EISCAT_3D
Final Report of Work Package 11

Written by Mikko Orispää, Ilkka Virtanen and Markku Lehtinen

Deliverable D11.2 of Work Package 11:
Software Theory and Implementation
of the European Commission 7th Framework Programme project

EISCAT_3D: A European three-dimensional imaging radar for atmospheric and geospace research (Preparatory Phase)
Project number: 261967

Report No 67
Sodankylä 2014
Project number: 261967

Authors:
Mikko Orispää, Ilkka Virtanen and Markku Lehtinen

Sodankylä Geophysical Observatory
Tähteläntie 62
FI-99600 Sodankylä
FINLAND

On behalf of:
EISCAT 3D Preparatory Phase
The EISCAT Scientific Association,
PO Box 812,
SE-981 28 Kiruna
SWEDEN
Tel: +46-980-79150
Fax: +46-980-79159

Sodankylä Geophysical Observatory Report No. 67
ISSN 0359-3657

Sodankylä, 2014
EISCAT_3D Preparatory Phase

Work Package 11: Software theory and implementation

Final Report

Deliverable 11.2

Mikko Orispää, Ilkka Virtanen and Markku Lehtinen
Contents

Work Package 11: Software theory and implementation 3

Task 11.1: Productification and parallelisation of FLIPS 4

FLIPS and statistical linear inverse problems 4
OpenCL 8
RLIPS implementation 10
Performance 13
Conclusions and things to do 14

Task 11.2: Development of signal processing and beam-forming software 17
beam-forming 17
Implementation and beam-forming simulations 21
Conclusions and things to do 24

Task 11.3: Development of new multi-purpose codes 29
Multipurpose modulations 29
  Requirements for transmission modulations 29
  Available modulation techniques 29
  EISCAT_3D multipurpose experiments 32

Task 11.4: Productification of the analysis software package 35
LPI 35

Task 11.5: Integration of hardware and software 38
Work Package 11: Software theory and implementation

The purpose of this Work Package is to develop the software modules that are required for the data processing and analysis tasks of the EISCAT_3D radar system. The development of new data algorithms and software is needed because of the complexity of a distributed phased-array incoherent radar system compared to earlier systems.

The Work Package consisted of five tasks:

**Task 11.1** Parallelisation of the basic inverse problem-solving tools for signal processing and data analysis for use in the EISCAT_3D radar system.

**Task 11.2** Production of software for the signal processing and beam-forming systems, which is done with tight connections to WP7 (Digital Signal Processing).

**Task 11.3** Development of multi-purpose codes to allow the EISCAT_3D system to be used in an optimised fashion.

**Task 11.4** Development of data analysis software to be applicable to multi-beam measurements and imaging applications, and to allow for some new experimental methods that the new system will allow.

**Task 11.5** Integration of EISCAT_3D hardware and software

In the following sections we give a summary of the work done. The actual theory behind the software and the documentation of their implementation is given as Appendices in separate documents, list of which is given at the end of this document. These Appendices are part of the Final Report.
Task 11.1: Productification and parallelisation of FLIPS

The purpose of this task was to modify the existing software module FLIPS to take advantage of the modern multicore CPU’s instead of using just one core. We first studied the possibility of modifying FLIPS for cluster computers consisting of multiple CPU’s but after a consideration decided instead to modify FLIPS for GPU (Graphics Processing Unit) architecture.

Modern-day GPU’s are so called many-core processors and they can contain up to thousands of cores. These GPU-cores are smaller and simpler than corresponding CPU-cores but their sheer number makes them suitable for many computing tasks. They are especially suitable for what is called embarrassingly parallel computational problems, i.e. problems that can be separated into parallel tasks without dependency between them.[20]

As a result we wrote a R package RLIPS (R Linear Inverse Problem Solver) which is a modification of FLIPS utilising the processing power of the GPU’s

FLIPS and statistical linear inverse problems

FLIPS or Fortran Linear Inverse Problem Solver [21] is a Fortran95 module for solving large scale overdetermined statistical linear inverse problems. Its development started in 2004 at Sodankylä Geophysical Observatory as an effort to build a free open source alternative for proprietary GULIPS software developed by Sodankylä based Invers Ltd.
Its development has since been funded by TEKES\(^1\)(MASI program, project "Inverse problems and reliability of modelling") and Academy of Finland (through the Finnish Centre of Excellence in Inverse Problems Research).

FLIPS was designed to numerically solve linear inverse problems of form

$$m = Ax + \epsilon,$$  \hspace{1cm} (1)

where \(m\) is typically a vector containing measured data, \(A\) is a known matrix called theory or model matrix, \(\epsilon\) is a vector containing errors, the exact values of which are normally unknown, and \(x\) is a vector called unknown. Typically, the inverse problem is to found out as much as possible about the unknown \(x\) when the measurement vector \(m\) is given and some statistical properties (for example the probability distribution) of the error \(\epsilon\) are known.

In Bayesian framework \([11]\) the measurement vector \(m\), the error vector \(\epsilon\) and the unknown \(x\) are taken to be realisations of real or complex valued random variables. Also, it is assumed that some \textit{a priori} information about the unknown \(x\) is available and it can be coded into a known probability density denoted by \(\pi(x)\). It is also assumed that the so called \textit{likelihood density} (the conditional probability density of the measurement given the unknown) denoted by \(\pi(m|x)\) is either known or it can be estimated. Then by the famous Bayes’ formula the \textit{a posteriori} density of the unknown (or the conditional probability density of the unknown \(x\) given the measurement data \(m\)) denoted by \(\pi(x|m)\) is given by

$$\pi(x|m) = \frac{\pi(x)\pi(m|x)}{\pi(m)}.$$  \hspace{1cm} (2)

Here the term \(\pi(m)\) is the probability density of the measurement. In many cases it acts as a norming constant and the equation (2) is written as

$$\pi(x|m) \propto \pi(x)\pi(m|x),$$  \hspace{1cm} (3)

i.e. the posteriori density is proportional to the product of the prior and likelihood densities. Instead of being a simple vector, the solution of a statistical inverse problem is given as a joint probability density of the components of the unknown random variable \(x\).

\(^1\)Finnish Funding Agency for Innovation
In many cases the probability distribution of the error vector $\epsilon$ is taken to be Gaussian, i.e. multivariate normal distribution, given as

$$
\pi(\epsilon) = \frac{1}{(2\pi)^{k/2}|\Sigma|^{1/2}} \exp\left\{ -\frac{1}{2} (\epsilon - \bar{\epsilon})^T \Sigma^{-1} (\epsilon - \bar{\epsilon}) \right\},
$$

(4)

where $k$ is the dimension of the error vector, and $\bar{\epsilon}$ and $\Sigma$ are the expectation value and covariance matrix of the multivariate normal distribution, respectively. This is denoted by

$$
\epsilon \sim \mathcal{N}(\bar{\epsilon}, \Sigma).
$$

(5)

Then also the likelihood $\pi(m|x)$ is Gaussian,

$$
\pi(m|x) \propto \exp\left\{ (m - Ax - \bar{\epsilon})^T \Sigma^{-1} (m - Ax - \bar{\epsilon}) \right\}.
$$

(6)

If also the prior density of the unknown $x$ can be presented as a Gaussian distribution,

$$
\pi(x) \sim \mathcal{N}(\bar{x}, \Sigma_{pr}),
$$

(7)

where vector $\bar{x}$ and matrix $\Sigma_{pr}$ are called a priori values and covariance, respectively, then it can be shown that the posterior density $\pi(x|m)$ is also Gaussian, or

$$
\pi(x|m) \sim \mathcal{N}(x_0, \Sigma_{post}),
$$

(8)

where the posterior values $x_0$ and the posterior covariance matrix are given by the formulae

$$
\Sigma_{post} = \left( \Sigma_{pr}^{-1} + A^T \Sigma^{-1} A \right)^{-1},
$$

(9)

$$
x_0 = \Sigma_{post} A^T \Sigma^{-1} \left( m - \bar{\epsilon} + \Sigma_{pr}^{-1} \bar{x} \right).
$$

(10)

This is considered to be the solution to the statistical linear inverse problem.

The above problem can be written as a overdetermined linear system

$$
y = Bx,
$$

(11)

where $B$ and $y$ are given by

$$
B = \begin{bmatrix} C^{-1}A \\ C_{pr}^{-1} \end{bmatrix}, \quad y = \begin{bmatrix} m \\ 0 \end{bmatrix},
$$

(12)

\footnote{Also in the case of radar measurements.}
where matrices $C$ and $C_{pr}$ are upper triangular Cholesky factors \([7]\) of the covariance matrices $\Sigma$ and $\Sigma_{pr}$, respectively, i.e.

$$\Sigma = CC^T, \quad \Sigma_{pr} = C_{pr}C_{pr}^T. \quad (13)$$

The solution of equation (11) is given in the least squares sense by formula

$$x_0 = (B^T B)^{-1} B^T y, \quad (14)$$

which reduces to equation (10), the posterior covariance matrix given as

$$\Sigma_{post} = (B^T B)^{-1}. \quad (15)$$

The formulae (9) and (10) (or (14) and (15)) are unpractical if the dimensions of the theory matrix $A$ and the covariance matrices $\Sigma$ and $\Sigma_{pr}$ are large, as the numerical inversion of large matrices is computationally expensive operation.

FLIPS was developed to overcome this problem. It transforms the overdetermined system (11) into a much simpler one

$$Y = Rx, \quad (16)$$

where the matrix $R$ is a square upper triangular matrix with positive diagonal elements. It contains as many rows and columns as the unknown vector $x$ and the vector $Y$ is of the same size as $x$. The least squares solution of the original system (11) is then simply

$$x_0 = R^{-1} Y, \quad (17)$$

as $R$ is invertible. The solution is easy to calculate using so called back substitution algorithm. Moreover, the posterior covariance matrix is given simply by

$$\Sigma_{post} = (R^T R)^{-1}. \quad (18)$$

FLIPS transforms the equation (11) into the equation (16) by using QR decomposition \([7]\), i.e. by decomposing the theory matrix $B$ into a form

$$B = QR \quad (19)$$

where $Q$ is orthogonal matrix and $R$ is upper triangular. Because the orthogonality of $Q$ the equation (11) can be written in form

$$Q^T y = Rx \quad (20)$$
and by denoting \( Y = Q^T y \), the equation (16) is obtained. FLIPS uses so called Givens rotations to calculate the QR decomposition of the matrix \( B \) row-by-row. See Appendix A for details.

Because the QR decomposition is calculated row-by-row it is possible to feed the matrix \( B \) and the vector \( y \) of equation (11) in pieces. This can decrease the memory footprint of the solver considerably if the matrix \( B \) is very large. Also, the vector \( Y = Q^T y \) is formed row-by-row so there is no need to store the whole matrix \( Q \) in the computer memory. Only the upper triangular matrix \( R \) is resident in the computer memory at all times.

**OpenCL**

OpenCL (Open Computing Language) is a framework for programming heterogeneous computer systems consisting different processing units like CPU’s, GPU’s and DSP’s. It was initially developed by Apple Inc, but is nowadays developed by Khronos Group, a not-for-profit member-funded industry consortium. It was first released in December 2008 [19].

OpenCL framework consists of C99-based programming language for writing kernels (functions executed on OpenCL devices) and API’s (Application Programming Interfaces) for defining and controlling the computing platforms. OpenCL is intended for parallel computing using task- and data-based parallelism.

OpenCL computing platform consist of a single host which is connected to one or more OpenCL devices (or compute devices). OpenCL devices are further divided into compute units which in turn are divided into processing elements.

Typically the host is a standard computer consisting of one or more CPU’s and OpenCL devices are the GPU’s in either standard graphics cards or dedicated computing cards like NVIDIA’s Tesla GPU Accelerators. Graphics and computing cards can have one or more GPU’s so a single graphics card can contain more than one OpenCL device. GPU’s then have—depending on their internal structure—several compute units and these compute units have several processing elements.

OpenCL programs consists of standard C or C++ code (also interfaces for other programming languages exist) that is run in host system, and OpenCL kernel functions that
Figure 1: OpenCL platform consisting of one host and three OpenCL devices. Each OpenCL device has three compute units. Each compute units has several processing elements.

are run in OpenCL devices and are written in OpenCL C language which is a variant of C99 with some extensions. Unlike normal C code, the OpenCL kernel functions are incorporated in the application as source code and are compiled, optimised for the existing hardware and transferred into the OpenCL devices at the runtime. The data which is going to be processed in the OpenCL devices must first be transferred from the host system to the device. host system then launches the OpenCL kernels in the device. The kernel functions process the data by work-items which are the smallest processing units in OpenCL program. These work-items can be divided into work-groups. There can be several work-groups in a kernel function. The optimal number of work-groups and work-items per work-group depend on the situation. After the kernel has run, the data must be transferred back to the host before it can be used by the host system. Especially if the amount of data processed by the OpenCL device is large, the overhead of data
transfers can be significant and must be taken into account when designing and writing OpenCL applications.

