0 , imaginary=0 )
+
+   # 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)
+
+   # TX2 is identical with TX1
+   outlist[['TX2']] <- outlist[['TX1']]
+
+   # Our coherent target is assumed to be at 200 km range,
+   # convert to sample intervals
+   rtarg <- floor( 200e3 / 2.99792458e8 * 2 * srates )
+
+   # 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))*0.3
+
+   # Receiver index vector is negation of the transmitter index vector
+   outlist[['RX1']]['idata'] <- !outlist[['TX1']]['idata']
+
+   # Add the ndata element
+   outlist[['RX1']]['ndata'] <- as.integer(nd)
+
+   # RX2 is identical with RX1
+   outlist[['RX2']] <- outlist[['RX1']]
+
+   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)
+ }

```

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

```

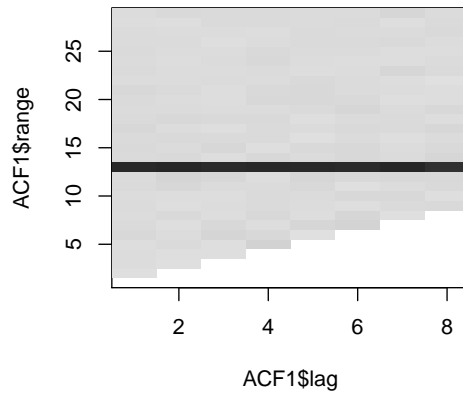


Figure 4.1: LPI detection of a simulated coherent point target.

```

        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 , ... ){
+
+   srate <- 1e4
+
+   # First pre-allocate the output list
+   outlist <- list( TX1=list() , TX2=list() , RX1=list() ,
+                   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 )
+
+   # Let us use 1 ms pulses at random positions with 25 % duty-cycle
+
+   # Pulse lengths counted as data samples
+   plen <- floor( 1e-3 * srate )
+
+   # Number of pulses in the whole data vectors
+   np <- round( nd * .25 / plen )
+
+   # Let us generate random pulse positions for TX1
+   pstarts <- floor(runif( np ) * ( nd - plen - 1 ) ) + 1
+
+   # Allocate the data and index vectors for TX1
+   outlist[['TX1']][['cdata']] <- complex( nd , real=0 , imaginary=0 )
+
+   # 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)
+
+ # TX2 is identical with TX1
+ outlist[['TX2']] <- outlist[['TX1']]
+
+ # 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))*0.5
+
+ # Receiver index vector is the negation of the transmitter index vector
+ outlist[['RX1']]['idata'] <- !outlist[['TX1']]['idata']
+
+ # Add the ndata element
+ outlist[['RX1']]['ndata'] <- as.integer(nd)
+
+ # RX2 is identical with RX1
+ outlist[['RX2']] <- outlist[['RX1']]

```



```

+
+   return(outlist)
+
+ }
+
>

```

We have now everything needed for the simulation run, let us call LPI, first without clutter suppression

```

> 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
+   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
+   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: 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: RX1:20 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))

```

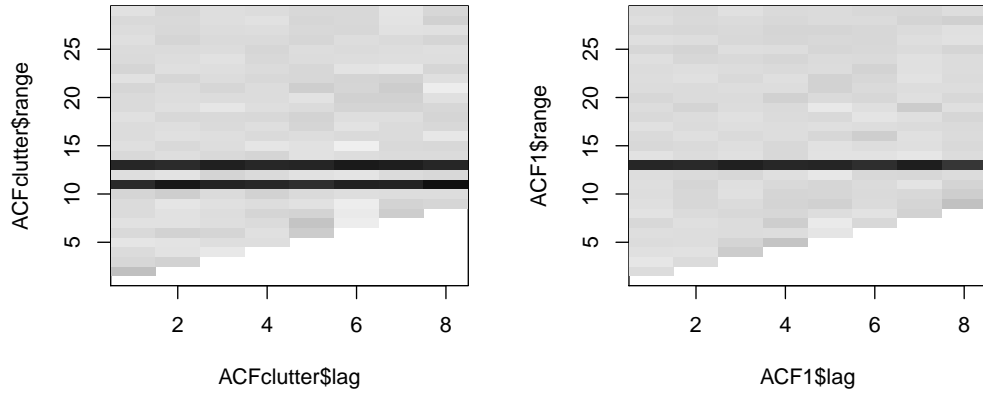


Figure 4.2: LPI detection of two simulated coherent point targets. The upper one has a small doppler shift whereas the lower one has zero doppler. When analysed without clutter suppression (left) both targets are detected. When the clutter suppression is applied (right) the lower one becomes subtracted 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 sampling times we can select arbitrary values for 'startTime', 'stopTime', and 'timeRes.s'.

```

> LPI(
+   startTime = 1356998400,
+   stopTime = 1356998401,
+   lagLimits = seq( 15 ) ,           # all intra-pulse lags
+   timeRes.s = 1 ,                   # 10 s integration time
+   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.

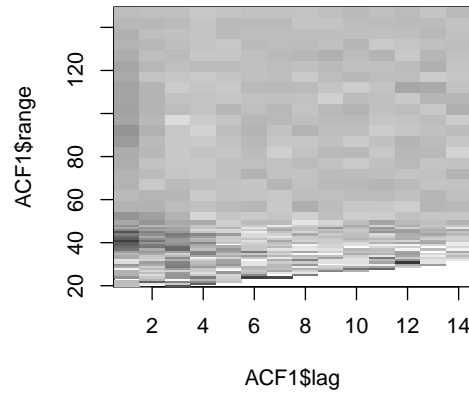


Figure 4.3: LPI analysis of one second of data from an EISCAT UHF beata experiment March 13 2013 22:02:36 UT. The results is very noisy because a very short period of data was used, but a clear E region is visible around range gate 40 and F region around range gate 100.

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,
13                 inputPackages=c(),
14                 startTime = 0, # 1st Jan 1970 00:00 UT
15                 stopTime = 4000000000, # 2nd Oct 2096 07:00
16                               UT
17                 nup = LPIexpand.input( 1 ),
18                 filterLength = LPIexpand.input( 1 ),
19                 decodingFilter = "none",
20                 lagLimits = c(1,2),
21                 rangeLimits = c(1,2),
22                 maxRanges = Inf,
23                 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)) )
32     ,
33     solver = "fishs",
34     nBuf = 10000,
35     fullCovar = FALSE,
36     rlips.options = list( type="c" , nbuf=1000 ,
37                           workgroup.size=128),
38     remoteRX = FALSE,
39     normTX = FALSE,
40     nCode = NA,
41     ambInterp = FALSE,
42     resultDir = paste(format(Sys.time(), "%Y-%m-%d
43                        _%H:%M"), 'LP', sep='_'),
44     dataEndTimeFunction="currentTimes",
45     resultSaveFunction = "LPIsaveACF",
46     paramUpdateFunction="noUpdate",
47     ...
48 )
49 {
50     # Collect all input in a list that is handy to pass
51     forwards
52     par1 <- formals()
53     par1['...'] <- NULL
54     par2 <- list(...)
55     par1names <- names(par1)
56     par1 <- lapply( names( par1 ) , FUN=function(x){ eval( as
57         .name( x ) ) } )
58     names(par1) <- par1names
59     LPIparam <- c(par1,par2)
60
61     # Expand parameters to LPI internal format and set
62     storage modes as necessary
63     LPIparam[["nup"]] <- LPIexpand.input( LPIparam[["nup"]] )
64     storage.mode( LPIparam[["nup"]] ) <- "integer"
65     LPIparam[["filterLength"]] <- LPIexpand.input( LPIparam[[
66         "filterLength"]] )
67     storage.mode( LPIparam[["filterLength"]] ) <- "integer"
68     storage.mode( LPIparam[["lagLimits"]] ) <- "integer"
69     storage.mode( LPIparam[["rangeLimits"]] ) <- "integer"

```

```

64 LPIparam[["maxClutterRange"]] <- LPIexpand.input(
    LPIparam[["maxClutterRange"]] )
65 storage.mode( LPIparam[["maxClutterRange"]] ) <- "integer"
66 LPIparam[["clutterFraction"]] <- LPIexpand.input(
    LPIparam[["clutterFraction"]] )
67 LPIparam[["freqOffset"]] <- LPIexpand.input( LPIparam[["
    freqOffset"]] )
68 if( ! is.list( LPIparam[["indexShifts"]] ) ){
69     LPIparam[["indexShifts"]] <- list(LPIparam[["
        indexShifts"]])
70 }
71 LPIparam[["indexShifts"]] <- LPIexpand.input( LPIparam[["
    indexShifts"]] )
72 for( dType in c("TX1","TX2","RX1","RX2")) storage.mode(
    LPIparam[["indexShifts"]][[dType]]) <- "integer"
73 storage.mode( LPIparam[["nCode"]] ) <- "integer"
74
75
76 # Print input arguments
77 cat(sprintf("%20s %f (%s UT)\n","startTime:",startTime,
    format(as.POSIXlt(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.POSIXlt(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')
82 cat(sprintf("%20s %s\n","dataInputFunction:",
    dataInputFunction))
83 cat(sprintf("%20s %s\n","dataEndTimeFunction:",
    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 }
90 cat(sprintf("%20s","nup:"));for(dType in c("RX1","RX2","
    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')}
92 cat(sprintf("%20s %s\n","decodingFilter:",decodingFilter
    [1]))

```

```

93   cat(lagLimits,fill=70,labels=c(sprintf("%20s","lagLimits:
    "),rep(' ',1000)))
94   cat(rangeLimits,fill=70,labels=c(sprintf("%20s","
    rangeLimits: "),rep(' ',1000)))
95   cat(maxRanges,fill=70,labels=c(sprintf("%20s","maxRanges:
    "),rep(' ',1000)))
96   cat(sprintf("%20s %f\n","timeRes.s:",timeRes.s))
97   cat(sprintf("%20s RX1:%i RX2:%i \n","maxClutterRange:",
    LPIparam$maxClutterRange["RX1"],LPIparam$
    maxClutterRange["RX2"]))
98   cat(sprintf("%20s RX1:%i RX2:%i \n","clutterFraction:",
    LPIparam$clutterFraction["RX1"],LPIparam$
    clutterFraction["RX2"]))
99   cat(sprintf("%20s %s\n","backgroundEstimate:",
    backgroundEstimate))
100  cat(sprintf("%20s %f\n","maxWait.s:",maxWait.s))
101  cat(sprintf("%20s RX1:%f RX2:%f TX1:%f TX2:%f\n","
    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))
104  cat(sprintf("%20s %i\n","nBuf:",nBuf))
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))
108  cat(sprintf("%20s %s\n","normTX:",normTX))
109  cat(sprintf("%20s %i\n","nCode:",nCode))
110  cat(sprintf("%20s %s\n","ambInterp:",ambInterp))
111  cat(sprintf("%20s %s\n","resultDir:",resultDir))
112  cat(sprintf("%20s %s\n","resultSaveFunction:",
    resultSaveFunction))
113  cat(sprintf("%20s %s\n","paramUpdateFunction:",
    paramUpdateFunction))
114  cat(sprintf("%20s %s\n","useXDR:",useXDR))
115
116  # Total number of integration periods requested
117  LPIparam[["lastIntPeriod"]] <- round( ( stopTime -
    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"]] ,
129     useXDR=useXDR )
129 ncl <- length( ctrlcl )
130
131 # A flag telling whether the analysis will be run in some
132 # kind of cluster configuration or in a single process
133 LPIparam[["iscluster"]] <- !all(is.na(ctrlcl))
134
135 # Initialize a list for unsolved integration periods
136 intPer.missing <- seq( LPIparam[["lastIntPeriod"]] )
137
138 # Run analysis loop until end of data
139 endOfData <- FALSE
140 repeat{
141
142     # Update the last available data samples
143     LPIparam[["dataEndTimes"]] <- eval( as.name( LPIparam[["
144         dataEndTimeFunction"]] ))( LPIparam )
145
146     # Latest integration period for which data is available
147     LPIparam[["maxIntPeriod"]] <- floor( ( min(unlist(
148         LPIparam[["dataEndTimes"]])) - LPIparam[["startTime"
149         ]] ) / LPIparam[["timeRes.s"]] )
150
151     # Select integration period numbers for the next
152     # analysis run
153     # Latest periods will be analysed first in order to
154     # simplify real-time analysis
155     waitSum <- 0
156     while( is.null( intPer.current <-
157         nextIntegrationPeriods( LPIparam , nodeMultip*ncl ,
158         intPer.missing ))) {
159
160         # Break the loop after waiting
161         # long enough for new data
162         if( waitSum > LPIparam[["maxWait.s"]] ){
163             endOfData <- TRUE
164             break
165         }
166
167         # Wait 10 seconds
168         Sys.sleep(10)
169
170         # Increment the wait time counter

```

```

164         waitSum <- waitSum + 10
165
166         # Update the last available data samples
167         LPIparam[["dataEndTimes"]] <- eval( as.name( LPIparam
168             [["dataEndTimeFunction"]] ))( LPIparam )
169
170         # Latest integration period for which data is
171         # available
172         LPIparam[["maxIntPeriod"]] <- floor( ( min(unlist(
173             LPIparam[["dataEndTimes"]])) - LPIparam[["
174             startTime"]] ) / LPIparam[["timeRes.s"]] )
175
176     }
177
178     if( endOfData ) break
179
180     # Run analysis on each parallel node, or locally if
181     # LPIparam[["iscluster"]]==FALSE
182     if( LPIparam[["iscluster"]] ){
183         clusterApplyLB( ctrlcl , intPer.current , fun=LPI:::
184             LPIsolve.acf , LPIparam )
185     }else{
186         for( iper in intPer.current ){
187             LPI:::LPIsolve.acf( iper , LPIparam )
188         }
189     }
190
191     # Print something to show that the analysis is running
192     for( k in seq(length(intPer.current))) cat('.'.')
193
194     # Remove the solved periods from the list of missing
195     # ones
196     intPer.missing <- setdiff( intPer.missing , intPer.
197         current )
198
199     # Stop if all integration periods are solved
200     if( length(intPer.missing)==0) break
201
202 } # repeat
203
204 # Shut down the cluster at end of analysis
205 if(!all(is.na(LPIparam[["clusterNodes"]])) stopCluster(
206     ctrlcl )
207
208 # This function does not return anything,
209 # results are written to files.
210 invisible()
211
212 }
```

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)
23
24   # If the input does not have names attributes, assume
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
29     outvec <- rep(parvec,length.out=4)
30     # Set names
31     names(outvec) <- c( "RX1" , "RX2" , "TX1" , "TX2" )
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)
40   names(outvec) <- c( "RX1" , "RX2" , "TX1" , "TX2" )
41
42   # First look if any of the internally used
43   # names is used in the input
44   if( any(namevec=="RX1")) outvec[1] <- parvec["RX1"]
45   if( any(namevec=="RX2")) outvec[2] <- parvec["RX2"]
46   if( any(namevec=="TX1")) outvec[3] <- parvec["TX1"]
47   if( any(namevec=="TX2")) outvec[4] <- parvec["TX2"]
```

```

48
49 # If the vector had elements "RX1" , "RX2" , "TX1" ,
50 # and "TX2", return them in correct order
51 if( !any(is.na(outvec))) return(outvec)
52
53 # If there are still missing values,
54 # look for elements "RX" and "TX"
55 if( is.na(outvec[1])){
56     if(any(namevec=="RX")) outvec[1] <- parvec["RX"]
57 }
58 if( is.na(outvec[2])){
59     if(any(namevec=="RX")) outvec[2] <- parvec["RX"]
60 }
61 if( is.na(outvec[3])){
62     if(any(namevec=="TX")) outvec[3] <- parvec["TX"]
63 }
64 if( is.na(outvec[4])){
65     if(any(namevec=="TX")) outvec[4] <- parvec["TX"]
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"]
74 }
75 if( is.na(outvec[2])){
76     if(any(namevec=="TR2")) outvec[2] <- parvec["TR2"]
77 }
78 if( is.na(outvec[3])){
79     if(any(namevec=="TR1")) outvec[3] <- parvec["TR1"]
80 }
81 if( is.na(outvec[4])){
82     if(any(namevec=="TR2")) outvec[4] <- parvec["TR2"]
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
90 parvec <- parvec[ nchar(namevec) == 0 ]
91 if( length(parvec) > 0 ) outvec[is.na(outvec)] <- rep(
    parvec,length.out=sum(is.na(outvec)))
92
93 # If the output is now full, return it
94 if( !any(is.na(outvec))) return(outvec)
95

```

