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

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 monostatic 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 Pacakge 11 (Software theory and implementation) of the European Union's Framework 7 project EISCAT_3D. This document serves as part of the final report of the Task 11.4 as well as a tutorial for the package and lag profile inversion in general. A standard user manual of the R package is provided as a separate document.

1.2 Lag Profile Inversion

LPI is an R[1] package for suppressing range ambiguities from incoherent scatter radar[2] 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.

[1]R 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.

[2]The 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 installed[3], 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.

[3]Latest 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 $\mathrm{env}(t)$. Because the carrier sigal contribution can be effectively removed by means of complex frequency mixing to baseband, we will neglect the carrier from this point on and consider only the transmitter envelope $\mathrm{env}(t)$.

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

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

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

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

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

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

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

where $t_i = i\Delta t$.

The receiver sample stream is not continuous in general, because monostatic radar systems cannot receive while transmitting. The transmitter sample stream will be effectively continuous because the transmitter envelope is known to be zero while the radar is not transmitting. The discontinuities in receiver sample stream have siginicant consequencies especially when detecting nearby targets with a monostatic high duty-cycle radar[1].

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

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

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

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

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

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

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

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

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

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

write the received signal as

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

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. Similarily, 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 fucntion of the scattering process at the given range.

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

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

2.5 Range ambiguity functions

Expectation value of the product

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

can be written as

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

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

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

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

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

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

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

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

$$\mathbf{m}_l = \mathbf{W}_l \rho_l + \varepsilon_l \tag{2.15}$$

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

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

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

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

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

2.7 Additional analysis steps

2.7.1 Ground clutter suppression

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

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

2.7.2 Voltage level decoding

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

Chapter 3

LPI implementation

3.1 Resampling and filtering

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

3.2 Ground clutter suppression

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

3.3 Voltage level decoding filters

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

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

3.4 Data correlation

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

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

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

3.5 Theory matrix

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

3.6 Lag profile inversion solvers

Altogether five different inverse problem solvers are supported.

- **rlips** R Linear Inverse Problem Solver[1]. 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 indepedently from LPI. Names of the I/O functions and packages including them are given as input arguments to the main solver function. It is usually practical to include also a simple wrapper function that generates a call to the main analysis loop of LPI. A raw data input function is mandatory, whereas a few other routines have defaults, see the LPI manual for details. Currently availabe I/O packages are `LPI.gdf` and `LPI.KAIRA`. See LPI-manual.pdf for detailed descriptions of the I/O functions.

Chapter 4

LPI in practice

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

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

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

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

and the help page of the main analysis function

```
> ?LPI
```

4.1 Examples with simulated data

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

4.1.1 A coherent point target

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

```
> library(LPI)
```

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

```
> datafun <- function( LPIparam , ... ){
+
+   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 receiver samples are simply
+   # a shifted copy of the transmitter samples
+   outlist[['RX1']][['cdata']] <-
+     c( rep( 0 , rtarg ) , outlist[['TX1']][['cdata']][1:(nd-rtarg)] )
+
+   # Let us add some random noise on top of the receiver samples
+   outlist[['RX1']][['cdata']] <-
+     outlist[['RX1']][['cdata']] + (rnorm(nd) + 1i*rnorm(nd))*.3
+
+   # Receiver index vector is negation of the transmitter index vector
+   outlist[['RX1']][['idata']] <- !outlist[['TX1']][['idata']]
+
+   # 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. 'startTime' 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
```

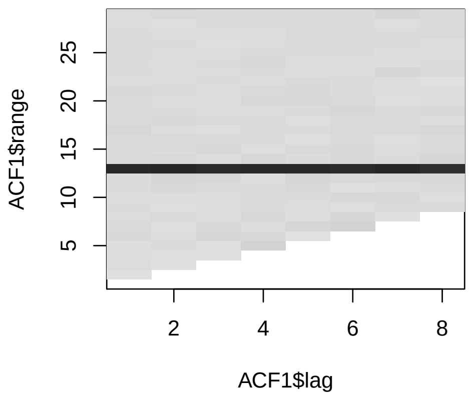

Figure 4.1: LPI detection of a simulated coherent point target.

```
             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))*.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))
```

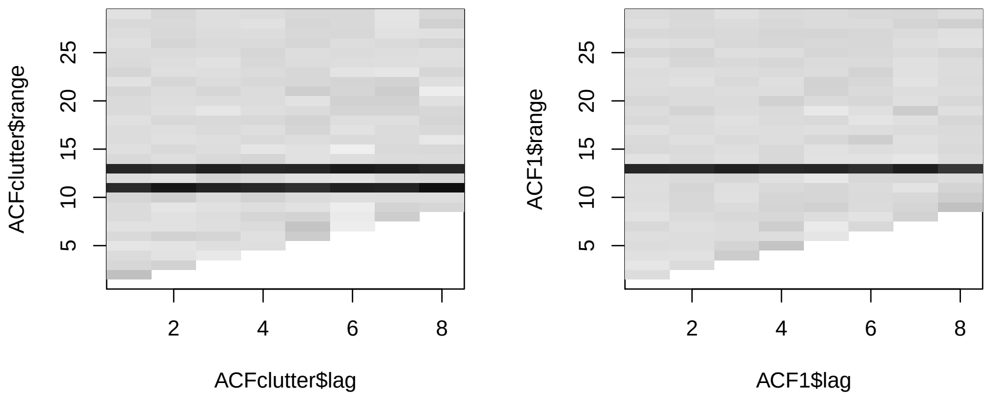

Figure 4.2: LPI detection of two simulated coherent point targets. The upper one has a small doppler shift whereas the lower one has zero doppler. When analysed without clutter suppresion (left) both targets are detected. When the clutter suppression is applied (right) the lower one becomes subtracted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

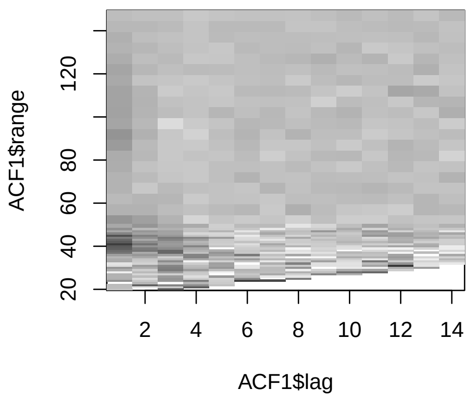

Figure 4.3: LPI analysis of one second of data from an EISCAT UHF beata experiment March 13 2013 22:02:36 UT. The results is very noisy because a very short period of data was used, but a clear E region is visible around range gate 40 and F region around range gate 100.

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.

```
## file:LPI.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.

##
## The main analysis loop of LPI
##
##
##

LPI <- function(dataInputFunction,
                inputPackages=c(),
                startTime = 0, # 1st Jan 1970 00:00 UT
                stopTime = 4000000000, # 2nd Oct 2096 07:00
                    UT
                nup = LPIexpand.input( 1 ),
                filterLength = LPIexpand.input( 1 ),
                decodingFilter = "none",
                lagLimits = c(1,2),
                rangeLimits = c(1,2),
                maxRanges = Inf,
                maxClutterRange = 0,
                 clutterFraction = 1,
                 timeRes.s = 10,
                 backgroundEstimate=TRUE,
                 clusterNodes = NA,
                 nodeMultip=1,
                 useXDR=FALSE,
                 maxWait.s = -1,
                 freqOffset = LPIexpand.input( 0 ),
                 indexShifts = LPIexpand.input( list(c(0,0)) )
                     ,
                 solver = "fishs",
                 nBuf = 10000,
                 fullCovar = FALSE,
                 rlips.options = list( type="c" , nbuf=1000 ,
                    workgroup.size=128),
                 remoteRX = FALSE,
                 normTX = FALSE,
                 nCode = NA,
                 ambInterp = FALSE,
                 resultDir = paste(format(Sys.time(),"%Y-%m-%d
                    _%H:%M"),'LP',sep='_'),
                 dataEndTimeFunction="currentTimes",
                 resultSaveFunction = "LPIsaveACF",
                 paramUpdateFunction="noUpdate",
                 ...
                 )
{

  # Collect all input in a list that is handy to pass
     forwards
  par1 <- formals()
  par1['...'] <- NULL
  par2 <- list(...)
  par1names <- names(par1)
  par1 <- lapply( names( par1 ) , FUN=function(x){ eval( as
     .name( x ) ) } )
  names(par1) <- par1names
  LPIparam <- c(par1,par2)

  # Expand parameters to LPI internal format and set
     storage modes as necessary
  LPIparam[["nup"]] <- LPIexpand.input( LPIparam[["nup"]] )
  storage.mode( LPIparam[["nup"]] ) <- "integer"
  LPIparam[["filterLength"]] <- LPIexpand.input( LPIparam[[
     "filterLength"]] )
  storage.mode( LPIparam[["filterLength"]] ) <- "integer"
  storage.mode( LPIparam[["lagLimits"]] ) <- "integer"
  storage.mode( LPIparam[["rangeLimits"]] ) <- "integer"
```

```r
    LPIparam[["maxClutterRange"]] <- LPIexpand.input( LPIparam[["maxClutterRange"]] )
    storage.mode( LPIparam[["maxClutterRange"]] ) <- "integer"
    LPIparam[["clutterFraction"]] <- LPIexpand.input( LPIparam[["clutterFraction"]] )
    LPIparam[["freqOffset"]] <- LPIexpand.input( LPIparam[["freqOffset"]] )
    if( ! is.list( LPIparam[["indexShifts"]] ) ){
      LPIparam[["indexShifts"]] <- list(LPIparam[["indexShifts"]])
    }
    LPIparam[["indexShifts"]] <- LPIexpand.input( LPIparam[["indexShifts"]] )
    for( dType in c("TX1","TX2","RX1","RX2")) storage.mode( LPIparam[["indexShifts"]][[dType]]) <- "integer"
    storage.mode( LPIparam[["nCode"]] ) <- "integer"


    # Print input arguments
    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")))
    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")))
    cat(sprintf("%20s"," inputPackages:"))
    for(n in inputPackages){cat(n,", ")}
    cat('\n')
    cat(sprintf("%20s %s\n","dataInputFunction:", dataInputFunction))
    cat(sprintf("%20s %s\n","dataEndTimeFunction:", dataEndTimeFunction))
    cat(sprintf("%20s"," clusterNodes:"))
    if( is.list(clusterNodes )){
      for(n in names(clusterNodes)){cat(sprintf("%s:",n));cat(clusterNodes[[n]],'  ')};cat('\n')
    }else{
      cat( clusterNodes ); cat('\n')
    }
    cat(sprintf("%20s","nup:"));for(dType in c("RX1","RX2","TX1","TX2")){cat(' ',dType,':',LPIparam[["nup"]][[dType]],sep='')};cat('\n')
    cat(sprintf("%20s","filterLength:"));for(dType in c("RX1","RX2","TX1","TX2")){cat(' ',dType,':',LPIparam[["filterLength"]][[dType]],sep='')};cat('\n')
    cat(sprintf("%20s %s\n","decodingFilter:",decodingFilter[1]))
```

```
  cat(lagLimits,fill=70,labels=c(sprintf("%20s","lagLimits:"),rep('                     ',1000)))
  cat(rangeLimits,fill=70,labels=c(sprintf("%20s","rangeLimits:"),rep('                     ',1000)))
  cat(maxRanges,fill=70,labels=c(sprintf("%20s","maxRanges:"),rep('                     ',1000)))
  cat(sprintf("%20s %f\n","timeRes.s:",timeRes.s))
  cat(sprintf("%20s RX1:%i RX2:%i \n","maxClutterRange:",LPIparam$maxClutterRange["RX1"],LPIparam$maxClutterRange["RX2"]))
  cat(sprintf("%20s RX1:%i RX2:%i \n","clutterFraction:",LPIparam$clutterFraction["RX1"],LPIparam$clutterFraction["RX2"]))
  cat(sprintf("%20s %s\n","backgroundEstimate:",backgroundEstimate))
  cat(sprintf("%20s %f\n","maxWait.s:",maxWait.s))
  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"]))
  cat(sprintf("%20s","indexShifts:"));for(dType in c("RX1","RX2","TX1","TX2")){cat(' ',dType,':',sep='');cat(LPIparam$indexShifts[[dType]])};cat('\n')
  cat(sprintf("%20s %s\n","solver:",solver))
  cat(sprintf("%20s %i\n","nBuf:",nBuf))
  cat(sprintf("%20s %s\n","fullCovar:",fullCovar))
  cat(sprintf("%20s","rlips.options:"));for(n in names(rlips.options)){cat(' ',n,':',rlips.options[[n]],sep='')};cat('\n')
  cat(sprintf("%20s %s\n","remoteRX:",remoteRX))
  cat(sprintf("%20s %s\n","normTX:",normTX))
  cat(sprintf("%20s %i\n","nCode:",nCode))
  cat(sprintf("%20s %s\n","ambInterp:",ambInterp))
  cat(sprintf("%20s %s\n","resultDir:",resultDir))
  cat(sprintf("%20s %s\n","resultSaveFunction:",resultSaveFunction))
  cat(sprintf("%20s %s\n","paramUpdateFunction:",paramUpdateFunction))
  cat(sprintf("%20s %s\n","useXDR:",useXDR))

  # Total number of integration periods requested
  LPIparam[["lastIntPeriod"]] <- round( ( stopTime - startTime ) / LPIparam[["timeRes.s"]] )

  # Create the result directory if a valid path was given
  if( is.character( resultDir ) ){
    if( nchar( resultDir ) > 0 ){
      dir.create( resultDir , recursive=TRUE , showWarnings=FALSE )
    }
  }

  # Initialise the computer cluster for LPI.
  # Save the cluster definitions to the global workspace
  ctrlcl <<- LPIinitCluster( LPIparam[["clusterNodes"]] ,
     useXDR=useXDR )
  ncl <- length( ctrlcl )

  # A flag telling whether the analysis will be run in some
  # kind of cluster configuration or in a single process
  LPIparam[["iscluster"]] <- !all(is.na(ctrlcl))

  # Initialize a list for unsolved integration periods
  intPer.missing <- seq( LPIparam[["lastIntPeriod"]] )

  # Run analysis loop until end of data
  endOfData <- FALSE
  repeat{

    # Update the last available data samples
    LPIparam[["dataEndTimes"]] <- eval( as.name( LPIparam[[
       "dataEndTimeFunction"]] ))( LPIparam )

    # Latest integration period for which data is available
    LPIparam[["maxIntPeriod"]] <- floor( ( min(unlist(
       LPIparam[["dataEndTimes"]])) - LPIparam[["startTime"
       ]] ) / LPIparam[["timeRes.s"]] )

    # Select integration period numbers for the next
       analysis run
    # Latest periods will be analysed first in order to
       simplify real-time analysis
    waitSum <- 0
    while( is.null( intPer.current <-
       nextIntegrationPeriods( LPIparam , nodeMultip*ncl ,
       intPer.missing ))){

      # Break the loop after waiting
      # long enough for new data
      if( waitSum > LPIparam[["maxWait.s"]] ){
        endOfData <- TRUE
        break
      }

      # Wait 10 seconds
      Sys.sleep(10)

      # Increment the wait time counter
```

```r
      waitSum <- waitSum + 10

      # Update the last available data samples
      LPIparam[["dataEndTimes"]] <- eval( as.name( LPIparam[["dataEndTimeFunction"]] ))( LPIparam )

      # Latest integration period for which data is available
      LPIparam[["maxIntPeriod"]] <- floor( ( min(unlist( LPIparam[["dataEndTimes"]])) - LPIparam[["startTime"]] ) / LPIparam[["timeRes.s"]] )

    }

    if( endOfData ) break

    # Run analysis on each parallel node, or locally if
    # LPIparam[["iscluster"]]==FALSE
    if( LPIparam[["iscluster"]] ){
      clusterApplyLB( ctrlcl , intPer.current , fun=LPI:::LPIsolve.acf , LPIparam )
    }else{
        for( iper in intPer.current ){
            LPI:::LPIsolve.acf( iper , LPIparam )
        }
    }

    # Print something to show that the analysis is running
    for( k in seq(length(intPer.current))) cat('.')

    # Remove the solved periods from the list of missing ones
    intPer.missing <- setdiff( intPer.missing , intPer.current )

    # Stop if all integration periods are solved
    if( length(intPer.missing)==0) break

  } # repeat

  # Shut down the cluster at end of analysis
  if(!all(is.na(LPIparam[["clusterNodes"]]))) stopCluster( ctrlcl )

  # This function does not return anything,
  # results are written to files.
  invisible()

}
```

5.1.2 LPIexpand.input

```
## file:LPIexpand.input.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.

##
## Expand input argument list or vector
## into the internally used format
##
## Arguments:
##  parvec A vector (or list)
##
## Returns:
##  outvec A named vector or list with elements
## "RX1", "RX2", "TX1", and "TX2".
##

LPIexpand.input <- function( parvec )
  {

    # Names of the input list / vector
    namevec <- names(parvec)

    # If the input does not have names attributes, assume
    # that the elements are in order RX1 , RX2 , TX1 , TX2
    # and repeat as necessary.
    if(is.null(namevec)){
      # Repeat the input
      outvec        <- rep(parvec,length.out=4)
      # Set names
      names(outvec) <- c( "RX1" , "RX2" , "TX1" , "TX2" )
      # Return the named vector / list
      return(outvec)
    }

    # If the input had names(s), start inspecting them

    # A vector for the output
    outvec <- rep(NA,4)
    names(outvec) <- c( "RX1" , "RX2" , "TX1" , "TX2" )

    # First look if any of the internally used
    # names is used in the input
    if( any(namevec=="RX1")) outvec[1] <- parvec["RX1"]
    if( any(namevec=="RX2")) outvec[2] <- parvec["RX2"]
    if( any(namevec=="TX1")) outvec[3] <- parvec["TX1"]
    if( any(namevec=="TX2")) outvec[4] <- parvec["TX2"]

    # If the vector had elements "RX1" , "RX2" , "TX1" ,
    # and "TX2", return them in correct order
    if( !any(is.na(outvec))) return(outvec)

    # If there are still missing values,
    # look for elements "RX" and "TX"
    if( is.na(outvec[1])){
      if(any(namevec=="RX")) outvec[1] <- parvec["RX"]
    }
    if( is.na(outvec[2])){
      if(any(namevec=="RX")) outvec[2] <- parvec["RX"]
    }
    if( is.na(outvec[3])){
      if(any(namevec=="TX")) outvec[3] <- parvec["TX"]
    }
    if( is.na(outvec[4])){
      if(any(namevec=="TX")) outvec[4] <- parvec["TX"]
    }

    # If the vector is now properly filled, return it
    if( !any(is.na(outvec))) return(outvec)

    # Now look for elements "TR1" and "TR2"
    if( is.na(outvec[1])){
      if(any(namevec=="TR1")) outvec[1] <- parvec["TR1"]
    }
    if( is.na(outvec[2])){
      if(any(namevec=="TR2")) outvec[2] <- parvec["TR2"]
    }
    if( is.na(outvec[3])){
      if(any(namevec=="TR1")) outvec[3] <- parvec["TR1"]
    }
    if( is.na(outvec[4])){
      if(any(namevec=="TR2")) outvec[4] <- parvec["TR2"]
    }

    # If the vector is now properly filled, return it
    if( !any(is.na(outvec))) return(outvec)

    # Finally remove the named elements from parvec and
    # try to fill the output vector
    parvec <- parvec[ nchar(namevec) == 0 ]
    if( length(parvec) > 0 ) outvec[is.na(outvec)] <- rep(
       parvec,length.out=sum(is.na(outvec)))

    # If the output is now full, return it
    if( !any(is.na(outvec))) return(outvec)

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

5.1.3 LPIinitCluster.R

```
## file:LPIinitCluster.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.

##
## Initialise the analysis cluster, which consists of:
##   - a master process (which calls this function)
##   - length(nodes) control processes running on the same
##     computer with the master process (if the input list
##     does not contain the entry localControl=FALSE)
##   - length(nodes) control processes running on any
##     computer on the cluster (names of these computers
##     are given in names(nodes) )
##   - length(nodes[[i]]) computing slaves running on each
##     remote computer nodes[i] (if nodes[[i]] is a
##     character vector, these nodes may be also on several
##     different computers)
##
##   The computing slaves do most of the  actual work,
##   the other processs are mainly for data transfer
##   (the control processes closest to master do all
##   disk I/O)
##
## Each remote control process will be given one
## integration period of raw data, whose ACF will be
## calculated in parallel by the computing slaves of the
## process. Thus, there are length(nodes) integration
## periods in parallel, and each of them has
## length(nodes[[i]]) parallel lag profiles inversions
## running.
##
## Arguments:
##  nodes    A list of remote control machine names and
##           definitions of computing slaves for each of them
##           e.g. nodes = list( tesla1=8, tesla2=8,
##                              tesla3=8, tesla4=8, tesla5=8)
##                or
##                nodes = list(
##              tesla1=c( rep('tesla1',8), rep('tesla3',8) ),
##              tesla4=c( rep('tesla4',8), rep('tesla5',8) )
##                           )
##
##           The former example starts a control process
##           on each computer of the tesla cluster, and
##           allocates one computing slave per core
##           (each of the computers has 8 cores).
##          Thus, five integration periods are analysed
##          in parallel with 8 lag profiles in parallel
##          in each of them. The latter one runs only two
##          integration periods at a time, but each of
##          them has 16 lag profiles in parallel.
##
##          Notice that the latter option leads to
##          signicantly larger amount of network traffic,
##          as the remote control nodes transfer the data
##          to each computing slave separately
##
##          Alternatively, one can give an integer number,
##          which will start the given number of parallel
##          processes, running one integration period each
##          on localhost. Any combination of the above
##          inputs are also accepted.
##
##          The list nodes is treated as follows:
##            1. Put NAs to values <= 0
##            2. If only NA's were left from 1.,, do not
##               start a cluster
##               (analysis sequentially in the main process)
##            3. Unnamed entries are replaced with equal
##               number of entries localhost=1
##
##          More examples:
##
##            Start 5 parallel integration periods on
##            localhost, and another 5 on "remotecomputer":
##              nodes=list(5,remotecomputer=1,
##                       remotecomputer=1,remotecomputer=1,
##                       remotecomputer=1,remotecomputer=1)
##
##            Start 1 integration period with five parallel
##            lags in localhost and another similar one on
##            "remotecomputer":
##              nodes=list(localhost=5,remotecomputer=5)
##
##            Start 4 parallel integration periods on both
##            remotecomputer1 and remotecomputer2, but do
##            not create the local control processes. This
##            requires that both computers have the input
##            and output data directories mounted on same
##            paths.
##              nodes=list(remotecomputer1=4,
##                         remotecomputer2=4,
##                         localControl=FALSE)
##
##            Do not use parallelism, solve everything
##              sequentially in the main process
##                nodes=NA
##
##
##
##
## Returns:
##    ctrlcl   A list of class cluster of
##             the local control nodes
##
##             The corresponding lists of remote control
##             clusters and computing slaves
##             are stored on the cluster nodes
##

LPIinitCluster <- function( nodes , useXDR=FALSE )
  {

    # Check if nodes has an entry "localControl",
    # if not, use default (TRUE)
    localControl <- TRUE
    if(is.list(nodes)){
      if(is.logical(nodes[["localControl"]])) localControl <-
          nodes[["localControl"]]
    }

    # Replace negative values with NAs
    for(k in seq(length(nodes))){
      if(is.numeric(nodes[[k]])){
        if(nodes[[k]]<=0)  nodes[[k]] <- NA
      }
    }

    # If only NA values, we will run locally
    if(all(is.na(nodes))) return(NA)

    # Strip off all NAs (original NAs
    # and those from non-positive values)
    nodes[is.na(nodes)] <- NULL

    # Named nodes are left as such, unnamed are
    # assumed to denote the number of local
    # parallel integration periods
    nnames <- names(nodes)
    if(is.null(nnames)){
      ncnames <- rep(0,length(nodes))
    }else{
      ncnames <- nchar(names(nodes))
    }
```

```r
  nodes2 <- c( nodes[ncnames>0] , rep( list( localhost=1 ) , sum( unlist( nodes[ ncnames==0 ] ) ) ) )

  # Remove the localControl entry
  nodes2[["localControl"]] <- NULL

  # Create the (optional) local control nodes
  if(localControl){
    # Create the local control nodes
    ctrlcl <- makeCluster( length( nodes2 ) , useXDR=useXDR )

    # Load packaages LPI and parallel to each of the local control nodes
    clusterEvalQ( ctrlcl , library(LPI) )

    # Create the remote computer control processes
    for(k in 1:length(nodes2)){
      # Run initialisation at the local control process to
      # create the remote control process and its slaves
      clusterCall( ctrlcl[k] , LPIinitRemoteNode , nodes2[k] , useXDR )
    }

  # Otherwise the remote nodes will
  # act as control nodes as well
  }else{

    # Create the remote control nodes directly
    ctrlcl <- makeCluster( names(nodes2) , useXDR=useXDR )

    # Load packaages LPI and parallel to
    # each of the remote control nodes
    clusterEvalQ( ctrlcl , library(LPI) )

    # Set remcl=NA on each node to notify that
    # the additional control step does not exist
    remcl <<- NA
    clusterExport( ctrlcl , 'remcl')

    # Initialise the computing slaves
    for(k in 1:length(nodes2)){
      clusterCall( ctrlcl[k] , LPI:::LPIinitComputingSlaves , nodes2[[k]] , useXDR )
    }
  }

  return(ctrlcl)

```

```
    }
```

5.1.4 LPIinitRemoteNode.R

```
## file:LPIinitRemoteNode.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.