Figure 2: The data processing is done by work-items which are divided into work-groups.

Another complication in OpenCL programming is the memory management. Both the host and the devices have separate memory spaces. The data transfer from the host to the device and vice versa needs to be done because of this. In addition, the OpenCL device has several different memory spaces that must be managed manually by the programmer. Every work-item has its own small private memory, every work-group has its own local memory which the work-items can access and the device itself has a global memory which can be accessed by every work-group and work-item.

**RLIPS implementation**

RLIPS is a R package that is modified from the original FLIPS. Also FLIPS had a very basic R interface that used binary files written on a hard disk to transfer data from R to FLIPS engine, a separate Fortran program that processed the data and wrote the
The purpose of Task 11.1 was to modify FLIPS to utilise parallel processing. This was done using OpenCL. The QR decomposition in FLIPS and RLIPS is done by using so-called Givens rotations. In RLIPS, these rotations were parallelised using OpenCL.

Givens rotations are orthogonal plane rotations of form

\[
\begin{bmatrix}
  c & s \\
  -s & c
\end{bmatrix}
\begin{bmatrix}
  a \\
  b
\end{bmatrix} =
\begin{bmatrix}
  r \\
  0
\end{bmatrix},
\]

(21)

where

\[
c = \frac{a}{\sqrt{a^2 + b^2}}, \quad s = \frac{b}{\sqrt{a^2 + b^2}} \quad \text{and} \quad r = \sqrt{a^2 + b^2}.
\]

(22)

These rotations can be used to introduce zero elements into matrices. These rotations affect only two matrix rows at the time, and by doing the rotations in a right order, any matrix can be transformed into a upper-triangular form [7].
In FLIPS, the rotations were made sequentially row-by-row. Table 1 shows how a 9 × 6-dimensional matrix is rotated. Numbers represent the order in which the matrix elements are rotated to zero. 33 rotations are needed to transform the matrix into a upper-triangular one.

<table>
<thead>
<tr>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>x</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 1: Sequential rotation scheme

In RLIPS, the rotations are parallelised using OpenCL. Table 2 shows how RLIPS would rotate a 9 × 6-matrix. Again, the numbers give the order in which the elements are zeroed. Now however multiple rotations are done at the same time. For example, at stage 9, four simultaneous rotations are made.

<table>
<thead>
<tr>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>10</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 2: Parallel rotation scheme

When a new RLIPS problem is initialised, the space for the upper-triangular matrix $R$ and vector $Y$ of equation (16) are allocated in the OpenCL device. In the host side,
memory space is allocated for buffers that can hold some predetermined number of the theory matrix rows and measurement vector elements. The data, i.e. the theory matrix rows and measurement vector elements of equation (1) together with either measurement error variances or whole covariance matrix of equation (5), are first fed into R either row-by-row or in chunks. In R, the data is first transformed into the form of (11) and then transferred to the OpenCL device. The data is then parallelly rotated into upper-triangular form.

When all data is fed in, R transfers the upper-triangular matrix $R$ and vector $Y$ from the OpenCL device back to the host system, and the least squares solution (17) and the posterior covariance matrix (18) are calculated.

**Performance**

We have made performance tests with the equipment we have had available. In these tests variable sized problems are solved. For GPU’s we used the RLIPS and for CPU’s we used an unpublished sequential version of the software which is written fully in R.

In Figures 4–7 are represented the results of performance tests for two CPU’s and two GPU cards. The CPU’s were Intel Core i7 2.5GHz and Intel Xeon E5630 2.53GHz processors, and the GPU’s were NVIDIA Tesla C2050 GPU Accelerator card, AMD Radeon 6670M graphics card and AMD FirePro D500 graphics card.

In the first test, we tested the solving of linear systems with real-valued single precision square theory matrix, i.e. a linear system

$$m = Ax,$$

where the matrix $A$ has the same number of rows and columns. It can be seen that parallelised RLIPS is up to 10 times faster than the sequential solver. The performance depends heavily on used GPU card, as was expected. For AMD Radeon 6670M graphics card the largest matrix that could be solved was $7000 \times 7000$. This was due to the low amount of GDDR memory the card contains.

In the second test, we tested overdetermined systems, i.e. systems in which the theory matrix has more rows than columns. Number of columns were fixed to 1000 and the
number of rows varied from 1000 to 50,000. Again, the parallel version of RLIPS achieves up to 10 times the performance of the CPU version. Note that the memory constrain of AMD Radeon 6670M graphics card was no issue in this test because RLIPS stores only the square upper-triangular part of the rotated matrix.

Conclusions and things to do

We developed R package RLIPS as a modified version of older FLIPS software. RLIPS is parallelised using OpenCL and is up to 10 times faster than the old sequential software. Documentation and full implementation including the source code is given in Appendix A. RLIPS is licensed under freeBSD type open source license.

The performance of RLIPS is satisfactory with regard to the requirements of EIS-CAT_3D. However, it would be very beneficial to test the software using the newest possible hardware, especially with the recently released Intel Xeon Phi coprocessor cards.

Also, the software itself has room for improvement. The code could and should be optimised further, and we could also consider completely different algorithm for performing the QR decomposition, instead of Givens rotations, that could be better suited for parallelisation. One of such alternative algorithm could be so called Householder reflections [7].
Figure 4: Solving times for square matrices.

Figure 5: Performance in gigaflops for square matrices.
Figure 6: Solving times for overdetermined systems.

Figure 7: Performance in gigaflops for overdetermined systems.
Task 11.2: Development of signal processing and beam-forming software

In this task the purpose was to develop DSP and beam-forming routines and software for EISCAT_3D radar. The developed routines are based on the theory given in the Chapters 4 and 5 of the EISCAT_3D Measurement Method Handbook by Markku Lehtinen (Work Package 6) [17]. We refer to the Handbook [17] for details.

beam-forming

By beam-forming we mean the combination of data from different antennas of a phased array in order to receive signals from a given direction. This is achieved by combining the different data streams in such a way that the signals originating from certain angle experience constructive interference while the other signals from different angles will be suppressed by the destructive interference.

The single antennas in a large antenna array are typically non-directional, i.e. they receive signals from different directions. beam-forming is the combination of signals from the set of single antennas to simulate a large directional antenna. The data streams from different antennas are summed up in a way that the maximum constructive interference is obtained at the wanted beam direction.

As the signals from different directions reach the antennas in the antenna array at different times, the data streams from different antennas must be delayed to compensate that time difference in order to create the maximum constructive interference of the incoming
signal. In Figure 8 is represented a 2D example of such delays. A signal coming from
a direction $\theta$ will reach the antennas $A_i$, $i = 0,1,\ldots,4$ at different times. The time
differences at antennas $A_i$, $i = 1,\ldots,4$ with respect to the antenna $A_0$—which will be
the last antenna to receive the signal—are $d_i/c$, where $c$ is the speed of light and $d_i$,
i = 1,\ldots,4 are the lengths of orthogonal projections of the relative position vectors of
antennas $A_i$, $i = 1,\ldots,4$ with respect to the position of antenna $A_0$, and the beam direction
(unit) vector, i.e.

$$d_i = s_i \cos \theta,$$  \hspace{1cm} (24)

where $s_i$, $i = 1,\ldots,4$, is the distance between antennas $A_i$ and $A_0$ and $\theta$ is the angle of
the beam.

In the more general 3D-case the beam direction is given by the polar angle $\theta$ and
azimuthal angle $\phi$ and the antenna vectors $a_i = (x_i, y_i, z_i)$ are given with respect to some
reference point $r = (x_0, y_0, z_0)$. Then the length of the orthogonal projection $p$ of the
antenna vector $a_i$ with respect to the beam (unit vector) is

$$|p| = x_i \sin \theta \sin \phi + y_i \sin \theta \cos \phi + z_i \cos \theta.$$  \hspace{1cm} (25)

See Figure 9 for illustration.

In order to create the final beam, all the delayed signal streams are summed together. For
example, if an antenna array contains $N$ antennas, each antenna records a signal stream
$S_i(t)$, $i = 1,\ldots,N$, $t$ is time, and the different antennas have delays $d_i$, $i = 1,\ldots,N$, with
respect to a beam direction $(\theta, \phi)$, then the beam-formed beam $S_{(\theta,\phi)}$ to the direction
$(\theta, \phi)$ would be

$$S_{(\theta,\phi)}(t) = \sum_{i=1}^{N} S_i(t - d_i).$$  \hspace{1cm} (26)

In practice, we are dealing with sampled data, i.e. discrete (real) voltage values recorded
in each antenna. As an example, let us consider two antennas situated at distance $s$ from
each other. As the electro-magnetic signal travels at the speed of light $c \approx 300,000$km/s
the time difference $\tau$ of a signal arriving from angle $\theta$ to the antennas is

$$\tau = \frac{d}{c}$$  \hspace{1cm} (27)

where $d$ is the difference in the distance,

$$d = s \cos \theta.$$  \hspace{1cm} (28)
If the signal is sampled at frequency $f_s$,

$$f_s = \frac{1}{T},$$

where $T$ is the sampling interval, the difference in sampling units will be

$$d_s = \tau / T = \frac{f_s d}{c} = \frac{f_s s \cos \theta}{c}. \quad (30)$$

In order to form a beam to direction $\theta$ the data received with the other antenna must be delayed $d_s$ samples.

For a 3-dimensional antenna array where antennas are positioned at points $a_i = (x_i, y_i, z_i)$, $x_i, y_i, z_i \in \mathbb{R}$, $i = 1, 2, \ldots, N$, and the beam direction is given by two parameters, the azimuthal angle $\phi$ and polar angle $\theta$, the relative delays can be calculated (with respect to origin) as

$$d_{s,i,\theta,\phi} = \frac{f_s}{c} (x_i \sin \theta \sin \phi + y_i \sin \theta \cos \phi + z_i \cos \theta). \quad (31)$$

Finally, the beam to the direction $(\theta, \phi)$ is constructed by summing up the delayed data streams from all the antennas. Let $S_{(\theta, \phi)}(n)$ denote the final (discrete) data stream formed from the signals coming from the direction $(\theta, \phi)$,

$$S_{(\theta, \phi)}(n) = \sum_{i=1}^{N} S_i(n - d_{s,i,\theta,\phi}), \quad (32)$$

where $S_i(n)$, $i = 1, \ldots, N$, is the original discrete data stream from the antenna $a_i$.

In general, the sample delays $d_{s,i,\theta,\phi}$ are decimal numbers. The integral part of such number corresponds to number of whole samples that are needed to be delayed and is easily implemented by shifting the memory address from where the data stream is read. The fractional part of the sample delay must be dealt differently.

For the fractional part of the delay, calculate the fractional sample delay $d_{f,i}$ by

$$d_{f,i} = d_{s,i} - \lfloor d_{s,i} \rfloor, \quad (33)$$

where $\lfloor d_{s,i} \rfloor$ is the closest integer less or equal to the sample delay $d_{s,i}$ (also known as floor function). Hence,

$$0 \leq d_{f,i} < 1. \quad (34)$$
Figure 8: In order to form a beam into a given direction $\theta$, the data streams from antennas $A_i$, $i = 1, 2, 3, 4$ must be delayed by $d_i/c$ seconds, where $c$ is the speed of light.

Following the ideas developed and presented in the EISCAT_3D Measurement Methods Handbook [17] Chapters 4 and 5, all the necessary filtering operations (mirror image deletion, pass-band filtering and time delay) to the sampled signal can be done with one (or in some cases two) specially designed FIR filters. A source code of a R function constructing such filters is given in the Appendix B. One of the design parameters of the FIR filter is the delay given in fractional sample units $d_{f,i}$. Hence every antenna in the array needs its own FIR filter. The sampled data from all the antennas is then filtered using their respective FIR filters (taking into account also the full sample delays $\lfloor d_{s,i} \rfloor$ by starting the filtering accordingly), and then summed together sample by sample to produce the final beam.
Implementation and beam-forming simulations

In Appendix B is presented algorithms for implementing beam-forming using the FIR filters described in the EISCAT_3D Measurement Methods Handbook [17].

Figure 9: Projection $p$ of antenna position vector $a$ and beam direction vector $b$. 
In order to demonstrate these algorithms we created simulated real data samples for an array of 80 antennas situated in a row about 0.65 meters apart (about half-wavelength of the used center frequency $f_c = 225\text{MHz}$). The antennas were taken to be isotropic radiators, i.e. the gain of the antennas is constant in all directions. We created total of 601 real-valued sinusoidal sources with frequencies $222\text{MHz}$–$228\text{MHz}$ situated 0.3 degrees apart from horizon to horizon over the antenna row, see Figure 10. The simulated samples from every source were calculated and summed up for all 80 antennas, constructing 80 separate data streams. The sampling and central frequencies were chosen to be $f_s = 100\text{MHz}$ and $225\text{MHz}$, respectively. Note that according to the EISCAT_3D Measurement Methods Handbook [17], this is an optimal sampling-central frequency pair, as $225 = (2 + 1/4) \times 100$.