```
96 |     # If still unsuccesfull, stop the whole analysis
97 |     stop("Cannot parse the input vector ",paste(substitute(
    parvec)))
98 | }
```


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 processes 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 ##      significantly 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 ##
75 ##          Start 5 parallel integration periods on
76 ##          localhost, and another 5 on "remotecomputer":
77 ##          nodes=list(5,remotecomputer=1,
78 ##                    remotecomputer=1,remotecomputer=1,
79 ##                    remotecomputer=1,remotecomputer=1)
80 ##
81 ##          Start 1 integration period with five parallel
82 ##          lags in localhost and another similar one on
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 ##           sequentially 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
115     # Check if nodes has an entry "localControl",
116     # if not, use default (TRUE)
117     localControl <- TRUE
118     if(is.list(nodes)){
119         if(is.logical(nodes[["localControl"]])) localControl <-
            nodes[["localControl"]]
120     }
121
122     # Replace negative values with NAs
123     for(k in seq(length(nodes))){
124         if(is.numeric(nodes[[k]])){
125             if(nodes[[k]]<=0) nodes[[k]] <- NA
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
135
136     # Named nodes are left as such, unnamed are
137     # assumed to denote the number of local
138     # parallel integration periods
139     nnames <- names(nodes)
140     if(is.null(nnames)){
141         ncnames <- rep(0,length(nodes))
142     }else{
143         ncnames <- nchar(names(nodes))
144     }

```

```

145 nodes2 <- c( nodes[ncnames>0] , rep( list( localhost=1 )
      , sum( unlist( nodes[ ncnames==0 ] ) ) ) )
146
147 # Remove the localControl entry
148 nodes2[["localControl"]] <- NULL
149
150 # Create the (optional) local control nodes
151 if(localControl){
152   # Create the local control nodes
153   ctrlcl <- makeCluster( length( nodes2 ) , useXDR=useXDR
      )
154
155   # Load packages LPI and parallel to each of the local
      control nodes
156   clusterEvalQ( ctrlcl , library(LPI) )
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     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 )
171
172   # Load packages LPI and parallel to
173   # each of the remote control nodes
174   clusterEvalQ( ctrlcl , library(LPI) )
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
181   # Initialise the computing slaves
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.
5
6 ##
7 ## Establish a socket 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
15 ##             computer and the list constains either
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
26   nodeName <- names(remNode)
27
28   # If nodeName is localhost, do not start the remote
29   # control process but make direct connections
30   # to the slaves instead.
31   if( nodeName == "localhost"){
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 )
39
40   # Load package LPI
41   clusterEvalQ( remcl , library(LPI) )
42
43   # Initialise the computing slaves
44   clusterCall( remcl , LPIinitComputingSlaves , remNode
45               [[1]] , useXDR )
46
47   return(remcl)
```

47 |
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.
5
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 )
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.
5
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:
12 ##     ...   An arbitrary list of arguments is accepted, but
13 ##           none
14 ##           of them will be used.
15 ##
16 ## Returns:
17 ##     curTimes A named vector ("TX1","TX2","RX1","RX2") with
18 ##             the current unix time -5 in each element.
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 ##   n              Number of new periods to search for
13 ##   intPer.ready  A list of solved period indices
14 ##
15 ## Returns:
16 ##   nextIpers     Indices of the integration periods to
17 ##                 be solved next.
18 ##
19
20 nextIntegrationPeriods <- function( LPIparam , n , intPer.
21   missing )
22 {
23
24   # Truly available periods
25   intPer.available <- intPer.missing[ which( intPer.missing
26     <= min( LPIparam[["maxIntPeriod"]] , LPIparam[["
27       lastIntPeriod"]]) ) ]
28
29   # We know that the integration periods are in order,
30   # simply pick the n last ones
31   nper <- length(intPer.available)
32   if(nper==0) return(NULL)
33   return(intPer.available[ max(1,( nper - n + 1 )) : nper
34     ])
35
36   ##
37   # A vector for the integration period numbers
38   nextIpers <- rep(0,n)
39   ##
40   # Counter for identified new periods
41   k <- 0
42   ##
43   # The period from which we will start seeking backwards
44   p <- min( LPIparam[["maxIntPeriod"]] , LPIparam[["
45     lastIntPeriod"]])
46   ##
```

```

43 ##      # If the last data sample or analysis end time is
      before
44 ##      # beginning of analysis, there will be nothing to do
45 ##      if( p < 0 ) return(NULL)
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
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
15 ##               LPIparam[["timeRes.s"]]
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
29   LPIparam <- eval( as.name( LPIparam[["paramUpdateFunction
30     "]] ))( LPIparam , intPeriod )
31
32   if( !is.null(LPIparam)){
33     # Read raw data, name of the data input function
34     # should be stored in a character string
35     LPIdatalist.raw <- eval(as.name(LPIparam[["
36       dataInputFunction"]]))( LPIparam , intPeriod )
37
38     # If data reading was successfull
39     if(LPIdatalist.raw[["success"]]){
40
41       # require that there are at least some TX and RX
42       # samples
43       if( (sum(LPIdatalist.raw[["RX1"]][["idata"]]) > 0)
44         &
45         (sum(LPIdatalist.raw[["RX2"]][["idata"]]) > 0)
46         &
```

```

42         (sum(LPIdatalist.raw[["TX1"]][["idata"]]) > 0)
43         &
44         (sum(LPIdatalist.raw[["TX2"]][["idata"]]) > 0))
45         {
46             # Frequency mixing, filtering, etc., the output
47             # is
48             # collected in a list and stored on the user
49             # workspace
50             LPIdatalist.final <- prepareLPIData( LPIparam ,
51             LPIdatalist.raw )
52
53             # Call the function that will send the data to
54             # proper place and run the actual analysis
55             ACF <- LPI:::LPIdrun.remote( substitute(
56                 LPIdatalist.final) )
57
58             # Store the results
59             eval( as.name( LPIparam[["resultSaveFunction"]])
60                 )( LPIparam , intPeriod , ACF )
61         }
62     }
63 }
64
65     # Return the integration period
66     # number to the main process
67     return(intPeriod)
68 }

```

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.
5 ##
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
18 ##           first call, NULL in second call with the
19 ##           same list
20 ##
21
22 noUpdate <- function( LPIparam , intPeriod )
23 {
24
25     if(is.null(LPIparam[["callN"]])){
26         LPIparam[["callN"]] <- 1
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 ##   ACFlst      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.
34         name )
35
36       # Return the ACF
37       return( ACF[[1]] )
38     }
39   }
40
41   # If the analysis is run in a single process,
42   # or if this is a remote control node,
43   # just run the analysis in this process
44   return( LPI:::LPIrun( LPIenv.name ) )
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.
5 ##
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
29   nlags <- eval(LPIenv.name)[["nLags"]]
30   x <- seq( nlags )
31
32   # Number of range gates
33   ngates <- eval(LPIenv.name)[['nGates']]
34   maxgates <- max(ngates)
35
36   # Are we going to calculate a full covariance matrix?
37   fullcovar <- eval(LPIenv.name)[['fullCovar']]
38
39   # Range-gate centre points
40   r <- eval(LPIenv.name)[['rangeLimits']]
41   rgates <- ( r[1:maxgates] + r[2:(maxgates+1)] -1 ) / 2
42
43   # Lag-gate centre points
44   l <- eval(LPIenv.name)[["lagLimits"]]
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
49 if(!exists('slavecl')) slavecl <- NA
50
51 # If we are running on a cluster
52 if( eval(LPIenv.name)[["iscluster"]] & ( !is.na( slavecl
53   ) ) ){
54
55   # Copy the data to all computing slaves
56   clusterExport( slavecl , paste( LPIenv.name ) )
57
58   # Allocate necessary vectors on each slave
59   clusterCall( slavecl , initLPIenv , LPIenv.name )
60
61   # Run the analysis processes on the slaves
62   ACFlist <- clusterApplyLB( slavecl , x , fun=LPI:::
63     LPIsolve , LPIenv.name=LPIenv.name )
64
65   # If not running on cluster, solve the lag profiles
66   # sequentially. Mimic the output list of cluster
67   # calls in order to simplify further processing.
68 }else{
69
70   # Create a list for the lag profiles
71   ACFlist <- vector(mode='list',length=nlags)
72
73   # Allocate vectors etc.
74   initLPIenv( LPIenv.name )
75
76   # Run the actual analysis sequentially
77   for( k in 1:nlags ){
78     ACFlist[[k]] <- LPI::LPIsolve( lag=x[k] , LPIenv.
79       name=LPIenv.name )
80   }
81
82   # Collect the lag numbers from ACF list
83   lagnums <- x
84   for(k in 1:nlags ){
85     lagnums[k] <- ACFlist[[k]][['lagnum']]
86   }
87
88   # Find correct order for the lag profiles
89   lagorder <- x[order(lagnums)]
90
91   # Order the ACF list
92   ACFlist <- ACFlist[lagorder]
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
98 for( k in 1:nlags){
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     ACFmat[maxgates+1,k] <- ACFlist[[k]][['lagprof']][
      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   VARmat <- matrix(NA,ncol=nlags,nrow=(maxgates+1))
111   COVARmat <- array(NA,dim=c((maxgates+1),(maxgates+1),
      nlags))
112   for( k in 1:nlags){
113     if(ngates[k]>0){
114       # Copy variances
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)]
121       COVARmat[(maxgates+1),(maxgates+1),k] <- ACFlist[[
        k]][['covariance']][(ngates[k]+1),(ngates[k]+1)]
122     }
123   }
124   # If only variances were solved
125 }else{
126   # Allocate a matrix for the variances,
127   # set COVARmat to NULL
128   VARmat <- matrix(NA,ncol=nlags,nrow=(maxgates+1))
129   COVARmat <- NULL
130   for( k in 1:nlags){
131     if( ngates[k] > 0 ){
132       # Copy the variances

```

```

133         VARmat[1:ngates[k],k] <- Re(ACFlist[[k]][['
           covariance']])[1:ngates[k]]
134         VARmat[(maxgates+1),k] <- Re(ACFlist[[k]][['
           covariance']])[ngates[k]+1]
135     }
136 }
137 }
138
139
140 # Collect the results in a list and return it.
141 # A list is used because an environment
142 # is much slower to transfer
143 ACFreturn <- list()
144 ACFreturn[["ACF"]] <- ACFmat
145 ACFreturn[["var"]] <- VARmat
146 ACFreturn[["covariance"]] <- COVARmat
147 ACFreturn[["lag"]] <- lgates
148 ACFreturn[["range"]] <- rgates
149 ACFreturn[["nGates"]] <- ngates
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.
5 ##
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 updated 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,
25   # convert into an environment first)
26   LPIenv <- as.environment( eval( LPIenv.name ) )
27
28   # Allocate vector for the range ambiguity function
29   assign( 'camb' , vector(mode='complex',length=(LPIenv[["
30     nData"]]*LPIenv[["nDecimTX"]])) , LPIenv )
31
32   # Range ambiguity indices
33   assign( 'iamb' , vector(mode='logical',length=(LPIenv[["
34     nData"]]*LPIenv[["nDecimTX"]])) , LPIenv )
35
36   # Laged products
37   assign( 'cprod', vector(mode='complex',length=LPIenv[["
38     nData"]]) , LPIenv )
39
40   # Lagged product indices
41   assign( 'iprod', vector(mode='logical',length=LPIenv[["
42     nData"]]) , LPIenv )
43
44   # Lagged product variances
45   assign( 'var' , vector(mode='numeric',length=LPIenv[["
46     nData"]]) , LPIenv )
47
48 }
```

```

43 # Theory matrix rows, one extra row because
44 # theory_rows needs a temp vector
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
49 assign( 'irows', vector(mode='logical',length=((max(
    LPIenv[["nGates"]])+1)*(LPIenv[["nBuf"]]+1))), LPIenv
    )
50
51 # Measurement vector
52 assign( 'meas' , vector(mode='complex',length=LPIenv[["
    nBuf"]])) , LPIenv )
53
54 # Measurement variances
55 assign( 'mvar' , vector(mode='numeric',length=LPIenv[["
    nBuf"]])) , LPIenv )
56
57 # Buffer row counter
58 assign( 'nrows', as.integer(0)
    , 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.
5 ##
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         Lag number, all fractional lags from
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 ##
23
24 LPIsolve <- function( lag , LPIenv.name )
25 {
26
27     # Get the LPI environment from the global workspace
28     LPIenv <- eval(LPIenv.name)
29
30     # Return immediately if number of gates is <= 0
31     if( LPIenv[["nGates"]][lag] <= 0 ) return(list(lagnum=lag
32     ))
33
34     # If rlisp is used, make sure it has been loaded.
35     # rlips is not required in startup in order to
36     # allow analysis without installing it. Other
37     # solvers are included in the LPI package.
38     # Switch quietly to fishs if rlips is not available.
39     if(LPIenv$solver=="rlips"){
40         require(rlips) -> rres
41         if( !rres ) assign( 'solver' , 'fishs' , LPIenv )
42     }
43
44     # Initialise the inverse problem solver
45     if(LPIenv$solver=="rlips"){
46         solver.env <- rlips.init( ncols = LPIenv$nGates[lag] +
47             1 , nrhs = 1 , type = LPIenv$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 )
48 }else if ( LPIenv[["solver"]]=="deco" ){
49   solver.env <- deco.init( LPIenv[["nGates"]][lag] + 1 )
50 }else if ( LPIenv[["solver"]]=="dummy" ){
51   solver.env <- dummy.init( range( LPIenv[["rangeLimits"
    ]][ 1 : (LPIenv[["nGates"]][lag]+1) ]) ) )
52 }else if ( LPIenv[["solver"]]=="ffts" ){
53   solver.env <- ffts.init( LPIenv[["nGates"]][lag] ,
    LPIenv[["TX1"]][["idata"]][1:LPIenv[["nData"]]])
54 }
55
56 # Copy of LPIenv[["nData"]]
57 ndcpy <- LPIenv[["nData"]]
58
59 # Walk through all fractional time-lags
60 for( l in seq( LPIenv[["lagLimits"]][lag] , ( LPIenv[["
    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 , l )
71
72   # Variances of lagged products
73   lagprodVar( LPIenv , l )
74
75   # Calculate range ambiguity function
76   rangeAmbiguity( LPIenv , l )
77
78   # Optional pre-averaging of lag-profiles
79   if( !is.null( LPIenv[["nCode"]] ) ){
80     if( !is.na( LPIenv[["nCode"]] ) ){
81       if( LPIenv[["nCode"]] > 0 ){
82         averageProfiles( LPIenv , l )
83         nd <- min( LPIenv[["nData"]] , which( diff(
            LPIenv[["TX1"]][["idata"]] ) == 1 )[ LPIenv[["
            nCode"]] + 1 ] )
84         LPIenv[["nData"]] <- ifelse( is.na(nd) , LPIenv[["
            nData"]] , nd )

```

```

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

```



```

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

```

```

166 }else if(LPIenv[["solver"]]=="deco"){
167     deco.solve( e = solver.env )
168 }else if(LPIenv[["solver"]]=="dummy"){
169     dummy.solve( e = solver.env , LPIenv[["rangeLimits"
170         ]][1:(LPIenv[["nGates"]][lag]+1)])
171 }else if( LPIenv[["solver"]]=="ffts"){
172     ffts.solve( e = solver.env , LPIenv[["rangeLimits"
173         ]][1:(LPIenv[["nGates"]][lag]+1)])
174 }
175
176 # Create the return environment
177 lagprof <- new.env()
178
179 # Assign the solution to the new environment
180 assign( "lagprof" , solver.env[["solution"]] , lagprof )
181 assign( "covariance" , solver.env[["covariance"]] ,
182     lagprof )
183 assign( "lagnum" , lag , lagprof )
184
185 # Kill the solver object
186 if(LPIenv$solver=="rlips") rlips.dispose(solver.env)
187
188 # Conversion to list because it is faster to transfer
189 return(as.list(lagprof))
190 }

```

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
25   dTypes <- c( "RX1" , "RX2" , "TX1" , "TX2" )
26
27   # An empty list for the output data
28   LPIdatalist.final <- vector(mode="list",length=4)
29   names(LPIdatalist.final) <- dTypes
30
31
32   # A list for TX1 pulse start positions in all data
33   # vectors (these will be different if sample rates
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
38   # be passed to c-routines as such, and 0 is thus the
39   # first index.
40   pulseStarts <- list( TX1 = c(0) , TX2 = c(0) , RX1 = c(0)
41                       , RX2 = c(0) )
42
43   # A list for first sample to use in decimation
44   # in each data vector
```