##
## Establish a scoket connection between a local
## control process and a remote control process
## and initialise the computing slaves at the remote
##
## Arguments:
##    remnode   a list of the form
##              remname = list(compslave1,compslave2,...)
##              where remname is the remote analysis
##              computer and the list constains either
##              the number of computing slaves
##              on that particular computer, or a vector
##              of computer names at which to create the
##              computing slaves
##

LPIinitRemoteNode <- function( remNode , useXDR )
  {

    # Node name
    nodeName <- names(remNode)

    # If nodeName is localhost, do not start the remote
    # control process but make direct connections
    # to the slaves instead.
    if( nodeName == "localhost"){
      remcl <<- NA
      LPI:::LPIinitComputingSlaves( remNode[[1]] , useXDR )
      return(remcl)
    }

    # Establish the connection to the remote control nodes
    remcl   <<- makeCluster( names(remNode) , useXDR=useXDR )

    # Load package LPI
    clusterEvalQ( remcl , library(LPI) )

    # Initialise the computing slaves
    clusterCall( remcl , LPIinitComputingSlaves , remNode
        [[1]] , useXDR )

    return(remcl)

  }
```

5.1.5 LPIinitComputingSlaves.R

```
## file:LPIinitComputingSlaves.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.

##
## Init the actual worker processes,
## "computing slaves", of a remote control node.
##
## Arguments:
##  slaveNodes Cluster node definition, either an integer
##             number of cluster nodes or a string vector
##             of host names.
##
## Returns:
##  slavecl    An object of class SOCKcluster. The same
##             object is also stored on the global workspace
##

LPIinitComputingSlaves <- function( slaveNodes , useXDR )
  {

    # If only one slave, do not allocate it but
    # run analysis in the control process
    if( slaveNodes == 1 ){
      return( slavecl <<- NA )
    }

    # Create the cluster of computing slaves
    slavecl <<- makeCluster( slaveNodes , useXDR=useXDR )

    # Load LPI package to each of the nodes
    clusterEvalQ( slavecl , library(LPI) )

    return(slavecl)

  }
```

5.1.6 currentTimes.R

```r
## file:currentTimes.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.

##
## Current unix time minus 5 seconds to be used for
## identifying the latest available data samples in
## real time analysis.
##
## Arguments:
##   ...  An arbitrary list of arguments is accepted, but
   none
##        of them will be used.
##
## Returns:
##  curTimes A named vector ("TX1","TX2","RX1","RX2") with
##           the current unix time -5 in each element.
##
##

currentTimes <- function( ... )
  {
    return( LPIexpand.input( as.numeric(Sys.time()-5) ) )
  }
```

5.1.7 nextIntegrationPeriods.R

```
## file:nextIntegrationPeriods.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.

##
## Indices of n latest integration periods
## that have not yet been analysed.
##
## Arguments:
##  LPIparam     A LPI parameter list
##  n            Number of new periods to search for
##  intPer.ready A list of solved period indices
##
## Returns:
##  nextIpers   Indices of the integration periods to
##              be solved next.
##

nextIntegrationPeriods <- function( LPIparam , n , intPer.
   missing )
  {


    # Truly available periods
    intPer.available <- intPer.missing[ which( intPer.missing
        <= min( LPIparam[["maxIntPeriod"]] , LPIparam[["
       lastIntPeriod"]]) ) ]

    # We know that the integration periods are in order,
    # simply pick the n last ones
    nper <- length(intPer.available)
    if(nper==0) return(NULL)
    return(intPer.available[ max(1,( nper - n + 1 )) : nper
       ])


##    # A vector for the integration period numbers
##    nextIpers <- rep(0,n)
##
##    # Counter for identified new periods
##    k <- 0
##
##    # The period from which we will start seeking backwards
##    p <- min( LPIparam[["maxIntPeriod"]] , LPIparam[["
   lastIntPeriod"]] )
##
##    # If the last data sample or analysis end time is
   before
##    # beginning of analysis, there will be nothing to do
##    if( p < 0 ) return(NULL)
##    # Start looking backwards from the last period
##    while( k < n ){
##      # Select periods that have not yet been analysed.
##      if(!any(intPer.ready == p)){
##        k <- k+1
##        nextIpers[k] <- p
##      }
##      # Stop looking if we hit the analysis start time
##      if( p == 1) break
##      p <- p - 1
##    }
##
##    # Return NULL if nothing was found
##    if( k== 0 ) return(NULL)
##
##    return( nextIpers[1:k] )

 }
```

5.1.8 LPIsolve.acf.R

```
## file:LPIsolve.acf.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

## This function is run in local control nodes.
## Read data for one integration period, send it for
## analysis in a remote computer, and write the returned
## ACF to file
##
## Arguments:
##  intPeriod  Integration period number, counted from
##             LPIparam[["firstTime"]] in steps of
##             LPIparam[["timeRes.s"]]
##
## Returns:
##  intPeriod  The integration period number.
##

LPIsolve.acf <- function( intPeriod , LPIparam )
  {
    # Load packages that are needed for reading the data
    for( pn in LPIparam[["inputPackages"]] ){
      require( pn , character.only=TRUE )
    }

    # Parameter list update
    LPIparam <- eval( as.name( LPIparam[["paramUpdateFunction
        "]] ))( LPIparam , intPeriod )

    if( !is.null(LPIparam)){
        # Read raw data, name of the data input function
        # should be stored in a character string
        LPIdatalist.raw    <- eval(as.name(LPIparam[["
            dataInputFunction"]]))( LPIparam , intPeriod )

        # If data reading was successfull
        if(LPIdatalist.raw[["success"]]){

          # require that there are at least some TX and RX
              samples
          if( (sum(LPIdatalist.raw[["RX1"]][["idata"]]) > 0)
              &
              (sum(LPIdatalist.raw[["RX2"]][["idata"]]) > 0)
                  &
            (sum(LPIdatalist.raw[["TX1"]][["idata"]]) > 0) &
            (sum(LPIdatalist.raw[["TX2"]][["idata"]]) > 0)) {

          # Frequency mixing, filtering, etc., the output is
          # collected in a list and stored on the user workspace
          LPIdatalist.final <<- prepareLPIdata( LPIparam , LPIdatalist.raw )

          # Call the function that will send the data to
          # proper place and run the actual analysis
          ACF <- LPI:::LPIrun.remote( substitute( LPIdatalist.final) )

          # Store the results
          eval( as.name( LPIparam[["resultSaveFunction"]]) )( LPIparam , intPeriod , ACF )

        }
      }
  }

  # Return the integration period
  # number to the main process
  return(intPeriod)

}
```

5.1.9 noUpdate.R

```r
## file:noUpdate.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## LPI parameter list update function
## Return the list itself in first call,
## NULL in the second call with the same list
##
## Arguments:
##  LPIparam  A LPI parameter list
##  intPeriod Integration period number
##
## Returns:
##  LPIparam An exact copy  of the input LPIparam in
##           first call, NULL in second call with the
##           same list
##

noUpdate <-  function( LPIparam , intPeriod )
    {

        if(is.null(LPIparam[["callN"]])){
            LPIparam[["callN"]] <- 1
            return(LPIparam)
        }

        return(NULL)

    }
```

5.1.10 LPIrun.remote.R

```
## file:LPIrun.remote.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Send data to a remote computer and run analysis in there
##
## Arguments:
##  LPIenv.name Name of the LPI environment to use, the
##              environment was copied on global workspace
##              by LPIsolve.ACF
##
## Returns:
##  ACFlist     A list that contains the solved ACF, its
##              covariance matrices, lags, etc.
##

LPIrun.remote <- function( LPIenv.name )
  {

    # Check if we are running in a cluster or not
    if( eval(LPIenv.name)[["iscluster"]]  ){

      # Check that this is a local control node
      if(!is.na(remcl)){

        # Send the data environment to the remote node
        clusterExport( remcl , paste(LPIenv.name)  )

        # Run the remote analysis
        ACF <-  clusterCall( remcl , LPI:::LPIrun , LPIenv.
           name )

        # Return the ACF
        return( ACF[[1]] )
      }

    }

    # If the analysis is run in a single process,
    # or if this is a remote control node,
    # just run the analysis in this process
    return( LPI:::LPIrun( LPIenv.name ) )

   }
```

5.1.11 LPIrun.R

```
## file:LPIrun.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Copy the raw data to computing slaves and solve lag
## profiles in there. Combine the solved profiles into
## a full ACF and return it.
##
## Arguments:
##  LPIenv.name Name of the LPI environment to use
##              for the analysis.
##
## Returns:
##  ACFlist     A list that contains the solved ACF, its
##              covariance matrices, lags, etc.
##
## The LPI environment, which may be a named list
## as well, must be stored on the global workspace.
##

LPIrun <- function( LPIenv.name)
  {

    # Number of lags, each full lag
    # will get its own call of LPIsolve
    nlags <- eval(LPIenv.name)[["nLags"]]
    x <- seq( nlags )

    # Number of range gates
    ngates <- eval(LPIenv.name)[['nGates']]
    maxgates <- max(ngates)

    # Are we going to calculate a full covariance matrix?
    fullcovar <- eval(LPIenv.name)[['fullCovar']]

    # Range-gate centre points
    r <- eval(LPIenv.name)[['rangeLimits']]
    rgates <- ( r[1:maxgates] + r[2:(maxgates+1)] -1 ) / 2

    # Lag-gate centre points
    l <- eval(LPIenv.name)[["lagLimits"]]
    lgates <- ( l[1:nlags] + l[2:(nlags+1)] -1 ) / 2

    # If the computing slaves do not exist, set
    # slavecl=NA and run the analysis on this process
    if(!exists('slavecl')) slavecl <- NA

    # If we are running on a cluster
    if( eval(LPIenv.name)[["iscluster"]]  & ( !is.na( slavecl
        ) ) ){

      # Copy the data to all computing slaves
      clusterExport( slavecl , paste( LPIenv.name ) )

      # Allocate necessary vectors on each slave
      clusterCall( slavecl , initLPIenv , LPIenv.name )

      # Run the analysis processes on the slaves
      ACFlist <-  clusterApplyLB( slavecl , x , fun=LPI:::
          LPIsolve , LPIenv.name=LPIenv.name )


    # If not running on cluster, solve the lag profiles
    # sequentially. Mimic the output list of cluster
    # calls in order to simplify further processing.
    }else{

      # Create a list for the lag profiles
      ACFlist <- vector(mode='list',length=nlags)

      # Allocate vectors etc.
      initLPIenv( LPIenv.name )

      # Run the actual analysis sequentially
      for( k in 1:nlags ){
        ACFlist[[k]] <- LPI:::LPIsolve( lag=x[k] , LPIenv.
            name=LPIenv.name )
      }

    }

    # Collect the lag numbers from ACF list
    lagnums <- x
    for(k in 1:nlags ){
      lagnums[k] <- ACFlist[[k]][['lagnum']]
    }

    # Find correct order for the lag profiles
    lagorder <- x[order(lagnums)]

    # Order the ACF list
    ACFlist <- ACFlist[lagorder]

```

```
  # Make ACF and variance matrices
  ACFmat <- matrix(NA,ncol=nlags,nrow=(maxgates+1))

  # Collect the lag profiles to the ACF matrix
  for( k in 1:nlags){
    if(ngates[k]>0){
      # Copy the solved lag profile
      ACFmat[1:ngates[k],k] <- ACFlist[[k]][['lagprof']][1:
         ngates[k]]
      # Copy the background ACF estimate
      ACFmat[maxgates+1,k]  <- ACFlist[[k]][['lagprof']][
         ngates[k]+1]
    }
  }

  # If full covariance matrices were solved
  if(fullcovar){
    # allocate matrix for variances and a cube for the
        covariance matrices
    VARmat   <- matrix(NA,ncol=nlags,nrow=(maxgates+1))
    COVARmat <- array(NA,dim=c((maxgates+1),(maxgates+1),
        nlags))
    for( k in 1:nlags){
      if(ngates[k]>0){
        # Copy variances
        VARmat[1:ngates[k],k]                   <- Re(diag(
           ACFlist[[k]][['covariance']]))[1:ngates[k]]
        VARmat[maxgates+1,k]                    <- Re(diag(
           ACFlist[[k]][['covariance']]))[ngates[k]+1]
        # Copy covariance matrices
        COVARmat[1:ngates[k],1:ngates[k],k]     <- ACFlist[[
           k]][['covariance']][1:ngates[k],1:ngates[k]]
        COVARmat[(maxgates+1),1:ngates[k],k]    <- ACFlist[[
           k]][['covariance']][(ngates[k]+1),1:ngates[k]]
        COVARmat[1:ngates[k],(maxgates+1),k]    <- ACFlist[[
           k]][['covariance']][1:ngates[k],(ngates[k]+1)]
        COVARmat[(maxgates+1),(maxgates+1),k]   <- ACFlist[[
           k]][['covariance']][(ngates[k]+1),(ngates[k]+1)]
      }
    }
  # If only variances were solved
  }else{
    # Allocate a matrix for the variances,
    # set COVARmat to NULL
    VARmat   <- matrix(NA,ncol=nlags,nrow=(maxgates+1))
    COVARmat <- NULL
    for( k in 1:nlags){
      if( ngates[k] > 0 ){
        # Copy the variances
```

```
          VARmat[1:ngates[k],k]   <- Re(ACFlist[[k]][['
             covariance']])[1:ngates[k]]
          VARmat[(maxgates+1),k]  <- Re(ACFlist[[k]][['
             covariance']])[ngates[k]+1]
        }
      }
    }


    # Collect the results in a list and return it.
    # A list is used because an environment
    # is much slover to transfer
    ACFreturn <- list()
    ACFreturn[["ACF"]]        <- ACFmat
    ACFreturn[["var"]]        <- VARmat
    ACFreturn[["covariance"]] <- COVARmat
    ACFreturn[["lag"]]        <- lgates
    ACFreturn[["range"]]      <- rgates
    ACFreturn[["nGates"]]     <- ngates

    return(ACFreturn)

  }
```

5.1.12 initLPIenv.R

```r
## file:initLPIenv.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Allocate and initialise necessary vectors and variables
## for the actual lag profile inversion. This function is
## called once per integration period in each computing slave
##
## Arguments:
##  LPIenv.name Name of the LPI environment used for
##              the analysis.
##
## Returns:
##    Nothing, the udpated environment is stored on
##    the global workspace.
##

initLPIenv <- function( LPIenv.name )
  {

    # Get the LPI environment (transferred as a list,
    # convert into an environment first)
    LPIenv <- as.environment( eval( LPIenv.name ) )

    # Allocate vector for the range ambiguity function
    assign( 'camb' , vector(mode='complex',length=(LPIenv[["
       nData"]]*LPIenv[["nDecimTX"]]))      , LPIenv )

    # Range ambiguity indices
    assign( 'iamb' , vector(mode='logical',length=(LPIenv[["
       nData"]]*LPIenv[["nDecimTX"]]))      , LPIenv )

    # Laged products
    assign( 'cprod', vector(mode='complex',length=LPIenv[["
       nData"]])                              , LPIenv )

    # Lagged product indices
    assign( 'iprod', vector(mode='logical',length=LPIenv[["
       nData"]])                              , LPIenv )

    # Lagged product variances
    assign( 'var'  , vector(mode='numeric',length=LPIenv[["
       nData"]])                              , LPIenv )

    # Theory matrix rows, one extra row because
    # theory_rows needs a temp vector
    assign( 'arows', vector(mode='complex',length=((max(
       LPIenv[["nGates"]])+1)*(LPIenv[["nBuf"]]+1))), LPIenv
       )

    # Indices for theory matrix rows, one extra row because
    # theory_rows needs a temp vector
    assign( 'irows', vector(mode='logical',length=((max(
       LPIenv[["nGates"]])+1)*(LPIenv[["nBuf"]]+1))), LPIenv
       )

    # Measurement vector
    assign( 'meas' , vector(mode='complex',length=LPIenv[["
       nBuf"]])                              , LPIenv )

    # Measurement variances
    assign( 'mvar' , vector(mode='numeric',length=LPIenv[["
       nBuf"]])                              , LPIenv )

    # Buffer row counter
    assign( 'nrows', as.integer(0)

       , LPIenv )

    # Copy the modified environment back
    # to the user workspace
    assign( paste(LPIenv.name) , LPIenv , envir=.GlobalEnv)

    return()

  }
```

5.1.13 LPIsolve.R

```
## file:LPIsolve.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Solve the MAP estimate of a lag profile,
## starting from raw voltage samples
##
## Arguments:
##    LPIenv      A lag profile inversion environment
##    lag         Lag number, all fractional lags from
##                LPIenv[["lagLimits"]][lag] to
##                LPIenv[["lagLimits"]][lag+1]-1
##                are integrated in the same profile
##
## Returns:
##   lagprof      A named list  containing the MAP estimate
##                of the lag profile together with
##                its (co)variance.
##

LPIsolve <- function( lag , LPIenv.name )
  {

    # Get the LPI environment from the global workspace
    LPIenv <- eval(LPIenv.name)

    # Return immediately if number of gates is <= 0
    if( LPIenv[["nGates"]][lag] <= 0 ) return(list(lagnum=lag
       ))

    # If rlisp is used, make sure it has been loaded.
    # rlips is not required in startup in order to
    # allow analysis without installing it. Other
    # solvers are included in the LPI package.
    # Switch quietly to fishs if rlips is not available.
    if(LPIenv$solver=="rlips"){
      require(rlips) -> rres
      if( !rres ) assign( 'solver' , 'fishs' , LPIenv )
    }

    # Initialise the inverse problem solver
    if(LPIenv$solver=="rlips"){
      solver.env <- rlips.init( ncols = LPIenv$nGates[lag] +
          1 , nrhs = 1 , type = LPIenv$rlips.options[["type"]]
```

```
        , nbuf = LPIenv$rlips.options[["nbuf"]] , workgroup
        .size = LPIenv$rlips.options[["workgroup.size"]] )
  }else if ( LPIenv$solver=="fishs" ){
    solver.env <- fishs.init( LPIenv[["nGates"]][lag] + 1 )
  }else if ( LPIenv[["solver"]]=="deco" ){
    solver.env <- deco.init( LPIenv[["nGates"]][lag] + 1 )
  }else if ( LPIenv[["solver"]]=="dummy" ){
    solver.env <- dummy.init( range( LPIenv[["rangeLimits"
        ]][ 1 : (LPIenv[["nGates"]][lag]+1) ]) )
  }else if ( LPIenv[["solver"]]=="ffts" ){
    solver.env <- ffts.init( LPIenv[["nGates"]][lag] ,
        LPIenv[["TX1"]][["idata"]][1:LPIenv[["nData"]]])
  }

  # Copy of LPIenv[["nData"]]
  ndcpy <- LPIenv[["nData"]]

  # Walk through all fractional time-lags
  for( l in seq( LPIenv[["lagLimits"]][lag] , ( LPIenv[["
     lagLimits"]][lag+1] - 1 ) )){

    # If the lag is longer than the data vector
    # it cannot be calculated
    if( l >= LPIenv[["nData"]]) break

    # Current position in data vector, we will skip the
        first nGates samples
    assign( "nCur" , as.integer(LPIenv[["rangeLimits"]][
        LPIenv[["nGates"]][lag]+1]+1) , LPIenv)

    # Calculate the lagged products
    laggedProducts( LPIenv , l )

    # Variances of lagged products
    lagprodVar( LPIenv , l )

    # Calculate range ambiguity function
    rangeAmbiguity( LPIenv , l )

    # Optional pre-averaging of lag-profiles
    if( !is.null( LPIenv[["nCode"]] )){
      if( !is.na( LPIenv[["nCode"]] )){
        if( LPIenv[["nCode"]] > 0 ){
          averageProfiles( LPIenv , l )
          nd <- min( LPIenv[["nData"]] , which( diff(
              LPIenv[["TX1"]][["idata"]] ) == 1 )[ LPIenv[["
              nCode"]] + 1 ] )
          LPIenv[["nData"]] <- ifelse( is.na(nd) , LPIenv[[
              "nData"]] , nd )
          # Approximate the variance.
          # This is not exactly accurate!
          if(!is.na(nd))  LPIenv[["var"]] <- LPIenv[["var"]]  / ( sum(diff(LPIenv[["TX1"]][["idata"]])==1) /  LPIenv[["nCode"]] )
        }
      }
    }

    # Solvers "dummy" and "ffts" operate
    # directly with the product vectors
    if( LPIenv[["solver"]]=="dummy" ){

      dummy.add( e       = solver.env          ,
                M.data   = LPIenv[["cprod"]] ,
                M.ambig = LPIenv[["camb"]]  ,
                I.ambig = LPIenv[["iamb"]]  ,
                I.prod   = LPIenv[["iprod"]] ,
                E.data   = LPIenv[["var"]] , nData = as.integer( LPIenv[["nData"]] - 1 ) )

    }else if( LPIenv[["solver"]]=="ffts"){

      ffts.add( e       = solver.env          ,
                M.data   = LPIenv[["cprod"]] ,
                M.ambig = LPIenv[["camb"]]  ,
                I.ambig = LPIenv[["iamb"]]  ,
                I.prod   = LPIenv[["iprod"]] ,
                E.data   = LPIenv[["var"]]   ,
                nData    = as.integer(LPIenv[["nData"]] - 1)
                )

    # Other solvers need theory matrix rows
    }else{

      # Produce theory matrix rows in
      # (small) sets and add them to the solver
      while( newrows <- theoryRows( LPIenv , lag ) ){

        # If new rows were produced
        if( LPIenv[["nrows"]]>0){

          # select the correct solver
          if(LPIenv$solver=="rlips"){

            rlips.add( e = solver.env ,
                      A.data = LPIenv[["arows"]][1:(LPIenv[["nrows"]]*(LPIenv[["nGates"]][lag]+1))] ,
                        M.data = LPIenv[["meas"]][1:LPIenv[["nrows"]]] ,
                        E.data = LPIenv[["mvar"]][1:LPIenv[["nrows"]]]
                        )

            }else if(LPIenv$solver=='fishs'){

              fishs.add( e = solver.env ,
                        A.data = LPIenv[["arows"]][1:(LPIenv[["nrows"]]*(LPIenv[["nGates"]][lag]+1))] ,
                        M.data = LPIenv[["meas"]][1:LPIenv[["nrows"]]] ,
                        E.data = LPIenv[["mvar"]][1:LPIenv[["nrows"]]]
                        )

            }else if(LPIenv[["solver"]] == "deco" ){

              deco.add( e = solver.env ,
                       A.data = LPIenv[["arows"]][1:(LPIenv[["nrows"]]*(LPIenv[["nGates"]][lag]+1))] ,
                       M.data = LPIenv[["meas"]][1:LPIenv[["nrows"]]] ,
                       E.data = LPIenv[["mvar"]][1:LPIenv[["nrows"]]]
                       )