For the FIR filters used in the beam-forming simulations, we used $4\text{MHz}$ half-bandwidth and $2\text{MHz}$ ramp (i.e. passband was $221\text{MHz}$–$229\text{MHz}$ and the transition band was $2\text{MHz}$ on both sides of the pass band). Using these parameters makes it possible to decimate the signal from $100\text{MHz}$ to $10\text{MHz}$ without losing any information. The downside of this one filter approach is that the lengths of the filters (i.e. the number of filter taps) is
Figure 11: Typical one-stage beam-forming FIR filter. Sampling frequency 100MHz, center frequency 225MHz, delay 0.5 sample units. Decimation from 100MHz to 10MHz.

Another possibility is to do the beam-forming in two stages. In the first stage the data streams are filtered using a FIR filter with wider 12MHz transition band (ramp) and zero delays. Using this kind of pass-band filter allows the decimation to 20MHz. After that another FIR filter with sampling frequency of 20MHz and 2MHz is used to delay (and further pass-band filter) the data streams. The resulting beam is identical to the beam one gets using single filter method, but the filters are considerably shorter, only about 20 taps long. The downside of this approach is that the first filter (with wide transition band) is much slower to construct. However, the first filter does not have any delays implemented, so it can be constructed in advance, and only the second stage filter needs to be constructed when the beam-direction is decided. Frequency responses of typical FIR filters used in the two-stage approach is shown in Figures 12 and 13.

Below is plotted power-spectra of beam-formed simulated data using different beam directions and antenna configurations. The red scale plotted on the top of the spectra shows the direction (in degrees) where the simulated source with the respective frequency
was situated. This visualises how the signals from different directions attenuate and how the beam is formed.

**Conclusions and things to do**

The FIR filter based beam-forming described in the EISCAT_3D Measurements Methods Handbook [17] works well. At this point the DSP and beam-forming routines are on a prototype stage. In order to make the routines fast and effective enough they must be optimised and written in some fast compiled language like C, C++ of Fortran. Also the parallelisation of the filter construction and beam-forming algorithms must be developed. The parallelisation method and how it is going to be implemented depends on the computer infrastructure of the coming EISCAT3D radar and can not be decided at this point.
Figure 13: Typical second stage FIR filter. Sampling frequency 20MHz (output of the first stage filter), center frequency 225MHz, delay 0.5 sample units. Decimation from 20MHz to 10MHz.

Figure 14: A beam formed in direction az = 0, inc = 0. 80 antennas spaced half-wavelength apart.
Figure 15: A beam formed in direction $az = 0, \text{inc} = 0$. 10 antennas spaced half-wavelength apart. The beam is much wider as is expected.

Figure 16: A beam tilted 30 degrees. 10 antennas spaced half-wavelength apart.
Figure 17: A beam tilted 60 degrees. 80 antennas spaced half-wavelength apart.

Figure 18: A beam formed in direction $\text{az} = 0$, $\text{inc} = 0$. 40 antennas spaced a wavelength apart. Artifact on the right is a grating-lobe. Note that in this simulation all antennas are taken to be isotropic. In reality, the antenna gain would go to (near) zero at $\pm 90$ degrees, attenuating the grating-lobe.
Figure 19: A beam tilted 30 degrees. 40 antennas spaced a wavelength apart. The main lobe is on the right and the one on the left is a grating-lobe.
Task 11.3: Development of new multi-purpose codes

Multipurpose modulations

Requirements for transmission modulations

EISCAT_3D will observe a variety of targets, each with different requirements for radar transmission modulation. The system will be used both for routine monitoring type observations and for special experiments dedicated for specific targets. Modulation techniques exploited in EISCAT_3D must thus be adaptable enough to allow experiment optimisation for targets with considerably different physical dimensions, radar cross sections, and scattering spectra. Furthermore, the routine monitoring requires multipurpose modulations that combine properties of the target-specific experiments, enabling observations of large variety of targets with the same modulation pattern. Design tools of the multipurpose experiments must also be adaptable enough to allow one to balance sensitivity in between different targets as necessary.

Available modulation techniques

EISCAT_3D will exploit a variety of modulation techniques, including the latest developments in this field, whose combinations will allow the system to be used in an optimal way. Modern transmitter technology of EISCAT_3D will enable use of several new
modulation techniques that are not in routine use in any incoherent scatter radar system in the world.

**Pulse codes** are the lowest level of radar transmission modulation. The transmitted waveform is divided into separate pulses, leaving periods for signal reception in between. In incoherent scatter radars the pulses are usually further coded by means of other modulation techniques. Monostatic radars can operate only with pulse-coded transmissions, but with multistatic systems one can use continuous wave transmission and receive echoes only at remote sites.

**Multipulse codes** [6], including simple pulse pairs, are sequences of short "elementary" pulses, which allow a collection of lag profiles to be measured with range resolution determined by the elementary pulse length. Multipulse codes are different from the pulse codes in the sense that separation of the elementary pulses is not long enough to facilitate signal reception. However, separation between the pulse codes and multipulse codes is not clear since aperiodic transmitter coding [24], also known as simultaneous multiple pulse-repetition frequency [22], combines properties of both techniques.

**Alternating codes**, including both the binary codes used in the present EISCAT system [14] and their polyphase extension [18] are cycles of phase-coded pulses. The transmitted pulses consist of short elementary pulses, and a simple decoding algorithm allows lag profiles to be resolved with range resolution corresponding to the elementary pulse length, or even better if fractional sampling is applied [10]. The decoded lag profiles have zero range sidelobes and, if signal-to-noise ratio is low, the decoded samples are uncorrelated. In high signal-to-noise conditions performance of alternating codes can be significantly improved by means of code randomisation [15]. Shorter sequences of alternating codes with imperfect decoding properties at edges of the target are available at some code and code cycle lengths [26]. In a recent test with the MIT Millstone Hill incoherent scatter radar [30] the polyphase alternating codes were found to perform as theoretically predicted.

**Random codes** [25] are cycles of phases-coded pulses with randomly chosen phases. Their behaviour is similar to alternating codes, except that the sidelobe suppression is not exact but happens only in statistical sense.

**Orthogonal polarisation coding** [9, 8] was recently introduced as a modulation technique that reduces the incoherent scatter self-noise and allows one to replace the standard
"strong" alternating codes with shorter cycles of "weak" codes, as well as to interleave multipulse codes in a novel way. Also Faraday rotation measurements are possible when the transmission is divided in between two orthogonal polarisation channels. Use of the polarisation coding is currently limited to monostatic measurements, since techniques for separating the two polarisations in remote reception are not known.

Barker codes [3], including their polyphase extensions [4, e.g.], are pulse compression codes decoded at voltage level. Barker-coded pulses must thus be shorter than decorrelation time of the scattering process, which limits their use in incoherent scatter applications. In addition, Barker codes are imperfect in the sense that their deciding will either produce range sidelobes or magnify the random measurement noise. Barker codes will be available in EISCAT_3D, but other modulations that provide better sidelobe suppression will be used in routine operation.

Near perfect polyphase codes [5, 16, e.g.] are numerically optimised phase codes or cycles of phase codes, either optimised for lag profile inversion and used as an alternative for the alternating codes, or optimised for voltage level decoding by means of inverse filtering and used as an alternative for Barker codes. Near perfect polyphase codes have been tested and found to perform as theoretically predicted with the MIT Millstone Hill incoherent scatter radar [30]. Near perfect binary codes for lag profile inversion have been used with the EISCAT VHF system in Finnish special experiments for several years [12, e.g.]. Near perfect codes and code cycles within few percent from optimal in terms of signal-to-noise ratio of decoded data can be found by means of numerical optimisation.

Almost perfect pulse compression codes [13, 23] are binary phase-coded and amplitude modulated pulses that allow optimal exploitation of the radar transmission power in voltage level pulse compression in the sense that their decoding can simultaneously provide both perfect range sidelobe suppression and optimal signal-to-noise ratio. The almost optimal pulse compression codes will be used as alternatives for Barker codes in EISCAT_3D. Whether the almost perfect codes or the near perfect polyphase codes are used will depend on application, since the larger backscatter power provided by the constant amplitude of the polyphase codes may compensate for the loss in their worse performance in decoding.
Multipurpose modulations exploit the phase-coded pulse aperiodic transmitter coding (PPATC) technique [31]. These modulations are combinations of aperiodic pulse codes and numerically optimised near perfect polyphase codes for lag profile inversion. In addition to the simple difference cover codes used in [31], other aperiodic pulse coding schemes are suitable for both ionospheric and space debris works [29, 27]. The possibly simplest approach, in which the pulse coding will be changed cyclically in periods much longer than the individual inter-pulse periods, is also very efficient and will be demonstrated in the next section.

EISCAT_3D multipurpose experiments

EISCAT_3D multipurpose codes will need to cover all altitudes from D region to upper F region, which requires in practice range coverage approximately from 50 km to at least 1000 km and measurements of autocovariance function lags from zero to tens of milliseconds. The first few hundred microseconds must be covered with high lag resolution, whereas about one millisecond lag spacing is sufficient for the longer lags used in the D region.

The requirements for range and lag coverage can be fulfilled with the PPATC technique. E.g. a simple difference cover code consisting of three different inter-pulse periods can combine range coverage from very low altitudes up to 1000 km, lags from zero to 500 µs with very high resolution, longer pulse-to-pulse lags with 1 ms resolution, and 21 % transmitter duty cycle. Numerically optimised near perfect phase code cycles providing lag profile variances within 10 percent from theoretical optimum can be found for the PPATC experiments [31].

As an alternative to the original PPATC technique, one can divide the incoherent integration period into short sections, each with different but uniform inter-pulse periods and pulse lengths. This approach makes the modulation design very simple, as it allows one to optimise most of the phase-codes with the simple FFT-based algorithm of [16] or to use e.g. alternating codes. Drawback of the technique is that measurements from different parts of the ionosphere are not truly simultaneous, which may be significant in some applications.

Range resolution requirements will have substantial differences in different parts of
Figure 20: Left: Lag profile matrix measured with EISCAT VHF on August 21 2012 using a multipurpose modulation. Each autocovariance function is multiplied with square of the radar range in order to make the upper ionosphere better visible in the plot. The white area at lags shorter than 1 ms is a measurement gap whose size could be reduced by including shorter inter-pulse periods in the modulation. Most of the white area at the upper right corner could be covered by the modulation, but is intentionally excluded from the lag profile inversion analysis because F region incoherent scatter signal is expected to be uncorrelated at the lags clearly longer than 1 ms. Right: Lag profile matrix measured with the KAIRA array from the same experiment. The result is not identical with that from the monostatic analysis because the scale is not corrected for KAIRA beamshape effects. The data gap at the short lags is not produced in remote reception because signal can be received also while transmitting. The upper right corner in the plot is again intentionally excluded from the analysis. The measurement was performed before proper calibration of the KAIRA array and the ACFs from different beams of KAIRA are simply summed instead of proper Faraday rotation estimation and selection of best beam for each height. The plot at right hand side is thus included only to demonstrate the lag coverage of remote receivers at different heights, far more accurate results can be achieved after careful system calibration and data analysis. Notice the logarithmic time-lag axes in both plots.
the ionosphere. They will also have substantial temporal variations, because the best achievable resolutions will depend on the ionospheric electron density. The lag profile inversion analysis and multipurpose modulations allow one to combine individual codes optimised for different resolutions in the same modulation pattern. Resolutions down to about 1 km can be handled directly with the near optimal code cycles.

Better resolutions are most conveniently achieved by means of sub-coding individual bits of the multipurpose modulations e.g. with almost perfect pulse compression codes or with polyphase codes optimised for voltage level decoding. The voltage level decoding is acceptable because decorrelation time of the incoherent scatter signal is always much longer than duration of a 1 km long pulse. Direct lag profile inversion down to very high range resolution is uneconomical, because it increases the number of lag profiles that must be estimated. The voltage level decoding is fast and reduces the number of individual lag profiles that needs to be inverted.

The multipurpose modulations can be theoretically optimised under low signal-to-noise ratio assumption. However, depending on ionospheric conditions, the EISCAT_3D system may produce very high signal-to-noise ratios. Final experiment optimisation will thus be possible only by means of test measurements after the system has been built. Most probably a set of different modulation patterns will need to be develop in order to adapt for the sometimes rapidly changing conditions in the ionosphere.