```

44 firstSample <- 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)
49 pulseStarts[["TX1"]] <- which( diff( LPIdatalist.raw[["
    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[["
        TX1"]]) / LPiParam[["filterLength"]][["TX1"]] *
        LPiParam[["nup"]][["TX1"]] * LPiParam[["filterLength"
        "]][[XXN]] / LPiParam[["nup"]][["XXN"]] )
55     firstSample[[XXN]] <- pulseStarts[[XXN]][1]
56 }
57
58 # The below fix does not work if 'nup' are not common for
59 # all data vectors.
60 # Disable in this case.
61 if(all(LPIparam[["nup"]]==LPIparam[["nup"]][["TX1"]])){
62     # Strip off samples to make each
63     # IPP a multiple of filter length
64     for( XXN in dTypes ){
65         # New pulse start positions that
66         # are even multiples of the filter length
67         pstarts2 <- pulseStarts[[XXN]] - round( ( pulseStarts
            [[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
75             ntx <- length(ncut)
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
81             # k-1 from the original ncut[k]. Then take
82             # modulus to make sure that no points will be
83             # cut unless really necessary and that number

```

```

84         # of points to cut is not negative
85         ncut[2:ntx] <- ncut[2:ntx] - ncut[1:(ntx-1)]
86         ncut <- ncut %% round( LPIparam[["filterLength"]][[
            XXN]] / LPIparam[["nup"]][[XXN]] )
87         ind <- rep( TRUE , LPIdatalist.raw[[XXN]][["ndata"
            ]])
88         for( k in seq(length(pstarts2)) ){
89             if( ( ncut[k] > 0 ) & (pulseStarts[[XXN]][k]<
                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
93         LPIdatalist.raw[[XXN]][["ndata"]] <- min(
            LPIdatalist.raw[[XXN]][["ndata"]] , sum(ind) )
94         LPIdatalist.raw[[XXN]][["cdata"]] <- LPIdatalist.
            raw[[XXN]][["cdata"]][ind][1:LPIdatalist.raw[[
            XXN]][["ndata"]]]
95         LPIdatalist.raw[[XXN]][["idata"]] <- LPIdatalist.
            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.
103 # Take this into account in firstSamples.
104 # Again keep 0 as the first index, because
105 # the indices will be passed to c-routines as such
106 firstSample[["TX1"]] <- firstSample[["TX1"]] + LPIparam[[
    "indexShifts"]][["TX1"]][1]
107 # while( firstSample[["TX1"]] < 0 ){
108 #     firstSample[["TX1"]] <- firstSample[["TX1"]] -
LPIparam[["filterLength"]][["TX1"]] / LPIparam[["nup"]][["
TX1"]]
109 while( firstSample[["TX1"]] < 0 ){
110     firstSample[["TX1"]] <- firstSample[["TX1"]] +
        LPIparam[["filterLength"]][["TX1"]] / LPIparam[["nup
        "]]["TX1"]]
111 }
112
113 firstFraction <- c( TX1 = 0 , TX2 = 0 , RX1 = 0 , RX2 = 0
    )
114 for( XXN in dTypes ){
115     firstSampleF <- firstSample[["TX1"]] * LPIparam[["
        filterLength"]][[XXN]] / LPIparam[["filterLength"

```

```

    ]][["TX1"]] / LPIparam[["nup"]][[XXN]] * LPIparam[["
    nup"]][["TX1"]]
116 firstSample[XXN] <- round( firstSampleF )
117 firstFraction[XXN] <- round( ( firstSample[XXN] -
    firstSampleF ) * LPIparam[["nup"]][[XXN]] )
118 }
119
120
121 # Conversions to integer mode
122 storage.mode( firstSample ) <- "integer"
123 storage.mode( LPIparam[["filterLength"]] ) <- "integer"
124 storage.mode( firstFraction ) <- "integer"
125
126 # Index corrections, frequency mixing,
127 # and filtering in C routines
128 for( XXN in dTypes ){
129
130     storage.mode( LPIparam[["indexShifts"]][[XXN]] ) <- "
        integer"
131
132     LPIdatalist.final[XXN] <-
133         .Call( "prepare_data"
134             , LPIdatalist.raw[XXN][["cdata"]]
135             , LPIdatalist.raw[XXN][["idata"]]
136             , LPIdatalist.raw[XXN][["ndata"]]
137             , LPIparam[["freqOffset"]][XXN]
138             , LPIparam[["indexShifts"]][[XXN]]
139             , LPIparam[["nup"]][XXN]
140             , LPIparam[["filterLength"]][XXN]
141             , firstSample[XXN]
142             , firstFraction[XXN]
143             , TRUE
144             )
145
146 }
147
148 # Use length of the shortest data vector
149 LPIdatalist.final[["nData"]] <-
150     min(
151         LPIdatalist.final[["RX1"]][["ndata"]],
152         LPIdatalist.final[["RX2"]][["ndata"]],
153         LPIdatalist.final[["TX1"]][["ndata"]],
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:
      LPIdatalist.final[["nData"]]])
161     itx2 <- which(LPIdatalist.final[["TX2"]][["idata"]][1:
      LPIdatalist.final[["nData"]]])
162     txamp1 <- mean(abs(LPIdatalist.final[["TX1"]][["cdata"
      ]][itx1]))
163     txamp2 <- mean(abs(LPIdatalist.final[["TX2"]][["cdata"
      ]][itx2]))
164     LPIdatalist.final[["TX1"]][["cdata"]][itx1] <- exp(1i*
      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
169   if( ( LPIparam[["maxClutterRange"]][["RX1"]] > 0 ) & (
      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"]],
      LPIdatalist.final[["RX2"]], LPIparam[["rangeLimits"
      ]][1], LPIparam[["maxClutterRange"]][["RX2"]],
      LPIdatalist.final[["nData"]], LPIparam[["
      clutterFraction"]][["RX2"]] )
174   }
175
176
177   # Optional voltage level decoding
178   if( is.numeric( LPIparam[["decodingFilter"]] ) ){
179
180     LPIdatalist.final[["RX1"]][["cdata"]][!LPIdatalist.
      final[["RX1"]][["idata"]]] <- 0+0i
181     LPIdatalist.final[["RX2"]][["cdata"]][!LPIdatalist.
      final[["RX2"]][["idata"]]] <- 0+0i
182     LPIdatalist.final[["TX1"]][["cdata"]][!LPIdatalist.
      final[["TX1"]][["idata"]]] <- 0+0i
183     LPIdatalist.final[["TX2"]][["cdata"]][!LPIdatalist.
      final[["TX2"]][["idata"]]] <- 0+0i
184
185     nd <- LPIdatalist.final[["nData"]]
186

```

```

187
188     LPIdatalist.final[["RX1"]][["cdata"]] <- LPI:::
        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:::
        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:::
        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:::
        decoFilter.cdata( LPIdatalist.final[["TX2"]][["
        cdata"]][1:nd] , LPIdatalist.final[["TX2"]][["
        cdata"]][1:nd] , LPIdatalist.final[["TX2"]][["
        idata"]][1:nd] , LPIparam[["decodingFilter"]][1] )
195
196 }else if( is.character( LPIparam[["decodingFilter"]] ) ){
197
198     if( any( LPIparam[["decodingFilter"]][1] == c("
        matched","inverse") ) ) {
199
200         LPIdatalist.final[["RX1"]][["cdata"]][!
            LPIdatalist.final[["RX1"]][["idata"]]] <- 0+0i
201         LPIdatalist.final[["RX2"]][["cdata"]][!
            LPIdatalist.final[["RX2"]][["idata"]]] <- 0+0i
202         LPIdatalist.final[["TX1"]][["cdata"]][!
            LPIdatalist.final[["TX1"]][["idata"]]] <- 0+0i
203         LPIdatalist.final[["TX2"]][["cdata"]][!
            LPIdatalist.final[["TX2"]][["idata"]]] <- 0+0i
204
205         nd <- LPIdatalist.final[["nData"]]
206
207
208         LPIdatalist.final[["RX1"]][["cdata"]] <- LPI:::
            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:::
          decoFilter.cdata( LPIdatalist.final[["TX1"]][[
            "cdata"]][1:nd] , LPIdatalist.final[["TX1"]][[
            "cdata"]][1:nd] , LPIdatalist.final[["TX1"]][[
            "idata"]][1:nd] , LPIparam[["decodingFilter"
            ]][1] )
211
212         LPIdatalist.final[["RX2"]][["cdata"]] <- LPI:::
          decoFilter.cdata( LPIdatalist.final[["RX2"]][[
            "cdata"]][1:nd] , LPIdatalist.final[["TX2"]][[
            "cdata"]][1:nd] , LPIdatalist.final[["TX2"]][[
            "idata"]][1:nd] , LPIparam[["decodingFilter"
            ]][1] )
213
214         LPIdatalist.final[["TX2"]][["cdata"]] <- LPI:::
          decoFilter.cdata( LPIdatalist.final[["TX2"]][[
            "cdata"]][1:nd] , LPIdatalist.final[["TX2"]][[
            "cdata"]][1:nd] , LPIdatalist.final[["TX2"]][[
            "idata"]][1:nd] , LPIparam[["decodingFilter"
            ]][1] )
215
216         LPIdatalist.final[["TX1"]][["idata"]] <- LPI:::
          decoFilter.idata( LPIdatalist.final[["TX1"]][[
            "idata"]][1:nd] )
217
218         LPIdatalist.final[["TX2"]][["idata"]] <- LPI:::
          decoFilter.idata( LPIdatalist.final[["TX2"]][[
            "idata"]][1:nd] )
219     }
220 }
221
222
223 # Largest range in rangeLimits
224 maxr <- as.integer(max(LPIparam[["rangeLimits"]]))
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
230     LPIdatalist.final[["RX1"]][["power"]] <-
        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
233     LPIdatalist.final[["RX2"]][["power"]] <-
        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,
236 # but only if there were reasonably many samples in the
    averages
237 if(LPIdatalist.final[["RX1"]][["power"]][1] < .05 ){
238     itx1 <- which( abs(LPIdatalist.final[["RX1"]][["
        cdata"]][1:LPIdatalist.final[["nData"]]]) > (
        sqrt(LPIdatalist.final[["RX1"]][["power"]])*4) )
239     LPIdatalist.final[["RX1"]][["idata"]][itx1] <-
        FALSE
240 }
241 if(LPIdatalist.final[["RX2"]][["power"]][1] < .05 ){
242     itx2 <- which( abs(LPIdatalist.final[["RX2"]][["
        cdata"]][1:LPIdatalist.final[["nData"]]]) > (
        sqrt(LPIdatalist.final[["RX2"]][["power"]])*4) )
243     LPIdatalist.final[["RX2"]][["idata"]][itx1] <-
        FALSE
244 }
245 }
246
247 # maxr points in the beginning will not have
248 # a reasonable power estimate, flag these points as well
249 LPIdatalist.final[["RX1"]][["idata"]][1:maxr] <- FALSE
250 LPIdatalist.final[["RX2"]][["idata"]][1:maxr] <- FALSE
251
252 #####
253 ## Copy parameters from LPiParam to ##
254 ## the final data list as necessary ##
255 #####
256
257 # Lag values
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"
    "]]
264 nmaxr <- length(LPIdatalist.final[["maxRanges"]])
265 if( nmaxr < LPIdatalist.final[["nLags"]] ){
266     LPIdatalist.final[["maxRanges"]] <- c( LPIdatalist.
        final[["maxRanges"]] , rep(LPIdatalist.final[["
        maxRanges"]][nmaxr],(LPIdatalist.final[["nLags"]]-

```

```

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

```

```

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

```

335 |
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.
5 ##
6
7 ##
8 ## Ground clutter suppression as follows:
9 ##
10 ## 1. Scattered signal in ranges between rmin and rmax is
11 ##    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 contains named
20 ##            vectors 'cdata' and 'idata'
21 ##   rmin     Smallest range from which clutter should
22 ##            be suppressed
23 ##   rmax     Largest range from which clutter should
24 ##            be suppressed
25 ##   ndata    Number of points in data vectors
26 ##   clutterFraction Fraction of the full integration
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 ,
38                             ndata , clutterFraction )
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
44     if( ndata <= 0 ) return()
45
46     # Set negative ranges to zero
```

```

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

```

```

95         txdata[["idata"]],
96         rxdata[["cdata"]],
97         rxdata[["idata"]],
98         ndata,
99         rmin,
100        rmax,
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.
5 ##
6
7 ##
8 ## Voltage level decoding, either matched or inverse
9 ## filtering, using measured transmitter samples.
10 ##
11 ## Arguments:
12 ##   cdata      A complex receiver data vector
13 ##   cenv       A complex transmitter data vector
14 ##   idata      A logical vector of transmitter data indices
15 ##   filterType Decoding filter. Either a complex vector of
16 ##             filter taps, 'matched' or 'inverse'
17 ##
18 ## Returns:
19 ##   cdata The complex receiver data vector after decoding
20 ##
21 decoFilter.cdata <- function( cdata , cenv , idata ,
22                               filterType='inverse')
23 {
24   # Pulse start positions and number of pulses
25   txstarts <- which( diff(idata>0) == 1 )
26   if(idata[1]) txstarts <- c(0,txstarts)
27   ntx <- length(txstarts)
28   txstarts <- c(txstarts,length(cdata))
29
30   # If there are no transmission pulses, then simply return
31   if(ntx<1) return(cdata)
32
33   # Set the data points before the first pulse to zero
34   if( txstarts[1] > 0 ) cdata[1:txstarts[1]] <- 0+0i
35
36   # Set transmitter data to zero at points that are not
37   # transmitter samples
38   cenv[!idata] <- 0+0i
39
40   # Filtering with user-defined coefficients
41   if( is.numeric( filterType ) ){
42     nfilter <- length(filterType)
43     for( k in seq( ntx ) ){
44       cenv[ (txstarts[k]+1) : (txstarts[k+1]) ] <- 0+0i
```

```

45         cenv[ (txstarts[k]+1) : (txstarts[k]+nfilter)]
46         <- filterType
47         cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] <-
48             fft(
49                 fft( cdata[ (txstarts[k]+1) : (txstarts[k]
50                     +1)] ) ) /
51                 fft( cenv[ (txstarts[k]+1) : (txstarts[k]
52                     +1)] ) )
53                 , inverse=TRUE ) /
54                 (txstarts[k+1]-txstarts[k]) * sqrt(
55                     sum(abs(cenv[ (txstarts[k]+1) : (
56                         txstarts[k+1]) ])**2))
57             }
58 }else if( is.character( filterType ) ){
59     # Inverse filtering
60     if(filterType=="inverse"){
61         for( k in seq( ntx ) ){
62             cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] <-
63                 fft(
64                     fft( cdata[ (txstarts[k]+1) : (
65                         txstarts[k+1]) ] ) ) /
66                     fft( cenv[ (txstarts[k]+1) : (
67                         txstarts[k+1]) ] ) )
68                     , inverse=TRUE ) /
69                     (txstarts[k+1]-txstarts[k]) *
70                     sqrt(sum(abs(cenv[ (txstarts[k]
71                         +1) : (txstarts[k+1]) ])**2))
72             }
73     # Matched filtering
74 }else if(filterType=="matched"){
75     for( k in seq( ntx ) ){
76         cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] <-
77             fft(
78                 fft( cdata[ (txstarts[k]+1) : (
79                     txstarts[k+1]) ] ) *
80                 Conj( fft( cenv[ (txstarts[k]+1) : (
81                     txstarts[k+1]) ] ) ) )
82                 , inverse=TRUE ) /
83                 (txstarts[k+1]-txstarts[k]) /
84                 sqrt(sum(abs(cenv[ (txstarts[k]
85                     +1) : (txstarts[k+1]) ])**2))
86             }
87     # Other filters are not supported at the moment
88 }else{
89     stop("Unknown decoding filter")
90 }

```

```

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)
104     ntx <- length(txstarts)
105     txstarts <- c(txstarts,length(idata))
106
107     # Each pulse should have been compressed into
108     # a single sample in the decoding
109     for( k in seq( ntx ) ){
110         idata[(txstarts[k]+2):txstarts[k+1]] <- FALSE
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.
5 ##
6
7 ##
8 ## Average power profiles
9 ##
10 ## Arguments:
11 ##   cdata      A complex data vector
12 ##   idatatx    A logical vector of transmitter pulse positions
13 ##   idatarx    A logical vector of usable receiver samples
14 ##   ndata      Number points in data vectors
15 ##   maxrange   Largest range from which the power is needed
16 ##
17 ## Returns:
18 ##   pdata      Average power profile vector
19 ##
20
21 LPIaveragePower <- function( cdata , idatatx , idatarx ,
22                             ndata , maxrange )
23 {
24   # Call the C function
25   pow <- .Call( "average_power" , cdata , idatatx , idatarx
26                 , ndata , maxrange )
27
28   # Check the first element, .01 means that number of
29   # summed power values is 10 in average.
30   # The first element will be NA if no pulses were found,
31   # then it does not really matter what we do..
32   if( is.na( pow[1] ) ){
33     pow[] <- mean( abs( cdata[idatarx])**2 )
34   }else if( pow[1] > .1 ){
35     pow[] <- mean( abs( cdata[idatarx])**2 )
36   }
37
38   return(pow)
39 }
```