            }
          }
        }
      }

      # Make sure that the original value is
      # stored in LPIenv[["nDataa"]]
      LPIenv[["nData"]] <- as.integer(ndcpy)


    }

    # Solve the inverse problem
    if(LPIenv$solver=="rlips"){
      rlips.solve2( e = solver.env ,full.covariance = LPIenv[["fullCovar"]])
    }else if(LPIenv$solver=="fishs"){
      fishs.solve( e = solver.env , full.covariance = LPIenv[["fullCovar"]] )
    }else if(LPIenv[["solver"]]=="deco"){
      deco.solve( e = solver.env )
    }else if(LPIenv[["solver"]]=="dummy"){
      dummy.solve( e = solver.env , LPIenv[["rangeLimits"
          ]][1:(LPIenv[["nGates"]][lag]+1)])
    }else if( LPIenv[["solver"]]=="ffts"){
      ffts.solve( e = solver.env , LPIenv[["rangeLimits"
          ]][1:(LPIenv[["nGates"]][lag]+1)])
    }

    # Create the return environment
    lagprof <- new.env()

    # Assign the solution to the new environment
    assign( "lagprof" , solver.env[["solution"]] , lagprof )
    assign( "covariance" , solver.env[["covariance"]] ,
        lagprof )
    assign( "lagnum" , lag , lagprof )

    # Kill the solver object
    if(LPIenv$solver=="rlips") rlips.dispose(solver.env)

    # Conversion to list because it is faster to transfer
    return(as.list(lagprof))

  }
```

5.2 Signal pre- and post-processing

5.2.1 prepareLPIdata.R

```
## file:prepareLPIdata.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Create the LPI environment that is passed from local
## control nodes to remote nodes. A list is created instead
## of the final environment because it is faster to transfer.
##
## Arguments:
##  LPIparam         An LPI parameter list
##  LPIdatalist.raw  A raw data list returned by a data input
##                   function. (See e.g. readLPIdata.gdf)
##
## Returns:
##  LPIdatalist.final The final data list that is transferred
##                       to the solver nodes.
##

prepareLPIdata <- function( LPIparam , LPIdatalist.raw )
  {
    # Internally used data vectors
    dTypes <- c( "RX1" , "RX2" , "TX1" , "TX2" )

    # An empty list for the output data
    LPIdatalist.final <- vector(mode="list",length=4)
    names(LPIdatalist.final) <- dTypes


    # A list for TX1  pulse start positions in all data
    # vectors (these will be different if sample rates
    # are different).
    # Initialise with zeros to handle data vectors without
    # pulses (they will also go through the whole system
    # and NA results will be written). The pulseStarts will
    # be passed to c-routines as such, and 0 is thus the
    # firs index.
    pulseStarts <- list( TX1 = c(0) , TX2 = c(0) , RX1 = c(0)
        , RX2 = c(0) )

    # A list for first sample to use in decimation
    # in each data vector
    firstSample <- c( TX1 = 0 , TX2 = 0 , RX1 = 0 , RX2 = 0 )

    # Pulse start positions in TX1 ( >0 used because
    # c-routines may have put values larger than one
    #  to the idata vector)
    pulseStarts[["TX1"]] <- which( diff( LPIdatalist.raw[["TX1"]][["idata"]][1:LPIdatalist.raw[["TX1"]][["ndata"]]] > 0 ) == 1 )

    # Calculate the corresponding pulse
    # start positions in other data vectors
    for( XXN in dTypes ){
      pulseStarts[[XXN]] <- round( as.numeric(pulseStarts[["TX1"]]) / LPIparam[["filterLength"]][["TX1"]] * LPIparam[["nup"]][["TX1"]] * LPIparam[["filterLength"]][[XXN]] / LPIparam[["nup"]][[XXN]] )
      firstSample[[XXN]] <- pulseStarts[[XXN]][1]
    }

    # The below fix does not work if 'nup' are not common for all data vectors.
    # Disable in this case.

    if(all(LPIparam[["nup"]]==LPIparam[["nup"]]["TX1"])){
      # Strip off samples to make each
      # IPP a multiple of filter length
      for( XXN in dTypes ){
        # New pulse start positions that
        # are even multiples of the filter length
        pstarts2 <- pulseStarts[[XXN]] - round( ( pulseStarts[[XXN]] - firstSample[[XXN]] ) %% ( LPIparam[["filterLength"]][[XXN]] / LPIparam[["nup"]][[XXN]] ) )

        # Do something only if the pulse positions
        # really need to be modified
        if( any( pstarts2 != pulseStarts[[XXN]] ) ){

          # Amount of shift needed in original data
          ncut <- pulseStarts[[XXN]] - pstarts2
          ntx <- length(ncut)

          # Because we are cutting off data samples,
          # the start point k-1 will already be adjusted
          # when handling point k. We will thus need to
          # subtract the number of points cut in point
          # k-1 from the original ncut[k]. Then take
          # modulus to make sure that no points will be
          # cut  unless really necessary and that number
```

```
     # of points to cut is not negative
     ncut[2:ntx] <- ncut[2:ntx] - ncut[1:(ntx-1)]
     ncut <- ncut %% round( LPIparam[["filterLength"]][[XXN]] / LPIparam[["nup"]][[XXN]] )
     ind <- rep( TRUE , LPIdatalist.raw[[XXN]][["ndata"]])
     for( k in seq(length(pstarts2)) ){
      if( ( ncut[k] > 0 ) & (pulseStarts[[XXN]][k]<LPIdatalist.raw[[XXN]][["ndata"]]) ) ind[(pulseStarts[[XXN]][k]-ncut[k]+1):pulseStarts[[XXN]][k]] <- FALSE
     }
     # Number of data points must have changed
     # as samples were cut off, update the values
     LPIdatalist.raw[[XXN]][["ndata"]] <- min( LPIdatalist.raw[[XXN]][["ndata"]] , sum(ind) )
     LPIdatalist.raw[[XXN]][["cdata"]] <- LPIdatalist.raw[[XXN]][["cdata"]][ind][1:LPIdatalist.raw[[XXN]][["ndata"]]]
     LPIdatalist.raw[[XXN]][["idata"]] <- LPIdatalist.raw[[XXN]][["idata"]][ind][1:LPIdatalist.raw[[XXN]][["ndata"]]]
    }
   }

  }

  # The idata vectors will be modified according
  # to LPIparam$indexShift before decimation.
  # Take this into account in firstSamples.
  # Again keep 0 as the first index, because
  # the indices will be passed to c-routines as such
  firstSample[["TX1"]] <- firstSample[["TX1"]] + LPIparam[["indexShifts"]][["TX1"]][1]
#  while( firstSample[["TX1"]] < 0 ){
#   firstSample[["TX1"]]  <- firstSample[["TX1"]] - LPIparam[["filterLength"]][["TX1"]] / LPIparam[["nup"]][["TX1"]]
  while( firstSample[["TX1"]] < 0 ){
   firstSample[["TX1"]]  <- firstSample[["TX1"]] + LPIparam[["filterLength"]][["TX1"]] / LPIparam[["nup"]][["TX1"]]
  }

  firstFraction <- c( TX1 = 0 , TX2 = 0 , RX1 = 0 , RX2 = 0 )
  for( XXN in dTypes ){
   firstSampleF <- firstSample[["TX1"]] * LPIparam[["filterLength"]][[XXN]] / LPIparam[["filterLength"
```

```
        ]][["TX1"]] / LPIparam[["nup"]][[XXN]] * LPIparam[["
        nup"]][["TX1"]]
      firstSample[[XXN]] <- round( firstSampleF )
      firstFraction[[XXN]] <- round( ( firstSample[[XXN]] -
        firstSampleF ) * LPIparam[["nup"]][[XXN]] )
    }


    # Conversions to integer mode
    storage.mode( firstSample ) <- "integer"
    storage.mode( LPIparam[["filterLength"]] ) <- "integer"
    storage.mode( firstFraction ) <- "integer"

    # Index corrections, frequency mixing,
    # and filtering in C routines
    for( XXN in dTypes ){

      storage.mode( LPIparam[["indexShifts"]][[XXN]] ) <- "
        integer"

      LPIdatalist.final[[XXN]] <-
        .Call( "prepare_data"                      ,
              LPIdatalist.raw[[XXN]][["cdata"]] ,
              LPIdatalist.raw[[XXN]][["idata"]] ,
              LPIdatalist.raw[[XXN]][["ndata"]] ,
              LPIparam[["freqOffset"]][XXN]     ,
              LPIparam[["indexShifts"]][[XXN]]  ,
              LPIparam[["nup"]][XXN]            ,
              LPIparam[["filterLength"]][XXN]   ,
              firstSample[[XXN]]                ,
              firstFraction[[XXN]]              ,
              TRUE
              )

    }

    # Use length of the shortest data vector
    LPIdatalist.final[["nData"]] <-
      min(
          LPIdatalist.final[["RX1"]][["ndata"]],
          LPIdatalist.final[["RX2"]][["ndata"]],
          LPIdatalist.final[["TX1"]][["ndata"]],
          LPIdatalist.final[["TX2"]][["ndata"]]
          )


    # Optional TX amplitude normalisation
    if( LPIparam[["normTX"]] ){
      itx1 <- which(LPIdatalist.final[["TX1"]][["idata"]][1:LPIdatalist.final[["nData"]]])
      itx2 <- which(LPIdatalist.final[["TX2"]][["idata"]][1:LPIdatalist.final[["nData"]]])
      txamp1 <- mean(abs(LPIdatalist.final[["TX1"]][["cdata"]][itx1]))
      txamp2 <- mean(abs(LPIdatalist.final[["TX2"]][["cdata"]][itx2]))
      LPIdatalist.final[["TX1"]][["cdata"]][itx1] <- exp(1i*Arg(LPIdatalist.final[["TX1"]][["cdata"]][itx1])) * txamp1
      LPIdatalist.final[["TX2"]][["cdata"]][itx2] <- exp(1i*Arg(LPIdatalist.final[["TX2"]][["cdata"]][itx2])) * txamp2
    }

    # Optional ground clutter suppression
    if( ( LPIparam[["maxClutterRange"]]["RX1"] > 0 ) & ( LPIparam[["clutterFraction"]][["RX1"]] > 0 )){
      clutterSuppress( LPIdatalist.final[["TX1"]] , LPIdatalist.final[["RX1"]] , LPIparam[["rangeLimits"]][1] , LPIparam[["maxClutterRange"]]["RX1"] , LPIdatalist.final[["nData"]] , LPIparam[["clutterFraction"]][["RX1"]] )
    }
    if( ( LPIparam[["maxClutterRange"]]["RX2"] > 0 ) & ( LPIparam[["clutterFraction"]][["RX2"]] > 0 )){
      clutterSuppress( LPIdatalist.final[["TX2"]] , LPIdatalist.final[["RX2"]] , LPIparam[["rangeLimits"]][1] , LPIparam[["maxClutterRange"]]["RX2"] , LPIdatalist.final[["nData"]] , LPIparam[["clutterFraction"]][["RX2"]] )
    }


    # Optional voltage level decoding
    if( is.numeric( LPIparam[["decodingFilter"]] ) ){

        LPIdatalist.final[["RX1"]][["cdata"]][!LPIdatalist.final[["RX1"]][["idata"]]] <- 0+0i
        LPIdatalist.final[["RX2"]][["cdata"]][!LPIdatalist.final[["RX2"]][["idata"]]] <- 0+0i
        LPIdatalist.final[["TX1"]][["cdata"]][!LPIdatalist.final[["TX1"]][["idata"]]] <- 0+0i
        LPIdatalist.final[["TX2"]][["cdata"]][!LPIdatalist.final[["TX2"]][["idata"]]] <- 0+0i

        nd <- LPIdatalist.final[["nData"]]


        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] )

        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] )

        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] )

        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] )

    }else if( is.character( LPIparam[["decodingFilter"]] )){

        if( any( LPIparam[["decodingFilter"]][1] == c("matched","inverse") ) ){

            LPIdatalist.final[["RX1"]][["cdata"]][!LPIdatalist.final[["RX1"]][["idata"]]] <- 0+0i
            LPIdatalist.final[["RX2"]][["cdata"]][!LPIdatalist.final[["RX2"]][["idata"]]] <- 0+0i
            LPIdatalist.final[["TX1"]][["cdata"]][!LPIdatalist.final[["TX1"]][["idata"]]] <- 0+0i
            LPIdatalist.final[["TX2"]][["cdata"]][!LPIdatalist.final[["TX2"]][["idata"]]] <- 0+0i

            nd <- LPIdatalist.final[["nData"]]


            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] )

            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] )

            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] )

            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] )

            LPIdatalist.final[["TX1"]][["idata"]] <- LPI:::
               decoFilter.idata( LPIdatalist.final[["TX1"]][[
               "idata"]][1:nd] )

            LPIdatalist.final[["TX2"]][["idata"]] <- LPI:::
               decoFilter.idata( LPIdatalist.final[["TX2"]][[
               "idata"]][1:nd] )
        }
    }


    # Largest range in rangeLimits
    maxr <- as.integer(max(LPIparam[["rangeLimits"]]))

    # Average signal powers, loop three times in order to
       make simple noise spike detection as well
    for(niter in seq(3)){

      # Average power in signal vector RX1
      LPIdatalist.final[["RX1"]][["power"]] <-
         LPIaveragePower( LPIdatalist.final[["RX1"]][["cdata"
         ]] , LPIdatalist.final[["TX1"]][["idata"]] ,
         LPIdatalist.final[["RX1"]][["idata"]] , LPIdatalist.
         final[["nData"]] , maxr )

      # Average power in signal vector RX2
      LPIdatalist.final[["RX2"]][["power"]] <-
         LPIaveragePower( LPIdatalist.final[["RX2"]][["cdata"
```

```r
]] , LPIdatalist.final[["TX2"]][["idata"]] , LPIdatalist.final[["RX2"]][["idata"]] , LPIdatalist.final[["nData"]] , maxr )

    # Flag data points whose power is more than four times the average at a given height,
    # but only if there were reasonably many samples in the averages
    if(LPIdatalist.final[["RX1"]][["power"]][1] < .05 ){
        itx1 <- which( abs(LPIdatalist.final[["RX1"]][["cdata"]][1:LPIdatalist.final[["nData"]]]) > (sqrt(LPIdatalist.final[["RX1"]][["power"]])*4) )
        LPIdatalist.final[["RX1"]][["idata"]][itx1] <- FALSE
    }
    if(LPIdatalist.final[["RX2"]][["power"]][1] < .05 ){
        itx2 <- which( abs(LPIdatalist.final[["RX2"]][["cdata"]][1:LPIdatalist.final[["nData"]]]) > (sqrt(LPIdatalist.final[["RX2"]][["power"]])*4) )
        LPIdatalist.final[["RX2"]][["idata"]][itx1] <- FALSE
    }
  }

  # maxr points in the beginning will not have
  # a reasonable  power estimate, flag these points as well
  LPIdatalist.final[["RX1"]][["idata"]][1:maxr] <- FALSE
  LPIdatalist.final[["RX2"]][["idata"]][1:maxr] <- FALSE

  #####################################
  ## Copy parameters from LPIparam to ##
  ## the final data list as necessary ##
  #####################################

  # Lag values
  LPIdatalist.final[["lagLimits"]] <- LPIparam[["lagLimits"]]
  LPIdatalist.final[["nLags"]]     <- length(LPIdatalist.final[["lagLimits"]]) - 1


  # Maximum ranges, repeat the last value as necessary
  LPIdatalist.final[["maxRanges"]] <-  LPIparam[["maxRanges"]]
  nmaxr <- length(LPIdatalist.final[["maxRanges"]])
  if( nmaxr < LPIdatalist.final[["nLags"]] ){
    LPIdatalist.final[["maxRanges"]] <- c( LPIdatalist.final[["maxRanges"]] , rep(LPIdatalist.final[["maxRanges"]][nmaxr],(LPIdatalist.final[["nLags"]]-
      nmaxr)))
  }




  # Range gate limits
  LPIdatalist.final[["rangeLimits"]] <- LPIparam[["
     rangeLimits"]]
  LPIdatalist.final[["nGates"]] <- rep( length(LPIparam[["
     rangeLimits"]]) - 1 , LPIdatalist.final[["nLags"]] )
  for( k in seq(LPIdatalist.final[["nLags"]]) ){
    LPIdatalist.final[["nGates"]][k] <- length( LPIparam[["
       rangeLimits"]][ LPIparam[["rangeLimits"]] <
       LPIdatalist.final[["maxRanges"]][k] ] ) - 1
  }

  # The TX vectors are always decimated
  # in the present version
  LPIdatalist.final[["nDecimTX"]] <- 1

  # Number of theory matrix rows to buffer
  LPIdatalist.final[["nBuf"]] <- LPIparam[["nBuf"]]

  # Inverse problem solver
  LPIdatalist.final[["solver"]] <- LPIparam[["solver"]]

  # Options to rlips
  LPIdatalist.final[["rlips.options"]] <- LPIparam[["rlips.
     options"]]

  # Do we calculate background ACF estimates
  LPIdatalist.final[["backgroundEstimate"]] <- LPIparam[["
     backgroundEstimate"]]

  # Should full covariance matrix or only its
  # diagonal be calculated
  LPIdatalist.final[["fullCovar"]] <- LPIparam[["fullCovar"
     ]]

  # Are we running in a cluster or locally
  LPIdatalist.final[["iscluster"]] <- LPIparam[["iscluster"
     ]]

  # Is the rx data from a remote site?
  LPIdatalist.final[["remoteRX"]] <- LPIparam[["remoteRX"]]

  # Number of codes if pre-averaging is being used
  LPIdatalist.final[["nCode"]] <- LPIparam[["nCode"]]

  # Should interpolation be used when calculating
  # the range ambiguity functions
  LPIdatalist.final[["ambInterp"]] <- LPIparam[["ambInterp"
     ]]

  # Make sure that the storage modes are correct
  storage.mode(LPIdatalist.final[["TX1"]][["cdata"]])  <- "
     complex"
  storage.mode(LPIdatalist.final[["TX2"]][["cdata"]])  <- "
     complex"
  storage.mode(LPIdatalist.final[["TX1"]][["idata"]])  <- "
     logical"
  storage.mode(LPIdatalist.final[["TX2"]][["idata"]])  <- "
     logical"
  storage.mode(LPIdatalist.final[["RX1"]][["cdata"]])  <- "
     complex"
  storage.mode(LPIdatalist.final[["RX2"]][["cdata"]])  <- "
     complex"
  storage.mode(LPIdatalist.final[["RX1"]][["idata"]])  <- "
     logical"
  storage.mode(LPIdatalist.final[["RX2"]][["idata"]])  <- "
     logical"
  storage.mode(LPIdatalist.final[["RX1"]][["power"]])  <- "
     double"
  storage.mode(LPIdatalist.final[["RX2"]][["power"]])  <- "
     double"
  storage.mode(LPIdatalist.final[["lagLimits"]])       <- "
     integer"
  storage.mode(LPIdatalist.final[["rangeLimits"]])     <- "
     integer"
  storage.mode(LPIdatalist.final[["nDecimTx"]])        <- "
     integer"
  storage.mode(LPIdatalist.final[["nBuf"]])            <- "
     integer"
  storage.mode(LPIdatalist.final[["nData"]])           <- "
     integer"
  storage.mode(LPIdatalist.final[["nGates"]])          <- "
     integer"
  storage.mode(LPIdatalist.final[["nLags"]])           <- "
     integer"
  storage.mode(LPIdatalist.final[["nCode"]])           <- "
     integer"
  storage.mode(LPIdatalist.final[["ambInterp"]])       <- "
     logical"
  storage.mode(LPIdatalist.final[["backgroundEstimate"]])
     <- "logical"

  return( LPIdatalist.final )

  }
```

5.2.2 clutterSuppress.R

```r
## file:clutterSuppress.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Ground clutter suppression as follows:
##
## 1. Scattered signal in ranges between rmin and rmax is
##    solved by means of voltage-level inversion.
## 2. The solved profile is convolved with the transmission
##    envelope and the convolution is subtracted from the
##    receiver samples.
##
## Arguments:
##  txdata   A transmitter data list that contains named
##           vectors 'cdata' and 'idata'
##  rxdata   A receiver data list that cntains named
##           vectors 'cdata' and 'idata'
##  rmin     Smallest range from which clutter should
##           be suppressed
##  rmax     Largest range from which clutter should
##           be suppressed
##  ndata    Number of points in data vectors
##  clutterFraction Fraction of the full integration
##           period used for the clutter profile estimation
##           A float from the interval (0,1]
##
## Returns:
##  solution The solved clutter profile
##
## Clutter-suppressed receiver data is written to the
## vector rxdata[["cdata"]]
##

clutterSuppress <- function( txdata , rxdata , rmin , rmax ,
   ndata , clutterFraction )
  {

    # If rmin > rmax there will be nothing to subtract
    if( rmin > rmax ) return()

    # No reason to continue if ndata is not positive
    if( ndata <= 0 ) return()

    # Set negative ranges to zero
    rmin <- max( rmin , 0 )
    rmax <- max( rmax , 0 )


    # Number of range gates to solve
    nr <- rmax - rmin + 1

    # Initialize a fishs object
    e <- fishs.init( ncols = nr )

    # number of points used in clutter profile estimation
    nclutter <- round( ndata * min( clutterFraction , 1 ) )

    # Set correct storage modes
    storage.mode( ndata ) <- "integer"
    storage.mode( nclutter ) <- "integer"
    storage.mode( rmin ) <- "integer"
    storage.mode( rmax ) <- "integer"

    # Add data to the inverse problem
    nrow <- .Call( "clutter_meas",
                  txdata[["cdata"]],
                  txdata[["idata"]],
                  rxdata[["cdata"]],
                  rxdata[["idata"]],
                  ndata,
                  rmin,
                  rmax,
                  e[["Qvec"]],
                  e[["y"]]
                  )

    # Do not subtract if the number of measurement rows
    # is smaller than number of unknowns
    if( nrow < nr ){
      warning("Not enough data points for clutter suppression
         .")
      invisible( NULL )
    }

    # Otherwise solve the inverse problem
    fishs.solve(e)

    # The unmeasured points should be zero instead of NA
    e[["solution"]][is.na(e[["solution"]])] <- 0+0i

    # Do the actual subtraction
    ncor <- .Call( "clutter_subtract",
                  txdata[["cdata"]],
                    txdata[["idata"]],
                    rxdata[["cdata"]],
                    rxdata[["idata"]],
                    ndata,
                    rmin,
                    rmax,
                    e[["solution"]]
                    )

  invisible(e$solution)

}
```

5.2.3 decoFilter.R

```r
## file:decoFilter.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Voltage level decoding, either matched or inverse
## filtering, using measured transmitter samples.
##
## Arguments:
##  cdata       A complex receiver data vector
##  cenv        A complex transmitter data vector
##  idata       A logical vector of transmitter data indices
##  filterType  Decoding filter. Either a complex vector of filter taps, 'matched' or 'inverse'
##
## Returns:
##  cdata The complex receiver data vector after decoding
##

decoFilter.cdata <- function( cdata , cenv , idata , filterType='inverse')
  {
    # Pulse start positions and number of pulses
    txstarts <- which( diff(idata>0) == 1 )
    if(idata[1]) txstarts <- c(0,txstarts)
    ntx <- length(txstarts)
    txstarts <- c(txstarts,length(cdata))