In August 21, 2012 a bistatic multipurpose experiment was conducted using the EISCAT VHF system for signal transmission and both the VHF system and the KAIRA array for reception. The modulation consisted of an F/E region part with long pulses and long inter-pulse periods, and a D/E region part with short pulses transmitted with short inter-pulse periods. Both parts of the modulation had duration of one second and they were transmitted subsequently, allowing analysis of full D/E/F region lag profiles with 2 second time resolution. A noticeable property of this combination is that the F region part allows the F region to be properly modelled in D region analysis, thus removing the risk of aliased F region echoes biasing the D region lag profiles. Examples of lag profile matrices from both the monostatic VHF measurement and the bistatic measurement with remote reception at KAIRA are shown in Figure 20. The bistatic operation with a multibeam remote receiver and bistatic multipurpose experiments are both completely new techniques that are currently unique to the combination of EISCAT VHF and KAIRA, but will be possible with the EISCAT_3D system.
Task 11.4: Productification of the analysis software package

LPI

LPI is a lag profile inversion [28] software package for EISCAT_3D. It uses voltage level samples of transmitted and received waveforms as input and produces estimates of the scattering autocovariance or crosscovariance function with selected resolutions in range, lag, and time as output.

In order to adapt for the requirements of interferometric imaging and orthogonal polarisation coding the package takes four data vectors, two vectors of samples of the transmitted waveform and another two vectors of samples of received signal, as input. Selection between crosscovariance function and autocovariance function estimation then reduces to using different combinations of recorded sample streams as inputs.

Optional ground clutter suppression is implemented as voltage level inversion of backscatter signal and successive subtraction of convolution of this estimate and the transmitted waveform. The clutter subtraction is thus essentially a notch filter at zero doppler frequency. The inversion approach allows simple implementation of the filter without need of any additional information about the applied modulation.

Optional voltage level decoding with either user-defined filter coefficients, or as either matched of inverse filter decoding of whole pulses, allows one to decode possible subcoding of bits of other modulations, as well as to gain analysis speed when estimating targets with narrow scattering spectra.
Figure 21: Schematic diagram of parallel LPI analysis. The user starts the analysis from R command line, after which the other processes are automatically started. If the data disk is mounted on same path on all remote computers one can optionally omit the local control nodes and do also the data I/O in remote nodes. This option allows one to use large clusters without running short of memory on the main node computer.

Several options for the actual lag profile inversion are provided. The package can exploit the RLIPS package – linear inverse problem solver that exploits GPUs. In addition, a fast linear inverse problem solver is included in the LPI package itself and an FFT-based option is available for applications in which its poor performance at edges of the target are not important, such as remote reception. For analysis of alternating codes a decoder is provided, and for the D region experiments decoded already at voltage level a dummy solver calculating simple lag profile averages is available. All solvers provide also variance estimates or covariance matrices of lag profiles. Depending on solver, the estimates are either based on measured lag profile variances or on background noise power and assumption of low signal-to-noise ratio.
In order to make the analysis fast enough for high-resolution analysis, the LPI package exploits parallelism at two levels. First of all, several incoherent integration periods can be analysed in parallel because they are completely independent. Secondly, different lags within each integration period can be analysed in parallel without need for intercommunication. The parallelism is implemented in practice using the standard R package ‘parallel’, which allows one to run several R processes on one or several computers and provides simple tools for transferring data and executing commands in each of them. Several processes for different integration periods are run in parallel, and lags of each integration period are further divided in between several processes (see Figure 21).

Because details of the EISCAT_3D raw data format are not known, the data input routines are not included in the core LPI package. Instead, separate I/O packages for different data formats can be developed independently from the core package. Packages for reading the file formats used in the Finnish USRP receivers and in the KAIRA array are currently available. Similarly, users have the possibility to replace the default output routines.

Routines for decoding different pulses with different filters, power calibration from noise injections, etc. are not included in the LPI package, because their implementation requires information that cannot be extracted from the transmission envelopes and echoes alone. Experiment specific parts of analysis can be implemented in the I/O routines, in which they can be stored and maintained independently from the main LPI package.

The LPI package is being used and further developed as the default analysis tool for the voltage level data recorded in Finnish EISCAT and KAIRA experiments. The package has thus been tested as far as possible in real scientific work and has proven its ability to provide high-quality results.
Task 11.5: Integration of hardware and software

Computing hardware

EISCAT_3D requires two separate computing infrastructures, one for beam-forming and other for data analysis. We consider them separately.

Beam-forming

Computing infrastructure for beam-forming is critical for the proper functioning of EISCAT_3D system. It is also challenging because of the very high data rates that are going to be used. We refer here to the calculations made in the EISCAT_3D Measurement Methods Handbook [17].

For the sake of argument, we assume that an antenna array contains 100 antenna groups each of which produce 10 pre-defined beams beam-formed within the antenna groups using for example the daisy-chain approach described in the EISCAT_3D Measurement Methods [17]. For the lack of better term, we call these 10 beams pre-beams. The single data stream from the antenna group to the final processing computer is transferred using 10Gb/s Ethernet. Hence, one pre-beam data stream rate is up to 1Gb/s. The total data stream from the antenna groups to the final processing unit will be $100 \times 10\text{Gb/s} = 1\text{Tb/s}$.

In the final beam-forming stage up to 10 beams are formed inside each of the 10 pre-beams totalling up to 100 final beams. This means that we need to be able to calculate up
Figure 22: Beam-forming cluster 1. The 10Gb/s data streams from each antenna group are connected to computing nodes each of which is capable of delaying 100 data streams (10 pre-beams, 10 final beams inside each pre-beam) producing total of 100Gb/s data stream. These 100Gb/s data streams must then be summed up to create the final 100 beams, requiring 100Gb/s interconnect between the nodes.

to 10000 time delays simultaneously (100 antenna groups, 100 final beams). According to the EISCAT_3D Measurement Methods [17] (Section 8.3.1) this totals up to 3.6TMAC/s or 3.6Tflop/s computing capability.

It is clear that a very large cluster computer is going to be needed to handle the task. The beam-forming (i.e. calculating the time delays) itself is straight-forward and easy to implement. Also the parallelising the delay calculations for separate beams is more or less trivial. The challenge is the communication requirements between the computing nodes. In Figures 22 and 23 are presented two possible beam-forming architectures.

**Data analysis**

The LPI analysis software is currently written as a R package and hence it can be run on multitude of platforms (Linux, Unix, Mac OS X, Windows, etc.). Also, the analysis on separate beams is completely independent on each other, so the parallelisation of
multiple beam analysis is quite trivial. Furthermore, the lag calculations of a single beam analysis can be easily parallelised. The speed of the analysis depends on the analysis parameters, for example, the number of range gates and the integration time, so very specific recommendations are not easy to give.

It can be concluded that a computer cluster build from standard reasonably powerful work stations can be used to analyse the data. More performance can be achieved by using more powerful CPU’s and GPU’s or by increasing the number of computing nodes in the cluster. The used operating system is not a critical issue.

For the maximum performance, the computer hardware must be decided first and the analysis software is then optimised to that particular hardware and infrastructure. However,

Figure 23: Beam-forming cluster 2. The 10Gb/s data streams containing 10 pre-beams from the antenna groups are first separated into 10 1Gb/s streams for each pre-beam. These 1Gb/s pre-beam streams are collected together (100 antenna groups, totalling up to 100Gb/s data stream) and sent to 10 separate computing clusters each responsible to create 10 final beams. If a single cluster consists of 10 computing nodes then the data stream rate for a single node could be 10Gb/s.
the computer technology is developing at such a high rate that the computer hardware should be decided and purchased in the latest stages of the EISCAT_3D construction. Analysis can be started quickly using the current software and the optimisation can be done later.

In conclusion, the construction and operation of the beam-forming cluster will be a major challenge for the coming EISCAT_3D system. Computing and data communication hardware and standards develop so fast that the final decisions about the beam-forming cluster should be done in the late stages of the final EISCAT_3D planning and construction.

**DSP hardware**

**Kilpisjärvi Test System**

Kilpisjärvi Test System (KTS) is an experimental antenna system designed and build by Finnish companies Codelma, RF-Shamaanit and Siru Innovations. It consists of two sets of $2 \times 2$ Yagi antenna arrays together with necessary sampling and DSP hardware. For a picture of the antenna rig see Figure 24.

KTS was tested ta KAIRA site in Kilpisjärvi, Finland, in the beginning of June, 2014. EISCAT Tromso VHF radar was used to send different pulse codings into ionosphere and the scattered signals were received using KST and KAIRA. The tests were successful and we refer to Deliverables D7.2 and D7.4 of Work Package 7 for details [1,2].

The KTS used two-stage FIR filtering in order to decimate the initial sampling frequency of 100MHz to 10MHz. This decimated data (total of 8 channels per $2 \times 2$ array) was the written to hard drives. The first FIR filter used 100MHz sampling frequency, 225MHz center frequency, 8MHz pass-band and 12MHz transition band. Frequency response of the filter is shown in the Figure 25. Length of this filter was 21 taps and it pass-band filters the sampled voltage data and after it the data can be decimated to 20MHz sampling frequency.

The second-stage FIR filter used sampling frequency 20MHZ (as it used the output of the first-stage FIR filter as its input), central frequency 225MHz, 8Mhz pass-band and narrower 2MHz transition band. The frequency response of this filter is shown in the
Length of the filter was 23 taps and after it the data can be decimated by 2 to achieve sample frequency of 10MHz.

As the FIR filters were implemented in FPGA in the sampling hardware, the filters were scaled to use integer values. The actual scaled tap values of the filters are given in the Table 3.

Note that actual beam-forming was not done in this experiment as the focus of the experiment was to test the basic functionality of the sampling system.

Table 3: Tap values (scaled) of the first and second-stage FIR filters used in the KTS experiment. The real and imaginary parts of the taps are listed separately. Note that the taps of the first stage filter are either purely real or purely imaginary, and in the second stage filter only one real-valued tap is non-zero and every second imaginary tap is zero.
Figure 24: One of the Kilpisjärvi Test System $2 \times 2$ Yagi arrays. Sampling and DSP hardware not attached.
Figure 25: Frequency response of the first-stage FIR filter used in the KTS experiment.
Figure 26: Frequency response of the second-stage FIR filter used in the KTS experiment.
Bibliography


Appendices
Appendix A

RLIPS documentation and R package manual
RLIPS
R Linear Inverse Problem Solver
EISCAT_3D, Work Package 11, Task 11.1

Mikko Orispää
Sodankylä Geophysical Observatory
University of Oulu, Finland
Notes

- **Version 0.1 (Writing started)** 2012-07-31
- **Version 1.0** 2013-04-10
License

RLIPS (R Linear Inverse Problem Solver)

Copyright 2010–2012 University of Oulu, Finland. All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

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

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

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

1 Introduction ........................................ 4
   1.1 Purpose of this document ......................... 4
   1.2 R Linear Inverse Problem Solver ................. 4

2 Linear inverse problem ............................... 5
   2.1 Definition ...................................... 5
   2.2 Formal Solution .................................. 6
   2.3 Overdetermined linear system ...................... 8
   2.4 QR decomposition .................................. 9
   2.5 Givens rotations .................................. 10
   2.6 Rlips algorithm .................................. 13

3 Implementation ....................................... 15
   3.1 R files ........................................... 15
      3.1.1 rips environment ............................. 15
      3.1.2 file: rips.init.R ............................. 16
      3.1.3 file: rips.add.R ............................. 19
      3.1.4 file: rips.rotate.R ......................... 24
      3.1.5 file: rips.solve.R ........................... 26
      3.1.6 file: rips.dispose.R ......................... 29
      3.1.7 file: rips.get.data.R ......................... 30
      3.1.8 file: rips.test.R ............................ 32
      3.1.9 file: rips.copy.R ............................ 35
   3.2 C/OpenCL files .................................... 36
      3.2.1 Header file: rips.h ........................... 36
      3.2.2 File: rips.c ................................ 40
      3.2.3 Header file: rotations.h ..................... 68
      3.2.4 File: rotations.c ............................ 68
      3.2.5 File: kernel sources.h ....................... 80
      3.2.6 File: cbacksolve.c ........................... 95
4 User manual

4.1 Installation .............................................. 99
4.2 Loading package in R .................................... 99
4.3 Using rlips package ..................................... 100
  4.3.1 rlips.init ........................................... 101
  4.3.2 rlips.dispose ....................................... 102
  4.3.3 rlips.add ........................................... 103
  4.3.4 rlips.solve ......................................... 105
  4.3.5 rlips.rotate ........................................ 107
  4.3.6 rlips.copy ......................................... 109
  4.3.7 rlips.test .......................................... 110
4.4 Example of use .......................................... 112
Chapter 1

Introduction

1.1 Purpose of this document

Development of R Linear Inverse Problem Solver (RLIPS) was the Task 11.1 of the Work Package 11 (Software theory and implementation) of the European Union’s Framework 7 project EISCAT_3D. This document serves as part of the final report of the Task 11.1 as well as the user and programmer’s manual of RLIPS.

1.2 R Linear Inverse Problem Solver

R Linear Inverse Problem Solver (RLIPS) is a R package for solving large scale (overdetermined) linear inverse problems. It provides an easy and efficient method of solving statistical linear inverse problems of the form

\[ m = Ax + \varepsilon, \]

where the measurement \( m \) and the theory matrix \( A \) are given and the probability distribution of the Gaussian error term \( \varepsilon \) is known or estimated.