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.
5 ##
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"]])
23
24   # Number of lags
25   nlags <- length(ACF[["lag"]])
26
27   # Seconds since 1970
28   ACF[["time.s"]] <- LPIparam[["startTime"]] + intPeriod*
     LPIparam[["timeRes.s"]]
29
30   # The same time as a string, useful for debugging
31   # time conversions and for plotting
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
36   resFile <- gsub(' ', '0', file.path( LPIparam[["resultDir"
     ]] , paste( sprintf( '%13.0f' , trunc( ACF[["time.s"]]
     * 1000 ) ) , "LP.Rdata" , sep='')) )
37
38   # Range
39   names(ACF[["range"]]) <- paste('gate', seq(ngates), sep='')
40
41   # Lag
42   names(ACF[["lag"]]) <- paste('lag', seq(nlags), sep='')
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),
    sep='')
48 names(ACF[["backgroundvar"]]) <- paste('lag',seq(nlags),
    sep='')
49
50 # ACF and variance without the background samples
51 ACF[["ACF"]] <- matrix(ACF[["ACF"]][1:ngates,],ncol=nlags
    )
52 ACF[["var"]] <- matrix(ACF[["var"]][1:ngates,],ncol=nlags
    )
53 dimnames(ACF[["ACF"]]) <- list(paste('gate',seq(ngates),
    sep=''),paste('lag',seq(nlags),sep=''))
54 dimnames(ACF[["var"]]) <- list(paste('gate',seq(ngates),
    sep=''),paste('lag',seq(nlags),sep=''))
55
56 # Dimnames for the optional full covariance matrix
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
60 # laginds <- apply( ACF[["ACF"]] , FUN=function(x){ any( !
    is.na( x ) ) } , MARGIN = 2 )
61 laginds <- which( c( LPIparam[["maxRanges"]] , rep(
    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
65 ACF[["rangeLimits"]] <- LPIparam[["rangeLimits"]]
66 names(ACF[["rangeLimits"]]) <- ""
67
68 # Lag integration limits
69 ACF[["lagLimits"]] <- LPIparam[["lagLimits"]]
70 names(ACF[["lagLimits"]]) <- ""
71
72 # Maximum ranges
73 ACF[["maxRanges"]] <- LPIparam[["maxRanges"]]
74 names(ACF[["maxRanges"]]) <- ""
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.
5 ##
6
7 ##
8 ## Return an ACF list with only selected ranges and lags
9 ##
10 ## Arguments:
11 ##   ACFlst      An ACF list returned by runLPI or stored by
12 ##               LPIsaveACF
13 ##   rgates      Range gate indices
14 ##   lags        Lag indices
15 ##   fullCovar   TRUE if the ACFlst contains the full
16 ##               covariance matrices
17 ##
18 ## Returns:
19 ##   ACFlst      A modified ACF list
20
21 stripACF <- function( ACFlst , rgates , lags , fullCovar=
22   FALSE)
23 {
24   # An empty list for the output
25   ACFlst2 <- list()
26
27   # If rgates and lags are logical vectors
28   # convert them into indices
29   if(is.logical(rgates)) rgates <- which(rgates)
30   if(is.logical(lags)) lags <- which(lags)
31
32   # Pick the ACF and variance values
33   ACFlst2[["ACF"]] <- ACFlst[["ACF"]][rgates,lags]
34   ACFlst2[["var"]] <- ACFlst[["var"]][rgates,lags]
35
36   # Make sure that ACF, var, and covariance are still
37   # arrays
38   dim(ACFlst2[["ACF"]]) <- c( length(rgates) , length(lags)
39     ) )
40   dim(ACFlst2[["var"]]) <- c( length(rgates) , length(lags)
41     ) )
42   if(fullCovar){
43     covdims <- dim(ACFlst[["covariance"]])
44     ACFlst2[["covariance"]] <- ACFlst[["covariance"]][c(
45       rgates,covdims[1]),c(rgates,covdims[2]),lags]
```



```

41     dim(ACFlist2[["covariance"]]) <- c( (length(rgates)+1)
42     , (length(rgates)+1) , length(lags) )
43 }
44
45 ACFlist2[["lag"]] <- ACFlist[["lag"]][lags]
46 ACFlist2[["range"]] <- ACFlist[["range"]][rgates]
47 ACFlist2[["nGates"]] <- pmin(rep(length(rgates),length(
48     lags)),ACFlist[["nGates"]][lags])
49 ACFlist2[["backgroundACF"]] <- ACFlist[["backgroundACF"
50     ]][lags]
51 ACFlist2[["backgroundvar"]] <- ACFlist[["backgroundvar"
52     ]][lags]
53 ACFlist2[["timeString"]] <- ACFlist[["timeString"]]
54 ACFlist2[["time.s"]] <- ACFlist[["time.s"]]
55
56 # Udpate names to match with the new indexing
57 nlags <- length(lags)
58 ngates <- length(rgates)
59
60 names(ACFlist2[["range"]]) <- paste('gate',seq(ngates),
61     sep='')
62 names(ACFlist2[["lag"]]) <- paste('lag',seq(nlags),sep='')
63
64 names(ACFlist2[["backgroundACF"]]) <- paste('lag',seq(
65     nlags),sep='')
66 names(ACFlist2[["backgroundvar"]]) <- paste('lag',seq(
67     nlags),sep='')
68
69 dimnames(ACFlist2[["ACF"]]) <- list(paste('gate',seq(
70     ngates),sep=''),paste('lag',seq(nlags),sep=''))
71 dimnames(ACFlist2[["var"]]) <- list(paste('gate',seq(
72     ngates),sep=''),paste('lag',seq(nlags),sep=''))
73
74 return(ACFlist2)
75 }

```

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:
11 ##   LPIenv An LPI environment
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"
28
29     # Call the c function
30     return( .Call( "lagged_products" ,
31                   LPIenv[["RX1"]][["cdata"]] ,
32                   LPIenv[["RX2"]][["cdata"]] ,
33                   LPIenv[["RX1"]][["idata"]] ,
34                   LPIenv[["RX2"]][["idata"]] ,
35                   LPIenv[["cprod"]] ,
36                   LPIenv[["iprod"]] ,
37                   LPIenv[["nData"]] ,
38                   LPIenv[["nData"]] ,
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.
5 ##
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 ##   success TRUE if a variance estimate was successfully
17 ##           calculated for at least one data point,
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
26     storage.mode(lag) <- "integer"
27
28     # Call the C function
29     return( .Call( "lagged_products_r"
30                   , LPIenv[["RX1"]][["power"]] ,
31                   , LPIenv[["RX2"]][["power"]] ,
32                   , LPIenv[["var"]] ,
33                   , LPIenv[["nData"]] ,
34                   , LPIenv[["nData"]] ,
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.
5 ##
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 ##   success  TRUE if at least one point was successfully
17 ##             calculated, FALSE otherwise.
18 ##             The range ambiguity function is
19 ##             (over)written to LPIenv$camb.
20 ##
21 ##
22 ##
23 ##
24 ##
25
26 rangeAmbiguity <- function( LPIenv , lag )
27 {
28
29     # True oversampling is not supported.
30     if( LPIenv[['nDecimTX']] != 1) stop("True transmitter
31         signal oversampling is not supported.")
32
33     # Make sure that lag is an integer
34     storage.mode(lag) <- "integer"
35
36     # Simulate oversampling by means of interpolation.
37     # This works well if the pulses have
38     # sharp edges and constant amplitude.
39     if( LPIenv[["ambInterp"]] ){
40         return( .Call( "range_ambiguity"
41             , LPIenv[["TX1"]][["cdata"]] ,
42             LPIenv[["TX2"]][["cdata"]] ,
43             LPIenv[["TX1"]][["idata"]] ,
44             LPIenv[["TX2"]][["idata"]] ,
45             LPIenv[["camb"]] ,
46             LPIenv[["iamb"]] ,
47             LPIenv[["nData"]]
```

```

47         LPIenv[["nData"]] ,
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"]] ,
58               LPIenv[["TX1"]][["idata"]] ,
59               LPIenv[["TX2"]][["idata"]] ,
60               LPIenv[["camb"]] ,
61               LPIenv[["iamb"]] ,
62               LPIenv[["nData"]] ,
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.
5 ##
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 accuracy and flexibility is acceptable.
15 ##
16 ##
17 ## Arguments:
18 ##   LPIenv A LPI environment
19 ##   l      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 , l )
30 {
31
32     s1 <- .Call( "average_profile" , LPIenv[["cprod"]] ,
33                 LPIenv[["TX1"]][["idata"]] , as.integer( LPIenv[["
34                 nData"]] - 1 ) , as.integer( LPIenv[["nCode"]] ) )
35
36     s2 <- .Call( "average_profile" , LPIenv[["camb"]] ,
37                 LPIenv[["TX1"]][["idata"]] , as.integer( LPIenv[["
38                 nData"]] - 1 ) , as.integer( LPIenv[["nCode"]] ) )
39
40     invisible( ( s1 & s2 ) )
41 }
```

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.
5 ##
6
7 ##
8 ## Form theory matrix rows for lag profile inversion
9 ##
10 ## Arguments:
11 ##   LPIenv      A LPI environment
12 ##   lag         Lag number
13 ##
14 ## Returns:
15 ##   success     TRUE if at least one theory matrix row
16 ##               was successfully produces, FALSE otherwise.
17 ##
18 ## The rows are written to LPIenv[["arows"]],
19 ## the corresponding measurements to LPIenv[["meas"]],
20 ## variance to LPIenv[["mvar"]], and number of rows
21 ## generated to LPIenv[["nrows"]]
22 ##
23 ##
24
25 theoryRows <- function( LPIenv , lag )
26 {
27
28   # Call the C routine
29   return( .Call( "theory_rows" ,
30                 LPIenv[['camb']] ,
31                 LPIenv[['iamb']] ,
32                 LPIenv[['cprod']] ,
33                 LPIenv[['iprod']] ,
34                 LPIenv[['var']] ,
35                 LPIenv[['nData']] ,
36                 LPIenv[['nCur']] ,
37                 as.integer(LPIenv[['nCur']] + LPIenv[['nBuf'
38                 ]]) ,
39                 LPIenv[['rangeLimits']] ,
40                 LPIenv[['nGates']][lag] ,
41                 LPIenv[['arows']] ,
42                 LPIenv[['irows']] ,
43                 LPIenv[['meas']] ,
44                 LPIenv[['mvar']] ,
45                 LPIenv[['nrows']] ,
46                 LPIenv[["backgroundEstimate"]],
47                 LPIenv[["remoteRX"]]
```

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.
5 ##
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 {
20   # New environment for the solver
21   s <- new.env()
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
34   # correct for later c function calls
35   storage.mode(s$Qvec) <- storage.mode(s$y) <- "complex"
36   storage.mode(s$ncol) <- "integer"
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 ##   e      A fishs solver environment
14 ##   A.data Theory matrix rows as a vector (row-by-row)
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))
27
28     # Variance vector
29     E.data <- rep(E.data,length.out=nrow)
30
31     # Check storage modes before calling the c function
32     storage.mode(A.data) <- "complex"
33     storage.mode(M.data) <- "complex"
34     storage.mode(E.data) <- "double"
35     storage.mode(nrow)   <- "integer"
36
37     # Call the c function
38     return( .Call( "fishs_add" , e[["Qvec"]] , e[["y"]] , A.
39         data , M.data , E.data , e[["ncol"]] , nrow ))
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.
5 ##
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 ##   e           A fishs solver environment
14 ##   full.covariance Logical, full covariance matrix is
15 ##                 calculated
16 ##                 if TRUE, otherwise only variances are
17 ##                 returned.
18 ## Returns:
19 ##   Nothing, the solution is assigned to
20 ##   the solver environment
21 ##
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"]] )
28
29   # Copy the upper triangular part form e$Qvec
30   i <- 1
31   for( k in seq( e$ncol ) ){
32     Q[ k , k : e[["ncol"]] ] <- e[["Qvec"]][ i : ( i + ( e
33       [["ncol"]] - k ) ) ]
34     i <- i + e[["ncol"]] - k + 1
35   }
36
37   # The lower triangular part is
38   # complex conjugate of the upper one
39   Q <- Q + Conj( t( Q ) )
40
41   # The above row multiplies the diagonal
42   # with 2, divide accordingly
43   diag( Q) <- diag( Q ) / 2
44
45   # Select points at which the diagonal of Q is zero,
```

```

45 # these points have not been measured at all and
46 # need to be regularized before inverting the matrix
47 nainds      <- Re( diag( Q ) ) == 0
48
49 # Set unit values on the diagonal at unmeasured points.
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
55 # the Fisher information matrix
56 # Even if there were measurements the matrix might not be
    invertible
57 # return NA matrix in this case
58 covariance      <- tryCatch( solve( Q ) , error=
    function(e){Q*NA})
59
60 # Multiply the covariance matrix with e$y from right.
61 # For some reason the direct matrix multiplication
62 # with %*% does not work properly in some machines.
63 solution <- rep(0+0i,e[["ncol"]])
64 for(k in seq(e[["ncol"]])) solution[k] <- sum( covariance
    [k,] * e[["y"]] )
65
66 # Set NAs to points that were not actually measured
67 solution[ nainds ] <- NA
68
69 # Assign the solution to the solver environment e
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 ){
76     covariance[ nainds ,      ] <- 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:
11 ##   ncols Number of unknowns (theory matrix columns)
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
26   # Number of columns in theory matrix
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
37   storage.mode(s$Qvec) <- storage.mode(s$y) <- "complex"
38   storage.mode(s$ncol) <- "integer"
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.
5 ##
6
7 ##
8 ## Matched filter decoder. Data accumulation function.
9 ##
10 ## Arguments:
11 ##   e      A deco solver environemnt
12 ##   A.data Theory matrix rows as a vector (row-by-row)
13 ##   M.data Measurement vector
14 ##   E.data Measurement variance vector
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))
25
26   # Measurement variance vector
27   E.data <- rep(E.data,length.out=nrow)
28
29   # Set storage modes
30   storage.mode(A.data) <- "complex"
31   storage.mode(M.data) <- "complex"
32   storage.mode(E.data) <- "double"
33   storage.mode(nrow)   <- "integer"
34
35   # Call the c routine
36   return( .Call( "deco_add" , e$Qvec , e$y , A.data , M.
37     data , E.data , e$ncol , nrow ))
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.
5 ##
6
7 ##
8 ## Matched filter decoder. Final solver function.
9 ##
10 ## Arguments:
11 ##   e                A deco solver environment
12 ##   full.covariance Logical, full covariance matrix is
13 ##                   calculated if TRUE, otherwise only
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
39   variance <- 1 / Qdiag
40
41   # Assign the solution to the solver environment
42   assign( 'solution' , variance * e[["y"]] , e )
43
44   # Set NAs to the unmeasured points
45   e[["solution"]][nainds] <- NA
46
47   # Same for the variances
```

```
48|     assign( 'covariance' , variance , e )
49|     e[["covariance"]][nainds] <- NA
50|
51|     invisible()
52|
53| }
```


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.
5 ##
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 ##   s      A dummy solver environment
17 ##
18
19 dummy.init <- function( rrange )
20 {
21
22   # A new environment for the solver
23   s <- new.env()
24
25   # Number of ranges (this is different
26   # from number of final range gates)
27   nr <- abs(diff(rrange))
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)
34
35   # Minimum range
36   rmin <- min(rrange)
37
38   # Maximum range
39   rmax <- max(rrange)
40
41   # Make sure that storage modes are correct
42   storage.mode(msum) <- "complex"
43   storage.mode(vsum) <- "double"
44   storage.mode(rmin) <- "integer"
45   storage.mode(rmax) <- "integer"
46
47   # Assign the variables to the environment
```