    # If there are no transmission pulses, then simply return
    if(ntx<1) return(cdata)

    # Set the data points before the first pulse to zero
    if( txstarts[1] > 0 ) cdata[1:txstarts[1]] <- 0+0i


    # Set transmitter data to zero at points that are not transmitter samples
    cenv[!idata] <- 0+0i

    # Filtering with user-defined coefficients
    if( is.numeric( filterType ) ){
        nfilter <- length(filterType)
        for( k in seq(  ntx ) ){
            cenv[ (txstarts[k]+1) : (txstarts[k+1]) ] <- 0+0i
            cenv[ (txstarts[k]+1) :  (txstarts[k]+nfilter)] <- filterType
            cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] <-
                fft(
                    fft( cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] ) /
                    fft(  cenv[ (txstarts[k]+1) : (txstarts[k+1]) ] )
                    , inverse=TRUE ) /
                        (txstarts[k+1]-txstarts[k]) * sqrt(sum(abs(cenv[ (txstarts[k]+1) : (txstarts[k+1]) ])**2))
        }
    }else if( is.character( filterType ) ){
        # Inverse filtering
        if(filterType=="inverse"){
            for( k in seq(  ntx ) ){
                cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] <-
                    fft(
                        fft( cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] ) /
                        fft(  cenv[ (txstarts[k]+1) : (txstarts[k+1]) ] )
                        , inverse=TRUE ) /
                            (txstarts[k+1]-txstarts[k]) * sqrt(sum(abs(cenv[ (txstarts[k]+1) : (txstarts[k+1]) ])**2))
            }
        # Matched filtering
        }else if(filterType=="matched"){
            for( k in seq( ntx ) ){
                cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] <-
                    fft(
                        fft( cdata[ (txstarts[k]+1) : (txstarts[k+1]) ] ) *
                        Conj( fft( cenv[ (txstarts[k]+1) : (txstarts[k+1]) ] ) )
                        , inverse=TRUE ) /
                            (txstarts[k+1]-txstarts[k]) / sqrt(sum(abs(cenv[ (txstarts[k]+1) : (txstarts[k+1]) ])**2))
            }
        # Other filters are not supported at the moment
        }else{
            stop("Unknown decoding filter")
        }
    }else{
        stop("Unknown decoding filter")
    }
```

```

  return(cdata)

}

##
## Index corrections for decoded receiver data
##
##
## Arguments:
##  idata A logical vector of transmitter data indices
##
## Returns:
##  idata A corrected index vector with only first index
##        of each pulse set.
##

decoFilter.idata <- function( idata )
  {

    # Pulse start positions
    txstarts <- which( diff(idata>0) == 1 )
    if(idata[1]) txstarts <- c(0,txstarts)
    ntx <- length(txstarts)
    txstarts <- c(txstarts,length(idata))

    # Each pulse should have been compressed into
    # a single sample in the decoding
    for( k in seq( ntx ) ){
      idata[(txstarts[k]+2):txstarts[k+1]] <- FALSE
    }

    return(idata)

}
```

5.2.4 LPIaveragePower.R

```
## file:LPIaveragePower.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Average power profiles
##
## Arguments:
##  cdata     A complex data vector
##  idatatx   A logical vector of transmitter pulse positions
##  idatarx   A logical vector of usable receiver samples
##  ndata     Number points in data vectors
##  maxrange Largest range from which the power is needed
##
## Returns:
##  pdata    Average power profile vector
##

LPIaveragePower <- function( cdata , idatatx , idatarx ,
    ndata , maxrange )
  {
    # Call the C function
    pow <- .Call( "average_power" , cdata , idatatx , idatarx
        , ndata , maxrange )

    # Check the first element, .01 means that number of
    # summed power values is 10 in average.
    # The first element will be NA if no pulses were found,
    # then it does not really matter  what we do..
    if( is.na( pow[1] ) ){
      pow[] <-  mean( abs( cdata[idatarx])**2 )
    }else if( pow[1] > .1 ){
      pow[] <-  mean( abs( cdata[idatarx])**2 )
    }

    return(pow)
  }
```

5.2.5 LPIsaveACF.R

```
## file:LPIsaveACF.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Save resolved ACF to file
##
## Arguments:
##  LPIparam  A LPI parameter list
##  intPeriod Integration period number
##  ACF       An ACF list returned by LPIsolve
##
## Returns:
##  resFile   Result file name
##

LPIsaveACF <- function( LPIparam , intPeriod , ACF )
  {
    # Number of range gates
    ngates <- length(ACF[["range"]])

    # Number of lags
    nlags  <- length(ACF[["lag"]])

    # Seconds since 1970
    ACF[["time.s"]] <-  LPIparam[["startTime"]] + intPeriod*
       LPIparam[["timeRes.s"]]

    # The same time as a string, useful for debugging
    # time conversions and for plotting
    ACF[["timeString"]] <-
      format( as.POSIXct( ACF[["time.s"]] , origin='
         1970-01-01' , tz='UTC' ) , "%Y-%m-%d %H:%M:%OS3 UT")

    # Result file name
    resFile <- gsub(' ','0',file.path( LPIparam[["resultDir"
       ]] , paste( sprintf( '%13.0f' , trunc( ACF[["time.s"]]
         * 1000 ) ) , "LP.Rdata" , sep='') ))

    # Range
    names(ACF[["range"]]) <- paste('gate',seq(ngates),sep='')

    # Lag
    names(ACF[["lag"]]) <- paste('lag',seq(nlags),sep='')

```

```r
    # Background ACF
    ACF[["backgroundACF"]] <- ACF[["ACF"]][(ngates+1),]
    ACF[["backgroundvar"]] <- ACF[["var"]][(ngates+1),]
    names(ACF[["backgroundACF"]]) <- paste('lag',seq(nlags),sep='')
    names(ACF[["backgroundvar"]]) <- paste('lag',seq(nlags),sep='')

    # ACF and variance without the background samples
    ACF[["ACF"]] <- matrix(ACF[["ACF"]][1:ngates,],ncol=nlags)
    ACF[["var"]] <- matrix(ACF[["var"]][1:ngates,],ncol=nlags)
    dimnames(ACF[["ACF"]]) <- list(paste('gate',seq(ngates),sep=''),paste('lag',seq(nlags),sep=''))
    dimnames(ACF[["var"]]) <- list(paste('gate',seq(ngates),sep=''),paste('lag',seq(nlags),sep=''))

    # Dimnames for the optional full covariance matrix
    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=''))

    # Strip off skipped time lags
#    laginds <- apply( ACF[["ACF"]] , FUN=function(x){ any( !is.na( x ) ) } , MARGIN = 2 )
    laginds <- which( c( LPIparam[["maxRanges"]] , rep( LPIparam[["maxRanges"]][length(LPIparam[["maxRanges"]])] , nlags ))[1:nlags] >= LPIparam[["rangeLimits"]][1] )
    ACF <- stripACF( ACF , rgates = seq( ngates ) , lags= laginds , fullCovar=LPIparam[["fullCovar"]])

    # Range gate limits
    ACF[["rangeLimits"]] <- LPIparam[["rangeLimits"]]
    names(ACF[["rangeLimits"]]) <- ""

    # Lag integration limits
    ACF[["lagLimits"]] <- LPIparam[["lagLimits"]]
    names(ACF[["lagLimits"]]) <- ""

    # Maximum ranges
    ACF[["maxRanges"]] <- LPIparam[["maxRanges"]]
    names(ACF[["maxRanges"]]) <- ""

    # Write the output list to the file
    save( ACF=ACF , file=resFile )

    # Return the file name invisibly
    invisible( resFile )

  }
```

5.2.6 stripACF.R

```r
## file:stripACF.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Return an ACF list with only selected ranges and lags
##
## Arguments:
##   ACFlist   An ACF list returned by runLPI or stored by
    LPIsaveACF
##   rgates    Range gate indices
##   lags      Lag indices
##   fulCovar  TRUE if the ACFlist contains the full
    covariance matrices
##
## Returns:
##   ACFlist   A modified ACF list
##

stripACF <- function( ACFlist , rgates , lags , fullCovar=
    FALSE)
  {

    # An empty list for the output
    ACFlist2 <- list()

    # If rgates and lags are logical vectors
    # convert them into indices
    if(is.logical(rgates)) rgates <- which(rgates)
    if(is.logical(lags)) lags <- which(lags)

    # Pick the ACF and variance values
    ACFlist2[["ACF"]] <- ACFlist[["ACF"]][rgates,lags]
    ACFlist2[["var"]] <- ACFlist[["var"]][rgates,lags]

    # Make sure that ACF, var, and covariance are still
        arrays
    dim(ACFlist2[["ACF"]]) <- c( length(rgates) , length(lags
        ) )
    dim(ACFlist2[["var"]]) <- c( length(rgates) , length(lags
        ) )
    if(fullCovar){
      covdims <- dim(ACFlist[["covariance"]])
      ACFlist2[["covariance"]] <- ACFlist[["covariance"]][c(
          rgates,covdims[1]),c(rgates,covdims[2]),lags]
      dim(ACFlist2[["covariance"]]) <- c( (length(rgates)+1) , (length(rgates)+1) , length(lags) )
    }


    ACFlist2[["lag"]] <- ACFlist[["lag"]][lags]
    ACFlist2[["range"]] <- ACFlist[["range"]][rgates]
    ACFlist2[["nGates"]] <- pmin(rep(length(rgates),length(lags)),ACFlist[["nGates"]][lags])
    ACFlist2[["backgroundACF"]] <- ACFlist[["backgroundACF"]][lags]
    ACFlist2[["backgroundvar"]] <- ACFlist[["backgroundvar"]][lags]
    ACFlist2[["timeString"]] <- ACFlist[["timeString"]]
    ACFlist2[["time.s"]] <- ACFlist[["time.s"]]

    # Udpate names to match with the new indexing
    nlags <- length(lags)
    ngates <- length(rgates)

    names(ACFlist2[["range"]]) <- paste('gate',seq(ngates),sep='')
    names(ACFlist2[["lag"]]) <- paste('lag',seq(nlags),sep='')
    names(ACFlist2[["backgroundACF"]]) <- paste('lag',seq(nlags),sep='')
    names(ACFlist2[["backgroundvar"]]) <- paste('lag',seq(nlags),sep='')
    dimnames(ACFlist2[["ACF"]]) <- list(paste('gate',seq(ngates),sep=''),paste('lag',seq(nlags),sep=''))
    dimnames(ACFlist2[["var"]]) <- list(paste('gate',seq(ngates),sep=''),paste('lag',seq(nlags),sep=''))


    return(ACFlist2)

  }
```

5.3 Correlation and inverse problem formulation

5.3.1 laggedProducts.R

```
## file:laggedProducts.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Calculation of lagged products
##
## Arguments:
##   LPIenv An LPI environment
##   lag    Lag number
##
##
## Returns:
##   success  TRUE if at least one lagged product was
##            successfully calculated, FALSE otherwise.
##
## The lagged products are (over)written to
## the vector LPIenv[["cprod."]]
##

laggedProducts <- function( LPIenv , lag )
  {

    # Make sure that the lag number is an integer
    storage.mode(lag) <- "integer"

    # Call the c function
    return( .Call( "lagged_products" ,
                  LPIenv[["RX1"]][["cdata"]] ,
                  LPIenv[["RX2"]][["cdata"]] ,
                  LPIenv[["RX1"]][["idata"]] ,
                  LPIenv[["RX2"]][["idata"]] ,
                  LPIenv[["cprod"]]          ,
                  LPIenv[["iprod"]]          ,
                  LPIenv[["nData"]]          ,
                  LPIenv[["nData"]]          ,
                  lag
                  )
           )
  }
```

5.3.2 lagprodVar.R

```
## file:lagprodVar.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Variances of lagged products. Calculated
## as lagged products of average power values.
##
## Arguments:
##   LPIenv A LPI environment
##   lag    Lag number
##
## Returns:
##   success  TRUE if a variance estimate was successfully
##            calculated for at least one data point,
##            FALSE otherwise.
## The variances are (over)written to LPIenv[["var"]]
##

lagprodVar <- function( LPIenv , lag )
  {

    # Make sure that lag is an integer
    storage.mode(lag) <- "integer"

    # Call the C function
    return( .Call( "lagged_products_r"         ,
                   LPIenv[["RX1"]][["power"]] ,
                   LPIenv[["RX2"]][["power"]] ,
                   LPIenv[["var"]]            ,
                   LPIenv[["nData"]]          ,
                   LPIenv[["nData"]]          ,
                   lag
                   )
           )
  }
```

5.3.3 rangeAmbiguity.R

```r
## file:rangeAmbiguity.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Calculation of range ambiguity functions.
##
## Arguments:
##  LPIenv   A LPI environment
##  lag      Lag number
##
##
## Returns:
##  success   TRUE if at least one point was successfully
##            calculated, FALSE otherwise.
##            The range ambiguity function is
##            (over)written to LPIenv$camb.
##
##
##
##
##

rangeAmbiguity <- function( LPIenv , lag )
  {

    # True oversampling is not supported.
    if( LPIenv[['nDecimTX']] != 1) stop("True transmitter
        signal oversampling is not supported.")

    # Make sure that lag is an integer
    storage.mode(lag) <- "integer"

    # Simulate oversampling by means of interpolation.
    # This works well if the pulses have
    # sharp edges and constant amplitude.
    if( LPIenv[["ambInterp"]] ){
      return( .Call( "range_ambiguity"          ,
                     LPIenv[["TX1"]][["cdata"]] ,
                     LPIenv[["TX2"]][["cdata"]] ,
                     LPIenv[["TX1"]][["idata"]] ,
                     LPIenv[["TX2"]][["idata"]] ,
                     LPIenv[["camb"]]           ,
                     LPIenv[["iamb"]]           ,
                     LPIenv[["nData"]]          ,
                     LPIenv[["nData"]]          ,
                     lag
                     )
              )
    }

    # Simple lagged products of decimated data,
    # works with strong codes.
    return( .Call( "lagged_products"          ,
                   LPIenv[["TX1"]][["cdata"]] ,
                   LPIenv[["TX2"]][["cdata"]] ,
                   LPIenv[["TX1"]][["idata"]] ,
                   LPIenv[["TX2"]][["idata"]] ,
                   LPIenv[["camb"]]           ,
                   LPIenv[["iamb"]]           ,
                   LPIenv[["nData"]]          ,
                   LPIenv[["nData"]]          ,
                   lag
                   )
            )

  }
```

5.3.4 averageProfiles.R

```
## file:averageProfiles.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Lag-profile pre-averaging before the actual inversion.
## Provides significant speed-up but may lead to somewhat
## reduced estimation accuracy
##
## This routine is intended to be used in real-time
## analysis with limited computing resources when speed
## gain with reduced accuaracy and flexibility is accepable.
##
##
## Arguments:
##  LPIenv A LPI environment
##  l      Lag number
##
## Returns:
##  success TRUE if both lagged products and range ambiguity
##          functions were successfully averaged.
##
## The averaged profiles are overwritten to
## LPIenv[["cprod"]] and LPIenv[["camb"]]
##

averageProfiles <- function( LPIenv , l )
  {

    s1 <- .Call( "average_profile" , LPIenv[["cprod"]] ,
       LPIenv[["TX1"]][["idata"]] , as.integer( LPIenv[["
       nData"]] - 1 ) , as.integer( LPIenv[["nCode"]] ) )

    s2 <- .Call( "average_profile" , LPIenv[["camb"]]  ,
       LPIenv[["TX1"]][["idata"]] , as.integer( LPIenv[["
       nData"]] - 1 ) , as.integer( LPIenv[["nCode"]] ) )

    invisible( ( s1 & s2 ) )

  }
```

5.3.5 theoryRows.R

```
## file:theoryRows.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Form theory matrix rows for lag profile inversion
##
## Arguments:
##   LPIenv    A LPI environment
##   lag       Lag number
##
## Returns:
##   success  TRUE if at least one theory matrix row
##            was successfully produces, FALSE otherwise.
##
## The rows are written to LPIenv[["arows"]],
## the correspoding measurements to LPIenv[["meas"]],
## variance to LPIen[["mvar"]], and number of rows
## generated to LPIenv[["nrows"]]
##
##

theoryRows <- function( LPIenv , lag )
  {

    # Call the C routine
    return( .Call( "theory_rows" ,
                  LPIenv[['camb']] ,
                  LPIenv[['iamb']] ,
                  LPIenv[['cprod']],
                  LPIenv[['iprod']],
                  LPIenv[['var']] ,
                  LPIenv[['nData']] ,
                  LPIenv[['nCur']] ,
                  as.integer(LPIenv[['nCur']]+LPIenv[['nBuf']]) ,
                  LPIenv[['rangeLimits']] ,
                  LPIenv[['nGates']][lag] ,
                  LPIenv[['arows']] ,
                  LPIenv[['irows']] ,
                  LPIenv[['meas']] ,
                  LPIenv[['mvar']],
                  LPIenv[['nrows']],
                  LPIenv[["backgroundEstimate"]],
                  LPIenv[["remoteRX"]]
```

```
                            )
                )
    }
```

5.4 Inverse problem solvers

5.4.1 fishs.init.R

```r
## file:fishs.init.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Linear inverse problem solution by means of direct
   calculation
## of Fisher information matrix. Initialization function.
##
## Arguments:
##  ncols Number of unknowns (theory matrix columns)
##
## Returns:
##  s     A fishs solver environment
##

fishs.init <- function( ncols , ... )
  {
    # New environment for the solver
    s <- new.env()

    # Number of columns in the theory matrix
    assign( 'ncol' , ncols , s )

    # A vector for upper triangular part of
    # the Fisher information matrix
    assign( 'Qvec' , rep(0+0i,(ncols*(ncols+1)/2)) , s )

    # A vector for weighted measurements
    assign( 'y'    , rep(0+0i,ncols) , s )

    # Make sure that the storage modes are
    # correct for later c function calls
    storage.mode(s$Qvec) <- storage.mode(s$y) <- "complex"
    storage.mode(s$ncol) <- "integer"

    return(s)

  }
```

5.4.2 fishs.add.R

```
## file:fishs.add.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Linear inverse problem solution by means of direct
## calculationof Fisher information matrix.
## Data accumulation function.
##
## Arguments:
##  e      A fishs solver environemnt
##  A.data Theory matrix rows as a vector (row-by-row)
##  M.data Measurement vector
##  E.data Measurement variance vector
##
## Returns:
##  success TRUE if the rows were successfully added.
##

fishs.add <- function( e , A.data ,  M.data ,  E.data=1 )
  {

    # Number of theory rows to add
    nrow <- as.integer(length(M.data))

    # Variance vector
    E.data <- rep(E.data,length.out=nrow)

    # Check storage modes before calling the c function
    storage.mode(A.data) <- "complex"
    storage.mode(M.data) <- "complex"
    storage.mode(E.data) <- "double"
    storage.mode(nrow)   <- "integer"

    # Call the c function
    return( .Call( "fishs_add" , e[["Qvec"]] , e[["y"]] , A.
       data , M.data , E.data , e[["ncol"]] , nrow ))

  }
```

5.4.3 fishs.solve.R

```
## file:fishs.solve.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Linear inverse problem solution by means of direct
## calculation of Fisher information matrix.
## Final solver function.
##
## Arguments:
##  e                 A fishs solver environment
##  full.covariance Logical, full covariance matrix is
   calculated
##                    if TRUE, otherwise only variances are
   returned.
##
## Returns:
##  Nothing, the solution is assigned to
##  the solver environment
##

fishs.solve <- function( e , full.covariance = TRUE , ... )
  {

    # Allocate a matrix for the full
    # Fisher information matrix
    Q <- matrix( 0 , ncol=e[["ncol"]] , nrow=e[["ncol"]] )

    # Copy the upper triangular part form e$Qvec
    i <- 1
    for( k in seq( e$ncol ) ){
      Q[ k , k : e[["ncol"]] ] <- e[["Qvec"]][ i : ( i + ( e
          [["ncol"]] - k ) ) ]
      i <- i + e[["ncol"]] - k + 1
    }

    # The lower triangular part is
    # complex conjugate of the upper one
    Q          <- Q + Conj( t( Q ) )

    # The above row multiplies the diagonal
    # with 2, divide accordingly
    diag( Q)  <- diag( Q ) / 2

    # Select points at which the diagonal of Q is zero,
    # these points have not been measured at all and
    # need to be regularized before inverting the matrix
    nainds     <- Re( diag( Q ) ) == 0

    # Set unit values on the diagonal at unmeasured points.
    # This will not affect the other unknowns because
    # they cannot correlate with this one
    diag( Q )[ nainds ] <- 1

    # Covariance matrix is inverse matrix of
    # the Fisher information matrix
    # Even if there were measurements the matrix might not be
       invertible
    # return NA matrix in this case
    covariance          <- tryCatch( solve( Q ) , error=
      function(e){Q*NA})

    # Multiply the covariance matrix with e$y from right.
    # For some reason the direct matrix multiplication
    # with %*% does not work properly in some machines.
    solution <- rep(0+0i,e[["ncol"]])
    for(k in seq(e[["ncol"]])) solution[k] <- sum( covariance
      [k,] * e[["y"]] )

    # Set NAs to points that were not actually measured
    solution[ nainds ]  <- NA

    # Assign the solution to the solver environment e
    assign( 'solution'   , solution , e )

    # The full covariance matrix was already calculated, pick
    # the diagonal if that is enough.
    # Put NA to unmeasured points.
    if( full.covariance ){
      covariance[ nainds ,        ] <- NA
      covariance[         , nainds ] <- NA
    }else{
      covariance                    <- diag( covariance )
      covariance[ nainds ]          <- NA
    }

    # Assign the covariance to the solver environment e
    assign( 'covariance' , covariance , e )

    invisible()

  }
```

5.4.4 deco.init.R

```r
## file:deco.init.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Matched filter decoder. Initialization function.
##
## Arguments:
##   ncols Number of unknowns (theory matrix columns)
##   ...   Additional arguments are allowed by not used
##         in order to make the solver more compatible
##         with others.
##
## Returns:
##   e    A deco solver environment
##

deco.init <- function( ncols , ... )
  {

    # A new environment for the solver
    s <- new.env()

    # Number of columns in theory matrix
    assign( 'ncol' , ncols , s )

    # Diagonal of the Fisher information matrix
    assign( 'Qvec' , rep(0,ncols) , s )

    # Scaled measurements
    assign( 'y'    , rep(0,ncols) , s )

    # Make sure that the storage modes are
    # correct for later c function calls
    storage.mode(s$Qvec) <- storage.mode(s$y) <- "complex"
    storage.mode(s$ncol) <- "integer"

    # return the environment
    return(s)

  }
```

5.4.5 deco.add.R

```
## file:deco.add.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Matched filter decoder. Data accumulation function.
##
## Arguments:
##  e      A deco solver environemnt
##  A.data Theory matrix rows as a vector (row-by-row)
##  M.data Measurement vector
##  E.data Measurement variance vector
##
## Returns:
##  success TRUE if the rows were successfully added.
##


deco.add <- function( e , A.data ,  M.data ,  E.data=1 )
  {
    # Number of theory rows
    nrow <- as.integer(length(M.data))

    # Measurement variance vector
    E.data <- rep(E.data,length.out=nrow)

    # Set storage modes
    storage.mode(A.data) <- "complex"
    storage.mode(M.data) <- "complex"
    storage.mode(E.data) <- "double"
    storage.mode(nrow)   <- "integer"

    # Call the c routine
    return( .Call( "deco_add" , e$Qvec , e$y , A.data , M.
        data , E.data , e$ncol , nrow ))

  }
```

5.4.6 deco.solve.R

```r
## file:deco.solve.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Matched filter decoder. Final solver function.
##
## Arguments:
##  e                 A deco solver environment
##  full.covariance Logical, full covariance matrix is
##                  calculated if TRUE, otherwise only
##                  variances are returned.
##
## Returns:
##  Nothing, the solution is assigned to
##  the solver environment
##

deco.solve <- function( e , ... )
  {
    # Diagonal of the Fisher information matrix
    # (Matched filter decoding is equivalent with assuming
    # that the nondiagonal elements are zeros)
    Qdiag <- e[["Qvec"]]

    # The points at which Qdiag is zero were not measured
    # at all, flag these points
    nainds <- Qdiag == 0

    # Put unit values to the unmeasured points. This does
    # not affect the other points as they cannot
    # correlated with the unmeasured ones.
    Qdiag[nainds] <- 1

    # Variance is simply the inverse of the diagonal
    # of the Fisher information
    variance  <- 1 / Qdiag

    # Assign the solution to the solver environment
    assign( 'solution'   , variance * e[["y"]] , e )

    # Set NAs to the unmeasured points
    e[["solution"]][nainds] <- NA

    # Same for the variances
    assign( 'covariance' , variance , e )
    e[["covariance"]][nainds] <- NA

    invisible()