RLIPS solves the above problem using QR-decomposition which is constructed row-by-row as the data (measurements \( m \) and the corresponding theory matrix rows) are fed into the RLIPS system.

\(^1\)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.
Chapter 2

Linear inverse problem

2.1 Definition

We define here the real valued inverse problem. The complex one is defined accordingly.

A general linear inverse problem can be presented as a matrix equation

\[ m = A \cdot x + \epsilon, \]  \hspace{1cm} (2.1)

where \( m \in \mathbb{R}^m \), \( A \in \mathbb{R}^{M \times N} \), \( x \in \mathbb{R}^N \) and \( \epsilon \in \mathbb{R}^N \). Here (and from now on) \( m \) is called the measurement, \( A \) is called the direct theory matrix, \( \epsilon \) is called the error and \( x \) is called the unknown. The problem is to find either the exact or best possible value for the unknown \( x \).

Depending on the integers \( M \) and \( N \) and the characteristics of the error \( \epsilon \), there are different possibilities:

If \( M = N \) and the values of \( m \) and \( \epsilon \) are known (in this case we can put \( \epsilon = 0 \) without loss of generality), we have a linear system with \( N \) equations. If \( A \) is invertible, i.e. \( \det(A) \neq 0 \), the system has an unique solution.

If \( A \) is singular, i.e. \( \det(A) = 0 \), its kernel is non-zero. In this case \( m \) either is or is not in the range of \( A \). If \( m \in \text{Ran}(A) \), then the system has more than one solution. However, it is possible to find the shortest vector in the solution space. On the other hand, if \( m \notin \text{Ran}(A) \), the system does not have an exact solution at all. In this case, it is possible to give ”solution” in the least squares sense, i.e., as the vector \( \hat{x} \) that minimizes the norm \( \| A \cdot \hat{x} - m \| \).
If $M < N$ and the values of $m$ and $\varepsilon$ are known (again, we can put $\varepsilon = 0$), we have a linear system with less equations than unknowns. In other words, the system is underdetermined. This problem can be reduced to the previous one by expanding matrix $A$ and vector $m$ with $N - M$ zero rows and elements, respectively. The resulting matrix $A$ will be singular, and the type of the approximate solution depends again if $m$ is in the range of $A$ or not.

If $M > N$ and the values of $m$ and $\varepsilon$ ($\varepsilon = 0$) are known, the system is overdetermined, unless $A$ is row-degenerate in such a way that the system reduces to one of the above ones. Overdetermined system has the solution only in the least squares sense.

If $m$ is known and $\varepsilon$ is not, but its components are random variables with a known joint distribution function, we have what will be called a linear stochastic inverse problem. In this case, the solution is given by the joint posteriori distribution of the unknowns. However, usually it is enough to obtain the maximum a posteriori estimate (MAP) and the posteriori (co)variances of the unknowns.

RLips is designed to solve all above problem types with Gaussian errors. It is able to give the MAP estimates of the unknowns and their posteriori covariance matrix. Note however that solving underdetermined problems with RLips requires some extra work from the user part, as some kind of regularization scheme is needed.

2.2 Formal Solution

We skip most of the mathematical details in what follows, and refer to standard mathematical and stochastic literature\(^1\).

As defined in the last section, the general stochastic linear inverse problem is to find "the best possible" solution to the equation

$$m = A \cdot x + \varepsilon,$$

where matrix $A$ is a known direct theory matrix, and the components of the vectors $m$, $x$ and $\varepsilon$ are random variables. Broadly speaking, this means finding out what kind of information about $x$ can be obtained when the measurement $m$ is fixed and the (statistical) distribution of $\varepsilon$ is known.

In the most general sense, the solution is given as the *posteriori density* of $x$, which is the conditional density of $x$ when the measurement $m$ is fixed. For this, it is necessary to know the distribution function of the error $\varepsilon$. Estimations of error variances can be obtained, for example, by repeating the measurement several times. Then if the mean values of the measurements are used as $m$, the density of errors $\varepsilon$ can be approximated by Gaussian distribution (using the central limit theorem).

If no *a priori* information about $x$ is available, the posteriori density is given by formula

$$\pi(x|m) = C \exp \left\{ -\frac{1}{2} (m - A \cdot x)^T \cdot \Sigma^{-1} \cdot (m - A \cdot x) \right\},$$

where $C$ is a normalization coefficient and $\Sigma$ is the *error covariance matrix*, defined by

$$\Sigma = \langle \varepsilon \cdot \varepsilon^T \rangle.$$

It is (quite) easy to show that the above posteriori density can be written as

$$\pi(x|m) = C \exp \left\{ -\frac{1}{2} \left[ (x - x_0)^T \cdot Q \cdot (x - x_0) + \chi^2 \right] \right\},$$

where

$$Q = A^T \cdot \Sigma^{-1} \cdot A,$$

$$x_0 = Q^{-1} \cdot A^T \cdot \Sigma^{-1} \cdot m,$$

$$\chi^2 = m^T \cdot \Sigma^{-1} \cdot m - x_0^T \cdot Q \cdot x_0.$$

The matrix $Q$ above is called the *Fisher information matrix*, and it is (as is $\Sigma$) positively definite and symmetric. Hence, the exponential function gets its maximum value $-\chi^2/2$ at point $x = x_0$, so $x_0$ is the maximum point of the posteriori density. The width of the distribution is determined by the inverse of $Q$, $Q^{-1} = \Sigma_x$, which is called the *posteriori error covariance matrix*. We can therefore say that the solution to the original inverse problem is given by $x_0$ and $\Sigma_x$. The scalar (or vector) $\chi^2$ is the smallest value (at $x = x_0$) of the quadratic form $(m - A \cdot x)^T \cdot \Sigma^{-1} \cdot (m - A \cdot x)$, and it is called the *residual of the solution*, since it gives some information how well $x_0$ approximates $x$. Note however, that the residual $\chi^2$ does not depend on $x$ and if it is not of particular interest, it is usually embedded in the norming constant $C$.

If, in addition, we have *a priori* information of the unknown $x$ which can be approximated by a Gaussian distribution

$$\pi(x) = C_0 \exp \left\{ -\frac{1}{2} (x - \bar{x}_0)^T \cdot \Sigma_0^{-1} \cdot (x - \bar{x}_0) \right\},$$
where $\bar{x}_0$ are the prior values and $\Sigma_0$ is the prior covariance matrix of $x$, they can be added easily to the formal solution formulae, namely,

$$Q = \Sigma_0^{-1} + A^T \cdot \Sigma^{-1} \cdot A,$$

$$x_0 = Q^{-1} \left( \Sigma_0^{-1} \cdot \bar{x}_0 + A^T \cdot \Sigma^{-1} \cdot m \right).$$

### 2.3 Overdetermined linear system

The above formal solution formulae can be reduced into a standard overdetermined linear system. To this end, write the *a priori* information as a stochastical model

$$x_0 = Ix + \eta, \quad \eta \sim \mathcal{N}(0, \Sigma_0),$$

where $x_0$ is the *a priori* values and $\Sigma_0$ is the *a priori* covariance of the prior. Since the covariance matrices are positive-definite and symmetric, they can be written in form

$$\Sigma = CC^T,$$

where $C$ is a lower-triangular matrix. This is so called *Cholesky factorization*.

The stochastic inverse problem consists now of two linear equations, namely the measurement and prior models,

$$y = Ax + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \Sigma),$$

$$x_0 = Ix + \eta, \quad \eta \sim \mathcal{N}(0, \Sigma_0),$$

for which we had the formal solution formulae given in the previous section.

Now however, this is equivalent of trying to solve the overdetermined problem

$$\begin{bmatrix} C^{-1}y \\ C_0^{-1}x_0 \end{bmatrix} = \begin{bmatrix} C^{-1}A \\ C_0^{-1} \end{bmatrix} x$$

in the least square sense (Here $C$ and $C_0$ are the Cholesky factors of the covariance matrices $\Sigma$ and $\Sigma_0$, respectively). Indeed, the least squares solution to a overdetermined problem

$$b = Bx$$

is given by the Moore-Penrose pseudoinverse of $B$,

$$B^+ = (B^T B)^{-1} B^T.$$
so the least squares solution is

\[ x = B^+y = (B^TB)^{-1}B^T y. \]

Now applying this to the original overdetermined problem gives the least squares solution

\[
\bar{x} = \left( \begin{bmatrix}
C^{-T}A \\
C^{-T}0
\end{bmatrix} \right) \left( \begin{bmatrix}
C^{-1}A \\
C^{-1}0
\end{bmatrix} \right)^{-1} \left( \begin{bmatrix}
A^T C^{-T} & C^{-T}0
\end{bmatrix} \right) \left( \begin{bmatrix}
C^{-1}y \\
C^{-1}x_0
\end{bmatrix} \right)
\]

\[ = (A^T \Sigma^{-1} A + \Sigma_0^{-1})^{-1} (A^T \Sigma^{-1} y + \Sigma_0^{-1} x_0), \]

i.e. the least squares solution \( \bar{x} \) is identical to the MAP-estimate given by the formal solution formulae.

### 2.4 QR decomposition

There exists several efficient numerical algorithms for solving overdetermined linear systems. The one that RLIPS uses is called QR decomposition.

It can be shown that every matrix \( A \in \mathbb{R}^{m \times n}, m \geq n \), can be uniquely written in form

\[ A = QR, \]

where \( Q \in \mathbb{R}^{m \times m} \) is an unitary matrix and \( R \in \mathbb{R}^{m \times n} \) is upper triangular with non-negative diagonal values. If \( m > n \), matrix \( R \) has form

\[ R = \begin{bmatrix}
R_1 \\
0
\end{bmatrix}, \]

where \( R_1 \in \mathbb{R}^{n \times n} \) is upper triangular and \( 0 \in \mathbb{R}^{(m-n) \times n} \) is a zero matrix. In this case, also the matrix \( Q \) can be composed in two parts

\[ Q = [Q_1 \ Q_2], \quad Q_1 \in \mathbb{R}^{m \times n}, \quad Q_2 \in \mathbb{R}^{m \times (m-n)}, \]

where both \( Q_1 \) and \( Q_2 \) have orthogonal columns and

\[ A = Q_1 R_1. \]

Using QR decomposition makes the solving of overdetermined problems numerically easy: consider a overdetermined problem

\[ Ax = b. \]
By using the QR decomposition of matrix $A$, we get
$$R_1x = Q^T b,$$
where the right side $\bar{b} = Q^T b$ is easy to calculate. Hence, we are left with a upper triangular equation
$$R_1x = \bar{b},$$
which is easy to solve numerically using backsubstitution, in which case it is not even necessary to explicitly calculate the inverse of matrix $R_1$. It is also easy to see, that the solution is actually the least squares solution of the original overdetermined problem, i.e.
$$\bar{x} = R_1^{-1} \bar{b}.$$
In other words, the original stochastic linear inverse problem (2.3)–(2.4) is first transformed into overdetermined linear system (2.5) which can then be solved using QR decomposition. Moreover, the \textit{a posteriori} covariance matrix $\Sigma_x$ is given simply as
$$\Sigma_x = (R_1^T R_1)^{-1}.$$
Hence, it still remains necessary to calculate the inverse of the matrix $R_1$ if one is interested in the posterior covariance matrix. However, since $R_1$ is upper triangular, it is doable quite easily numerically.