```
48 |     assign( 'msum' , msum , s )
49 |     assign( 'vsum' , vsum , s )
50 |     assign( 'rmin' , rmin , s )
51 |     assign( 'rmax' , rmax , s )
52 |
53 |     # Return the environment
54 |
55 |     return(s)
56 |
57 | }
```

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.
5 ##
6
7 ##
8 ## Dummy inverse problem solver that
9 ## calculates simple averages.
10 ## Data accumulation function.
11 ##
12 ## Arguments:
13 ##   e      A dummy solver environemnt
14 ##   M.data  Measurement vector
15 ##   M.ambig Range ambiguity function
16 ##   I.ambig Indices of non-zero ambiguity values
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.
26   prod , E.data , nData )
27 {
28   # Call the C routine
29   return( .Call( "dummy_add" ,
30     e[["msum"]] ,
31     e[["vsum"]] ,
32     e[["rmin"]] ,
33     e[["rmax"]] ,
34     M.data ,
35     M.ambig ,
36     I.ambig ,
37     I.prod ,
38     E.data ,
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.
5 ##
6
7 ##
8 ## Dummy inverse problem solver that
9 ## calculates simple averages.
10 ## Final solver function.
11 ##
12 ## Arguments:
13 ##   e      A dummy solver environment
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
32
33   # Vectors for the solution and variance
34   solution <- rep(0+0i,nr)
35   covariance <- rep(0,nr)
36
37   # Range integration for the data points that have
38   # the best possible resolution at this point.
39   for( r in seq(nr) ){
40
41     # Lower limit of this range gates
42     r1 <- rlims[r] - rlims[1] + 1
43
44     # Upper limit of this range gate
45     r2 <- rlims[r+1] - rlims[1]
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])
50
51     # The vector e$vsum contains informations, sum them.
52     covariance[r] <- sum(e[["vsum"]][r1:r2])
53
54 }
55
56 # Variance is inverse of the information
57 covariance <- c( 1/covariance , NA )
58
59 # Multiply the solution with the final variances
60 solution <- c( solution , NA ) * covariance
61
62 # Vectors solution and covariance will now contain
63 # variance-weighted averages of the lag profiles
64 # and their variances. Assign to the solver environment
65 assign( 'solution' , solution , e )
66 assign( 'covariance' , covariance , e )
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.
5 ##
6
7 ##
8 ## FFT deconvolution.
9 ## Initialization function.
10 ##
11 ## Arguments:
12 ##   rrange Extreme ranges to be solved c(rmin,rmax)
13 ##   itx     A logical vector of transmitter pulse positions.
14 ##
15 ## Returns:
16 ##   s       A ffts solver environment
17 ##
18
19 ffts.init <- function( rrange , itx )
20 {
21   # Minimum range
22   rmin      <- min( rrange )
23
24   # Maximum range
25   rmax      <- max( rrange )
26
27   # longest inter-pulse period
28   ippmax <- max( diff( which( diff( itx > 0 ) == 1 ) ) ,
29                 showWarnings=FALSE )
30
31   # Select the FFT length
32   n <- max( nextn( ippmax ) , nextn( rmax*2 ) )
33
34   # Allocate vectors
35   fy      <- rep( 0+0i , n )
36   amb.tmp <- famb.tmp      <- rep( 0+0i , n )
37   meas.tmp <- rep( 0+0i , n )
38   sqfamb   <- rep( 0 , n )
39   varsum   <- 0
40   nmeas    <- 0
41
42   # Set storage modes
43   storage.mode( rmin ) <- "integer"
44   storage.mode( rmax ) <- "integer"
45   storage.mode( n )    <- "integer"
46   storage.mode(nmeas)  <- "integer"
47   storage.mode( fy )   <- "complex"
```

```

47 storage.mode( amb.tmp ) <- "complex"
48 storage.mode( famb.tmp ) <- "complex"
49 storage.mode( meas.tmp ) <- "complex"
50 storage.mode( sqfamb ) <- "double"
51 storage.mode( varsum ) <- "double"
52
53 # Create a new environment and assign everything to it
54 s <- new.env()
55 assign( 'n' , n , s )
56 assign( 'rmin' , rmin , s )
57 assign( 'rmax' , rmax , s )
58 assign( 'fy' , fy , s )
59 assign( 'sqfamb' , sqfamb , s )
60 assign( 'amb.tmp' , amb.tmp , s )
61 assign( 'famb.tmp' , famb.tmp , s )
62 assign( 'meas.tmp' , meas.tmp , s )
63 assign( 'nmeas' , nmeas , s )
64 assign( 'varsum' , varsum , s )
65
66 # return the environment
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.
5 ##
6
7 ##
8 ## FFT deconvolution.
9 ## Data accumulation function.
10 ##
11 ## Arguments:
12 ##   e      An ffts solver environemnt
13 ##   M.data  Measurement vector
14 ##   M.ambig Range ambiguity function
15 ##   I.ambig Indices of non-zero ambiguity values
16 ##   I.prod  Indices of usable lagged products
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 ##
23
24 ffts.add <- function( e , M.data , M.ambig , I.ambig , I.prod
25                       , E.data , nData )
26 {
27   #
28   # FFT deconvolution. Data accumulation function.
29   #
30   # I. Virtanen 2012
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
38   M.data[ which(!I.prod) ] <- 0+0i
39   E.data[ which(!I.prod) ] <- 0
40   M.ambig[ which(!I.ambig) ] <- 0+0i
41
42   # Locate pulse start positions
43   ps <- which( diff( I.ambig[1:nData] > 0 ) == 1 )
44
45   # The first point should be adjusted to pulse start,
46   # so it is safe to use if the index is set
```



```

47 if( I.ambig[1] ) ps <- c( 1 , ps )
48 npulse <- length( ps )
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...
55 npulse <- min( length(pe) , length(ps) )
56
57 # Add data from one IPP at a time
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,
64     # but check anyway)
65     pe1 <- min( nData , pe[k] )
66
67     # max range or data end
68     pe2 <- min( nData , ( ps[k] + e[["n"]][ ] - 1
69         ) )
70
71     # Copy one pulse
72     e[["amb.tmp"]][ 1 : ( pe1 - ps[k] + 1 ) ] <- M.ambig[
73         ps[k] : pe1 ]
74
75     # Take fft
76     e[["famb.tmp"]][ ] <- fft( e[["amb.tmp"]][ ] )
77
78     # Copy data
79     e[["meas.tmp"]][ 1 : ( pe2 - ps[k] + 1 ) ] <- M.data[
80         ps[k] : pe2 ]
81
82     # Actual addition to the solver
83     e[["fy"]][ ] <- e[["fy"]][ ] + Conj( e[["famb.tmp"]][ ] ) *
84         fft( e[["meas.tmp"]][ ] )
85     e[["sqfamb"]][ ] <- e[["sqfamb"]][ ] + abs( e[["famb.tmp"]][ ]
86         )**2
87
88 }
89
90 # Variances
91 e[["varsum"]][ ] <- e[["varsum"]][ ] + sum( E.data[ 1 : nData ]
92     )
93 e[["nmeas"]][ ] <- e[["nmeas"]][ ] + sum( ( I.prod[ 1 : nData
94     ] > 0 ) )
95

```

```
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.
5 ##
6
7 ##
8 ## FFT deconvolution.
9 ## Final solver function.
10 ##
11 ## Arguments:
12 ##   e      A ffts solver environment
13 ##   rlims  Range gate limits
14 ##
15 ## Returns:
16 ##   Nothing, the solution is assigned to the solver
17 ##   environment
18 ##
19 ffts.solve <- function( e , rlims )
20 {
21   #
22   # FFT deconvolution. Final solver function.
23   #
24   # I. Virtanen 2012
25   #
26   # Solve the lag profile by means of FFT
27   sol <- fft( e[["fy"]] / e[["sqfamb"]] , inverse=TRUE )
28   / e[["n"]]
29
30   # Variance, the same value will be repeated at all ranges
31   var <- e[["varsum"]] / as.double(e[["nmeas"]]) * mean( 1
32   / e[["sqfamb"]] )
33
34   # Number of range gates
35   nr <- length(rlims) - 1
36
37   # Final solution and variance vectors
38   solution <- rep(0+0i,nr)
39   covariance <- rep(0,nr)
40
41   for( r in seq(nr) ){
42     # Lower limit of range gate
43     r1 <- rlims[r] + 1
44
45     # Upper limit of range gate
```

```

45         r2                <- rlims[r+1]
46
47         # All points have equal variances, calculate simple
           average
48         solution[r]      <- mean( sol[r1:r2] , na.rm=TRUE )
49
50         # Scale the variance
51         covariance[r] <- var/(r2-r1+1)
52     }
53
54     # The background ACF cannot be measured with this
           technique, set it to NA.
55     covariance <- c( covariance , NA )
56     solution   <- c( solution , NA )
57
58     # Assign the results to the solver environment.
59     assign( 'solution'   , solution   , e )
60     assign( 'covariance' , covariance , e )
61
62     invisible()
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 ##   e                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$R.mat ) ) == 0 )
28
29   # Add regularizing imaginary measurements
30   regrow <- rep(0+0i,e$ncols)
31   for( n in nainds ){
32     regrow[] <- 0+0i
33     regrow[n] <- 1+0i
34     rlips.add( e , A.data = regrow , M.data = 1.0+0.0i )
35   }
36
37   # Solve the problem
38   rlips.solve( e , calculate.covariance = TRUE , full.
39     covariance = full.covariance )
40
41   # Set NAs to appropriate points in the solution
42   sol <- e$solution
43   sol[nainds] <- NA
44   assign( 'solution' , sol , e )
45
46   # Set the unmeasured points to NA
47   # in the covariance matrix as well.
```

```

47 | covar <- e$covariance
48 | if( full.covariance ){
49 |     covar[ , nainds ] <- NA
50 |     covar[ nainds , ] <- NA
51 | }else{
52 |     covar[nainds] <- NA
53 | }
54 |
55 | # 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 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.
5
6 // Data types and function prototypes
7
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 5
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 cdata , 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 );
```

```

35
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 with 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
    nup , SEXP nfilter , SEXP nfirst , SEXP nfirstfrac , SEXP
    ipartial );
56 SEXP resample_R( SEXP cdata , SEXP idata , SEXP ndata , SEXP
    nup , SEXP nfilter , SEXP nfirst , SEXP 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 ,

```



```

        SEXP ndata2 ,  SEXP lag );
60
61 // Ground clutter suppression
62 SEXP clutter_meas( const SEXP tcdata , const SEXP tidata ,
        const SEXP rcdata , const SEXP ridata , const SEXP ndata ,
        const SEXP rmin ,  const SEXP rmax , const SEXP Qvec ,
        const SEXP yvec );
63 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 )
        ;
64 void fishs_add_clutter( const SEXP Qvec , const SEXP yvec ,
        Rcomplex * arow , Rcomplex * meas , const int nx );

```

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.
5
6 // R registration of C functions
7
8 #include "LPI.h"
9 static const R_CallMethodDef callMethods[20] = {
10 { "read_gdf_data_R"      , (DL_FUNC) & read_gdf_data_R
11   , 6 } ,
12 { "mix_frequency_R"      , (DL_FUNC) & mix_frequency_R
13   , 3 } ,
14 { "index_adjust_R"       , (DL_FUNC) & index_adjust_R
15   , 3 } ,
16 { "lagged_products_alloc" , (DL_FUNC) &
17   lagged_products_alloc , 7 } ,
18 { "lagged_products"      , (DL_FUNC) & lagged_products
19   , 9 } ,
20 { "lagged_products_r"    , (DL_FUNC) & lagged_products_r
21   , 6 } ,
22 { "fishs_add"            , (DL_FUNC) & fishs_add
23   , 7 } ,
24 { "theory_rows_alloc"    , (DL_FUNC) & theory_rows_alloc
25   , 13} ,
26 { "theory_rows"          , (DL_FUNC) & theory_rows
27   , 17} ,
28 { "prepare_data"         , (DL_FUNC) & prepare_data
29   , 10} ,
30 { "average_power"        , (DL_FUNC) & average_power
31   , 5 } ,
32 { "deco_add"             , (DL_FUNC) & deco_add
33   , 7 } ,
34 { "average_profile"      , (DL_FUNC) & average_profile
35   , 4 } ,
36 { "dummy_add"            , (DL_FUNC) & dummy_add
37   , 10} ,
38 { "resample"             , (DL_FUNC) & resample
39   , 8 } ,
40 { "resample_R"           , (DL_FUNC) & resample_R
41   , 8 } ,
42 { "range_ambiguity"      , (DL_FUNC) & range_ambiguity
43   , 9 } ,
44 { "clutter_meas"         , (DL_FUNC) & clutter_meas
45   , 9 } ,
46 { "clutter_subtract"     , (DL_FUNC) & clutter_subtract
47   , 8 } ,
```

```
29 | { NULL , NULL , 0 }
30 | };
31 |
32 | void R_init_LPI(DllInfo *info)
33 | {
34 |     R_registerRoutines( info , NULL , callMethods , NULL , NULL
35 |         );
}
```

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.
5
6 #include "LPI.h"
7
8 /*
9
10 Ground clutter suppression. This function adds clutter
11 signal measurements to an inverse problem. The function
12 clutter_subtract subtracts clutter contribution from a
13 signal.
14
15 Arguments:
16   tcdata   Complex transmitter samples
17   tidata   Transmitter sample indices
18   rcdata   Complex receiver samples
19   ridata   Receiver sample indices
20   ndata    Data vector length
21   rmin     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 ,
31                   const SEXP rcdata , const SEXP ridata , const SEXP ndata ,
32                   const SEXP rmin , const SEXP rmax , const SEXP Qvec ,
33                   const SEXP yvec )
34 {
35   Rcomplex *tcd = COMPLEX( tcdata );
36   int *tid = LOGICAL( tidata );
37   Rcomplex *rcd = COMPLEX( rcdata );
38   int *rid = LOGICAL( ridata );
39   const int nd = *INTEGER( ndata );
40   const int r0 = *INTEGER( rmin );
41   const int r1 = *INTEGER( rmax );
42
43   int i;
44   int j;
45   int k;
46   int r;
47   int isum;
```

```

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

```

```

94      // Is this receiver sample usable?
95      if( rid[i] ){
96          // Add a measurement
97          fishs_add_clutter( Qvec , yvec , tcd , rcd , nx );
98          // Increment measurement row counter
99          ++nr;
100     }
101 }
102 }
103 }
104 // Update counters if this was not the last sample
105 if( i < nd ){
106     isum -= tid[ i - r1 ];
107     isum += tid[ i - r0 + 1 ];
108     ++r;
109     ++rcd;
110     ++tcd;
111 }
112 }
113
114 // Copy the number of rows to output
115 *INTEGER( nrow ) = nr;
116
117 UNPROTECT(1);
118
119 // Return number of measured rows
120 return( nrow );
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.
5
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
14   tidata   Transmitter sample indices
15   rcdata   Complex receiver samples
16   ridata   Receiver sample indices
17   ndata    Data vector length
18   rmin     Minimum range
19   rmax     Maximum range
20   cldata   Measured clutter signal profile
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
29                        , const SEXP rcdata , const SEXP ridata , const SEXP ndata
30                        , const SEXP rmin , const SEXP rmax , const SEXP cldata )
31 {
32   Rcomplex *tcd = COMPLEX( tcdata );
33   int *tid = LOGICAL( tidata );
34   Rcomplex *rcd = COMPLEX( rcdata );
35   int *rid = LOGICAL( ridata );
36   Rcomplex *cld = COMPLEX( cldata );
37   const int nd = *INTEGER( ndata );
38   const int r0 = *INTEGER( rmin );
39   const int r1 = *INTEGER( rmax );
40
41   int i;
42   int j;
43   int k;
44   int r;
45   int isum;
46   int nx;
47   SEXP nrow;
```

```

46  int nr;
47  Rcomplex clsum;
48  Rcomplex * tcd2;
49  Rcomplex * cld2;
50
51  // Output
52  PROTECT( nrow = allocVector( INTSXP , 1 ) );
53
54  // Initialization
55  nr = 0;
56  nx = r1 - r0 + 1;
57  r = 0;
58  isum = 0;
59  // Sum tx indices and set r
60  for( i = 0 ; i <= r1 ; ++i ){
61      // The largest range is corresponds to index 0,
62      // after nx samples we will be below rmin.
63      if( i < nx ) isum += tid[i];
64      // Increment r
65      ++r;
66      // Set 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?
81              if( isum ){
82                  // Is this receiver sample usable?
83                  if( rid[i] ){
84                      // Calculate clutter contribution and subtract it
85                      clsum.r = 0.;
86                      clsum.i = 0.;
87                      tcd2 = tcd;
88                      cld2 = cld;
89                      for( j = 0 ; j < nx ; ++j ){
90                          clsum.r += tcd2->r * cld2->r - tcd2->i * cld2->i;
91                          clsum.i += tcd2->r * cld2->i + tcd2->i * cld2->r;
92                          ++tcd2;
93                          ++cld2;
94                      }