  }
```

5.4.7 dummy.init.R

```
## file:dummy.init.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Dummy inverse problem solver that calculates
## simple averages.
## Initialization function.
##
## Arguments:
##  rrange extreme ranges to be solved c(rmin,rmax)
##
## Returns:
##  s      A dummy solver environment
##

dummy.init <- function( rrange )
  {

    # A new environment for the solver
    s <- new.env()

    # Number of ranges (this is different
    # from number of final range gates)
    nr <- abs(diff(rrange))

    # A vector for sum of weighted measurements
    msum <- rep(0+0i,nr)

    # A vector for sum of information
    vsum <- rep(0,nr)

    # Minimum range
    rmin <- min(rrange)

    # Maximum range
    rmax <- max(rrange)

    # Make sure that storage modes are correct
    storage.mode(msum) <- "complex"
    storage.mode(vsum) <- "double"
    storage.mode(rmin) <- "integer"
    storage.mode(rmax) <- "integer"

    # Assign the variables to the environment
    assign( 'msum' , msum , s )
    assign( 'vsum' , vsum , s )
    assign( 'rmin' , rmin , s )
    assign( 'rmax' , rmax , s )

    # Return the environment

    return(s)

  }
```

5.4.8 dummy.add.R

```
## file:dummy.add.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Dummy inverse problem solver that
## calculates simple averages.
## Data accumulation function.
##
## Arguments:
##  e       A dummy solver environemnt
##  M.data  Measurement vector
##  M.ambig Range ambiguity function
##  I.ambig Indices of non-zero ambiguity values
##  I.prod  Indices of usable lagged products
##  E.data  Measurement variance vector
##  nData   Number of points in data vectors
##
## Returns:
##  success TRUE if the data was successfully added
##

dummy.add <- function( e , M.data , M.ambig , I.ambig , I.
   prod , E.data , nData )
  {

    # Call the C routine
    return( .Call( "dummy_add" ,
                  e[["msum"]] ,
                  e[["vsum"]] ,
                  e[["rmin"]] ,
                  e[["rmax"]] ,
                  M.data ,
                  M.ambig ,
                  I.ambig ,
                  I.prod ,
                  E.data ,
                  nData)
           )

  }
```

5.4.9 dummy.solve.R

```
## file:dummy.solve.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## Dummy inverse problem solver that
## calculates simple averages.
## Final solver function.
##
## Arguments:
##  e      A dummy solver environment
##  rlims  Range gate limits
##
## Returns:
##  Nothing, the solution is assigned to
##  the solver environment
##

dummy.solve <- function( e , rlims )
  {
    #
    #
    # Final solver function.
    #
    # I. Virtanen 2012
    #

    # Number of range gates
    nr <- length(rlims) - 1

    # Vectors for the solution and variance
    solution <- rep(0+0i,nr)
    covariance <- rep(0,nr)

    # Range integration for the data points that have
    # the best possible resolution at this point.
    for( r in seq(nr) ){

      # Lower limit of this range gates
      r1 <- rlims[r] - rlims[1] + 1

      # Upper limit of this range gate
      r2 <- rlims[r+1] - rlims[1]

      # The vector e$msum contains variance weighted sum,
      # we can simply sum its elements.
      solution[r] <- sum(e[["msum"]][r1:r2])

      # The vector e$vsum contains informations, sum them.
      covariance[r] <- sum(e[["vsum"]][r1:r2])

    }

    # Variance is inverse of the information
    covariance <- c( 1/covariance , NA )

    # Multiply the solution with the final variances
    solution <- c( solution , NA ) * covariance

    # Vectors solution and covariance will now contain
    # variance-weighted averages of the lag profiles
    # and their variances. Assign to the solver environment
    assign( 'solution'   , solution   , e )
    assign( 'covariance' , covariance , e )

    invisible()

  }
```

5.4.10 ffts.init.R

```
## file:ffts.init.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## FFT deconvolution.
## Initialization function.
##
## Arguments:
##  rrange Extreme ranges to be solved c(rmin,rmax)
##  itx    A logical vector of transmitter pulse positions.
##
## Returns:
##  s      A ffts solver environment
##

ffts.init <- function( rrange , itx )
  {
    # Minimum range
    rmin         <- min( rrange )

    # Maximum range
    rmax         <- max( rrange )

    # longest inter-pulse period
    ippmax <- max( diff( which( diff( itx > 0 ) == 1 ) ) ,
       showWarnings=FALSE )

    # Select the FFT length
    n <- max( nextn( ippmax ) , nextn( rmax*2 ) )

    # Allocate vectors
    fy           <- rep( 0+0i , n )
    amb.tmp      <- famb.tmp          <- rep( 0+0i , n )
    meas.tmp     <- rep( 0+0i , n )
    sqfamb       <- rep( 0 , n )
    varsum       <- 0
    nmeas        <- 0

    # Set storage modes
    storage.mode( rmin )      <- "integer"
    storage.mode( rmax )      <- "integer"
    storage.mode( n )         <- "integer"
    storage.mode(nmeas)       <- "integer"
    storage.mode( fy )        <- "complex"
    storage.mode( amb.tmp )  <- "complex"
    storage.mode( famb.tmp ) <- "complex"
    storage.mode( meas.tmp ) <- "complex"
    storage.mode( sqfamb )   <- "double"
    storage.mode(varsum)     <- "double"

    # Create a new environment and assign everything to it
    s <- new.env()
    assign( 'n'          , n          , s )
    assign( 'rmin'       , rmin       , s )
    assign( 'rmax'       , rmax       , s )
    assign( 'fy'         , fy         , s )
    assign( 'sqfamb'     , sqfamb     , s )
    assign( 'amb.tmp'    , amb.tmp    , s )
    assign( 'famb.tmp'   , famb.tmp   , s )
    assign( 'meas.tmp'   , meas.tmp   , s )
    assign( 'nmeas'      , nmeas      , s )
    assign( 'varsum'     , varsum     , s )

    # return the environment
    return( s )

  }
```

5.4.11 ffts.add.R

```
## file:ffts.add.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## FFT deconvolution.
## Data accumulation function.
##
## Arguments:
##  e        An ffts solver environemnt
##  M.data   Measurement vector
##  M.ambig  Range ambiguity function
##  I.ambig  Indices of non-zero ambiguity values
##  I.prod   Indices of usable lagged products
##  E.data   Measurement variance vector
##  nData    Number of points in data vectors
##
## Returns:
##  success TRUE if the data was successfully added
##

ffts.add <- function( e , M.data , M.ambig , I.ambig , I.prod
     , E.data , nData )
  {
    #
    # FFT deconvolution. Data accumulation function.
    #
    # I. Virtanen 2012
    #

    # Return immediately if the ambiguity
    # function is zero at all points
    if( ! any( I.ambig[1:nData] ) ) return()

    # Remove possibly remaining non-zero values
    # from points with unset index vector
    M.data[  which(!I.prod)  ] <- 0+0i
    E.data[  which(!I.prod)  ] <- 0
    M.ambig[ which(!I.ambig) ] <- 0+0i

    # Locate pulse start positions
    ps <- which( diff( I.ambig[1:nData] > 0 ) == 1 )

    # The first point should be adjusted to pulse start,
    # so it is safe to use if the index is set
    if( I.ambig[1] ) ps <- c( 1 , ps )
    npulse <- length( ps )

    # Locate pulse end positions
    pe <- which( diff( I.ambig[1:nData] > 0 ) == -1 )

    # pe and ps should be of the same length,
    # but check anyway...
    npulse <- min( length(pe) , length(ps) )

    # Add data from one IPP at a time
    for( k in seq( npulse ) ){

      # Set temporary vectors to zero
      e[["amb.tmp"]][] <- e[["meas.tmp"]][] <- 0.+0.i

      # Pulse end or data end (should always be pulse end,
      # but check anyway)
      pe1              <- min( nData , pe[k] )

      # max range or data end
      pe2              <- min( nData , ( ps[k] + e[["n"]] - 1
          ) )

      # Copy one pulse
      e[["amb.tmp"]][ 1 : ( pe1 - ps[k] + 1 ) ]   <- M.ambig[
          ps[k] : pe1 ]

      # Take fft
      e[["famb.tmp"]][] <- fft( e[["amb.tmp"]] )

      # Copy data
      e[["meas.tmp"]][ 1 : ( pe2 - ps[k] + 1 ) ] <- M.data[
         ps[k] : pe2 ]

      # Actual addition to the solver
      e[["fy"]][]     <- e[["fy"]]     + Conj( e[["famb.tmp"
         ]] ) * fft( e[["meas.tmp"]] )
      e[["sqfamb"]][] <- e[["sqfamb"]] + abs( e[["famb.tmp"]]
          )**2

    }

    # Variances
    e[["varsum"]] <- e[["varsum"]] + sum( E.data[ 1 : nData ]
        )
    e[["nmeas"]]  <- e[["nmeas"]]  + sum( ( I.prod[ 1 : nData
        ] > 0 ) )

    invisible()

  }
```

5.4.12 ffts.solve.R

```
## file:ffts.solve.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

##
## FFT deconvolution.
## Final solver function.
##
## Arguments:
##  e      A ffts solver environment
##  rlims  Range gate limits
##
## Returns:
##  Nothing, the solution is assigned to the solver
   environment
##
ffts.solve <- function( e , rlims )
  {
    #
    # FFT deconvolution. Final solver function.
    #
    # I. Virtanen 2012
    #

    # Solve the lag profile by means of FFT
    sol <- fft( e[["fy"]]   / e[["sqfamb"]] , inverse=TRUE )
       / e[["n"]]

    # Variance, the same value will be repeated at all ranges
    var <- e[["varsum"]] / as.double(e[["nmeas"]])  * mean( 1
       / e[["sqfamb"]] )

    # Number of range gates
    nr  <- length(rlims) - 1

    # Final solution and variance vectors
    solution   <- rep(0+0i,nr)
    covariance <- rep(0,nr)

    for( r in seq(nr) ){

      # Lower limit of range gate
      r1             <- rlims[r]  + 1

      # Upper limit of range gate
      r2            <- rlims[r+1]

      # All points have equal variances, calculate simple average
      solution[r]   <- mean( sol[r1:r2] , na.rm=TRUE )

      # Scale the variance
      covariance[r] <- var/(r2-r1+1)
    }

    # The background ACF cannot be measured with this technique, set it to NA.
    covariance <- c( covariance , NA )
    solution   <- c( solution , NA )

    # Assign the results to the solver environment.
    assign( 'solution'   , solution   , e )
    assign( 'covariance' , covariance , e )

    invisible()

  }
```

5.4.13 rlips.solve2.R

```
## file:rlips.solve2.R
## (c) 2010- University of Oulu, Finland
## Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
## Licensed under FreeBSD license.
##

## Call rlips.solve after regularization for
## unknowns that were not measured at all
## Set the corresponding values to NA before returning
##
## Arguments:
##  e               An rlips solver environment
##  fullCovariance  Logical, if TRUE full covariance matrix
##                  is calculated, otherwise only the
##                  variances.
## Returns:
##  Nothing, the solution is assigned to the
##  solver environment.
##

rlips.solve2 <- function( e , full.covariance = TRUE )
  {
    # Read data from gpu memory
    rlips.get.data( e )

    # Select non-measured points
    nainds <- which( Re( diag( e$R.mat ) ) == 0 )

    # Add regularizing imaginary measurements
    regrow <- rep(0+0i,e$ncols)
    for( n in nainds ){
      regrow[]  <- 0+0i
      regrow[n] <- 1+0i
      rlips.add( e , A.data = regrow , M.data = 1.0+0.0i )
    }

    # Solve the problem
    rlips.solve( e , calculate.covariance = TRUE , full.
       covariance = full.covariance )

    # Set NAs to appropriate points in the solution
    sol <- e$solution
    sol[nainds] <- NA
    assign( 'solution' , sol , e )

    # Set the unmeasured points to NA
    # in the covariance matrix as well.
    covar <- e$covariance
    if( full.covariance ){
      covar[ , nainds ] <- NA
      covar[ nainds , ] <- NA
    }else{
      covar[nainds] <- NA
    }

    # Assign the covariance matrix to the solver environment
    assign( 'covariance' , covar , e )

    invisible()

  }
```

5.5 C functions and headers

5.5.1 src/LPI.h

```
// file:LPI.h
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

// Data types and function prototypes

#include <R.h>
#include <math.h>
#include <stdint.h>
#include <Rinternals.h>
#include <Rdefines.h>
#include <R_ext/Rdynload.h>

static const double pi=3.1415926535;
#define AMB_N_INTERP  5


// gdf file input
SEXP read_gdf_data_R( SEXP ndata , SEXP nfiles , SEXP
    filepaths , SEXP istart , SEXP iend , SEXP bigendian);
SEXP read_gdf_data( SEXP cata , SEXP idatar , SEXP idatai ,
    SEXP ndata , SEXP nfiles, SEXP filepaths , SEXP istart ,
    SEXP iend , SEXP bigendian);

// Frequency mixing
SEXP mix_frequency_R( SEXP cdata , SEXP ndata , SEXP
    frequency);
SEXP mix_frequency( SEXP cdata , SEXP ndata , SEXP frequency)
    ;

// Index adjustments
SEXP index_adjust_R( SEXP idata , SEXP ndata , SEXP shifts );
SEXP index_adjust( SEXP idata , SEXP ndata , SEXP shifts );

// Lagged products
SEXP lagged_products_alloc( SEXP cdata1 , SEXP cdata2 , SEXP
    idata1 , SEXP idata2 , SEXP ndata1 , SEXP ndata2 , SEXP
    lag);
SEXP lagged_products( SEXP cdata1 , SEXP cdata2 , SEXP idata1
     , SEXP idata2 , SEXP cdatap , SEXP idatap , SEXP ndata1 ,
     SEXP ndata2 , SEXP lag );
SEXP lagged_products_r( SEXP rdata1 , SEXP rdata2 , SEXP
    prdata , SEXP ndata1 , SEXP ndata2 , SEXP lag );
```

```c

// Theory matrix construction
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 );
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 );

// Inverse problem solvers
SEXP fishs_add( const SEXP Qvec, const SEXP yvec , const SEXP
     arows , const SEXP meas , const SEXP var , const SEXP nx
    , const SEXP nrow  );
SEXP deco_add(  const SEXP Qvec , const SEXP yvec , const
    SEXP arows , const SEXP meas , const SEXP var , const SEXP
     nx , const SEXP nrow );
SEXP dummy_add( SEXP msum , SEXP vsum , SEXP rmin , SEXP rmax
     , SEXP mdata , SEXP mambig , SEXP iamb , SEXP iprod ,
    SEXP edata , SEXP ndata );

// All data preparations collected together
SEXP prepare_data( SEXP cdata , SEXP idata , SEXP ndata ,
    SEXP frequency, SEXP shifts , SEXP nup , SEXP nfilter ,
    SEXP nfirst , SEXP nfirstfrac , SEXP ipartial );

// Average signal power in points withe identical IPPs and
    pulse lengths
SEXP average_power( SEXP cdata , SEXP idatatx , SEXP idatarx
    , SEXP ndata , SEXP maxrange);

// Average lag profile
SEXP average_profile( SEXP cdata , SEXP idata , SEXP ndata ,
    SEXP N_CODE);

// Resampling
SEXP resample( SEXP cdata , SEXP idata , SEXP ndata , SEXP
    nup , SEXP nfilter , SEXP nfirst , SEXP nfirstfrac , SEXP
    ipartial);
SEXP resample_R( SEXP cdata , SEXP idata , SEXP ndata , SEXP
    nup , SEXP nfilter , SEXP nfirst , SEXP nfirstfrac , SEXP
    ipartial);

// Range ambiguity function calculation with optional
    interpolation
SEXP range_ambiguity( SEXP cdata1 ,SEXP cdata2 , SEXP idata1
    , SEXP idata2 , SEXP cdatap , SEXP idatap , SEXP ndata1 ,
     SEXP ndata2 ,  SEXP lag );

// Ground clutter suppression
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 );
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 )
    ;
void fishs_add_clutter( const SEXP Qvec , const SEXP yvec ,
    Rcomplex * arow , Rcomplex * meas , const int nx );
```

5.5.2 register.c

```
// file:register.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

// R registration of C functions

#include "LPI.h"
static const R_CallMethodDef callMethods[20] = {
  { "read_gdf_data_R"        , (DL_FUNC) & read_gdf_data_R
          , 6 } ,
  { "mix_frequency_R"        , (DL_FUNC) & mix_frequency_R
          , 3 } ,
  { "index_adjust_R"         , (DL_FUNC) & index_adjust_R
           , 3 } ,
  { "lagged_products_alloc" , (DL_FUNC) &
     lagged_products_alloc , 7 } ,
  { "lagged_products"        , (DL_FUNC) & lagged_products
          , 9 } ,
  { "lagged_products_r"      , (DL_FUNC) & lagged_products_r
        , 6 } ,
  { "fishs_add"              , (DL_FUNC) & fishs_add
                , 7 } ,
  { "theory_rows_alloc"      , (DL_FUNC) & theory_rows_alloc
        , 13} ,
  { "theory_rows"            , (DL_FUNC) & theory_rows
            , 17} ,
  { "prepare_data"           , (DL_FUNC) & prepare_data
             , 10} ,
  { "average_power"          , (DL_FUNC) & average_power
            , 5 } ,
  { "deco_add"               , (DL_FUNC) & deco_add
                 , 7 } ,
  { "average_profile"        , (DL_FUNC) & average_profile
          , 4 } ,
  { "dummy_add"              , (DL_FUNC) & dummy_add
                , 10} ,
  { "resample"               , (DL_FUNC) & resample
                 , 8 } ,
  { "resample_R"             , (DL_FUNC) & resample_R
               , 8 } ,
  { "range_ambiguity"        , (DL_FUNC) & range_ambiguity
          , 9 } ,
  { "clutter_meas"           , (DL_FUNC) & clutter_meas
             , 9 } ,
  { "clutter_subtract"       , (DL_FUNC) & clutter_subtract
         , 8 } ,
  { NULL , NULL , 0 }
};

void R_init_LPI(DllInfo *info)
{
  R_registerRoutines( info , NULL , callMethods , NULL , NULL
       );
}
```

5.5.3 clutter_meas.c

```c
// file:clutter_meas.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*

  Ground clutter suppression. This function adds clutter
  signal measurements to an inverse problem. The function
  clutter_subtract subtracts clutter contribution from a
  signal.

  Arguments:
   tcdata  Complex transmitter samples
   tidata  Transmitter sample indices
   rcdata  Complex receiver samples
   ridata  Receiver sample indices
   ndata   Data vector length
   rmin    Minimum range
   rmax    Maximum range
   Qvec    Upper triangular part of Fisher information matrix
   yvec    Modified measurement vector

  Returns:
   nrow    Number of measurement rows in the inverse problem

*/
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 )
{
  Rcomplex *tcd = COMPLEX( tcdata );
  int *tid = LOGICAL( tidata );
  Rcomplex * rcd = COMPLEX( rcdata );
  int *rid = LOGICAL( ridata );
  const int nd = *INTEGER( ndata );
  const int r0 = *INTEGER( rmin );
  const int r1 = *INTEGER( rmax );

  int i;
  int j;
  int k;
  int r;
  int isum;
  int nx;
  SEXP nrow;
  int nr;

  // Output
  PROTECT( nrow = allocVector( INTSXP , 1 ) );

  // Make sure that the data vectors contain non-zero
  // values only at points in which the logical vectors
  // are not set
  for( i = 0 ; i < nd ; ++i){
    if( tid[i]==0 ){
      tcd[i].r = 0.0;
      tcd[i].i = 0.0;
    }
    if( rid[i]==0 ){
      rcd[i].r = 0.0;
      rcd[i].i = 0.0;
    }
  }

  // Initialization
  nr = 0;
  nx = r1 - r0 + 1;
  r = 0;
  isum = 0;
  // Sum tx indices and set r
  for( i = 0 ; i <= r1 ; ++i ){
    // The largest range is corresponds to index 0,
    // after nx samples we will be below rmin.
    if( i < nx ) isum += tid[i];
    // Increment r
    ++r;
    // Set r to zero if a transmitter sample is meat
    if( tid[i] ) r = 0;
    // increment the rx data pointer
    ++rcd;
  }

  // Go through all data points
  for( i = r1 ; i < nd ; ++i ){
    // Set r = 0 if a transmitter sample is meat
    if( tid[i] ) r = 0;
    // Are we below rmax?
    if( r <= r1 ){
      // Are we above rmin?
      if( r >= r0 ){
    // Are the pulses within the clutter ranges?
    if( isum ){
      // Is this receiver sample usable?
      if( rid[i] ){
        // Add a measurement
        fishs_add_clutter( Qvec , yvec , tcd , rcd , nx );
        // Increment measurement row counter
        ++nr;
      }
    }
      }
    }
    // Update counters if this was not the last sample
    if( i < nd ){
      isum -= tid[ i - r1 ];
      isum += tid[ i - r0 + 1 ];
      ++r;
      ++rcd;
      ++tcd;
    }
  }

  // Copy the number of rows to output
  *INTEGER( nrow ) = nr;

  UNPROTECT(1);

  // Return number of measured rows
  return( nrow );

}
```

5.5.4 clutter_subtract.c

```
// file:clutter_subtract.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"
/*

  Ground clutter suppression. This function subtracts clutter
  signal from data.

  Arguments:
   tcdata  Complex transmitter samples
   tidata  Transmitter sample indices
   rcdata  Complex receiver samples
   ridata  Receiver sample indices
   ndata   Data vector length
   rmin    Minimum range
   rmax    Maximum range
   cldata  Measured clutter signal profile

  Returns:
   nrow    Number of points at which clutter
           signal was suppressed

*/

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 )
{
  Rcomplex *tcd = COMPLEX( tcdata );
  int *tid = LOGICAL( tidata );
  Rcomplex * rcd = COMPLEX( rcdata );
  int *rid = LOGICAL( ridata );
  Rcomplex *cld = COMPLEX( cldata );
  const int nd = *INTEGER( ndata );
  const int r0 = *INTEGER( rmin );
  const int r1 = *INTEGER( rmax );

  int i;
  int j;
  int k;
  int r;
  int isum;
  int nx;
  SEXP nrow;
  int nr;
  Rcomplex clsum;
  Rcomplex * tcd2;
  Rcomplex * cld2;

  // Output
  PROTECT( nrow = allocVector( INTSXP , 1 ) );

  // Initialization
  nr = 0;
  nx = r1 - r0 + 1;
  r = 0;
  isum = 0;
  // Sum tx indices and set r
  for( i = 0 ; i <= r1 ; ++i ){
    // The largest range is corresponds to index 0,
    // after nx samples we will be below rmin.
    if( i < nx ) isum += tid[i];
    // Increment r
    ++r;
    // Set r to zero if a transmitter sample is meat
    if( tid[i] ) r = 0;
    // increment the rx data pointer
    ++rcd;
  }

  // Go through all data points
  for( i = r1 ; i < ( nd - nx )  ; ++i ){
    // Set r = 0 if a transmitter sample is meat
    if( tid[i] ) r = 0;
    // Are we below rmax?
    if( r <= r1 ){
      // Are we above rmin?
      if( r >= r0 ){
    // Are the pulses within the clutter ranges?
    if( isum ){
      // Is this receiver sample usable?
      if( rid[i] ){
        // Calculate clutter contribution and subtract it
        clsum.r = 0.;
        clsum.i = 0.;
        tcd2 = tcd;
        cld2 = cld;
        for( j = 0 ; j < nx ; ++j ){
          clsum.r += tcd2->r * cld2->r - tcd2->i * cld2->i;
          clsum.i += tcd2->r * cld2->i + tcd2->i * cld2->r;
          ++tcd2;
          ++cld2;
        }
        rcd->r -= clsum.r;
        rcd->i -= clsum.i;
        // Increment measurement row counter
        ++nr;
      }
    }
      }
    }
    // Update counters if this was not the last sample
    if( i < nd ){
      isum -= tid[ i - r1 ];
      isum += tid[ i - r0 + 1 ];
      ++r;
      ++rcd;
      ++tcd;
    }
  }

  // Copy the number of rows to output
  *INTEGER( nrow ) = nr;

  UNPROTECT(1);

  // Return number of measured rows
  return( nrow );

}
```

5.5.5 dummy_add.c

```
// file:dummy_add.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
   Simple variance- and power-weighted average lag profile.
   Works only below one IPP range.