Finally, note that sometimes the prior might be given as a full stochastic model
$$x_0 = Lx + \eta, \quad \eta \sim \mathcal{N}(0, \Sigma_0),$$
where $L$ is some invertible matrix\(^2\). This is equivalent to the model
$$L^{-1}x_0 = Ix + \bar{\eta}, \quad \bar{\eta} \sim \mathcal{N}(0, L^{-1} \Sigma_0 L^{-T}).$$
The equivalent overdetermined system is then
$$\begin{bmatrix} C^{-1}y \\ C_0^{-1} x_0 \end{bmatrix} = \begin{bmatrix} C^{-1}A \\ C_0^{-1}L \end{bmatrix}^x \quad (2.6)$$

#### 2.5 Givens rotations

In RLIPS, the QR decomposition is implemented by Givens rotations. Givens rotations are simple orthogonal plane rotations of form
$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix},$$

\(^2\)If $L$ is invertible, it defines so called improper prior.
where
\[ c = \frac{a}{\sqrt{a^2 + b^2}} \text{ and } s = \frac{b}{\sqrt{a^2 + b^2}}. \]

It is easy to see that the rotation matrix
\[
G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}
\]
is orthogonal, i.e. \( G^{-1} = G^T \). Note that when this transformation is applied to vector \((a \ b)^T\) it introduces a zero in the second element of the vector. This can be used to transform a matrix equation into a upper triangular form. Let us consider a matrix equation
\[ Ax = b, \]
where \( A \in \mathbb{R}^{m \times n} \) and \( b \in \mathbb{R}^m \). Define rotation matrix \( G(i, j) \) by
\[
G(i, j) = \begin{bmatrix} 1 & & & \cdots & & 1 \\ & \ddots & & & \vdots & & \vdots \\ & & 1 & & c & \cdots & s \\ & & \vdots & \ddots & \vdots & \vdots & \vdots \\ & & -s & \cdots & c & & 1 \\ & & & & \ddots & & 1 \end{bmatrix},
\]
where the diagonal consists of 1’s except
\[
G(i, j)_{i,i} = G(i, j)_{j,j} = c = \frac{A_{i,i}}{\sqrt{A_{i,i}^2 + A_{j,j}^2}}
\]
and the off-diagonals are zeroes, except
\[
G(i, j)_{i,j} = s \quad \text{and} \quad G(i, j)_{j,i} = -s, \quad s = \frac{A_{i,j}}{\sqrt{A_{i,i}^2 + A_{j,j}^2}}.
\]
This is an orthogonal matrix that rotates the matrix \( A \) such that the matrix \( G(i, j)A \) will have a zero in element \((j, i)\). Note that only rows \( i \) and \( j \) of matrix \( A \) are affected in this transformation and all other rows will remain intact. It is easy to find a way to make consecutive transformations so that eventually the transformed
matrix would be upper triangular. For example, RLIPS makes the rotations in the following way: starting from the equation

\[ Ax = b, \]

first transformation is made using matrix \( G(1, 2) \), i.e. we get

\[ A_1x = y_1, \]

where \( A_1 = G(1, 2)A \) and \( b_1 = G(1, 2)b \). Now the element \((2, 1)\) of matrix \( A_1 \) will be zero. The transformation has only altered the first and second rows of the original matrix \( A \) and the vector \( b \). Next we apply transformation \( G(1, 3) \) to get

\[ A_2x = b_2, \]

where \( A_2 = G(1, 3)A_1 \) and \( b_2 = G(1, 3)b_1 \). Now the matrix \( A_2 \) will have two rotated zeros, in elements \((2, 1)\) and \((3, 1)\). The zero in \((2, 1)\) remains because the transformation \( G(3, 1) \) alters only the first and third rows of the matrix \( A_1 \) and the vector \( b_1 \). Next, we can rotate the second and third rows of \( A_2 \) by matrix \( G(2, 3) \) to get

\[ A_3x = b_3, \]

where \( A_3 = G(2, 3)A_2 \) and \( b_3 = G(2, 3)b_2 \). Note that the zeros in \((2, 1)\) and \((3, 1)\) of matrix \( A_2 \) are preserved in \( A_3 \) and there is a new rotated zero at \((3, 2)\) of matrix \( A_3 \). Continuing these rotations row-by-row we are eventually left with equation

\[ Rx = y, \]

where \( R \) is upper triangular. In Figure 2.1 is shown how rotations could be made row-by-row for a \( 6 \times 4 \) matrix.

RLIPS, however, makes the rotations in parallel using OpenCL and the GPU’s. This boosts the performance considerably. The rotation scheme for a \( 6 \times 4 \) matrix performed by RLIPS is presented in Figure 2.2. The rotations marked with the same number are done in parallel.

The parallelisation of the rotations is made in OpenCL. In every rotation stage as many OpenCL workgroups as there are parallel rotations to be made are launched, and the separate rotations are made in their own workgroups. For example, in Figure 2.2, at stage 5, three workgroups would perform the three rotations \( G(1, 6) \), \( G(2, 4) \) and \( G(3, 4) \) in parallel. Note that there is no need to actually construct the full rotation matrices \( G(\cdot, \cdot) \). Only \( 2 \times 2 \) rotation matrices are needed as the individual rotations only affect the rows that are being rotated.
2.6 Rlips algorithm

RLIPS solves the linear stochastical inverse problems in the following way:

First, the problem is initialized. User gives RLIPS the number of unknowns (number of columns in the theory matrix), the number of columns in the measurement matrix/vector and the type (real/complex) of the problem. Optionally user can also set the size of so called rotation buffer and the size of OpenCL workgroup. RLIPSA then allocates the necessary arrays for the data and sets several internal parameters.

After the initialization data can be added to the problem. Data consists of theory/prior matrix rows, measurements/prior values and the error/prior variances/covariances. Note that all the data is handled row-by-row, so it can be given to RLIPS by one or several rows at the time. The exception is the case where the error/prior covariance is given in full matrix form. Then also the corresponding theory matrix and all the measurements must be given at once.

When data rows are fed into RLIPS, they are first “whitened”, meaning that the theory matrix rows, measurements and the error term are multiplied by the inverse of the Cholesky factor of the covariance matrix, i.e. if the original equation was

\[ y = Ax + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \sigma), \quad \Sigma = CC^T, \]

we now have

\[ C^{-1}y = C^{-1}Ax + C^{-1}\varepsilon. \]
This is called whitening, because now the transformed noise term is actually white noise: the covariance of the term $C^{-1}\varepsilon$ is

\[
\langle C^{-1}\varepsilon \cdot (C^{-1}\varepsilon)^T \rangle = C^{-1}\langle \varepsilon \cdot \varepsilon^T \rangle C^{-T} = C^{-1}\Sigma C^{-T} = C^{-1}CC^TC^{-T} = I,
\]

i.e.

\[C^{-1}\varepsilon \sim \mathcal{N}(0,I).\]

Note that if the ovarian matrix $\Sigma$ is diagonal, the above transformation is simply multiplication by the reciprocal of the square root of the diagonal element.

After the data is whitened, it is moved into the oration buffer. When the buffer becomes full, or the rotations are forced by the user, the buffer is moved from R memory into GPU memory, and the Givens rotations are made. The rotated data rows are stored in GPU memory and the buffer rows are discarded. Only the rotated rows are stored which can save huge amounts of memory if the problem is overdetermined and large.

More measurement or prior data can then be added. It is first stored in the buffer and rotated in GPU when necessary. When all the data is fed in, it will be in upper triangular form. When user wants to calculate the solution the rotated data is first copied from the GPU memory into R memory. Then the least squares solution is calculated using a backsubstitution algorithm. If user wants also calculate the \textit{a posteriori} covariance matrix, the upper triangular matrix $R$ is first inverted and the posterior covariance $\Sigma_p$ is calculated by $\Sigma_p = R^{-1}R^{-T}$. It is also possible to calculate only the \textit{a posteriori} variances (i.e. the diagonal elements of the \textit{a posteriori} covariance matrix). It is considerably faster than calculating the full \textit{a posteriori} covariance matrix.

Note that the rotated data remains still in the GPU memory, so it is possible to add measurements or prior data to the old problem even if the solution is already calculated. Also, RLIPS contains a command to copy an existing problem data into a new one, so the user can for example test different prior models without having to start all over every time.
Chapter 3

Implementation

3.1 R files

3.1.1 rlips environment

When a new inverse problem is initialised using the command `rlips.init`, it returns a R environment that contains all necessary variables and data arrays for RLIPS to operate. The environment contains the following objects:

- **ref** A 2-vector containing the 64-bit address of the C struct associated with the current problem
- **ncols** Number of columns in the theory matrix \( A \)
- **nrhs** Number of columns in the measurement matrix \( m \)
- **nbuf** Number of road in the data buffer matrix
- **wg.size** OpenCL workgroup size
- **type** Problem type; ’s’ for real valued problem, ’c’ for complex valued problem
- **brows** Total number of data rows fed into this problem
- **brows** Total number of data rows *currently* in the data buffer
- **rows** Total number of rows *currently* in the target matrix \( R \)
- **buffer.cols** Number of columns in the buffer matrix
- **buffer** The array for data buffer
active Flag (TRUE/FALSE) for active rlips environment

After the problem is solved (at least once), the environment contains also the objects:

**solution** Matrix containing the solution

**covariance** Posteriori covariance matrix or the posteriori variances (if calculated)

**R.mat** The upper triangular target matrix $R$

**Y.mat** The target vector $Y$

### 3.1.2 file: rlips.init.R

R function `rlips.init` is used to initialise a new linear inverse problem. It returns the R environment holding all data for the problem.

```r
## Initialize RLIPS object
## Arguments:
## ncols number of unknowns/columns in the theory matrix
## nrhs number of measurements/columns in the measurement vector/matrix
## type problem type; 's' for single precision real,
## 'c' for single precision complex
## nbuf data buffer rows
## workgroup.size OpenCL workgroup size, best choice depends on the used GPU
##
## Returns:
## e Initialized RLIPS environment

rlips.init <- function(ncols, nrhs, type='s',
                       nbuf=ncols, workgroup.size=128)
{
    # Create new environment for the problem data
    e <- new.env()
    # Set problem parameters.
    # e$ref holds an integer vector of length 2 used
    # to form the C pointer for this

    # Set problem parameters.
    # e$ref holds an integer vector of length 2 used
    # to form the C pointer for this
}
```

# particular problem instance.
ref <- c(0,0)
ncols <- ncols
nrhs <- nrhs
nbuf <- nbuf

# Check workgroup size. Usually, it should be
# a multiple of 16. The optimal
# size depends on the used GPU.
if (workgroup.size%%16 != 0)
{
  warning("Workgroup size REALLY should be multiple of 16! 
")
}
wg.size <- workgroup.size
type <- type

# Check type parameter. At this point, only single precision
# real and complex problems are implemented. Also, only
# high-end GPU's are capable to use double precision.
if (type != 's' && type != 'c')
{
  stop('Only single precision real and complex types 
implemented! Exiting!')
}

# Initialize some problem parameters
nrows <- 0 # Number of total rows fed into system
brows <- 0 # Number of rows in the buffers
rrows <- 0 # Number of rows in R matrix

# Number of columns in the buffer matrix.
# Holds both data and measurements and is a multiple of 
# workgroup size.
buffer.cols <- floor((ncols + nrhs + 32 - 1)/32) * 32

# At this point, only single precision real is implemented.
# Call OpenCL initialization routine using .Call
# See file rlips.c
if (type == 's')
{
  ref = .Call("sInitRlips",
    ncols,
    nrhs,
    ...
else if (type == 'c') {
   e$ref = .Call("cInitRlips",
                ncols,
                nrhs,
                e$nbbuf,
                e$wg.size,
                PACKAGE="rlips")
}
else {
   stop('rlips.init: type not recognized!')
}

# Flag current RLIPS environment as active.
# We need this to ensure that we do not reallocate
# allocated objects or deallocate already deallocated
# objects. Doing so would cause R to crash.
e$active <- TRUE

# Return the RLIPS environment
return(e)
3.1.3 file: rlips.add.R

R function rlips.add is used to add new data (theory matrix, measurements, error covariances/variances) to the inverse problem.

```r
rlips.add <- function(e, A.data, M.data, E.data = 1) {
  # Make sure that we are using an active environment
  if (!e$active)
    stop("Not an active rlips environment! Nothing done!")
  
  # Check that the arguments are in right form
  # Is error given as a matrix
  if (is.matrix(E.data))
    Emat = TRUE
  else
    Emat = FALSE
  
  # Check given theory matrix data
  if (is.vector(A.data))
    # If given as a vector check that its length is
    # a multiple of e$ncols
```

if (length(A.data)/e$ncols != as.integer(length(A.data)/e$ncols))
    stop('rlips.add: theory matrix has wrong size!')
else
    # Calculate the number of matrix rows fed in
    num.rows <- length(A.data)/e$ncols

    # Reshape vector into a matrix
    A.data <- matrix(A.data,ncol=e$ncols,nrow=num.rows,byrow =T)
else
    # Theory matrix data given as a matrix. Check
    # number of columns.
    if (ncol(A.data) != e$ncols)
        stop('rlips.add: theory matrix has wrong number of columns!')
    else
        num.rows <- nrow(A.data)

    # Check given measurement matrix
    if (is.vector(M.data))
        # If given as a vector, check that its length
        # is right (given the rows in A.data)
        if (length(M.data) != e$nrhs * num.rows)
            stop('rlips.add: measurement vector has wrong size!'
        )
    else
        # reshape as matrix
        if (e$nrhs>1)
            M.data <- matrix(M.data,}
M.data,
ncol=e$nrhs,
nrows=num.rows,
byrow=T)

else
{
  dim(M.data) <- c(num.rows,1)
}
}

else # If given as a matrix, check its dimensions
{
  if (!all(c(num.rows,e$nrhs)==dim(M.data)))
  {
    stop('rlips.add: measurement matrix has wrong shape!')
  }
}

# Check given error data
if (Emat)
  # If errors are given as a covariance matrix,
  # check its dimensions.
  # NB: Positive definitiveness or symmetricity
  # is not checked!
  {
    if (!all(c(num.rows,num.rows)==dim(E.data)))
    {
      stop('rlips.add: error covariance matrix has wrong size!')
    }
  }

  # Multiply theory matrix and measurement data
  # with the inverse of the Cholesky factor of
  # the error covariance matrix
  # (Whitening of the noise)
  C <- chol(E.data)
  A.data <- backsolve(C,A.data,transpose=T)
  M.data <- backsolve(C,M.data,transpose=T)

else # Errors given as a vector or a scalar value
{
  if (length(E.data)==1)
    # If error variance is given as a single number,
# expand it to a vector.