```



```

95         rcd->r -= clsum.r;
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 ){
105     isum -= tid[ i - r1 ];
106     isum += tid[ i - r0 + 1 ];
107     ++r;
108     ++rcd;
109     ++tcd;
110 }
111 }
112
113 // Copy the number of rows to output
114 *INTEGER( nrow ) = nr;
115
116 UNPROTECT(1);
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.
5
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
14   vsum   sum of normalised inverse variances
15   rmin   Lower edge of the measurement
16   rmax   Upper edge
17   mdata  Complex measurement vector (lag profile)
18   mamb   Complex range ambiguity function
19   iamb   Range ambiguity function indices
20   iprod  Lagged product indices
21   edata  Measurement variances
22   ndata  Data vector length
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
30                , SEXP mdata , SEXP mamb , SEXP iamb , SEXP iprod , SEXP
31                edata , SEXP ndata )
32 {
33   Rcomplex *ms = COMPLEX(msum);
34   double *vs = REAL(vsum);
35   int r1 = *INTEGER(rmin);
36   int r2 = *INTEGER(rmax);
37   Rcomplex *cd = COMPLEX(mdata);
38   Rcomplex *ad = COMPLEX(mamb);
39   int *ia = LOGICAL(iamb);
40   int *ip = LOGICAL(iprod);
41   double *vd = REAL(edata);
42   int nd = *INTEGER(ndata);
43
44   int i, j, r, r0;
45
46   SEXP success;
47   int * restrict i_success;
```

```

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
54 *i_success = 1;
55
56 // Skip first r2 points, their range ambiguity function
57 // is not known
58 r = r2+1;
59 r0 = 0;
60
61 // Walk through the data vector
62 for( i = 0 ; i < nd ; ++i ){
63     // Check that we are above r1
64     if( r >= r1 ){
65         // Check that we are below r2
66         if( r < r2 ){
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;
71                 // Divide the lagged product with its variance and
72                 // multiply with TX power
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;
85         r0 = i;
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.
5
6 #include "LPI.h"
7
8 /*
9     Matched filter decoding, modified from fishs_add.
10
11     Arguments:
12         Qvec   Diagonal of the Fisher information matrix
13         yvec   Modified measurement vector
14         arows  Theory matrix rows
15         meas   Measurements
16         var    Measurement variances
17         nx     Number of unknowns
18         nrow   Number of theory rows in arows
19
20     Returns:
21         success 1 if the processing was succesful, 0 otherwise
22 */
23
24 SEXP deco_add( const SEXP Qvec , const SEXP yvec , const
25               SEXP arows , const SEXP meas , const SEXP var , const
26               SEXP nx , const SEXP nrow )
27 {
28     Rcomplex *q = COMPLEX(Qvec);
29     Rcomplex *y = COMPLEX(yvec);
30     int n = *INTEGER(nx);
31     int nr = *INTEGER(nrow);
32     int i = 0;
33     int j = 0;
34     int l = 0;
35
36     Rcomplex * restrict qtmp;
37     Rcomplex * restrict acpy = COMPLEX(arows);
38     Rcomplex * restrict atmp;
39     Rcomplex * restrict ytmp;
40     Rcomplex * restrict mcpy = COMPLEX(meas);
41     double * restrict vcpy = REAL(var);
42
43     SEXP success;
44     int * restrict i_success;
45
46     // Success output
```

```

46 PROTECT( success = allocVector( LGLSXP , 1 ) );
47
48 // Local pointer to the success output
49 i_success = LOGICAL( success );
50
51 // Set the success output
52 *i_success = 1;
53
54 // Go through all theory matrix rows
55 for( l = 0 ; l < nr ; ++l ){
56
57     // Pointers to y-vector and Fisher information matrix
58     // diagonal
59     ytmp = y;
60     qtmp = q;
61
62     // Go through all range gates
63     for( i = 0 ; i < n ; ++i ){
64
65         // Second pointer to the theory matrix
66         // (Strictly speaking not needed...)
67         atmp = acpy;
68
69         // Add information (only diaonal)
70         qtmp->r += ( acpy->r * atmp->r + acpy->i * atmp->i ) /
71             *vcpy;
72         qtmp->i += ( acpy->r * atmp->i - acpy->i * atmp->r ) /
73             *vcpy;
74
75         // Increment the second theory matrix counter
76         ++atmp;
77
78         // Increment information matrix counter (only diagonal)
79         ++qtmp;
80
81         // Add the corresponding measurement to the y-vector
82         ytmp->r += ( mcpy->r * acpy->r + mcpy->i * acpy->i ) /
83             *vcpy;
84         ytmp->i += ( mcpy->i * acpy->r - mcpy->r * acpy->i ) /
85             *vcpy;
86
87         // Increment the y-vector counter
88         ++ytmp;
89
90         // Increment the theory matrix counter
91         ++acpy;
92     }
93 }

```

```
90      // Increment the variance and measurement vector counters
91      ++mcpy;
92      ++vcpy;
93
94  }
95
96  UNPROTECT(1);
97
98  return(success);
99
100 }
```

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.
5
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
15   yvec  Modified measurement vector
16   arows Theory matrix rows
17   meas  Measurements
18   var   Measurement variances
19   nx    Number of unknowns
20   nrow  Number of theory rows in arows
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
28                SEXP arows , const SEXP meas , const SEXP var , const
29                SEXP nx , const SEXP nrow )
30 {
31   Rcomplex *q = COMPLEX(Qvec);
32   Rcomplex *y = COMPLEX(yvec);
33   int n = *INTEGER(nx);
34   int nr = *INTEGER(nrow);
35   int i = 0;
36   int j = 0;
37   int l = 0;
38
39   Rcomplex * restrict qtmp;
40   Rcomplex * restrict acpy = COMPLEX(arows);
41   Rcomplex * restrict atmp;
42   Rcomplex * restrict ytmp;
43   Rcomplex * restrict mcpy = COMPLEX(meas);
44   double * restrict vcpy = REAL(var);
45
46   SEXP success;
47   int * restrict i_success;
```



```

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
54 *i_success = 1;
55
56 // Go through all theory matrix rows
57 for( l = 0 ; l < nr ; ++l ){
58
59     // Pointers to y-vector and Fisher information matrix
60     ytmp = y;
61     qtmp = q;
62
63     // Go through all range gates
64     for( i = 0 ; i < n ; ++i ){
65
66         // Second pointer to the theory matrix
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 ) / *
                        vcpy;
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         // Add the corresponding measurement to the y-vector
84         ytmp->r += ( mcpy->r * acpy->r + mcpy->i * acpy->i ) /
                    *vcpy;
85         ytmp->i += ( mcpy->i * acpy->r - mcpy->r * acpy->i ) /
                    *vcpy;
86
87         // Increment the y-vector counter
88         ++ytmp;
89
90

```

```

91         // Increment the theory matrix counter
92         ++acpy;
93
94     }
95
96     // Increment the variance and measurement vector counters
97     ++mcpy;
98     ++vcpy;
99
100 }
101
102 UNPROTECT(1);
103
104 return(success);
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.
5
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:
14     Qvec    Upper triangular part of Fisher information matrix
15     yvec    Modified measurement vector
16     arow    One row of theory matrix
17     meas    Measurement
18     nx      Number of unknowns
19
20 */
21
22 void fishs_add_clutter( const SEXP Qvec , const SEXP yvec ,
23     Rcomplex * arow , Rcomplex * meas , const int nx )
24 {
25     Rcomplex *q = COMPLEX(Qvec);
26     Rcomplex *y = COMPLEX(yvec);
27     int n = nx;
28     int i = 0;
29     int j = 0;
30
31     Rcomplex * restrict qtmp;
32     Rcomplex * restrict acpy = arow;
33     Rcomplex * restrict atmp;
34     Rcomplex * restrict ytmp;
35     Rcomplex * restrict mcpy = meas;
36
37     // Pointers to y-vector and Fisher information matrix
38     ytmp = y;
39     qtmp = q;
40
41     // Go through all range gates
42     for( i = 0 ; i < n ; ++i ){
43
44         // Second pointer to the theory matrix
45         atmp = acpy;
46
47         // Go through all columns in the upper triangular part
```

```

47     for( j = 0 ; j < ( n - i ) ; ++j ){
48
49         // Add information
50         qtmp->r += ( acpy->r * atmp->r + acpy->i * atmp->i );
51         qtmp->i += ( acpy->r * atmp->i - acpy->i * atmp->r );
52
53         // Increment the second theory matrix counter
54         ++atmp;
55
56         // Increment the information matrix counter
57         ++qtmp;
58
59     }
60
61     // Add the corresponding measurement to the y-vector
62     ytmp->r += ( mcpy->r * acpy->r + mcpy->i * acpy->i );
63     ytmp->i += ( mcpy->i * acpy->r - mcpy->r * acpy->i );
64
65     // Increment the y-vector counter
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.
5
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]
11 samples towards larger indices. Also negative
12 shifts are allowed.
13
14 This function allocates new data vectors
15
16 Arguments:
17   idata   ndata integer vector of TX pulse / RX positions
18   ndata   Number of data points in idata
19   shifts  2-vector of shifts
20           (shifts at rising and falling edges)
21
22 Returns:
23   ans      A list with elements
24           idata   Index vector after adjustments
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;
33   SEXP s;
34   SEXP names;
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
48 inew = LOGICAL( idata_new );
49
50 // A pointer to the old data vector
51 iold = LOGICAL( idata );
52
53 // Copy data from old to new
54 for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
55     inew[k] = iold[k];
56 }
57
58 // The success logical
59 PROTECT( s = allocVector( LGLSXP , 1 ) );
60
61 // The actual work
62 s = index_adjust( idata_new , ndata , shifts );
63
64 // Collect the data into the return list
65 SET_VECTOR_ELT( ans , 0 , idata_new );
66 SET_VECTOR_ELT( ans , 1 , s );
67
68 // Set the name attributes
69 PROTECT( names = allocVector( STRSXP , 2 ));
70 SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
71 SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
72 setAttrib( ans , R_NamesSymbol , names);
73
74 UNPROTECT(4);
75
76 return(ans);
77
78 }
79
80 /*
81 Adjust TX / RX indices. The rising edges are shifted
82 shifts[0] samples and the falling edges shifts[1]
83 samples towards larger indices.
84 Also negative shifts are allowed.
85
86 This function overwrites the idata vector
87
88 Arguments:
89 idata    ndata integer vector of TX pulse / RX positions
90 ndata    Number of data points in idata
91 shifts   2-vector of shifts
92           (shifts at rising and falling edges)
93
94 Returns:
95 success 1 if all processing was successful, 0 otherwise

```

```

96
97 */
98
99 SEXP index_adjust( SEXP idata , SEXP ndata , SEXP shifts)
100 {
101     int *id = INTEGER(idata);
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;
110     // for the return value
111     SEXP success;
112     int *isuccess;
113
114     // Allocate the return value and initialise it
115     PROTECT(success = allocVector(LGLSXP,1));
116     isuccess = LOGICAL(success);
117     *isuccess = 1;
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,
123     // it will be needed later
124     lasttrue = 0;
125     for( k = ( *nd - 1 ) ; k >= 0 ; --k ){
126         if( id[k] ){
127             lasttrue = k;
128             break;
129         }
130     }
131
132     // If sh[0] < 0, shift towards smaller indices
133     if( sh[0] < 0 ){
134         for( k = 0 ; k < ( *nd + sh[0] ) ; ++k ){
135             id[k] = id[ k - sh[0] ];
136         }
137         // The last value is repeated in the remaining points
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
144     if( sh[0] > 0 ){

```

```

145     for( k = ( *nd - 1 ) ; k >= sh[0] ; --k ){
146         id[k] = id[ k - sh[0] ];
147     }
148     // The first value is repeated in the first sh[0] points
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]
155 sh1 = sh[1] - sh[0];
156
157 // If sh1 < 0 we are supposed to shift
158 // the falling edges towards smaller indices
159 if( sh1 < 0 ){
160     ncut = 0;
161     for( k = ( *nd - 1 ) ; k >= 0 ; --k ){
162         if( id[ k ] == 0 ){
163             ncut = 0;
164         }else{
165             --ncut;
166         }
167         if( ncut >= sh1 ) id[k] = 0;
168     }
169 }
170 // If sh1 > 0 we are supposed to shift
171 // the falling edges towards larger indices
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
186 // vector, correct using the stored index lasttrue
187 for( k = ( lasttrue + sh[1] + 1 ) ; k < *nd ; ++k ){
188     id[k] = 0;
189 }
190
191 // Remove protection from the return value
192 UNPROTECT(1);
193

```



```
194 | // Return the variable success only, the data is stored
195 | // in the R vectors 'cdata', 'idatar', and 'idat'ai'
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.
5
6 #include "LPI.h"
7
8 /*
9  Average power vector for variance estimation
10
11  The algorithm proceeds as follows
12
13  1. Locate falling edges of pulses from idatatx
14  2. locate the first falling edge at least maxrange samples
15     from the beginning, give this pulse the pulse index 0
16  3. Pick maxrange samples from idatatx from immediately
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)
20  5. If the vectors compared in (4) are identical, also this
21     pulse is given pulse index 0, repeat for all pulses
22  6. If there pulses are left without an index, select the
23     first of them and repeat steps (4) and (5) to give
24     these pulses the index 1.
25  7. Continue with indices 2, 3, ...
26     until all pulses have an index
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  ndata      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
40             the
41             ratio largest pulse index / number of pulses.
42  */
43
44 SEXP average_power( SEXP cdata , SEXP idatatx , SEXP idatarx
45                   , SEXP ndata , SEXP maxrange)
```

```

46 Rcomplex * cd = COMPLEX( cdata );
47 int * idtx = LOGICAL( idatatx );
48 int * idrx = LOGICAL( idatarx );
49 int nd = *INTEGER( ndata );
50 int maxr = *INTEGER( maxrange );
51
52 SEXP pdata;
53 double *pd;
54 double *ptmp;
55 int *pedges;
56 int nedges;
57 int *pinds;
58 int *nsamp;
59 int k, i, j;
60 int pindcur;
61 int pindmax;
62 int npulse;
63 int indprev;
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
85 for( k = 0 ; k < nd ; ++k ) pd[ k ] = 0.;
86
87 // Allocate a temporary vector for
88 // power profile calculation
89 ptmp = R_Calloc( nd , double );
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
98 for( k = 0 ; k < nd ; ++k ) nsamp[ k ] = 0;
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
105 for( k = 0 ; k < nd ; ++k ) pedges[ k ] = 0;
106
107 // Allocate a vector for pulse indices
108 pinds = R_Calloc( nd , int );
109
110 // Initialise to -1
111 for( k = 0 ; k < nd ; ++k ) pinds[ k ] = -1;
112
113
114 // Locate all falling edges of pulses
115 nedges = 0;
116 for( k = 0 ; k < ( nd - 1 ) ; ++k )
117 {
118     if( idtx[ k ] )
119     {
120         if( !(idtx[ k + 1] ) )
121         {
122             pedges[ nedges++ ] = k;
123         }
124     }
125 }
126
127 // The first falling pulse edge at least
128 // maxr samples from the beginning
129 for( k = 0 ; k < nedges ; ++k )
130 {
131     if( pedges[ k ] > maxr )
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 )
145     {
146         // Go through all the pulses
147         for( i = k ; i < nedges ; ++i )
148         {
149             // Compare only with pulses that
150             // do not yet have an index
151             if( pinds[ i ] < 0 )
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[
158                         i ] - j ] ) )
159                     {
160                         sameamb = 0;
161                         break;
162                     }
163                     // If the ambiguities were identical,
164                     // assign the pulse with the index pindcur
165                     if( sameamb ) pinds[ i ] = pindcur;
166                 }
167             }
168             // Increment pindcur
169             ++pindcur;
170         }
171     }
172
173     // There may be a pulse / pulses without an index
174     // in the begin of data vector.
175     // Give them an index if possible
176     if( p1 > 0 )
177     {
178         for( i = p1 ; i < nedges ; ++i )
179         {
180             sameamb = 1;
181             for( j = 0 ; j < pedges[ p1 - 1 ] ; ++j )
182             {
183                 if( idtx[ pedges[ p1 - 1 ] - j ] != idtx[ pedges[ i
184                     ] - j ] )
185                 {
186                     sameamb = 0;
187                     break;
188                 }
189             }
190             if( sameamb )
191             {

```