   Arguments:
    msum   Sum of normalised measurements
    vsum   sum of normalised inverse variances
    rmin   Lower edge of the measurement
    rmax   Upper edge
    mdata  Complex measurement vector (lag profile)
    mamb   Complex range ambiguity function
    iamb   Range ambiguity function indices
    iprod  Lagged product indices
    edata  Measurement variances
    ndata  Data vector length

   Returns:
    success 1 if the processing was succesful, 0 otherwise

*/

SEXP dummy_add( SEXP msum , SEXP vsum , SEXP rmin , SEXP rmax
     , SEXP mdata , SEXP mamb , SEXP iamb , SEXP iprod , SEXP
    edata , SEXP ndata )
{
  Rcomplex *ms = COMPLEX(msum);
  double *vs = REAL(vsum);
  int r1 = *INTEGER(rmin);
  int r2 = *INTEGER(rmax);
  Rcomplex *cd = COMPLEX(mdata);
  Rcomplex *ad = COMPLEX(mamb);
  int *ia = LOGICAL(iamb);
  int *ip = LOGICAL(iprod);
  double *vd = REAL(edata);
  int nd = *INTEGER(ndata);

  int i, j, r, r0;

  SEXP                 success;
  int       * restrict i_success;

  // success output
  PROTECT( success = allocVector( LGLSXP , 1 ) );

  // local pointer to the success output
  i_success = LOGICAL( success );

  // set the success output
  *i_success = 1;

  // Skip first r2 points, their range ambiguity function
  // is not known
  r = r2+1;
  r0 = 0;

  // Walk through the data vector
  for( i = 0 ;  i < nd ; ++i ){
    // Check that we are above r1
    if( r >= r1 ){
      // Check that we are below r2
      if( r < r2 ){
    // Check that the point is flagged as usable
    if(ip[i]){
      // The average vector starts from range r1
      j = r-r1;
      // Divide the lagged product with its variance and
      // multiply with TX power
      ms[j].r += cd[i].r  / vd[i] * ad[r0].r;
      ms[j].i += cd[i].i  / vd[i] * ad[r0].r;
      // Inverse of variance scaled accordingly
      vs[j] += ad[r0].r * ad[r0].r / vd[i];
    }
      }
    }

    // If a new pulse is transmitted set range to zero,
    // otherwise increment the range counter.
    if( ia[i] ){
      r = 0;
      r0 = i;
    }else{
      ++r;
    }
  }

  UNPROTECT(1);

  return(success);

```

```
}
```

5.5.6 deco_add.c

```c
// file:deco_add.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
   Matched filter decoding, modified from fishs_add.

   Arguments:
    Qvec   Diagonal of the Fisher information matrix
    yvec   Modified measurement vector
    arows  Theory matrix rows
    meas   Measurements
    var    Measurement variances
    nx     Number of unknowns
    nrow   Number of theory rows in arows

   Returns:
    success 1 if the processing was succesful, 0 otherwise

*/

SEXP deco_add(  const SEXP Qvec , const SEXP yvec , const
    SEXP arows , const SEXP meas  , const SEXP var  , const
    SEXP nx   , const SEXP nrow   )
{
  Rcomplex *q = COMPLEX(Qvec);
  Rcomplex *y = COMPLEX(yvec);
  int n  = *INTEGER(nx);
  int nr = *INTEGER(nrow);
  int i  = 0;
  int j  = 0;
  int l  = 0;

  Rcomplex * restrict qtmp;
  Rcomplex * restrict acpy = COMPLEX(arows);
  Rcomplex * restrict atmp;
  Rcomplex * restrict ytmp;
  Rcomplex * restrict mcpy = COMPLEX(meas);
  double   * restrict vcpy = REAL(var);

  SEXP                success;
  int      * restrict i_success;

  // Success output
  PROTECT( success = allocVector( LGLSXP , 1 ) );

  // Local pointer to the success output
  i_success = LOGICAL( success );

  // Set the success output
  *i_success = 1;

  // Go through all theory matrix rows
  for( l = 0 ; l < nr ; ++l ){

    // Pointers to y-vector and Fisher information matrix
       diagonal
    ytmp = y;
    qtmp = q;

    // Go through all range gates
    for( i = 0 ; i < n ; ++i ){

      // Second pointer to the theory matrix
      // (Strictly speaking not needed...)
      atmp = acpy;

      // Add information (only diaonal)
      qtmp->r += ( acpy->r * atmp->r + acpy->i * atmp->i ) /
          *vcpy;
      qtmp->i += ( acpy->r * atmp->i - acpy->i * atmp->r ) /
          *vcpy;

      // Increment the second theory matrix counter
      ++atmp;

      // Increment information matrix counter (only diagonal)
      ++qtmp;

      // Add the corresponding measurement to the y-vector
      ytmp->r += ( mcpy->r * acpy->r + mcpy->i * acpy->i ) /
          *vcpy;
      ytmp->i += ( mcpy->i * acpy->r - mcpy->r * acpy->i ) /
          *vcpy;

      // Increment the y-vector counter
      ++ytmp;

      // Increment the theory matrix counter
      ++acpy;

    }

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

  }

  UNPROTECT(1);

  return(success);

}
```

5.5.7 fishs_add.c

```
// file:fishs_add.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
   Inverse problem solver using direct calculation of the
   Fisher information matrix. Data accumulation.

   Arguments:
    Qvec   Upper triangular part of the Fisher
           information matrix as a vector
    yvec   Modified measurement vector
    arows  Theory matrix rows
    meas   Measurements
    var    Measurement variances
    nx     Number of unknowns
    nrow   Number of theory rows in arows

   Returns:
    success 1 if the processing was successful, 0 otherwise

*/

SEXP fishs_add( const SEXP Qvec , const SEXP yvec , const
    SEXP arows , const SEXP meas  , const SEXP var  , const
    SEXP nx    , const SEXP nrow   )
{
  Rcomplex *q = COMPLEX(Qvec);
  Rcomplex *y = COMPLEX(yvec);
  int n  = *INTEGER(nx);
  int nr = *INTEGER(nrow);
  int i  = 0;
  int j  = 0;
  int l  = 0;

  Rcomplex * restrict qtmp;
  Rcomplex * restrict acpy = COMPLEX(arows);
  Rcomplex * restrict atmp;
  Rcomplex * restrict ytmp;
  Rcomplex * restrict mcpy = COMPLEX(meas);
  double   * restrict vcpy = REAL(var);

  SEXP                success;
  int      * restrict i_success;

  // success output
  PROTECT( success = allocVector( LGLSXP , 1 ) );

  // local pointer to the success output
  i_success = LOGICAL( success );

  // set the success output
  *i_success = 1;

  // Go through all theory matrix rows
  for( l = 0 ; l < nr ; ++l ){

    // Pointers to y-vector and Fisher information matrix
    ytmp = y;
    qtmp = q;

    // Go through all range gates
    for( i = 0 ; i < n ; ++i ){

      // Second pointer to the theory matrix
      atmp = acpy;

      // Go through all columns in the upper triangular part
      for( j = 0 ; j < ( n - i ) ; ++j ){

    // Add information
    qtmp->r += ( acpy->r * atmp->r + acpy->i * atmp->i ) / *
        vcpy;
    qtmp->i += ( acpy->r * atmp->i - acpy->i * atmp->r ) / *
        vcpy;

    // Increment the second theory matrix counter
    ++atmp;

    // Increment the information matrix counter
    ++qtmp;

      }

      // Add the corresponding measurement to the y-vector
      ytmp->r += ( mcpy->r * acpy->r + mcpy->i * acpy->i ) /
          *vcpy;
      ytmp->i += ( mcpy->i * acpy->r - mcpy->r * acpy->i ) /
          *vcpy;

      // Increment the y-vector counter
      ++ytmp;

      // Increment the theory matrix counter
      ++acpy;

    }

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

  }

  UNPROTECT(1);

  return(success);

}
```

5.5.8 fishs_add_clutter.c

```c
// file:fishs_add_clutter.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"
/*

   A special version of fisher solver for ground clutter
   estimation. Assumes unit variance and adds only one
   row at a time.

   Arguments:
    Qvec   Upper triangular part of Fisher information matrix
    yvec   Modified measurement vector
    arow   One row of theory matrix
    meas   Measurement
    nx     Number of unknowns

*/

void fishs_add_clutter( const SEXP Qvec , const SEXP yvec ,
    Rcomplex * arow , Rcomplex * meas , const int nx )
{
  Rcomplex *q = COMPLEX(Qvec);
  Rcomplex *y = COMPLEX(yvec);
  int n  = nx;
  int i  = 0;
  int j  = 0;

  Rcomplex * restrict qtmp;
  Rcomplex * restrict acpy = arow;
  Rcomplex * restrict atmp;
  Rcomplex * restrict ytmp;
  Rcomplex * restrict mcpy = meas;

  // Pointers to y-vector and Fisher information matrix
  ytmp = y;
  qtmp = q;

  // Go through all range gates
  for( i = 0 ; i < n ; ++i ){

    // Second pointer to the theory matrix
    atmp = acpy;

    // Go through all columns in the upper triangular part
    for( j = 0 ; j < ( n - i ) ; ++j ){

      // Add information
      qtmp->r += ( acpy->r * atmp->r + acpy->i * atmp->i );
      qtmp->i += ( acpy->r * atmp->i - acpy->i * atmp->r );

      // Increment the second theory matrix counter
      ++atmp;

      // Increment the information matrix counter
      ++qtmp;

    }

    // Add the corresponding measurement to the y-vector
    ytmp->r += ( mcpy->r * acpy->r + mcpy->i * acpy->i );
    ytmp->i += ( mcpy->i * acpy->r - mcpy->r * acpy->i );

    // Increment the y-vector counter
    ++ytmp;

    // Increment the theory matrix counter
    ++acpy;

  }

}
```

5.5.9 index_adjust.c

```
// file:index_adjust.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
  Adjust tx / rx indices. The rising edges are shifted
  shifts[0] samples and the falling edges shifts[1]
  samples towards larger indices. Also negative
  shifts are allowed.

  This function allocates new data vectors

  Arguments:
   idata   ndata integer vector of TX pulse / RX positions
   ndata   Number of data points in idata
   shifts  2-vector of shifts
           (shifts at rising and falling edges)

  Returns:
   ans       A list with elements
               idata   Index vector after adjustements
            success Logical, set if all processing
                    was successful
*/

SEXP index_adjust_R( SEXP idata , SEXP ndata , SEXP shifts )
{
  SEXP ans;
  SEXP idata_new;
  SEXP s;
  SEXP names;
  char *cnames[2] = {"idata","success"};
  int *inew;
  int *iold;
  register uint64_t k;


  // Output list ans[[1]] = idata , ans[[2]] = success
  PROTECT( ans = allocVector( VECSXP , 2 ) );

  // Allocate the new logical vector
  PROTECT( idata_new = allocVector( LGLSXP , *(INTEGER(ndata)
     ) ) );

  // A pointer to the new data vector
  inew = LOGICAL( idata_new );

  // A pointer to the old data vector
  iold = LOGICAL( idata );

  // Copy data from old to new
  for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
    inew[k] = iold[k];
  }

  // The success logical
  PROTECT( s = allocVector( LGLSXP , 1 ) );

  // The actual work
  s = index_adjust( idata_new , ndata , shifts );

  // Collect the data into the return list
  SET_VECTOR_ELT( ans , 0 , idata_new );
  SET_VECTOR_ELT( ans , 1 , s );

  // Set the name attributes
  PROTECT( names = allocVector( STRSXP , 2 ));
  SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
  SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
  setAttrib( ans , R_NamesSymbol , names);

  UNPROTECT(4);

  return(ans);

}

/*
  Adjust TX / RX indices. The rising edges are shifted
  shifts[0] samples and the falling edges shifts[1]
  samples towards larger indices.
  Also negative shifts are allowed.

  This function overwrites the idata vector

  Arguments:
   idata   ndata integer vector of TX pulse / RX positions
   ndata   Number of data points in idata
   shifts  2-vector of shifts
           (shifts at rising and falling edges)

  Returns:
   success 1 if all processing was successful, 0 otherwise
```

```
*/

SEXP index_adjust( SEXP idata , SEXP ndata , SEXP shifts)
{
  int *id = INTEGER(idata);
  int *nd = INTEGER(ndata);
  int *sh = INTEGER(shifts);
  // temporary variables
  int sh1;
  register int64_t k;
  int lasttrue;
  int ncut;
  int nadd;
  // for the return value
  SEXP success;
  int *isuccess;

  // Allocate the return value and initialise it
  PROTECT(success = allocVector(LGLSXP,1));
  isuccess = LOGICAL(success);
  *isuccess = 1;

  // The shift on rising edges is done by
  //shifting the whole index vector

  // Find the last true index in the whole vector,
  // it will be needed later
  lasttrue = 0;
  for( k = ( *nd - 1 ) ; k >= 0 ; --k ){
    if( id[k] ){
      lasttrue = k;
      break;
    }
  }

  // If sh[0] < 0, shift towards smaller indices
  if( sh[0] < 0 ){
    for( k = 0 ; k < ( *nd + sh[0] ) ; ++k ){
      id[k] = id[ k - sh[0] ];
    }
    // The last value is repeated in the remaining points
    for( k = ( *nd + sh[0]) ; k < *nd ; ++k){
      id[k] = id[( *nd - 1 )];
    }
  }

  // If sh[0] > 0, shift towards larger indices
  if( sh[0] > 0 ){
    for( k = ( *nd - 1 ) ; k >= sh[0] ; --k ){
      id[k] = id[ k - sh[0] ];
    }
    // The first value is repeated in the first sh[0] points
    for( k = ( sh[0] - 1 ) ; k > 0 ; --k ){
      id[k] = id[0];
    }
  }

  // Add the shift that was already done to sh[1]
  sh1 = sh[1] - sh[0];

  // If sh1 < 0  we are supposed to shift
  // the falling edges towards smaller indices
  if( sh1 < 0 ){
    ncut = 0;
    for( k = ( *nd - 1 ) ; k >= 0 ; --k ){
      if( id[ k ] == 0 ){
        ncut = 0;
      }else{
        --ncut;
      }
      if( ncut >= sh1 ) id[k] = 0;
    }
  }
  // If sh1 > 0 we are supposed to shift
  // the falling edges towards larger indices
  if( sh1 > 0 ){
    nadd = 0;
    for( k = 0 ; k < *nd ; ++k ){
      if( id[ k ] == 0 ){
        ++nadd;
      }else{
        nadd = 0;
      }
      if( nadd <= sh1 ) id[k] = 1;
    }

  }

  // Now there may be errors in the very end of the index
  //  vector, correct using the stored index lasttrue
  for( k = ( lasttrue + sh[1] + 1 ) ; k < *nd ; ++k ){
    id[k] = 0;
  }

  // Remove protection from the return value
  UNPROTECT(1);

  // Return the variable success only, the data is stored
  // in the R vectors 'cdata', 'idatar', and 'idatai'
  return(success);

}
```

5.5.10 average_power.c

```c
// file:average_power.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
  Average power vector for variance estsimation

  The algorithm proceeds as follows

  1. Locate falling edges of pulses  from idatatx
  2. locate the first falling edge at least maxrange samples
     from the beginning, give this pulse the pulse index 0
  3. Pick maxrange samples from idatatx from  immediately
     *before* the first falling edge
  4. At all other falling edges, compare the maxrange points
     before the edge with the samples picked in (3)
  5. If the vectors compared in (4) are identical, also this
     pulse is given pulse index 0, repeat for all pulses
  6. If there pulses are left without an index, select the
     first of them and repeat steps (4) and (5) to give
     these pulses the index 1.
  7. Continue with indices 2, 3, ...
     until all pulses have an index
  8. When all pulses have indices, calculate average
     power profiles from pulses with identical indices


  Arguments:
   cdata    Complex receiver samples
   idatatx  Transmitter sample index vector
   idatarx  Receiver sample index vector
   ndata    Number of points in data vectors
   maxrange Maximum range for power profile estimation

  Returns:
   pdata    Average power vector. The first element contains
      the
            ratio largest pulse index / number of pulses.

 */

SEXP average_power( SEXP cdata , SEXP idatatx , SEXP idatarx
    , SEXP ndata , SEXP maxrange)
{
  Rcomplex * cd = COMPLEX( cdata );
  int * idtx = LOGICAL( idatatx );
  int * idrx = LOGICAL( idatarx );
  int nd = *INTEGER( ndata );
  int maxr = *INTEGER( maxrange );

  SEXP pdata;
  double *pd;
  double *ptmp;
  int *pedges;
  int nedges;
  int *pinds;
  int *nsamp;
  int k, i, j;
  int pindcur;
  int pindmax;
  int npulse;
  int indprev;
  int p1;
  int sameamb;
  int r;
  int ippend;
  int ntot;
  double ptot;

  ntot = 0;
  ptot = .0;

  // Inspect the TX index vector
  // to make sure that 1 is exactly 1
  for( k = 0 ; k < nd ; ++k ) idtx[ k ] = idtx[ k ] ? 1 : 0 ;

  // Allocate the power vector
  PROTECT( pdata = allocVector( REALSXP , nd ) );

  // A pointer to the power vector
  pd = REAL( pdata );

  // Initialise to zero
  for( k = 0 ; k < nd ; ++k ) pd[ k ] = 0.;

  // Allocate a temporary vector for
  // power profile calculation
  ptmp = R_Calloc( nd , double );

  // Initialise to zero
  for( k = 0 ; k < nd ; ++k ) ptmp[ k ] = 0.;

  // Allocate a vector for sample counter
```

```
nsamp = R_Calloc( nd , int );

// Initialise to zero
for( k = 0 ; k < nd ; ++k ) nsamp[ k ] = 0;

// Allocate a vector for pulse edge positions
// (this could be shorter if needed)
pedges = R_Calloc( nd , int );

// Initialise to zero
for( k = 0 ; k < nd ; ++k ) pedges[ k ] = 0;

// Allocate a vector for pulse indices
pinds = R_Calloc( nd , int );

// Initialise to -1
for( k = 0 ; k < nd ; ++k ) pinds[ k ] = -1;


// Locate all falling edges of pulses
nedges = 0;
for( k = 0 ; k < ( nd - 1 ) ; ++k )
  {
    if( idtx[ k ] )
  {
    if( !(idtx[ k + 1] ) )
      {
        pedges[ nedges++ ] = k;
      }
  }
  }

// The first falling pulse edge at least
// maxr samples from the beginning
for( k = 0 ; k < nedges ; ++k )
  {
    if( pedges[ k ] > maxr )
  {
    p1 = k;
    break;
  }
  }

// Inspect the tx indices and give a unique index for
// each unique 0-lag range-ambiguity function
pindcur = 0;
for( k = p1 ; k < nedges ; ++k )
  {
    // pinds < 0 for pulses that do not yet have an index
```

```
      if( pinds[ k ] < 0 )
    {
      // Go through all the pulses
      for( i = k ; i < nedges ; ++i )
        {
          // Compare only with pulses that
          // do not yet have an index
          if( pinds[ i ] < 0 )
        {
          // Inspect the points just before this pulse
          sameamb = 1;
          for( j = 0 ; j < maxr ; ++j )
            {
              if( (idtx[ pedges[ k ] - j ]) != (idtx[ pedges[
                  i ] - j ]) )
            {
              sameamb = 0;
              break;
            }
            }
          // If the ambiguities were identical,
          // assign the pulse with the index pindcur
          if( sameamb ) pinds[ i ] = pindcur;
        }
        }
      // Increment pindcur
      ++pindcur;
    }
    }

  // There may be a pulse / pulses without an index
  // in the begin of data vector.
  // Give them an index if possible
  if( p1 > 0 )
    {
      for( i = p1 ; i < nedges ; ++i )
    {
      sameamb = 1;
      for( j = 0 ; j < pedges[ p1 - 1 ] ; ++j )
        {
          if( idtx[ pedges[ p1 - 1 ] - j ] != idtx[ pedges[ i
              ] - j ] )
        {
          sameamb = 0;
          break;
        }
        }
      if( sameamb )
        {
          pinds[ p1 - 1 ] = pinds[ i ];
          break;
        }
    }
      // Give a new index for the  pulse p1-1 if it did not
         match
      // with any of the exisiting ones. Pulses before p1-1
         will
      // not be used and they do not need an index.
      if( pinds[ p1 - 1 ] < 0 ) pinds[ p1 - 1 ] = pindcur;
    }

  // Store the largest pind
  pindmax = pindcur;

  // We have now an index for each pulse that needs one.
     Pulses
  // with equal indices have similar power profile range
     ambiguity
  // functions and their signal powers can be averaged.
  // Now we will walk through all different pulse indices,
  // calculate the correspondign power-profiles, and
  // store the results in appropriate places in the average
  // power vector

  // Start from the first falling edge, or
  // one point before if necessary
  if( p1 > 0 ) --p1 ;

  // Go through all pulses
  for( k = p1 ; k < nedges ; ++k )
    {

      // The indices will be set to -1 after processing,
      // an index >= indicates that the point has not
      // yet been processed
      if( pinds[ k ] >= 0 )
    {

      // Initialise the temporary power vector to zero
      for( i = 0 ; i < nd ; ++i ) ptmp[ i ] = 0.;

      // Initialise the sample counter to zero
      for( i = 0 ; i < nd ; ++i ) nsamp[ i ] = 0 ;

      // Check remaining pulses and try to find
      // the same index
      for( j = k ; j < nedges ; ++j )
        {
          // If a matching index is found, add power from the
          // ipp to the temporary profile and increment
             sample
          // counter accordingly
          if( pinds[ j ] == pinds[ k ] )
        {

          // Find distance to the next pulse end  (must not
          // stop at pulse start in order to facilitate
          // bistatic operation)
          if( ( j + 1 ) >= nedges )
            {
              ippend = nd - pedges[ j ];
            }
          else
            {
              ippend = pedges[ j + 1 ] - pedges[ j ];
            }
          for( i = 0 ; i < ippend ; ++i )
            {
              r = pedges[ j ] + i;
              // This cuts off points that are too close to
              // the beginning of the data vector
              if( r >= maxr )
            {
              if( idrx[ r ] )
                {
                  ptmp[ i ]  += cd[ r ].r * cd[ r ].r + cd[ r
                      ].i * cd[ r ].i;
                  nsamp[ i ] += 1;
                  ptot +=  cd[ r ].r * cd[ r ].r + cd[ r ].i
                      * cd[ r ].i;
                  ++ntot;
                }
            }
            }
        }
        }

      // Divide the summed powers by
      // the number of summed samples
      for( i = 0 ; i < nd ; ++i )
        {
          if( nsamp[ i ] > 100 ){
        ptmp[ i ] /= (double) nsamp[ i ];
          }else{
        ptmp[ i ] = -1.;
          }
        }

      // Go through the indices again and copy the power
      //  values to appropriate places Set pinds to -1 at
      //  points that have already been visited
      pindcur = pinds[ k ];
      for( j = k ;  j < nedges ; ++j )
        {
          if( pinds[ j ] == pindcur )
        {
          if( ( j + 1 ) >= nedges )
            {
              ippend = nd - pedges[ j ];
            }
          else
            {
              ippend = pedges[ j + 1 ] - pedges[ j ];
            }
          for( i = 0 ; i < ippend ; ++i )
            {
              r = pedges[ j ] + i;
              pd[ r ] = ptmp[ i ];
            }
          pinds[ j ] = -1;
        }
        }

    }

    }

  // Put the grand average power to points that did not have
  // enough averaged samples (they are set to -1 at this
     point)
  ptot /= (float)ntot;
  for( i = 0 ;  i < nd ; ++i ){
    if( pd[ i ] < 0.) pd[ i ] = ptot;
  }

  // Store the ratio pindmax / nedges to the first data point
     .
  // If the ratio is large the power estimation will not
     perform
  // very well.
  // The power value in this point cannot ever be needed in
     LPI.
  pd[0] = (float)pindmax / (float)nedges;

  // Free the temporary allocations
  Free(ptmp);
  Free(nsamp);
  Free(pinds);
  Free(pedges);

  UNPROTECT(1);
  return(pdata);

}
```

5.5.11 lagged_products.c

```c
// file:lagged_products.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"
/*
  Calculate lagged products of a signal
  and its complex conjugate.