E.data <- rep(E.data, num.rows)

else
  # if given as a vector, check its length.

  if (length(E.data) != num.rows)
    stop('rlips.add: error vector has wrong length!

  }

# Divide both theory matrix rows and measurements
# by the square root of the inverse of the variance
# (whitening the noise)

if(num.rows>1){
  A.data <- apply(A.data,
    FUN= function(x,y){return(x/y)},
    y=sqrt(E.data),
    MARGIN=2)

  M.data <- apply(M.data,
    FUN= function(x,y){return(x/y)},
    y=sqrt(E.data),
    MARGIN=2)
}
else{
  A.data <- A.data / sqrt(E.data)
  M.data <- M.data / sqrt(E.data)
}

# Move data to buffer row by row
for (i in seq(num.rows))
  {
    # First ncols columns of the buffer contains
    # the theory matrix and the next n rhs columns
    # the measurement data
    e$buffer[e$brows+1:e$ncols] <- A.data[i,]
    e$buffer[e$brows+1,(e$ncols+1):(e$ncols+e$nrhs)] <- M.data[i,]

    e$brows <- e$brows + 1

    # If buffer becomes full, rotate it
    if (e$brows >= e$buf)
      { rlips.rotate(e) }
185  }
186  )
187
188
189  )
190
3.1.4 file: rlips.rotate.R

R function `rlips.rotate` performs the Givens rotations to the data stored in
the buffers. Rotations are usually made automatically but with this command the
end user has the possibility to rotate the data at will.

```r
## file:rlips.rotate.R
## (c) 2011- University of Oulu, Finland
## Written by Mikko Orispaa <mikko.orispaa@oulu.fi>
## Licensed under FreeBSD license. See file LICENSE for details.

## Do the rotations

## Arguments:
## e Active RLIPS environment
rlips.rotate <- function(e)
{
  # Make sure that we are using an active environment
  if (!e$active)
    stop("Not an active rlips environment! Nothing done!")

  # Rotate only, if there is something to rotate
  if (e$brows > 0)
    {
      # Depending on e$type, use the right C routine
      # <s|c>RotateRlips
      # Notice that the buffer is transposed, because
      # R stores matrices in column-major order and
      # C in row-major order.
      if (e$type == 's')
        {
          .Call("sRotateRlips",
           e$ref,
           t(e$buffer[1:e$brows,]),
           e$brows,
           PACKAGE="rlips")
        }
      else if (e$type == 'c')
        {
          .Call("cRotateRlips",
           e$ref,
           # Complex data is separated into real and
           # imaginary parts
           Re(t(e$buffer[1:e$brows,]),),
           Im(t(e$buffer[1:e$brows,])),
        }
    }
```
```r

e$brows,

PACKAGE="rlips")

# After rotations, update internal variables and
# empty the buffer matrix.
e$buffer <- matrix(0,e$nbuf,e$buffer.cols)
e$nrows <- e$nrows + e$brows
e$rrows <- min(e$ncols,e$nrows)
e$brows <- 0
```

3.1.5  file: rlips.solve.R

R function rlips.solve solves the current inverse problem. Oplionally, it can also calculate the full \textit{a posteriori} covariance matrix or only its diagonal elements, i.e. \textit{a posteriori} variances.

```r
rlips.solve <- function(e, calculate.covariance=FALSE, full.covariance=FALSE) {
  # Make sure that we are using an active environment
  if (!e$active) {
    stop("Not an active rlips environment! Nothing done!")
  }
  # If buffer has non-rotated rows, rotate them now
  if (e$brows > 1) rlips.rotate(e)
  # Get the problem data from the GPU
  # i.e. the target matrix R and the target vector Y
  rlips.get.data(e)
  # Use backsubstitution algorithm to get the solution
  # For complex backsubstitution we use our own routine
  # since it is missing in R
  if (e$type == 's') {
    e$solution <- backsolve(e$R.mat, e$Y.mat)
  }
}
```

else if (e$type == 'c')
{
    e$solution <- .Call("cbacksolve",
                        e$R.mat,
                        e$Y.mat,
                        PACKAGE="rlips")
}

# A posteriori covariance calculation
if (calculate.covariance)
{
    # Calculate full covariance matrix
    if (full.covariance)
    {
        # Real problem
        if (e$type == 's')
        {
            # Calculate the inverse of target matrix R
            e$covariance <- backsolve(
                              e$R.mat,
                              diag(rep(1,e$ncols)))
            # Covariance = (R\(^{-1}\)) \* (R\(^{-1}\))^T
            e$covariance <- e$covariance %*%
                            t(e$covariance)
        }
        # Complex problem
        else if (e$type == 'c')
        {
            # Here we use solve instead of complex backsolve
            # because it is not optimised for such a large
            # measurement vectors
            e$covariance <- solve(
                              e$R.mat,
                              diag(rep(1,e$ncols)))
            # Covariance = (R\(^{-1}\)) \* (R\(^{-1}\))^H
            e$covariance <- e$covariance %*%
                            Conj(t(e$covariance))
        }
    }
    # Calculate only the diagonal of the covariance matrix
    else
    {
        # Real problem
        if (e$type == 's')
        {
            e$iR.mat <- backsolve(e$R.mat,
                                  diag(rep(1,e$ncols)))
            e$covariance <- rep(0,e$ncols)
        }
    }
}
# Calculate only diagonal terms of the covariance matrix for (i in 1:e$ncols)
{
  e$covariance[i] <-
  sum(e$iR.mat[i,i:e$ncols]**2)
}
}

else if (e$type == 'c')
{
e$iR.mat <- solve(e$R.mat, diag(rep(1,e$ncols)))
e$covariance <- rep(0,e$ncols)
# Calculate only diagonal terms of the covariance matrix for (i in 1:e$ncols)
{
  e$covariance[i] <-
  sum(e$iR.mat[i,i:e$ncols] * Conj(e$iR.mat[i,i:e$ncols]))
}
}
### 3.1.6 file: rlips.dispose.R

R function `rlips.dispose` disposes the current inverse problem and deallocates all allocated arrays from the CPU and GPU memory.

```r
## dispose RLIPS object
## Arguments:
## e  rlips environment
rlips.dispose <- function(e)
{
  # Check that e actually is an active rlips environment
  if (e$active)
  {
    # Depending on the type of the environment call
    # the proper C routine which deallocates memory
    # and tidies things up.
    if (e$type == 's')
    {
      .Call("sKillRlips",
             e$ref,PACKAGE="rlips")
    }
    else if (e$type == 'c')
    {
      .Call("cKillRlips",
             e$ref,PACKAGE="rlips")
    }
    else
    {
      stop('rlips.dispose: type not recognized! Nothing done!')
    }

    # Set the rlips environment e inactive.
    e$active <- FALSE
  }
  else
  {
    stop('rlips.dispose: Not an active RLIPS environment! Nothing done!')
  }
}
```

---

30
3.1.7 file: rlips.get.data.R

R function `rlips.get.data` is an auxiliary routine for fetching the current theory matrix and theory vector from the GPU to the CPU memory.

```r
## file:rlips.get.data.R
## (c) 2011- University of Oulu, Finland
## Written by Mikko Orispaa <mikko.orispaa@oulu.fi>
## Licensed under FreeBSD license. See file LICENSE for details.

## Fetch problem data from GPU
## Arguments:
## e rlips environment
rlips.get.data <- function(e)
{
  if (e$brows > 0) rlips.rotate(e)

  # According to the problem type, call the appropriate
  # C routine which fetches the data from GPU to
  # CPU memory
  if (e$type == 's')
  {
    data <- .Call("sGetDataRlips",
                    e$ref,
                    PACKAGE="rlips")

    # Reshape the received vector as a matrix
    data.mat <- matrix(data, e$ncols, e$buffer.cols, byrow=TRUE)
  }
  else if (e$type == 'c')
  {
    data <- .Call("cGetDataRlips",
                   e$ref,
                   PACKAGE="rlips")

    # Transform the received real vector into a complex
    # vector
    data <- data[seq(1, 2 * e$ncols * e$buffer.cols, by=2)] +
             1i * data[seq(2, 2 * e$ncols * e$buffer.cols, by=2)]

    # Reshape as a matrix
    data.mat <- matrix(data, e$ncols, e$buffer.cols, byrow=TRUE)
  }

  # Get target matrix and set its dimensions
  e$R.mat <- data.mat[,1:e$ncols]
  dim(e$R.mat) <- c(e$ncols, e$ncols)
}
```

# Get target vector and set its dimensions
Y.mat <- data.mat[, (e$ncols+1):(e$ncols+e$nrhs)]

dim(e$Y.mat) <- c(e$ncols, e$nrhs)

##


### 3.1.8 file: rlips.test.R

R function `rlips.test` is a simple test program for checking RLIPS performance and accuracy.

```r
# Construct a random theory matrix and solution vector
ncols <- size[2]
rows <- size[1]
A <- matrix(rnorm(ncols*rows),rows,ncols)
sol <- rnorm(ncols)
if (type == 'c')
  A <- A + 1i*matrix(rnorm(ncols*rows),rows,ncols)
sol <- sol + 1i*rnorm(ncols)
```

## Arguments:
- **type** Type of the inverse problem
  - 's' for single precision real
  - 'c' for single precision complex
- **size** 2-vector containing the size of the theory matrix
- **buffersize** number of buffer rows used
- **loop** number of loops performed. for small problems solving the problem several times and averaging the results will give more accurate results. Default = 1
- **wg.size** OpenCL work group size. Should be a multiple of 16. The maximum size depends on the used GPU
- **return.data** If TRUE, the target matrix and vector are returned
- **averaging.fun** Used averaging function. Default is 'mean'. If loop = 1, this has no effect.

```r
rlips.test <- function(type,size,buffersize = size[2],loop=1,
                       wg.size=128,return.data=FALSE,
                       averaging.fun=mean)
  {
    # Construct a random theory matrix and solution vector
    ncols <- size[2]
    rows <- size[1]
    A <- matrix(rnorm(ncols*rows),rows,ncols)
sol <- rnorm(ncols)
    if (type == 'c')
      A <- A + 1i*matrix(rnorm(ncols*rows),rows,ncols)
sol <- sol + 1i*rnorm(ncols)
  }
```
# NB: No added noise

m <- A %*% sol

# Set up arrays for results
acc <- rep(0, loop)
times <- rep(0, loop)

# Calculate the approximate number of floating point operations (flops) needed to solve the problem
flops <- (2 * ncols^3 + 3 * ncols^2 - 5 * ncols + 6 * (rows - ncols) * ncols + 3 * (rows - ncols) * ncols * (ncols + 1))

# Solve the problem 'loop' times
for(k in 1:loop) {
  # Call rlips.problem, which solves the inverse problem and returns the solution and elapsed time
  ss <- rlips.problem(type, A, m, buffersize, wg.size, return.data)

  # Store the results
  times[k] <- ss$time[3]
  acc[k] <- max(abs(sol - ss$sol))
}

# Get the averaged results
a.time <- averaging.fun(times)
a.acc <- averaging.fun(acc)

# Calculate the approximate gigaflops per second (GFlop/s)
Gflops <- flops/1.0E9 / a.time

# If return.data = TRUE, return a list containing the results and the target matrix and vector
if (return.data) {
  return(list(times=a.time, accuracy=a.acc, Gflops=Gflops, R=ss$R, Y=ss$Y, sol=sol))
} else {
  # return only the timing and accuracy results
  return(list(times=a.time, accuracy=a.acc, Gflops=Gflops))
}

# 34
# Simple rlips problem solving script used by rlips.test

```r
rlips.problem <- function(type, A, m, bsize, wg.size, return.data) {
  ncols <- ncol(A)
  h <- rlips.init(ncols, 1, type, bsize, wg.size)
  tt <- proc.time()
  rlips.add(h, A, m, 1)
  rlips.solve(h)
  tt2 <- proc.time()
  aa <- h$solution
  Rmat <- 0
  Ymat <- 0
  if (return.data) {
    rlips.get.data(h)
    Rmat <- h$R.mat
    Ymat <- h$Y.mat
  }
  rlips.dispose(h)
  return(list(sol = aa, time = tt2 - tt, R = Rmat, Y = Ymat))
}
```

3.1.9 file: rlips.copy.R

R function rlips.copy creates a new RLIPS environment and copies all data from an old RLIPS environment into the new one.

```r
rlips.copy <- function(oenv) {
  nenv <- rlips.init(oenv$ncols, oenv$nrhs, oenv$type, 
                    oenv$nbuf, oenv$wg.size)
  rlips.rotate(oenv)
  rlips.get.data(oenv)
  rlips.add(nenv, oenv$R.mat, oenv$Y.mat)
  return(nenv)
}
```

## file: rlips.add.R

(c) 2011- University of Oulu, Finland

Written by Mikko Orispaa <mikko.orispaa@oulu.fi>

Licensed under FreeBSD license. See file LICENSE for details.