```

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

```

```

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

```

```

282
283 // Go through the indices again and copy the power
284 // values to appropriate places Set pinds to -1 at
285 // points that have already been visited
286 pindcur = pinds[ k ];
287 for( j = k ; j < nedges ; ++j )
288 {
289     if( pinds[ j ] == pindcur )
290     {
291         if( ( j + 1 ) >= nedges )
292         {
293             ippend = nd - pedges[ j ];
294         }
295         else
296         {
297             ippend = pedges[ j + 1 ] - pedges[ j ];
298         }
299         for( i = 0 ; i < ippend ; ++i )
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
313 // enough averaged samples (they are set to -1 at this
314 // point)
315 ptot /= (float)ntot;
316 for( i = 0 ; i < nd ; ++i ){
317     if( pd[ i ] < 0.) pd[ i ] = ptot;
318 }
319
320 // Store the ratio pindmax / nedges to the first data point
321 //
322 // If the ratio is large the power estimation will not
323 // perform
324 // very well.
325 // The power value in this point cannot ever be needed in
326 // LPI.
327 pd[0] = (float)pindmax / (float)nedges;
328
329 // Free the temporary allocations
330 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.
5
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:
14  cdata1  ndata1 vector of complex signal samples
15  cdata2  ndata2 vector of complex signal samples
16  idata1  ndata1 integer vector of usable
17          RX sample positions
18  idata2  ndata2 integer vector of usable
19          RX sample positions
20  ndata1  Number of samples in cdata1 and idata1
21  ndata2  Number of samples in cdata2 and idata2
22  lag     Lag
23
24  Returns:
25  ans     A list with elements
26          cdata  Complex vector of lagged products
27          idata  Index vector for cdata
28          ndata  Data vector length
29          success Logical, set if all processing
30                was successful
31 */
32
33 SEXP lagged_products_alloc( SEXP cdata1 , SEXP cdata2 , SEXP
34   idata1 , SEXP idata2 , SEXP ndata1 , SEXP ndata2 , SEXP
35   lag)
36 {
37   Rcomplex *cd1 = COMPLEX(cdata1);
38   Rcomplex *cd2 = COMPLEX(cdata2);
39   int *id1 = LOGICAL(idata1);
40   int *id2 = LOGICAL(idata2);
41   int *nd1 = INTEGER(ndata1);
42   int *nd2 = INTEGER(ndata2);
43   int *l = INTEGER(lag);
44
45   SEXP ans;
46   SEXP lcdata;
47   Rcomplex *lcd;
```

```

46  SEXP lidata;
47  int *lid;
48  SEXP success;
49  int *isuccess;
50  SEXP ndata;
51  int *nd;
52  SEXP names;
53  char *cnames[4] = {"cdata","idata","ndata","success"};
54  int k=0;
55
56  // Allocate the return value list
57  PROTECT( ans = allocVector( VECSXP , 4 ) );
58
59  // Allocate the ndata output
60  PROTECT( ndata = allocVector( INTSXP , 1 ) );
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
67  *nd = *nd1 - *l;
68  if( *nd1 > *nd2 ) *nd = *nd2 - *l;
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
80  lid = LOGICAL( lidata );
81
82  // Allocate the success return value
83  PROTECT( success = allocVector( LGLSXP , 1 ) );
84
85  // A local pointer to the success value
86  isuccess = LOGICAL( success );
87  *isuccess = 1;
88
89  // The actual lagged product calculation
90  for( k = 0 ; k < *nd ; ++k ){
91
92      // Calculate the index vector point
93      lid[k] = (id1[k] * id2[k+ *l]);
94

```

```

95     // Calculate the actual data product only if the index
        vector is set
96     if(lid[k]){
97         lcd[k].r = cd1[k].r * cd2[k+ *l].r + cd1[k].i * cd2[k+
            *l].i;
98         lcd[k].i = -cd1[k].r * cd2[k+ *l].i + cd1[k].i * cd2[k+
            *l].r;
99     }
100 }
101
102
103 // Collect the return values under the list "ans"
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
110 PROTECT( names = allocVector( STRSXP , 4 ) );
111 SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
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:
133 cdata1  ndata1 vector of complex signal samples
134 cdata2  ndata2 vector of complex signal samples
135 idata1  ndata1 integer vector of usable
136         RX sample positions
137 idata2  ndata2 integer vector of usable
138         RX sample positions
139 cdatap  complex vector for the lagged products
140 idatap  integer vector for the lagged product indices

```

```

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

```

```

188         cdp[k].r = cd1[k].r * cd2[k+ 1].r + cd1[k].i * cd2[k+ 1
           ].i;
189         cdp[k].i = cd1[k].r * cd2[k+ 1].i - cd1[k].i * cd2[k+ 1
           ].r;
190     }
191 }
192
193 // Set the logical vector to false at
194 // points where it cannot be calculated
195 for( k = 0 ; k < 1 ; ++k ){
196     idp[npr+k] = 0;
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
217 rdata2  ndata2 vector of real signal samples
218 prdata  real vector for the lagged products
219 ndata1  Number of samples in rdata1
220 ndata2  Number of samples in rdata2
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 int      l        = *INTEGER(lag)   ;
236 SEXP     success   ;
237 int      *isuccess ;
238 int      k          = 0              ;
239 int      npr        ;
240
241 // Output data length will be minimum of the two input
242 // data lengths, minus the time-lag
243 npr = nd1 - 1;
244 if( nd1 > nd2 ) npr = nd2 - 1;
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
253 // The actual lagged product calculation
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.
5
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
12 the repeated code cycle
13
14 The complicated structure is used because
15 measurements may contain additional sync
16 times which need to be skipped.
17
18 Arguments:
19   cdata  Complex lagged product vector
20   idata  Index vector for cdata
21   ndata  Data vector length
22   N_CODE Code cycle length
23
24 Returns:
25   success 1 if the processing was successful, 0 otherwise
26 */
27
28 SEXP average_profile( SEXP cdata , SEXP idata , SEXP ndata ,
29                      SEXP N_CODE)
30 {
31   Rcomplex * cd = COMPLEX( cdata );
32   int * id = LOGICAL( idata );
33   int nd = *INTEGER( ndata );
34   int ncode = *INTEGER( N_CODE );
35
36   double *aver;
37   double *avei;
38   Rcomplex *ad;
39   R_len_t *nave;
40   R_len_t k;
41   R_len_t ind1 , ind2, ipp_count;
42   SEXP success;
43   int *isuccess;
44
45   // Allocate the return value and initialise it
```



```

47 PROTECT(success = allocVector(LGLSXP,1));
48 isuccess = LOGICAL(success);
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
57 for( k = 0 ; k < nd ; ++k ){
58     aver[ k ] = 0.;
59     avei[ k ] = 0.;
60 }
61
62 // Allocate vector for data sample counter
63 nave = R_Calloc( nd , R_len_t );
64
65 // Initialise to zero
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
73 while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;
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
81     // we hit a point at which both pulses have ended.
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;
88         if(ind2==nd) break;
89     }
90
91     if(ind2==nd) break;
92
93     // Add power values until either of the indices
94     // hits the next pulse
95     while( (id[ind1]==0) & (id[ind2]==0)){

```

```

96         aver[ind1] += cd[ind2].r;
97         avei[ind1] += cd[ind2].i;
98         ++nave[ind1];
99         ++ind1;
100        ++ind2;
101        if(ind2==nd) break;
102    }
103
104    if(ind2==nd) break;
105
106    // Make sure that both indices point to a pulse start,
107    // increment if necessary (This takes possible sync
108    // times into account)
109    while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;
110    while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;
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 furhter analysis
134 // simpler. Start from beginning of the data vector.
135 ind1 = 0;
136 ind2 = 0;
137
138 // Search for the start of the first pulse
139 while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;
140 while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;
141 ipp_count = 0;
142
143 // Repeat until end of data
144 while( ind2 < nd ){

```

```

145 // At this point we should be at pulse starts,
146 // loop until both pulses have ended
147 while( id[ind1] | id[ind2]){
148     cd[ind2].r = aver[ind1];
149     cd[ind2].i = avei[ind1];
150     ++ind1;
151     ++ind2;
152     if(ind2==nd) break;
153 }
154
155 if(ind2==nd) break;
156
157 // Add power values until either of
158 // the indices hits the next pulse
159 while( (id[ind1]==0) & (id[ind2]==0)){
160     cd[ind2].r = aver[ind1];
161     cd[ind2].i = avei[ind1];
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
170 while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;
171 while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;
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);
186 Free(aver);
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.
5
6 #include "LPI.h"
7
8 /*
9  Frequency mixing for IQ data
10
11  This function allocates new vectors
12
13  Argumnets:
14      cdata      ndata complex vector of data samples
15      ndata      Number of samples in cdata
16      frequency   The mixing frequency
17
18  Returns:
19      ans        A list with elements
20                  cdata      Complex data samples after
21                          frequency mixing
22                  success Logical, set if all processing
23                          was successful
24 */
25
26
27 SEXP mix_frequency_R( SEXP cdata , SEXP ndata , SEXP
28     frequency )
29 {
30     SEXP ans;
31     SEXP cdata_new;
32     SEXP s;
33     SEXP names;
34     char *cnames[2] = {"cdata","success"};
35     Rcomplex *cnew;
36     Rcomplex *cold;
37     register uint64_t k;
38
39     // Output list ans[[1]] = cdata , ans[[2]] = success
40     PROTECT( ans = allocVector( VECSXP , 2 ) );
41
42     // Allocate the new complex vector
43     PROTECT( cdata_new = allocVector( CPLXSXP , *(INTEGER(ndata
44         )) ) );
45
46     // A pointer to the new data vector
```

```

46 cnew = COMPLEX( cdata_new );
47
48 // A pointer to the old data vector
49 cold = COMPLEX( cdata );
50
51 // Copy data from old to new
52 for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
53     cnew[k].r = cold[k].r;
54     cnew[k].i = cold[k].i;
55 }
56
57 // The success logical
58 PROTECT( s = allocVector( LGLSXP , 1 ) );
59
60 // The actual frequency mixing
61 s = mix_frequency( cdata_new , ndata , frequency );
62
63 // Collect the data into the return list
64 SET_VECTOR_ELT( ans , 0 , cdata_new );
65 SET_VECTOR_ELT( ans , 1 , s );
66
67 // Set the name attributes
68 PROTECT( names = allocVector( STRSXP , 2 ));
69 SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
70 SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
71 setAttrib( ans , R_NamesSymbol , names);
72
73 UNPROTECT(4);
74
75 return(ans);
76
77 }
78
79 /*
80 Frequency mixing for IQ data
81
82 This function overwrites the cdata vector
83
84 Argumnets:
85     cdata      ndata complex vector of data samples
86     ndata      Number of samples in cdata
87     frequency   The mixing frequency
88
89 Returns:
90     success    1 if all processing was successful, 0 otherwise
91 */
92 SEXP mix_frequency( SEXP cdata , SEXP ndata , SEXP frequency)
93 {
94     // Pointers to the R variables

```

```

95  Rcomplex *cd = COMPLEX(cdata);
96  int *nd = INTEGER(ndata);
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;
104 double idiff;
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);
120     idiff = tmpprod - (double)((int)(tmpprod));
121     if( fabs(idiff) <= FLT_MIN ){
122         ncycle = k;
123         break;
124     }
125 }
126
127 // If the cycle length is one, the mixing would not change
    anything
128 if( ncycle == 1 ){
129     UNPROTECT(1);
130     return(success);
131 }
132
133 // Tabulate the cyclic coefficients.
134 // This usually saves time as radar engineers tend to
135 // select nice numerical values for the frequencies
136 coefr = (double*) R_Calloc( ncycle , double );
137 coefi = (double*) R_Calloc( ncycle , double );
138 for( k = 0 ; k < ncycle ; ++k ){
139     arg      = 2.0 * pi * *fr * (double)(k);
140     coefr[k] = cos(arg);
141     coefi[k] = sin(arg);
142 }

```

```

143
144 // Actual mixing
145 nc = 0;
146 for( k = 0 ; k < *nd ; ++k ){
147     tmp.r = cd[k].r;
148     tmp.i = cd[k].i;
149     cd[k].r = tmp.r * coefr[nc] - tmp.i * coefi[nc];
150     cd[k].i = tmp.i * coefr[nc] + tmp.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
162 // Return the variable success only, the data is stored in
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.
5
6 #include "LPI.h"
7
8 /*
9  Resampling with linear interpolation. Reduces to a simple
10 boxcar filter when the filter length is an integer
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
20   idata    Index vector for cdata
21   ndata    Data vector length
22   nup      Upsampling factor
23   nfilter  Filter length on upsampled data
24             (final length is nfilter / nup)
25   nfirst   Decimation start index
26   nfirstfrac start point within the boxcar filter in
27             upsampled units
28   ipartial 0 if partial matched with filter
29             should not be accepted in idata vector
30
31 Returns:
32   success  1 if resampling was successful, 0 otherwise
33 */
34
35 SEXP resample( SEXP cdata , SEXP idata , SEXP ndata , SEXP
36               nup , SEXP nfilter , SEXP nfirst , SEXP nfirstfrac, SEXP
37               ipartial )
38 {
39   Rcomplex * restrict cd = COMPLEX(cdata);
40   int * restrict id = LOGICAL(idata);
41   int nd = *INTEGER(ndata);
42   const int nu = *INTEGER(nup);
43   const int nf = *INTEGER(nfilter);
44   const int ns = *INTEGER(nfirst);
45   const int nsf = *INTEGER(nfirstfrac);
```



```

45  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
55  // Allocate the return value and initialise it
56  PROTECT(success = allocVector(LGLSXP,1));
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
63   k the current point within the original data vector
64   l the current point within the resampled data vector
65  */
66
67  i = ns * nu ; // Starting point in upsampled units
68  // j = nu-1; // We are originally at the
69  // // beginning of the boxcar filter
70  j = nsf+nu-1; // increment with nu-1, we will use the full
    sample
71
    // cd[k] in any case. The first resampled
    one will
72
    // be wrong if nsf/=0, but we could not help
    this if
73
    // nsf < 0 in any case.
74  k = ns; // Starting point in original sampling
75  l = 0; // Current point in the final filtered and
76
    // decimated data vector, start filling
77
    // from beginning
78  tmpsum.r = 0.; // Initialise the temp filter sum to zero
79  tmpsum.i = 0.;
80  tmpi[0] = 1;
81  tmpi[1] = 0;
82
83
84  while( ( ( i + nf ) / nu ) <= nd ){ // Current filter start
    + filter length <= data length
85    while( j < nf ){ // One filter length of data
86      tmpsum.r += cd[ k ].r; // Add the current point to the
        filter sum
87      tmpsum.i += cd[ k ].i;
88      tmpi[0] *= id[k];

```

```

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

```

```

133     j -= nf;
134     // The sample where we ended in the previous step was
135     // already added to tmpsum, jump to the next one
136     j += nu;
137     // Move one filter length forwards
138     /*
139     i += nf;
140     ++k;
141     */
142     ++l;
143 }
144
145 // i and k must be incremented also at end of data to get
    us out of the loop
146 i += nf;
147 ++k;
148 }
149
150 // If we were exactly at end of data frac is unity, we will
    still get one more sample
151 // k was incremented after hitting the end of data
152 if( k == ( nd + 1 ) ){
153     if( frac > .9999999 ){
154         cd[l].r = tmpsum.r;
155         cd[l].i = tmpsum.i;
156         id[l] = ipar ? tmpi[1] : tmpi[0];
157         ++l;
158     }
159 }
160
161 *(INTEGER(ndata)) = l;
162
163 // remove protection from the return value
164 UNPROTECT(1);
165
166 // return the variable success only, the data is now stored
167 // in the R vectors 'cdata', 'idatar', and 'idatai'
168 return(success);
169
170 }
171
172
173 /*
174 Resampling with linear interpolation. Reduces to a simple
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   cdata    Complex data samples
185   idata    Index vector for cdata
186   ndata    Data vector length
187   nup      Upsampling factor
188   nfilter  Filter length on upsampled data (final length
189           is nfilter / nup)
190   nfirst   Decimation start index
191   ipartial 0 if partial matched with filter should not be
192           accepted in idata vector
193
194 Returns:
195   ans      A list with components:
196           cdata    Resampled complex data vector
197           idata    Index vector for cdata
198           ndata    Data vector length
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;
209   SEXP idata_new;
210   SEXP ndata_new;
211   SEXP s;
212   SEXP names;
213   char *cnames[4] = {"cdata","idata","ndata","success"};
214   Rcomplex * restrict cnew;
215   Rcomplex * restrict cold;
216   int * restrict inew;
217   int * restrict iold;
218   uint64_t k;
219   PROTECT_INDEX cpind=0;
220   PROTECT_INDEX ipind=0;
221
222
223   // Output list ans[[1]] = cdata , ans[[2]] = idata ,
224   // ans[[3]] = ndata , ans[[4]] = success
225   PROTECT( ans = allocVector( VECSXP , 4 ) );
226

```

```

227 // Allocate the new complex vector
228 PROTECT_WITH_INDEX( cdata_new = allocVector( CPLXXP , *(
      INTEGER(ndata)) ) , &cpind );
229
230 // Allocate the new logical vector
231 PROTECT_WITH_INDEX( idata_new = allocVector( LGLXP , *(
      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
243 inew = LOGICAL( idata_new );
244
245 // A pointer to the old idata vector
246 iold = LOGICAL( idata );
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;
251     cnew[k].i = cold[k].i;
252 }
253
254 // Copy data from old idata to new idata
255 for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
256     inew[k] = iold[k];
257 }
258
259 // Use the same pointers to copy old ndata to new ndata
260 inew = INTEGER( ndata_new );
261 iold = INTEGER( ndata );
262 *inew = *iold;
263
264 // The success logical
265 PROTECT( s = allocVector( LGLXP , 1 ) );
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
271 SET_LENGTH( cdata_new , *INTEGER(ndata_new) );
272 REPROTECT( cdata_new , cpind );

```