  This function allocates new data vectors

  Arguments:
   cdata1  ndata1 vector of complex signal samples
   cdata2  ndata2 vector of complex signal samples
   idata1  ndata1 integer vector of usable
           RX sample positions
   idata2  ndata2 integer vector of usable
           RX sample positions
   ndata1  Number of samples in cdata1 and idata1
   ndata2  Number of samples in cdata2 and idata2
   lag     Lag

  Returns:
   ans       A list with elements
              cdata   Complex vector of lagged products
              idata   Index vector for cdata
              ndata   Data vector length
              success Logical, set if all processing
                      was successful
*/

SEXP lagged_products_alloc( SEXP cdata1 , SEXP cdata2 , SEXP
    idata1 , SEXP idata2 , SEXP ndata1 , SEXP ndata2 , SEXP
    lag)
{
  Rcomplex *cd1 = COMPLEX(cdata1);
  Rcomplex *cd2 = COMPLEX(cdata2);
  int *id1 = LOGICAL(idata1);
  int *id2 = LOGICAL(idata2);
  int *nd1 = INTEGER(ndata1);
  int *nd2 = INTEGER(ndata2);
  int *l = INTEGER(lag);

  SEXP ans;
  SEXP lcdata;
  Rcomplex *lcd;
  SEXP lidata;
  int *lid;
  SEXP success;
  int *isuccess;
  SEXP ndata;
  int *nd;
  SEXP names;
  char *cnames[4] = {"cdata","idata","ndata","success"};
  int k=0;

  // Allocate the return value list
  PROTECT( ans  = allocVector( VECSXP , 4 ) );

  // Allocate the ndata output
  PROTECT( ndata = allocVector( INTSXP , 1 ) );

  // A local pointer to ndata
  nd = INTEGER( ndata );

  // Output data length will be minimum of the two
  //  input data lengths, minus the time-lag
  *nd = *nd1 - *l;
  if( *nd1 > *nd2 ) *nd = *nd2 - *l;

  // Allocate the lagged product vector
  PROTECT( lcdata = allocVector( CPLXSXP , *nd ) );

  // A local pointer to the lagged product vector
  lcd = COMPLEX( lcdata );

  // Allocate an index vector for the lagged products
  PROTECT( lidata = allocVector( LGLSXP , *nd ) );

  // A local pointer to the lagged product vector
  lid = LOGICAL( lidata );

  // Allocate the success return value
  PROTECT( success = allocVector( LGLSXP , 1 ) );

  // A local pointer to the success value
  isuccess = LOGICAL( success );
  *isuccess = 1;

  // The actual lagged product calculation
  for( k = 0 ; k < *nd ; ++k ){

    // Calculate the index vector point
    lid[k] = (id1[k] * id2[k+ *l]);

    // Calculate the actual data product only if the index
       vector is set
    if(lid[k]){
      lcd[k].r = cd1[k].r * cd2[k+ *l].r + cd1[k].i * cd2[k+
         *l].i;
      lcd[k].i = -cd1[k].r * cd2[k+ *l].i + cd1[k].i * cd2[k+
         *l].r;
    }
  }


  // Collect the return values under the list "ans"
  SET_VECTOR_ELT( ans , 0 , lcdata );
  SET_VECTOR_ELT( ans , 1 , lidata );
  SET_VECTOR_ELT( ans , 2 , ndata );
  SET_VECTOR_ELT( ans , 3 , success );

  // Set the name attributes
  PROTECT( names = allocVector( STRSXP , 4 ) );
  SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
  SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
  SET_STRING_ELT( names , 2 , mkChar( cnames[2] ) );
  SET_STRING_ELT( names , 3 , mkChar( cnames[3] ) );
  setAttrib( ans , R_NamesSymbol , names );

  UNPROTECT(6);

  return(ans);

}




/*
  Calculate lagged products of a signal
  and its complex conjugate.

  This function overwrites existing data vectors

  Arguments:
   cdata1  ndata1 vector of complex signal samples
   cdata2  ndata2 vector of complex signal samples
   idata1  ndata1 integer vector of usable
           RX sample positions
   idata2  ndata2 integer vector of usable
           RX sample positions
   cdatap  complex vector for the lagged products
   idatap  integer vector for the lagged product indices
```

```c
   ndata1  Number of samples in cdata1 and idata1
   ndata2  Number of samples in cdata2 and idata2
   lag     Lag

  Returns:
   success 1 if processing was succesful, 0 otherwise

*/

SEXP lagged_products( SEXP cdata1 , SEXP cdata2 , SEXP idata1 , SEXP idata2 , SEXP cdatap ,\
              SEXP idatap , SEXP ndata1 , SEXP ndata2 , SEXP lag )
{
  Rcomplex *cd1      =  COMPLEX(cdata1);
  Rcomplex *cd2      =  COMPLEX(cdata2);
  int      *id1      =  LOGICAL(idata1);
  int      *id2      =  LOGICAL(idata2);
  Rcomplex *cdp      =  COMPLEX(cdatap);
  int      *idp      =  LOGICAL(idatap);
  int       nd1      = *INTEGER(ndata1);
  int       nd2      = *INTEGER(ndata2);
  int       l        = *INTEGER(lag)   ;
  SEXP      success                    ;
  int      *isuccess                   ;
  int       k        =  0              ;
  int       npr                        ;

  // Output data length will be minimum of the
  //  two input data lengths, minus the time-lag
  npr = nd1 - l;
  if( nd1 > nd2 ) npr = nd2 - l;

  // Allocate the success return value
  PROTECT( success = allocVector( LGLSXP , 1 ) );

  // A local pointer to the success value
  isuccess = LOGICAL( success );
  *isuccess = 1;

  // The actual lagged product calculation
  for( k = 0 ; k < npr ; ++k ){

    // The logical vector
    idp[k] = (id1[k] * id2[k+ l]);

    // Multiply the actual data points only
    //  if the logical vector is set
    if(idp[k]){
      cdp[k].r = cd1[k].r * cd2[k+ l].r + cd1[k].i * cd2[k+ l
         ].i;
      cdp[k].i = cd1[k].r * cd2[k+ l].i - cd1[k].i * cd2[k+ l
         ].r;
    }
  }

  // Set the logical vector to false at
  // points where it cannot be calculated
  for( k = 0 ; k < l ; ++k ){
    idp[npr+k] = 0;
  }

  UNPROTECT(1);

  return(success);

}



/*
  Real-valued lagged products for variance estimation.

  No Index vectors, because they are carried with
  the complex vectors.

  This function overwrites existing data vectors

  Arguments:
   rdata1  ndata1 vector of real signal samples
   rdata2  ndata2 vector of real signal samples
   prdata  real vector for the lagged products
   ndata1  Number of samples in rdata1
   ndata2  Number of samples in rdata2
   lag     Lag

  Returns:
   success 1 if processing was successful, 0 otherwise

*/
SEXP lagged_products_r( SEXP rdata1 , SEXP rdata2 , SEXP
    prdata , SEXP ndata1 ,\
                 SEXP ndata2 , SEXP lag )
{
  double *rd1      =  REAL(rdata1)   ;
  double *rd2      =  REAL(rdata2)   ;
  double *prd      =  REAL(prdata)   ;
  int     nd1      = *INTEGER(ndata1);
  int      nd2       = *INTEGER(ndata2);
  int      l         = *INTEGER(lag)    ;
  SEXP     success                      ;
  int     *isuccess                     ;
  int      k         =  0               ;
  int      npr                          ;

  // Output data length will be minimum of the two input
  // data lengths, minus the time-lag
  npr = nd1 - l;
  if( nd1 > nd2 ) npr = nd2 - l;

  // Allocate the success return value
  PROTECT( success = allocVector( LGLSXP , 1 ) );

  // A local pointer to the success value
  isuccess = LOGICAL( success );
  *isuccess = 1;

  // The actual lagged product calculation
  for( k = 0 ; k < npr ; ++k ){
    prd[k] =  rd1[k] * rd2[k+ l];
  }

  UNPROTECT(1);

  return(success);

}
```

5.5.12 average_profile.c

```c
// file:average_profile.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
  Average lag-profile vector for speeding up
  the inversion process. Each average is
  calculated over samples from  the same point in
  the repeated code cycle

  The complicated structure is used because
  measuremnts may contain additional sync
  times which need to be skipped.


  Arguments:
   cdata  Complex lagged product vector
   idata  Index vector for cdata
   ndata  Data vector length
   N_CODE Code cycle length

  Returns:
   success 1 if the processing was successful , 0 otherwise

*/

SEXP average_profile( SEXP cdata , SEXP idata , SEXP ndata ,
    SEXP N_CODE)
{
  Rcomplex * cd = COMPLEX( cdata );
  int * id = LOGICAL( idata );
  int nd = *INTEGER( ndata );
  int ncode = *INTEGER( N_CODE );

  double *aver;
  double *avei;
  Rcomplex *ad;
  R_len_t *nave;
  R_len_t k;
  R_len_t ind1 , ind2, ipp_count;
  SEXP success;
  int *isuccess;

  // Allocate the return value and initialise it
  PROTECT(success = allocVector(LGLSXP,1));
  isuccess = LOGICAL(success);
  *isuccess = 1;

  // Allocate the average vectors,
  // real and imaginary parts separately
  aver = (double*) R_Calloc( nd , double );
  avei = (double*) R_Calloc( nd , double );

  // Initialise to zero
  for( k = 0 ; k < nd ; ++k ){
    aver[ k ] = 0.;
    avei[ k ] = 0.;
  }

  // Allocate vector for data sample counter
  nave = R_Calloc( nd , R_len_t );

  // Initialise to zero
  for( k = 0 ; k < nd ; ++k ) nave[ k ] = 0;

  // Start from begniing of the data vctor
  ind1 = 0;
  ind2 = 0;

  // Search for the start of the first pulse
  while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;
  while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;
  ipp_count = 0;

  // Repeat until end of data
  while( ind2 < nd ){

    // At this point we should be at pulse starts, loop until
    // we hit a point at which both pulses have ended.
    while( id[ind1] | id[ind2]){
      aver[ind1] += cd[ind2].r;
      avei[ind1] += cd[ind2].i;
      ++nave[ind1];
      ++ind1;
      ++ind2;
      if(ind2==nd) break;
    }

    if(ind2==nd) break;

    // Add power values until either of the indices
    // hits the next pulse
    while( (id[ind1]==0) & (id[ind2]==0)){
      aver[ind1] += cd[ind2].r;
      avei[ind1] += cd[ind2].i;
      ++nave[ind1];
      ++ind1;
      ++ind2;
      if(ind2==nd) break;
    }

    if(ind2==nd) break;

    // Make sure that both indices point to a pulse start,
    // increment if necessary (This takes possible sync
    // times into account)
    while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;
    while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;

    if(ind2==nd) break;

    // Increment the ipp counter
    ++ipp_count;
    if( ipp_count == ncode ){
      ipp_count = 0;
      ind1 = 0;
      while( id[ind1] == 0 ) ++ind1;
    }
  }

  // Divide the summed values with number of summed pulses
  for( k = 0 ; k < nd ; ++k ){
    if( nave[ k ] ){
      aver[k] /= (double)nave[k];
      avei[k] /= (double)nave[k];
    }
  }


  // Now there are averaged values available for one code
  // cycle, copy the valeus to make furhter analysis
  // simpler. Start from beginning of the data vector.
  ind1 = 0;
  ind2 = 0;

  // Search for the start of the first pulse
  while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;
  while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;
  ipp_count = 0;

  // Repeat until end of data
  while( ind2 < nd ){
```

```
    // At this point we should be at pulse starts,
    // loop until both pulses have ended
    while( id[ind1] | id[ind2]){
      cd[ind2].r = aver[ind1];
      cd[ind2].i = avei[ind1];
      ++ind1;
      ++ind2;
      if(ind2==nd) break;
    }

    if(ind2==nd) break;

    // Add power values until either of
    // the indices hits the next pulse
    while( (id[ind1]==0) & (id[ind2]==0)){
      cd[ind2].r = aver[ind1];
      cd[ind2].i = avei[ind1];
      ++ind1;
      ++ind2;
      if(ind2==nd) break;
    }

    if(ind2==nd) break;

    // Make sure that both indices point to a pulse start
    while( ( id[ind1] == 0 ) & ( ind1 < nd ) ) ++ind1;
    while( ( id[ind2] == 0 ) & ( ind2 < nd ) ) ++ind2;

    if(ind2==nd) break;

    // Increment the ipp counter
    ++ipp_count;
    if( ipp_count == ncode ){
      ipp_count = 0;
      ind1 = 0;
      while( id[ind1] == 0 ) ++ind1;
    }
  }

  // Free the temporary vectors
  Free(nave);
  Free(aver);
  Free(avei);

  UNPROTECT(1);

  return( success );

}
```

5.5.13 mix_frequency.c

```c
// file:mix_frequency.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
  Frequency mixing for IQ data

  This function allocates new vectors

  Argumnets:
   cdata      ndata complex vector of data samples
   ndata      Number of samples in cdata
   frequency  The mixing frequency

  Returns:
   ans       A list with elements
              cdata   Complex data samples after
                      frequency mixing
           success Logical, set if all processing
                      was successful
*/


SEXP mix_frequency_R( SEXP cdata , SEXP ndata , SEXP
    frequency )
{
  SEXP ans;
  SEXP cdata_new;
  SEXP s;
  SEXP names;
  char *cnames[2] = {"cdata","success"};
  Rcomplex *cnew;
  Rcomplex *cold;
  register uint64_t k;


  // Output list ans[[1]] = cdata , ans[[2]] = success
  PROTECT( ans = allocVector( VECSXP , 2 ) );

  // Allocate the new complex vector
  PROTECT( cdata_new = allocVector( CPLXSXP , *(INTEGER(ndata
      )) ) );

  // A pointer to the new data vector
  cnew = COMPLEX( cdata_new );

  // A pointer to the old data vector
  cold = COMPLEX( cdata );

  // Copy data from old to new
  for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
    cnew[k].r = cold[k].r;
    cnew[k].i = cold[k].i;
  }

  // The success logical
  PROTECT( s = allocVector( LGLSXP , 1 ) );

  // The actual frequency mixing
  s = mix_frequency( cdata_new , ndata , frequency );

  // Collect the data into the return list
  SET_VECTOR_ELT( ans , 0 , cdata_new );
  SET_VECTOR_ELT( ans , 1 , s );

  // Set the name attributes
  PROTECT( names = allocVector( STRSXP , 2 ));
  SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
  SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
  setAttrib( ans , R_NamesSymbol , names);

  UNPROTECT(4);

  return(ans);

}

/*
  Frequency mixing for IQ data

  This function overwrites the cdata vector

  Argumnets:
   cdata      ndata complex vector of data samples
   ndata      Number of samples in cdata
   frequency  The mixing frequency

  Returns:
   success    1 if all processing was successful, 0 otherwise
*/
SEXP mix_frequency( SEXP cdata , SEXP ndata , SEXP frequency)
{
  // Pointers to the R variables
  Rcomplex *cd = COMPLEX(cdata);
  int *nd = INTEGER(ndata);
  double *fr = REAL(frequency);
  register uint64_t k, nc;
  double arg;
  Rcomplex ctmp;
  // Temporary variables
  int ncycle;
  double tmpprod;
  double idiff;
  double *coefr;
  double *coefi;
  // For the return value
  SEXP success;
  int *isuccess;

  // Allocate the return value and initialise it
  PROTECT(success = allocVector(LGLSXP,1));
  isuccess = LOGICAL(success);
  *isuccess = 1;

  // The multiplicand will be cyclic, find the cycle length
  ncycle = *nd;
  for( k = 1 ; k < *nd ; ++k){
    tmpprod = *fr * (double)(k);
    idiff = tmpprod - (double)((int)(tmpprod));
    if( fabs(idiff) <= FLT_MIN ){
      ncycle = k;
      break;
    }
  }

  // If the cycle length is one, the mixing would not change
     anything
  if( ncycle == 1 ){
    UNPROTECT(1);
    return(success);
  }

  // Tabulate the cyclic coefficients.
  // This usually saves time as radar engineers tend to
  // select nice numerical values for the frequencies
  coefr = (double*) R_Calloc( ncycle , double );
  coefi = (double*) R_Calloc( ncycle , double );
  for( k = 0 ; k < ncycle ; ++k ){
    arg      = 2.0 * pi * *fr * (double)(k);
    coefr[k] = cos(arg);
    coefi[k] = sin(arg);
  }

  // Actual mixing
  nc = 0;
  for( k = 0 ; k < *nd ; ++k ){
    ctmp.r = cd[k].r;
    ctmp.i = cd[k].i;
    cd[k].r = ctmp.r * coefr[nc] - ctmp.i * coefi[nc];
    cd[k].i = ctmp.i * coefr[nc] + ctmp.r * coefi[nc];
    ++nc;
    if( nc == ncycle ) nc = 0;
  }

  // Free the memory allocated for the coefficient tables
  Free(coefr);
  Free(coefi);

  // Remove protection from the return value
  UNPROTECT(1);

  // Return the variable success only, the data is stored in
  // the R vectors 'cdata', 'idatar', and 'idatai'
  return(success);

}
```

5.5.14 resample.c

```c
// file:resample.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
  Resampling with linear interpolation. Reduces to a simple
  boxcar filter when the filter length is an integer
  multiple of the original sample interval.

  Final sample rate must be smaller than or
  equal to the original one.

  This function overwrites existing data vectors

  Arguments:
   cdata     Complex data samples
   idata     Index vector for cdata
   ndata     Data vector length
   nup       Upsamling factor
   nfilter   Filter length on upsampled data
             (final length is nfilter / nup)
   nfirst    Decimation start index
   nfirstfrac start point within the boxcar filter in
      upsampled units
   ipartial 0 if partial matched with filter
             should not be accepted in idata vector

  Returns:
   success  1 if resampling was successful, 0 otherwise

*/

SEXP resample( SEXP cdata , SEXP idata , SEXP ndata , SEXP
    nup , SEXP nfilter , SEXP nfirst , SEXP nfirstfrac, SEXP
    ipartial )
{

  Rcomplex * restrict cd = COMPLEX(cdata);
  int * restrict id = LOGICAL(idata);
  int nd = *INTEGER(ndata);
  const  int nu = *INTEGER(nup);
  const int nf = *INTEGER(nfilter);
  const int ns = *INTEGER(nfirst);
  const int nsf = *INTEGER(nfirstfrac);
  const int ipar = *LOGICAL(ipartial);
  uint64_t i, j, k, l, m, n;
  double frac;
  Rcomplex tmpsum;
  int tmpi[2];

  // For the return value
  SEXP success;
  int * restrict isuccess;

  // Allocate the return value and initialise it
  PROTECT(success = allocVector(LGLSXP,1));
  isuccess = LOGICAL(success);
  *isuccess = 1;

  /*
    i the current filter start point in upsampled data
    j the current point inside the (upsampled) boxcar filter
    k the current point within the original data vector
    l the current point within the resampled data vector
   */

  i = ns * nu ;  // Starting point in upsampled units
  //  j = nu-1;       // We are originally at the
  //                  // beginning of the boxcar filter
  j = nsf+nu-1;  // increment with nu-1, we will use the full
      sample
                 // cd[k] in any case. The first resampled
                     one will
                 // be wrong if nsf/=0, but we could not help
                      this if
                 // nsf < 0 in any case.
  k = ns;        // Starting point in original sampling
  l = 0;         // Current point in the final filtered and
                 // decimated data vector, start filling
                 // from beginning
  tmpsum.r = 0.; // Initialise the temp filter sum to zero
  tmpsum.i = 0.;
  tmpi[0] = 1;
  tmpi[1] = 0;


  while( ( ( i + nf ) / nu ) <= nd ){ // Current filter start
      + filter length <= data length
    while( j < nf ){          // One filter length of data
      tmpsum.r += cd[ k ].r; // Add the current point to the
          filter sum
      tmpsum.i += cd[ k ].i;
      tmpi[0] *= id[k];
      tmpi[1] += id[k];
      j += nu;               // Jump  to the next point that actually needs to be calculated
      ++k ;                  // Increment the sample counter of the original data vector
    }
    //    // Fraction of the k'th sample in the original data
    //    // vector that will go to l+1'th resampled point
    //    frac = ( (double)( j - nf + 1 ) ) / (double)nu;
    // not like this, it will create effectively two filters ...
    // this should be better
    frac = 0.;
    if( ( j - nf + 1 ) ==  nu ) frac = 1.;

       //
       // the whole fraction thing could be removed, but the above lines will fix
       // this for the time being. IV 2016-02-16.
       //
       // ... on the other hand, this will be rather easy to convert into upsamling, if that
       // would ever be needed?
       //





    // Now k could be beyond the data vector length,
    // check that it is not
    if( k < nd ){
      // Add the fraction that belongs to the k'th point
      tmpsum.r += ( 1. - frac )*cd[k].r;
      tmpsum.i += ( 1. - frac )*cd[k].i;
      if( frac < .99999 ) tmpi[0] *= id[k];
      if( frac < .99999 ) tmpi[1] += id[k];
      // Now tmpsum is ready, copy its contents to
      // the l'th element of the data vector
      cd[l].r = tmpsum.r;
      cd[l].i = tmpsum.i;
      id[l] = ipar ? tmpi[1] : tmpi[0];
      // Put the remaining fraction of
      // k'th sample to the tmpsum
      tmpsum.r = frac*cd[k].r;
      tmpsum.i = frac*cd[k].i;
      tmpi[0] = ( frac < .00001 ) ? 1 : id[k];
      tmpi[1] = ( frac < .00001 ) ? 0 : id[k];
      // One filter length backwards
      j -= nf;
      // The sample where we ended in the previous step was
      //  already added to tmpsum, jump to the next one
      j += nu;
      // Move one filter length forwards
      /*
      i += nf;
      ++k;
      */
      ++l;
    }

    // i and k must be incremented also at end of data to get
        us out of the loop
    i += nf;
    ++k;
  }

  // If we were exactly at end of data frac is unity, we will
      still get one more sample
  // k was incremented after hitting the end of data
  if( k == ( nd + 1 ) ){
    if( frac > .9999999 ){
      cd[l].r = tmpsum.r;
      cd[l].i = tmpsum.i;
      id[l] = ipar ? tmpi[1] : tmpi[0];
      ++l;
    }
  }

  *(INTEGER(ndata)) = l;

  // remove protection from the return value
  UNPROTECT(1);

  // return the variable success only, the data is now stored
  // in the R vectors 'cdata', 'idatar', and 'idatai'
  return(success);

}


/*
  Resampling with linear interpolation. Reduces to a simple
  boxcar filter when the filter length is an integer
  multiple of the original sample interval.

  Final sample rate must be smaller than
  or equal to the original one.