```

273 SET_LENGTH( idata_new , *INTEGER(ndata_new) );
274 REPROTECT( idata_new , ipind );
275
276 // Collect the data into the return list
277 SET_VECTOR_ELT( ans , 0 , cdata_new );
278 SET_VECTOR_ELT( ans , 1 , idata_new );
279 SET_VECTOR_ELT( ans , 2 , ndata_new );
280 SET_VECTOR_ELT( ans , 3 , s );
281
282 // Set the name attributes
283 PROTECT( names = allocVector( STRSXP , 4 ));
284 SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
285 SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
286 SET_STRING_ELT( names , 2 , mkChar( cnames[2] ) );
287 SET_STRING_ELT( names , 3 , mkChar( cnames[3] ) );
288 setAttrib( ans , R_NamesSymbol , names);
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.
5
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
14      idata      Integer vector of usable data indices
15      ndata      Data vector length
16      frequency  Frequency offset
17      shifts     Corrections to idata
18      nup        Upsampling factor in resampling
19      nfilter    Filter length (for upsampled data, final
20                  filter length is nfilter / nup)
21      nfirst     Decimation start index
22      ipartial   Logical, are partial matches of
23                  idata with the filter accepted?
24
25  Returns:
26      ans        A list with elements
27                  cdata    Final complex data vector
28                  idata    Final index vector
29                  ndata    Final data vector length
30                  success  Logical, set if all processing
31                          was successfull
32
33  */
34
35
36 SEXP prepare_data( SEXP cdata , SEXP idata , SEXP ndata ,
37                   SEXP frequency, SEXP shifts , SEXP nup , SEXP nfilter ,
38                   SEXP nfirst , SEXP nfirstfrac , SEXP ipartial )
39 {
40     SEXP ans;
41     SEXP cdata_new;
42     SEXP idata_new;
43     SEXP ndata_new;
44     SEXP s;
45     SEXP names;
46     char *cnames[4] = {"cdata","idata","ndata","success"};
47     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 ,
56 // ans[[3]] = idata , ans[[4]] = ndata , ans[[5]] = success
57 PROTECT( ans = allocVector( VECSXP , 5 ) );
58
59 // Allocate the new complex vector
60 PROTECT_WITH_INDEX( cdata_new = allocVector( CPLXSCP , *(
    INTEGER(ndata)) ) , &cpind );
61
62 // Allocate the new logical vector
63 PROTECT_WITH_INDEX( idata_new = allocVector( LGLSCP , *(
    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
69 cnew = COMPLEX( cdata_new );
70
71 // A pointer to the old cdata vector
72 cold = COMPLEX( cdata );
73
74 // A pointer to the new idata vector
75 inew = LOGICAL( idata_new );
76
77 // A pointer to the old idata vector
78 iold = LOGICAL( idata );
79
80 // Copy data from old cdata to new cdata
81 for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
82     cnew[k].r = cold[k].r;
83     cnew[k].i = cold[k].i;
84 }
85
86 // Copy data from old idata to new idata
87 for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
88     inew[k] = iold[k];
89 }
90
91 // Use the same pointers to copy old ndata to new ndata
92 inew = INTEGER( ndata_new );

```



```

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
103 s = index_adjust( idata_new , ndata_new , shifts );
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 );
110 for( k = 0 ; k < *INTEGER(ndata_new) ; ++k ){
111     if( inew[k] == 0 ){
112         cnew[k].r = .0;
113         cnew[k].i = .0;
114     }
115 }
116
117 // Reallocate the vectors to match with the new data length
118 SET_LENGTH( cdata_new , *INTEGER(ndata_new) );
119 REPROTECT( cdata_new , cpind );
120 SET_LENGTH( idata_new , *INTEGER(ndata_new) );
121 REPROTECT( idata_new , ipind );
122
123 // Collect the data into the return list
124 SET_VECTOR_ELT( ans , 0 , cdata_new );
125 SET_VECTOR_ELT( ans , 1 , idata_new );
126 SET_VECTOR_ELT( ans , 2 , ndata_new );
127 SET_VECTOR_ELT( ans , 3 , s );
128
129 // Set the name attributes
130 PROTECT( names = allocVector( STRSXP , 4 ));
131 SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
132 SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
133 SET_STRING_ELT( names , 2 , mkChar( cnames[2] ) );
134 SET_STRING_ELT( names , 3 , mkChar( cnames[3] ) );
135 setAttrib( ans , R_NamesSymbol , names);
136
137 UNPROTECT(6);
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.
5
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:
14      camb          Complex range ambiguity functions
15      iamb          Index vector of range ambiguity functions
16      cprod         Complex lagged product vector
17      iprod         Index vector of lagged products
18      rvar          Measurement variance vector
19      ndata         Data vector length
20      ncur          Current sample index
21      nend          Last sample index to use (in this call)
22      rlims         Range gate limits
23      nranges       Number of range gates
24      fitsize       0 if the vectors should not be reallocated to
25                   match the final data size.
26      background    0 if additional background term is not used
27      remoterx      0 if measurements TX times should not be used
28
29
30  Returns:
31      ans          A list with elements
32                   arows    Theory matrix rows
33                   irows    Theory row indices
34                   m        Inversion measurement vector
35                   var       Measurement variances
36                   nrows    Number of theory rows produced
37                   success   Logical, set if all processing
38                           was successful
39
40  */
41
42 SEXP theory_rows_alloc( SEXP camb , SEXP iamb , SEXP cprod ,
43                        SEXP iprod , SEXP rvar , SEXP ndata , SEXP ncur , SEXP
44                        nend , SEXP rlims , SEXP nranges , SEXP fitsize , SEXP
45                        background , SEXP remoterx )
46 {
47     const int n_cur = *INTEGER(ncur);
```

```

45  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;
54  SEXP nrows;
55  SEXP names;
56  int n_rows;
57  const char * c_names[6] = {"arows","irows","m","var","
    nrows","success"};
58  PROTECT_INDEX arind = 0;
59  PROTECT_INDEX irind = 0;
60  PROTECT_INDEX mind = 0;
61  PROTECT_INDEX vind = 0;
62
63
64  // Output list
65  PROTECT( ans = allocVector( VECSXP , 5 ) );
66
67  // A vector for the theory matrix rows
68  PROTECT_WITH_INDEX( arows = allocVector( CPLXSXP , ( (
    n_end - n_cur + 1 ) * ( n_ranges + 1 ) ) ) , & arind );
69
70  // A vector for the theory matrix indices
71  PROTECT_WITH_INDEX( irows = allocVector( LGLSXP , ( ( n_end
    - 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
77  PROTECT_WITH_INDEX( mvar = allocVector( REALSXP , ( n_end -
    n_cur + 1 ) ) , & vind );
78
79  // Number of rows for the R output
80  PROTECT( nrows = allocVector( INTSXP , 1 ) );
81
82  // Success output
83  PROTECT( success = allocVector( LGLSXP , 1 ) );
84
85  // Call the theory_rows function to actually make the rows
86  success = theory_rows( camb , iamb , cprod , iprod , rvar ,
    ndata , ncur , nend , rlms ,\

```

```

87         nranges , arows , irows , mvec ,
            mvar , nrows , background ,
            remoterx );
88
89 // Read the row count
90 n_rows = *(INTEGER(nrows));
91
92 // Reallocate the vectors to match with the data lengths
93 if(fit_size){
94     SET_LENGTH( arows , ( n_rows * ( n_ranges + 1 ) ) );
95     REPROTECT( arows , arind );
96     SET_LENGTH( irows , ( n_rows * ( n_ranges + 1 ) ) );
97     REPROTECT( irows , irind );
98     SET_LENGTH( mvec , n_rows );
99     REPROTECT( mvec , mind );
100    SET_LENGTH( mvar , n_rows );
101    REPROTECT( mvar , vind );
102 }
103
104 // Collect the data into the return list
105 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 );
110 SET_VECTOR_ELT( ans , 5 , success );
111
112 // Set the names attributes
113 PROTECT( names = allocVector( STRSXP , 5 ) );
114 SET_STRING_ELT( names , 0 , mkChar( c_names[0] ) );
115 SET_STRING_ELT( names , 1 , mkChar( c_names[1] ) );
116 SET_STRING_ELT( names , 2 , mkChar( c_names[2] ) );
117 SET_STRING_ELT( names , 3 , mkChar( c_names[3] ) );
118 SET_STRING_ELT( names , 4 , mkChar( c_names[4] ) );
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     cprod          Complex lagged product vector
142     iprod          Index vector of lagged products
143     rvar           Measurement variance vector
144     ndata          Data vector length
145     ncur           Current sample index
146     nend           Last sample index to use (in this call)
147     rlims          Range gate limits
148     nranges        Number of range gates
149     arows          Complex theory rows
150     irows          Theory row indices
151     mvec           Inversion measurement vector
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       0 if measurements TX times should not be used
157
158 Returns:
159     success        0 if no theory rows were produced _and_ end of
160                   data was reached, 1 otherwise
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);
170     const double * restrict var = REAL(rvar);
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;
189 R_len_t subi;
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
199 n_end = ( n_data > n_end ? n_end : n_data );
200
201 // Check that n_cur <= n_data
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
218 // will not be used in the theory matrix
219 // Initialize r_min for monostatic reception
220 r_lim = r_min;
221 // -1 (all samples accepted) 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
231 if( n_start < r_lims[ n_ranges ] ) n_start = r_lims[
    n_ranges ];
232
233 // Make sure that we did not yet pass the end point
234 if( n_start < n_end ){
235     // Go through all range-gates
236     for( i = 0 ; i < n_ranges ; ++i ){
237         // Initialize the theory matrix to zero
238         a_rows[i].r = .0;
239         a_rows[i].i = .0;
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;
252                 a_rows[i].i += amb[ n_start - j ].i;
253                 i_rows[i] += amb_i[ n_start - j ];
254             }
255         }
256     }
257
258     // The last gate will be 1 or 0, depending on whether
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
274 // Number of stored rows
275 n_rows = 0;

```



```

276
277 // Range from the latest pulse
278 r_cur = r_max;
279 for( k = (n_start-r_min) ; k < n_start ; ++k ){
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)
294     if( (prod_i[k] != 0) & (r_cur > r_lim) & (r_cur < r_max))
295     {
296
297         // Copy data to the measurement vector
298         m_vec[n_rows].r = prod[k].r;
299         m_vec[n_rows].i = prod[k].i;
300         m_var[n_rows] = var[k];
301
302         // Copy the current theory vectors to the next one.
303         for( i = 0 ; i < ( n_ranges + 1 ) ; ++i ){
304             i_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ] =
305                 i_rows[ n_rows * ( n_ranges + 1 ) + i ];
306             // Set the theory rows exactly to zero at points
307             // where the index vector is zero. This makes
308             // identification of blind ranges much easier.
309             if(i_rows[ n_rows * ( n_ranges + 1 ) + i ]==0){
310                 a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].r =
311                     0.0;
312                 a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].i =
313                     0.0;
314                 a_rows[ n_rows * ( n_ranges + 1 ) + i ].r = 0.0;
315                 a_rows[ n_rows * ( n_ranges + 1 ) + i ].i = 0.0;
316             }
317             // Otherwise copy the theory matrix row
318             }else{
319                 a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].r =
320                     a_rows[ n_rows * ( n_ranges + 1 ) + i ].r;
321                 a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].i =
322                     a_rows[ n_rows * ( n_ranges + 1 ) + i ].i;
323             }
324         }
325     }
326 }

```

```

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)
329         gati = n_rows * ( n_ranges + 1 ) + i;
330         // Index of the data point that
331         // will be added to this gate
332         addi = k - r_lims[i] + 1;
333         // Index of the data point that
334         // will be subtracted from this gate
335         subi = k - r_lims[i+1] + 1;
336
337         // Do additions / subtractions only if the point
338         // contains a non-zero ambiguity value
339         if( amb_i[ addi ] ){
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
353 // echoes from below the first gate
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( n_rows ) ) = n_rows;
364
365 // Update the current position in the data vector
366 *( INTEGER( ncur ) ) = n_end;
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.
5
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
14   cdata2  Second complex transmitter samples
15   idata1  First transmitter sample indices
16   idata2  Seconds transmitter sample indices
17   cdatap  Complex range ambiguity function
18   idatap  Range ambiguity index vector
19   ndata1  Length of vectors cdata1 and idata1
20   ndata2  Length of vectors cdata2 and idata2
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
29                      , SEXP idata2 , SEXP cdatap , SEXP idatap , SEXP ndata1 ,
30                      SEXP ndata2 , SEXP lag )
31 {
32   Rcomplex *cd1 = COMPLEX(cdata1);
33   Rcomplex *cd2 = COMPLEX(cdata2);
34   int *id1 = LOGICAL(idata1);
35   int *id2 = LOGICAL(idata2);
36   Rcomplex *cdp = COMPLEX(cdatap);
37   int *idp = LOGICAL(idatap);
38   int nd1 = *INTEGER(ndata1);
39   int nd2 = *INTEGER(ndata2);
40   int l = *INTEGER(lag);
41   SEXP success;
42   int *isuccess;
43   int k = 0;
44   int npr;
45   int ninterp = AMB_N_INTERP;
46   int i;
47   double * tmpr1;
```

```

46 double * tmpi1;
47 double * tmpr2;
48 double * tmpi2;
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 );
54 tmpi2 = (double*) R_Calloc( 2*ninterp , double );
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
61 // Allocate the success return value
62 PROTECT( success = allocVector( LGLSXP , 1 ) );
63
64 // A local pointer to the success value
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
75         for( i = 0 ; i < ( 2 * ninterp ) ; ++i ){
76             tmpr1[i] = .0;
77             tmpi1[i] = .0;
78             tmpr2[i] = .0;
79             tmpi2[i] = .0;
80         }
81         // Linear interpolation towards the previous data point
82         if( k > 1 ){
83             for( i = 0 ; i < ninterp ; ++i ){
84                 tmpr1[i] = cd1[k-1].r + ( cd1[k].r - cd1[k-1].r ) * (
85                     1. - (double)i / (double)( 2 * ninterp ) );
86                 tmpi1[i] = cd1[k-1].i + ( cd1[k].i - cd1[k-1].i ) * (
87                     1. - (double)i / (double)( 2 * ninterp ) );
88                 tmpr2[i] = cd2[k-1+1].r + ( cd2[k+1].r - cd2[k-1+1].r )
89                     * ( 1. - (double)i / (double)( 2 * ninterp ) );
90                 tmpi2[i] = cd2[k-1+1].i + ( cd2[k+1].i - cd2[k-1+1].i )
91                     * ( 1. - (double)i / (double)( 2 * ninterp ) );
92             }
93         }
94         // Linear interpolation towards the next data point

```

```

91     if( k < npr ){
92         for( i = 0 ; i < ninterp ; ++i ){
93             tmpr1[i+ninterp] = cd1[k].r + ( cd1[k+1].r - cd1[k
94             ].r ) * ( (double)i / (double)( 2 * ninterp ) );
95             tmpi1[i+ninterp] = cd1[k].i + ( cd1[k+1].i - cd1[k
96             ].i ) * ( (double)i / (double)( 2 * ninterp ) );
97             tmpr2[i+ninterp] = cd2[k+1].r + ( cd2[k+1+1].r -
98             cd2[k+1].r ) * ( (double)i / (double)( 2 *
99             ninterp ) );
100            tmpi2[i+ninterp] = cd2[k+1].i + ( cd2[k+1+1].i -
101            cd2[k+1].i ) * ( (double)i / (double)( 2 *
102            ninterp ) );
103        }
104    }
105    // Initialize the final data value to zero
106    cdp[k].r = .0;
107    cdp[k].i = .0;
108    // Add products of the interpolated data
109    for( i = 0 ; i < ( 2 * ninterp ) ; ++i ){
110        cdp[k].r += tmpr1[i] * tmpr2[i] + tmpi1[i] * tmpi2[i
111        ];
112        cdp[k].i += tmpr1[i] * tmpi2[i] - tmpi1[i] * tmpr2[i
113        ];
114    }
115    // Divide with number of summed values
116    cdp[k].r /= (double)(2*ninterp);
117    cdp[k].i /= (double)(2*ninterp);
118 }
119 }
120 // Set l index values from the beginning to false
121 for( k = 0 ; k < l ; ++k ){
122     idp[npr+k] = 0;
123 }
124 // Free the temporary vectors
125 Free(tmpr1);
126 Free(tmpi1);
127 Free(tmpr2);
128 Free(tmpi2);
129 UNPROTECT(1);
130 return(success);
131 }

```