  This function allocates new data vectors

  Arguments:
   cdata     Complex data samples
   idata     Index vector for cdata
   ndata     Data vector length
   nup       Upsamling factor
   nfilter   Filter length on upsampled data (final length
             is nfilter / nup)
   nfirst    Decimation start index
   ipartial  0 if partial matched with filter should not be
             accepted in idata vector

  Returns:
   ans       A list with components:
             cdata     Resampled complex data vector
             idata     Index vector for cdata
             ndata     Data vector length
             success   1 if resampling was successful,
                       0 otherwise

*/


SEXP resample_R( SEXP cdata , SEXP idata , SEXP ndata , SEXP
    nup , SEXP nfilter , SEXP nfirst , SEXP nfirstfrac, SEXP
    ipartial)
{
  SEXP ans;
  SEXP cdata_new;
  SEXP idata_new;
  SEXP ndata_new;
  SEXP s;
  SEXP names;
  char *cnames[4] = {"cdata","idata","ndata","success"};
  Rcomplex * restrict cnew;
  Rcomplex * restrict cold;
  int * restrict inew;
  int * restrict iold;
  uint64_t k;
  PROTECT_INDEX cpind=0;
  PROTECT_INDEX ipind=0;


  // Output list ans[[1]] = cdata , ans[[2]] = idata ,
  // ans[[3]] = ndata , ans[[4]] = success
  PROTECT( ans = allocVector( VECSXP , 4 ) );

```

```
// Allocate the new complex vector
PROTECT_WITH_INDEX( cdata_new = allocVector( CPLXSXP , *(
    INTEGER(ndata)) ) , &cpind );

// Allocate the new logical vector
PROTECT_WITH_INDEX( idata_new = allocVector( LGLSXP , *(
    INTEGER(ndata)) ) , &ipind );

// Allocate the new ndata variable
PROTECT( ndata_new = allocVector( INTSXP , 1 ) );

// A pointer to the new cdata vector
cnew = COMPLEX( cdata_new );

// A pointer to the old cdata vector
cold = COMPLEX( cdata );

// A pointer to the new idata vector
inew = LOGICAL( idata_new );

// A pointer to the old idata vector
iold = LOGICAL( idata );

// Copy data from old cdata to new cdata
for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
  cnew[k].r = cold[k].r;
  cnew[k].i = cold[k].i;
}

// Copy data from old idata to new idata
for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
  inew[k] = iold[k];
}

// Use the same pointers to copy old ndata to new ndata
inew = INTEGER( ndata_new);
iold = INTEGER( ndata );
*inew = *iold;

// The  success logical
PROTECT( s = allocVector( LGLSXP , 1 ) );

// The actual resampling
s = resample( cdata_new , idata_new , ndata_new , nup ,
    nfilter ,  nfirst , nfirstfrac , ipartial );

// Reallocate the vectors to match with the new data length
SET_LENGTH( cdata_new , *INTEGER(ndata_new) );
REPROTECT( cdata_new , cpind );
```

```
  SET_LENGTH( idata_new , *INTEGER(ndata_new) );
  REPROTECT( idata_new , ipind );

  // Collect the data into the return list
  SET_VECTOR_ELT( ans , 0 , cdata_new );
  SET_VECTOR_ELT( ans , 1 , idata_new );
  SET_VECTOR_ELT( ans , 2 , ndata_new );
  SET_VECTOR_ELT( ans , 3 , s );

  // Set the name attributes
  PROTECT( names = allocVector( STRSXP , 4 ));
  SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
  SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
  SET_STRING_ELT( names , 2 , mkChar( cnames[2] ) );
  SET_STRING_ELT( names , 3 , mkChar( cnames[3] ) );
  setAttrib( ans , R_NamesSymbol , names);

  UNPROTECT(6);

  return(ans);

}
```

5.5.15 prepare_data.c

```
// file:prepare_data.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
  Frequency mixing, index adjustments,
  and filtering in a single function

  Arguments:
   cdata     Complex voltage data vector
   idata     Integer vector of usable data indices
   ndata     Data vector length
   frequency Frequency offset
   shifts    Corrections to idata
   nup       Upsaling factor in resampling
   nfilter   Filter length (for upsampled data, final
             filter length is nfilter / nup)
   nfirst    Decimation start index
   ipartial  Logical, are partial matches of
             idata with the filter accepted?

  Returns:
   ans       A list with elements
              cdata   Final complex data vector
              idata   Final index vector
              ndata   Final data vector length
          success Logical, set if all processing
                      was successfull

 */


SEXP prepare_data( SEXP cdata , SEXP idata , SEXP ndata ,
    SEXP frequency, SEXP shifts , SEXP nup , SEXP nfilter ,
    SEXP nfirst , SEXP nfirstfrac , SEXP ipartial )
{
  SEXP ans;
  SEXP cdata_new;
  SEXP idata_new;
  SEXP ndata_new;
  SEXP s;
  SEXP names;
  char *cnames[4] = {"cdata","idata","ndata","success"};
  Rcomplex * restrict cnew;
  Rcomplex * restrict cold;
  int * restrict inew;
  int * restrict iold;
  uint64_t k;
  PROTECT_INDEX cpind=0;
  PROTECT_INDEX ipind=0;



  // Output list ans[[1]] = cdata ans[[2]] = pdata ,
  // ans[[3]] = idata , ans[[4]] = ndata , ans[[5]] = success
  PROTECT( ans = allocVector( VECSXP , 5 ) );

  // Allocate the new complex vector
  PROTECT_WITH_INDEX( cdata_new = allocVector( CPLXSXP , *(
      INTEGER(ndata)) ) , &cpind );

  // Allocate the new logical vector
  PROTECT_WITH_INDEX( idata_new = allocVector( LGLSXP , *(
      INTEGER(ndata)) ) , &ipind );

  // Allocate the new ndata variable
  PROTECT( ndata_new = allocVector( INTSXP , 1 ) );

  // A pointer to the new cdata vector
  cnew = COMPLEX( cdata_new );

  // A pointer to the old cdata vector
  cold = COMPLEX( cdata );

  // A pointer to the new idata vector
  inew = LOGICAL( idata_new );

  // A pointer to the old idata vector
  iold = LOGICAL( idata );

  // Copy data from old cdata to new cdata
  for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
    cnew[k].r = cold[k].r;
    cnew[k].i = cold[k].i;
  }

  // Copy data from old idata to new idata
  for( k = 0 ; k < *(INTEGER(ndata)) ; ++k ){
    inew[k] = iold[k];
  }

  // Use the same pointers to copy old ndata to new ndata
  inew = INTEGER( ndata_new);
```

```
  iold = INTEGER( ndata );
  *inew = *iold;

  // The  success logical
  PROTECT( s = allocVector( LGLSXP , 1 ) );

  // Frequency mixing
  s = mix_frequency( cdata_new , ndata_new , frequency );

  // Index adjustments
  s = index_adjust( idata_new , ndata_new , shifts );

  // Filtering
  s = resample( cdata_new , idata_new , ndata_new , nup ,
      nfilter , nfirst , nfirstfrac , ipartial );

  // Set cdata_new to zero at all points where idata_new==0
  inew = LOGICAL( idata_new );
  for( k = 0 ; k < *INTEGER(ndata_new) ; ++k ){
    if( inew[k] == 0 ){
      cnew[k].r = .0;
      cnew[k].i = .0;
    }
  }

  // Reallocate the vectors to match with the new data length
  SET_LENGTH( cdata_new , *INTEGER(ndata_new) );
  REPROTECT( cdata_new , cpind );
  SET_LENGTH( idata_new , *INTEGER(ndata_new) );
  REPROTECT( idata_new , ipind );

  // Collect the data into the return list
  SET_VECTOR_ELT( ans , 0 , cdata_new );
  SET_VECTOR_ELT( ans , 1 , idata_new );
  SET_VECTOR_ELT( ans , 2 , ndata_new );
  SET_VECTOR_ELT( ans , 3 , s );

  // Set the name attributes
  PROTECT( names = allocVector( STRSXP , 4 ));
  SET_STRING_ELT( names , 0 , mkChar( cnames[0] ) );
  SET_STRING_ELT( names , 1 , mkChar( cnames[1] ) );
  SET_STRING_ELT( names , 2 , mkChar( cnames[2] ) );
  SET_STRING_ELT( names , 3 , mkChar( cnames[3] ) );
  setAttrib( ans , R_NamesSymbol , names);

  UNPROTECT(6);

  return(ans);

```

```
}
```

5.5.16 theory_rows.c

```c
// file:theory_rows.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.


#include "LPI.h"
/*
  Make theory matrix rows and measurement vectors.

  This function allocates new data vectors.

  Arguments:
   camb         Complex range ambiguity functions
   iamb         Index vector of range ambiguity functions
   cprod        Complex lagged product vector
   iprod        Index vector of lagged products
   rvar         Measurement variance vector
   ndata        Data vector length
   ncur         Current sample index
   nend         Last sample index to use (in this call)
   rlims        Range gate limits
   nranges      Number of range gates
   fitsize      0 if the vectors should not be reallocated to
                match the final data size.
   background   0 if additional background term is not used
   remoterx     0 if measurements TX times should not be used


  Returns:
   ans        A list with elements
               arows    Theory matrix rows
               irows    Theory row indices
               m        Inversion measurement vector
               var      Measurement variances
               nrows    Number of theory rows produced
               success  Logical, set if all processing
                        was successful

 */

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 )
{
  const int n_cur = *INTEGER(ncur);
  const int n_end = *INTEGER(nend);
  const int n_ranges = *INTEGER(nranges);
  const int fit_size = *LOGICAL(fitsize);
  SEXP ans;
  SEXP arows;
  SEXP irows;
  SEXP mvec;
  SEXP mvar;
  SEXP success;
  SEXP nrows;
  SEXP names;
  int n_rows;
  const char * c_names[6] =  {"arows","irows","m","var","
     nrows","success"};
  PROTECT_INDEX arind =  0;
  PROTECT_INDEX irind = 0;
  PROTECT_INDEX mind = 0;
  PROTECT_INDEX vind = 0;


  // Output list
  PROTECT( ans = allocVector( VECSXP , 5 ) );

  // A vector for the theory matrix rows
  PROTECT_WITH_INDEX( arows = allocVector( CPLXSXP , ( (
     n_end - n_cur + 1 ) * ( n_ranges + 1) ) ) , & arind );

  // A vector for the theory matrix indices
  PROTECT_WITH_INDEX( irows = allocVector( LGLSXP , ( ( n_end
      - n_cur + 1 ) * ( n_ranges + 1) ) ) , & irind );

  // A vector for the measurements
  PROTECT_WITH_INDEX( mvec = allocVector( CPLXSXP , ( n_end -
      n_cur + 1 ) ) , & mind );

  // A vector for the measurement errors
  PROTECT_WITH_INDEX( mvar = allocVector( REALSXP , ( n_end -
      n_cur + 1 ) ) , & vind );

  // Number of rows for the R output
  PROTECT( nrows = allocVector( INTSXP , 1 ) );

  // Success output
  PROTECT( success = allocVector( LGLSXP , 1 ) );

  // Call the theory_rows function to actually make the rows
  success = theory_rows( camb , iamb , cprod , iprod , rvar ,
      ndata , ncur , nend , rlims ,\
                         nranges , arows , irows , mvec ,
                            mvar , nrows , background ,
                            remoterx );

  // Read the row count
  n_rows = *(INTEGER(nrows));

  // Reallocate the vectors to match with the data lengths
  if(fit_size){
    SET_LENGTH( arows , ( n_rows * ( n_ranges + 1 ) ) );
    REPROTECT( arows , arind );
    SET_LENGTH( irows , ( n_rows * ( n_ranges + 1 ) ) );
    REPROTECT( irows , irind );
    SET_LENGTH( mvec , n_rows );
    REPROTECT( mvec , mind );
    SET_LENGTH( mvar , n_rows );
    REPROTECT( mvar , vind );
  }

  // Collect the data into the return list
  SET_VECTOR_ELT( ans , 0 , arows   );
  SET_VECTOR_ELT( ans , 1 , irows   );
  SET_VECTOR_ELT( ans , 2 , mvec    );
  SET_VECTOR_ELT( ans , 3 , mvar    );
  SET_VECTOR_ELT( ans , 4 , nrows   );
  SET_VECTOR_ELT( ans , 5 , success );

  // Set the names attributes
  PROTECT( names = allocVector( STRSXP , 5 ) );
  SET_STRING_ELT( names , 0 , mkChar( c_names[0] ) );
  SET_STRING_ELT( names , 1 , mkChar( c_names[1] ) );
  SET_STRING_ELT( names , 2 , mkChar( c_names[2] ) );
  SET_STRING_ELT( names , 3 , mkChar( c_names[3] ) );
  SET_STRING_ELT( names , 4 , mkChar( c_names[4] ) );
  SET_STRING_ELT( names , 5 , mkChar( c_names[5] ) );
  setAttrib( ans , R_NamesSymbol , names);


  UNPROTECT(7);

  return(ans);

}





/*
```

```
  Make theory matrix rows and measurement vectors.

  This function overwrites existing data vectors

  Arguments:
   camb         Complex range ambiguity functions
   iamb         Index vector of range ambiguity functions
   cprod        Complex lagged product vector
   iprod        Index vector of lagged products
   rvar         Measurement variance vector
   ndata        Data vector length
   ncur         Current sample index
   nend         Last sample index to use (in this call)
   rlims        Range gate limits
   nranges      Number of range gates
   arows        Complex theory rows
   irows        Theory row indices
   mvec         Inversion measurement vector
   mvar         Inversion measurement variances
   nrows        Number of theory rows produced during
                this call
   background   0 if additional background term is not used
   remoterx     0 if measurements TX times should not be used

  Returns:
   success      0 if no theory rows were produced _and_ end of
                data was reached, 1 otherwise
 */


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 )
{
  const Rcomplex * restrict amb = COMPLEX(camb);
  const int * restrict amb_i = LOGICAL(iamb);
  const Rcomplex * restrict prod =  COMPLEX(cprod);
  const int * restrict prod_i = LOGICAL(iprod);
  const double * restrict var =  REAL(rvar);
  int n_cur = *INTEGER(ncur);
  int n_end = *INTEGER(nend);
  const int * restrict r_lims = INTEGER(rlims);
  const int n_ranges = *INTEGER(nranges);
  const int n_data = *INTEGER(ndata);
  const int bg = *LOGICAL(background);
  const int remrx = *LOGICAL(remoterx);
  Rcomplex * restrict a_rows = COMPLEX(arows);
  int * restrict i_rows = LOGICAL(irows);
  Rcomplex * restrict m_vec = COMPLEX(mvec);
  double * restrict m_var = REAL(mvar);
  SEXP success;
  int * restrict i_success;
  int n_rows;
  R_len_t k;
  R_len_t n_start;
  R_len_t i;
  R_len_t j;
  R_len_t subi;
  R_len_t addi;
  R_len_t gati;
  int r_min;
  int r_lim;
  int r_max;
  int r_cur;


  // Check that n_end <= n_data
  n_end = ( n_data > n_end ? n_end : n_data );

  // Check that n_cur <= n_data
  n_cur = ( n_data > n_cur ? n_cur : n_data );

  // Success output
  PROTECT( success = allocVector( LGLSXP , 1 ) );

  // Local pointer to the success output
  i_success = LOGICAL( success );

  // Set the success output
  *i_success = 1;

  // The lowest range gate limnit - 1
  r_min = r_lims[0] - 2 ;

  // Samples with non-zero range ambiguity
  // function at heights below r_lim
  // will not be used in the theory matrix
  // Initialize r_min for monostatic reception
  r_lim = r_min;
  //  -1 (all samples accpected) for remote reception
  if( remrx ) r_lim = -1;

  // The highest range gate limit
  r_max = r_lims[n_ranges] + 1;

  // Make the first theory row.
```

```
n_start = n_cur;
// If we are too close to start of data
// skip points as necessary
if( n_start < r_lims[ n_ranges ] ) n_start = r_lims[
    n_ranges ];

// Make sure that we did not yet pass the end point
if( n_start < n_end ){
  // Go through all range-gates
  for( i = 0 ; i <  n_ranges ; ++i ){
    // Initialize the theory matrix to zero
    a_rows[i].r = .0;
    a_rows[i].i = .0;
    i_rows[i] = 0;

    // Add contribution from all ranges
    // integrated to this gate
    for( j = r_lims[i] ; j < r_lims[ i + 1 ] ; ++j ){

      // In amb_i == 0 points there might be erroneous
  // values from previously calculated lags,
      // it is thus extremely important to check
  // amb_i before addition / subtraction!
      if(amb_i[ n_start - j ]){
        a_rows[i].r += amb[ n_start - j ].r;
        a_rows[i].i += amb[ n_start - j ].i;
        i_rows[i] += amb_i[ n_start - j ];
      }
    }
  }

  // The last gate will be 1 or 0, depending on whether
  // the background ACF will be suppressed or not.
  a_rows[ n_ranges ].r = ( bg == 0 ? 0.0 : 1.0);
  a_rows[ n_ranges ].i = 0.0;
  i_rows[ n_ranges ]   = ( bg == 0 ? 0 : 1 );

// If the first row could not be formed
// set success to false and return
}else{
  *i_success = 0;
}

// From this point on all possible theory rows will  be
// formed but only those with indprod set are stored,
// others are immediately overwritten

// Number of stored rows
n_rows = 0;
```

```

  // Range from the latest pulse
  r_cur = r_max;
  for( k = (n_start-r_min) ; k < n_start ; ++k ){
    if( k >= 0 ){
      if(amb_i[k]){
        r_cur = 0;
      }else{
        ++r_cur;
      }
    }
  }

  // Use all data points from n_start to n_end
  for( k = n_start ; k < n_end ; ++k ){

    // If this data point will be used (!=0 for clarity,
    // the prod_i vector may contains values larger than 1)
    if( (prod_i[k] != 0) & (r_cur > r_lim) & (r_cur < r_max)) {

      // Copy data to the measurement vector
      m_vec[n_rows].r = prod[k].r;
      m_vec[n_rows].i = prod[k].i;
      m_var[n_rows]   = var[k];

      // Copy the current theory vectors to the next one.
      for( i = 0 ; i <  ( n_ranges + 1 ) ; ++i ){
        i_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ]   = i_rows[ n_rows * ( n_ranges + 1 ) + i ];
        // Set the theory rows exactly to zero at points
    // where the index vector is zero. This makes
    // identification of blind ranges much easier.
        if(i_rows[ n_rows   * ( n_ranges + 1 ) + i ]==0){
          a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].r = 0.0;
          a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].i = 0.0;
          a_rows[ n_rows * ( n_ranges + 1 ) + i ].r = 0.0;
          a_rows[ n_rows * ( n_ranges + 1 ) + i ].i = 0.0;
        // Otherwise copy the theory matrix row
        }else{
          a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].r = a_rows[ n_rows * ( n_ranges + 1 ) + i ].r;
          a_rows[ ( n_rows + 1 ) * ( n_ranges + 1 ) + i ].i = a_rows[ n_rows * ( n_ranges + 1 ) + i ].i;
        }
      }

      // Increment the theory row counter
      ++n_rows;

    }

    // Now form the next theory row using the previous
    // one and the range limit indices
    for( i = 0 ; i < n_ranges ; ++i ){
      // Index in the theory matrix
      // (that is stored as a vector)
      gati = n_rows * ( n_ranges + 1 ) + i;
      // Index of the data point that
      // will be added to this gate
      addi = k - r_lims[i] + 1;
      // Index of the data point that
      // will be subtracted from this gate
      subi = k - r_lims[i+1] + 1;

      // Do additions / subtractions only if the point
      // contains a non-zero ambiguity value
      if( amb_i[ addi ] ){
        a_rows[ gati ].r += amb[ addi ].r;
        a_rows[ gati ].i += amb[ addi ].i;
        i_rows[ gati ]   += amb_i[ addi ];
      }
      if( amb_i[ subi ] ){
        a_rows[ gati ].r -= amb[ subi ].r;
        a_rows[ gati ].i -= amb[ subi ].i;
        i_rows[ gati ]   -= amb_i[ subi ];
      }

    }

    // Count samples to exclude everything that contains
    // echoes from below the first gate
    if( amb_i[ k ] ){
      r_cur = 0;
    }else{
      ++r_cur;
    }

  }

  // Write the row count to the output variable
  *( INTEGER( nrows ) ) = n_rows;

  // Update the current position in the data vector
  *( INTEGER( ncur ) ) = n_end;

  UNPROTECT(1);

  return(success);

}
```

5.5.17 range_ambiguity.c

```c
// file:range_ambiguity.c
// (c) 2010- University of Oulu, Finland
// Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
// Licensed under FreeBSD license.

#include "LPI.h"

/*
  Range ambiguity function with linear
  interpolation of TX data

  Arguments:
   cdata1  First complex transmitter samples
   cdata2  Second complex transmitter samples
   idata1  First transmitter sample indices
   idata2  Seconds transmitter sample indices
   cdatap  Complex range ambiguity function
   idatap  Range ambiguity index vector
   ndata1  Length of vectors cdata1 and idata1
   ndata2  Length of vectors cdata2 and idata2
   lag     Lag

  Returns:
   success 1 if all processing was successful, 0 otherwise

*/

SEXP range_ambiguity( SEXP cdata1 , SEXP cdata2 , SEXP idata1
    , SEXP idata2 , SEXP cdatap , SEXP idatap , SEXP ndata1 ,
    SEXP ndata2 , SEXP lag )
{
  Rcomplex *cd1 = COMPLEX(cdata1);
  Rcomplex *cd2 = COMPLEX(cdata2);
  int *id1 = LOGICAL(idata1);
  int *id2 = LOGICAL(idata2);
  Rcomplex *cdp = COMPLEX(cdatap);
  int *idp =  LOGICAL(idatap);
  int nd1 = *INTEGER(ndata1);
  int nd2 = *INTEGER(ndata2);
  int l = *INTEGER(lag);
  SEXP success;
  int *isuccess;
  int k = 0;
  int npr;
  int ninterp = AMB_N_INTERP;
  int i;
  double * tmpr1;
  double * tmpi1;
  double * tmpr2;
  double * tmpi2;

  // Allocate temporary vectors for interpolated data
  tmpr1 = (double*) R_Calloc( 2*ninterp , double );
  tmpi1 = (double*) R_Calloc( 2*ninterp , double );
  tmpr2 = (double*) R_Calloc( 2*ninterp , double );
  tmpi2 = (double*) R_Calloc( 2*ninterp , double );

  // Output data length will be minimum of the
  // two input data lengths, minus the lag
  npr = nd1 - l;
  if( nd1 > nd2 ) npr = nd2 - l;

  // Allocate the success return value
  PROTECT( success = allocVector( LGLSXP , 1 ) );

  // A local pointer to the success value
  isuccess = LOGICAL( success );
  *isuccess = 1;

  // The actual lagged product calculation
  for( k = 0 ; k < npr ; ++k ){
    // The index vector
    idp[k] = (id1[k] * id2[k+ l]);
    // Multiply data values only if the index vector was set
    if(idp[k]){
      // Initialize the temporary vectors to zero
      for( i = 0 ; i < ( 2 * ninterp ) ; ++i ){
    tmpr1[i] = .0;
    tmpi1[i] = .0;
    tmpr2[i] = .0;
    tmpi2[i] = .0;
      }
      // Linear interpolation towards the previous data point
      if( k > 1 ){
    for( i = 0 ; i < ninterp ; ++i ){
      tmpr1[i] = cd1[k-1].r + ( cd1[k].r - cd1[k-1].r ) * (
          1. - (double)i / (double)( 2 * ninterp ) );
      tmpi1[i] = cd1[k-1].i + ( cd1[k].i - cd1[k-1].i ) * (
          1. - (double)i / (double)( 2 * ninterp ) );
      tmpr2[i] = cd2[k-1+l].r + ( cd2[k+l].r - cd2[k-1+l].r )
           * ( 1. - (double)i / (double)( 2 * ninterp ) );
      tmpi2[i] = cd2[k-1+l].i + ( cd2[k+l].i - cd2[k-1+l].i )
           * ( 1. - (double)i / (double)( 2 * ninterp ) );
    }
      }
      // Linear interpolation towards the next data point
      if( k < npr ){
        for( i = 0 ; i < ninterp ; ++i ){
          tmpr1[i+ninterp] = cd1[k].r + ( cd1[k+1].r - cd1[k].r ) * ( (double)i / (double)( 2 * ninterp ) );
          tmpi1[i+ninterp] = cd1[k].i + ( cd1[k+1].i - cd1[k].i ) * ( (double)i / (double)( 2 * ninterp ) );
          tmpr2[i+ninterp] = cd2[k+l].r + ( cd2[k+1+l].r - cd2[k+l].r ) * ( (double)i / (double)( 2 * ninterp ) );
          tmpi2[i+ninterp] = cd2[k+l].i + ( cd2[k+1+l].i - cd2[k+l].i ) * ( (double)i / (double)( 2 * ninterp ) );
        }
      }
      // Initialize the final data value to zero
      cdp[k].r = .0;
      cdp[k].i = .0;
      // Add products of the interpolated data
      for( i = 0 ; i < ( 2 * ninterp ) ; ++i ){
        cdp[k].r += tmpr1[i] * tmpr2[i] + tmpi1[i] * tmpi2[i];
        cdp[k].i += tmpr1[i] * tmpi2[i] - tmpi1[i] * tmpr2[i];
      }
      // Divide with number of summed values
      cdp[k].r /= (double)(2*ninterp);
      cdp[k].i /= (double)(2*ninterp);
    }
  }

  // Set l index values from the beginning to false
  for( k = 0 ; k < l ; ++k ){
    idp[npr+k] = 0;
  }

  // Free the temporary vectors
  Free(tmpr1);
  Free(tmpi1);
  Free(tmpr2);
  Free(tmpi2);

  UNPROTECT(1);

  return(success);

}
```