Copy RLIPS environment

Arguments:

- `oenv`: rlips environment to be copied

Returns:

- `nenv`: new rlips environment that is identical with `oenv`

This is a quick hack! A better one should be written ASAP!
3.2 C/OpenCL files

3.2.1 Header file: rlips.h

Header file rlips.h contains flips C function prototypes and flips structured data types.

```c
// file: rlips.h

// RLIPS C data types, structs, function prototypes

// (c) 2011- University of Oulu, Finland
// Written by Mikko Orispaa <mikko.orispaa@oulu.fi>
// Licensed under FreeBSD license. See file LICENSE for details.

#ifndef __RLIPS_H
#define __RLIPS_H

#include <R.h>
#include <Rinternals.h>

#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif

// *************************************
// *** Rlips structures ***
// *************************************

// For splitting 64-bit addresses into two int’s
typedef union _split_t
{
    int II[2];
    long longValue;
} addr;

// RLIPS data structures

// Single precision real
typedef struct _sRlips
{
    // User given parameters (in rlips.init)
    int numCols; // Number of columns/unknowns
    int numRHS; // Number of right hand sides,
```
```

// i.e. number of columns in the measurement
int sizeBuffer; // Number of rows in the rotation buffer
int sizeWorkgroup; // OpenCL workgroup size
// (should be of form 2^n)

// Internal parameters
float zThreshold; // Threshold value for zero.
int numTotRows; // Total number of data rows
    // fed into problem
int numRmatRows; // Number of rows currently
    // in the target matrix
int numBufferRows; // Number of data rows currently
    // in the rotation buffer
long flops; // Number of floating point
    // operations made (NOT USED)

// Constant parameters depending on user given parameters
int numRmatCols; // number of columns in the R matrix
int sizeRmat; // Total size (cols x rows) of R matrix
int sizeBufferMat; // Total size of the buffer matrix

// Device buffers
cl_mem dRmat; // OpenCL buffer for R matrix
c_mem dBufferMat; // OpenCL buffer for buffer matrix

// OpenCL structures
cl_platform_id* platform_id;
c_device_id* device_id;
c_context* context;
c_command_queue* commandqueue;
c_program* kernel_program;
cl_kernel* fullRotKernel;
c_kernel* partRotKernel;
}

// Single precision complex
typedef struct cRlips
{
    // User given parameters (in rrlips.init)
    int numCols; // Number of columns/unknowns
    int numRHS; // Number of right hand sides,
    // i.e. number of columns in the measurement
    int sizeBuffer; // Number of rows in the rotation buffer
    int sizeWorkgroup; // OpenCL workgroup size
    // (should be of form 2^n)
// Internal parameters
float zThreshold; // Threshold value for zero.
int numTotRows; // Number of data rows fed into problem
int numRmatRows; // Number of rows currently in
    // the target matrix
int numBufferRows; // Number of data rows currently in
    // the rotation buffer
long flops; // Number of floating point operations made
    // (NOT USED)

// Constant parameters depending on user given parameters
int numRmatCols; // number of columns in the R matrix
int sizeRmat; // Total size (cols x rows) of R matrix
int sizeBufferMat; // Total size of the buffer matrix

// Device buffers
cl_mem dRmat_r; // OpenCL buffer for R matrix (real part)
cl_mem dRmat_i; // OpenCL buffer for R matrix (imag part)
cl_mem dBufferMat_r; // OpenCL buffer for buffer matrix
    // (real part)
cl_mem dBufferMat_i; // OpenCL buffer for buffer matrix
    // (imag part)

// OpenCL structures
cl_platform_id* platform_id;
cl_device_id* device_id;
cl_context* context;
cl_command_queue* commandqueue;
cl_program* kernel_program;
cl_kernel* fullRotKernel;
cl_kernel* partRotKernel;
} cRlips;

// ***********************************************
// *** Rlips function prototypes ***
// ***********************************************
SEXP sInitRlips( SEXP, SEXP, SEXP, SEXP);
SEXP sKillRlips(SEXP);
SEXP sRotateRlips(SEXP, SEXP, SEXP);
SEXP sGetDataRlips(SEXP);
SEXP cInitRlips(SEXP, SEXP, SEXP, SEXP);
SEXP cKillRlips(SEXP);
SEXP cRotateRlips(SEXP, SEXP, SEXP, SEXP);
SEXP cGetDataRlips(SEXP);
SEXP cbacksolve(SEXP, SEXP);

#endif
3.2.2 File: rlips.c

File rlips.c contains the main C routines of RLIPS:

- `<r|c>InitRlips` Initialize a new linear problem
- `<r|c>KillRlips` Dispose and deallocate problem
- `<r|c>RotateRlips` Perform the Givens rotations for the data in the buffer
- `<r|c>GetDataRlips` Fetch the rotated data from GPU to R

The first letter is either `r` or `c` meaning real and complex problem, respectively.

```c
// index of a row-major stored upper triangular matrix
```

// A row
// B column
// C number of columns
#define ridx(A,B,C) ((B)+(A)*(2*(C)-(A)-1)/2)

// Index of a row-major stored matrix
// A row
// B column
// C number of columns
#define yidx(A,B,C) ((A)*(C)+(B))

#define min(A,B) (((A)<=(B))?(A):(B))
#define max(A,B) (((A)>(B))?(A):(B))

/*
Routines for single precision real problems
*/

/*
Initialize a new RLIPS problem and structure
Arguments:
NCOLS Number of columns in the theory matrix, i.e.
number of unknowns
NRHS Number of columns in the measurement matrix
NBUF Rotation buffer size, i.e. number of data rows
stored into a buffer matrix
BLOCKSIZE OpenCL work group size, should currently be
a multiple of 16. The optimal value depends
on the GPU used.

Returns a integer vector of length 2.
The vector contains the (64bit) address of
the initialized structure.
*/
SEXP sInitRlips(SEXP NCOLS, SEXP NRHS, SEXP NBUF,
    SEXP BLOCKSIZE)
{
    // Definitions required by R’s rCall functionality
    SEXP ref;
    PROTECT(ref = allocVector(INTSXP,2));
NCOLS = coerceVector(NCOLS, INTSXP);
NRHS = coerceVector(NRHS, INTSXP);
NBUF = coerceVector(NBUF, INTSXP);
BLOCKSIZE = coerceVector(BLOCKSIZE, INTSXP);
int ncols = INTEGER(NCOLS)[0];
int nrhs = INTEGER(NRHS)[0];
int nbuf = INTEGER(NBUF)[0];
int blocksize = INTEGER(BLOCKSIZE)[0];

// Allocate new sRlips struct K
sRlips * restrict K = (sRlips *) malloc(sizeof(sRlips));

// Allocate OpenCL structures in K
K->platform_id = malloc(sizeof(cl_platform_id));
K->device_id = malloc(sizeof(cl_device_id));
K->context = malloc(sizeof(cl_context));
K->commandqueue = malloc(sizeof(cl_command_queue));
K->kernel_program = malloc(sizeof(cl_program));
K->fullRotKernel = malloc(sizeof(cl_kernel));
K->partRotKernel = malloc(sizeof(cl_kernel));

// Set the user provided parameters
K->numCols = ncols; // number of unknowns
K->numRHS = nrhs; // number of measurements
K->sizeBuffer = nbuf; // rotation buffer rows
K->sizeWorkgroup = blocksize; // OpenCL work group size

// Column size of OpenCL buffers (smallest multiple of
// workgroup size that contains both theory matrix
// columns and measurements. Also, we want it to be
// a multiple of 32)
K->numRmatCols = (ncols + nrhs + 32 - 1) / 32 * 32;

// Numbers which absolute value is smaller than zThreshold
// are considered as zeroes. Hard coded at the moment.
// Could/should be a user provided parameter in the future.
K->zThreshold = 1.0E-8f;

// Set initial values for status parameters
K->numTotRows = 0; // Total number of data rows fed
// into the problem
K->numBufferRows = 0; // Number of rows currently in
// the buffer
K->numRmatRows = 0; // Number of rows currently in
// the target matrix
K->flops = 0L; // Number of floating point
// operations used. This

// Allocate new sRlips struct K
sRlips * restrict K = (sRlips *) malloc(sizeof(sRlips));

// Allocate OpenCL structures in K
K->platform_id = malloc(sizeof(cl_platform_id));
K->device_id = malloc(sizeof(cl_device_id));
K->context = malloc(sizeof(cl_context));
K->commandqueue = malloc(sizeof(cl_command_queue));
K->kernel_program = malloc(sizeof(cl_program));
K->fullRotKernel = malloc(sizeof(cl_kernel));
K->partRotKernel = malloc(sizeof(cl_kernel));

// Set the user provided parameters
K->numCols = ncols; // number of unknowns
K->numRHS = nrhs; // number of measurements
K->sizeBuffer = nbuf; // rotation buffer rows
K->sizeWorkgroup = blocksize; // OpenCL work group size

// Column size of OpenCL buffers (smallest multiple of
// workgroup size that contains both theory matrix
// columns and measurements. Also, we want it to be
// a multiple of 32)
K->numRmatCols = (ncols + nrhs + 32 - 1) / 32 * 32;

// Numbers which absolute value is smaller than zThreshold
// are considered as zeroes. Hard coded at the moment.
// Could/should be a user provided parameter in the future.
K->zThreshold = 1.0E-8f;

// Set initial values for status parameters
K->numTotRows = 0; // Total number of data rows fed
// into the problem
K->numBufferRows = 0; // Number of rows currently in
// the buffer
K->numRmatRows = 0; // Number of rows currently in
// the target matrix
K->flops = 0L; // Number of floating point
// operations used. This
// Set OpenCL buffersizes
K->sizeRmat = K->numRmatCols * K->numCols;
K->sizeBufferMat = K->numRmatCols * K->sizeBuffer;

// ********************************************
// Initialize OpenCL
// ********************************************
cl_int error;

// Get OpenCL platform
error = clGetPlatformIDs(1,K->platform_id,NULL);
if (error != CL_SUCCESS)
{
    printf("Did not get OpenCL platform! Error code %d. Exiting sInitRlips.\n",
            error);
    exit(1);
}

// Ask for one GPU
error = clGetDeviceIDs(*K->platform_id,CL_DEVICE_TYPE_GPU,1,
                        K->device_id,NULL);
if (error != CL_SUCCESS)
{
    printf("Did not get OpenCL device! Error code %d. Exiting sInitRlips.\n",
            error);
    exit(1);
}

// Create OpenCL context
*K->context = clCreateContext(0,1,K->device_id,NULL,NULL,&
error);
if (error != CL_SUCCESS)
{
    printf("Did not create OpenCL context! Error code %d. Exiting sInitRlips.\n",
            error);
    if (*K->context) clReleaseContext(*K->context);
    exit(1);
}

// Create OpenCL command queue
*K->commandqueue = clCreateCommandQueue(*K->context,*K->
device_id,0,&error);
if (error != CL_SUCCESS)
{
    printf("Did not create OpenCL command queue! Error code
%ld. Exiting sInitRlips.\n",
    error);
    if (*K->commandqueue) clReleaseCommandQueue(*K->
    commandqueue);
    if (*K->context) clReleaseContext(*K->context);
    exit(1);
}

// Create kernel program (KernelSource in kernelsources.h)
*K -> kernel_program =
    clCreateProgramWithSource(  
    *K->context,  
    1,  
    (const char **)&sKernelSource,  
    NULL,  
    &error);
if (!*K->kernel_program)
{
    printf("Could not create compute program! Exiting
sInitRlips.\n");
    if (*K->commandqueue)
        clReleaseCommandQueue(*K->commandqueue);
    if (*K->context)
        clReleaseContext(*K->context);
    exit(1);
}

// Build kernel executable
error = clBuildProgram(*K->kernel_program,  
    0,NULL,"-w",NULL,NULL);
if (error != CL_SUCCESS)
    // If there were errors,
    // print info
    {
        printf("Error code: %ld\n",error);
        size_t len;
        char buffer[2048];
        printf("Failed to build program executable! Exiting
sInitRlips.\n");
        clGetProgramBuildInfo(*K->kernel_program,
            *K->device_id,
printf("%s\n",buffer);

if (*K→kernel_program)
    clReleaseProgram(*K→kernel_program);
if (*K→commandqueue)
    clReleaseCommandQueue(*K→commandqueue);
if (*K→context)
    clReleaseContext(*K→context);
exit(1);

// Create the kernel functions
*K→fullRotKernel =
    clCreateKernel(*K→kernel_program,
                  "s_full_rotations",&error);
cl_int error2;
*K→partRotKernel =
    clCreateKernel(*K→kernel_program,
                  "s_partial_rotations",&error2);
if (error != CL_SUCCESS || error2 != CL_SUCCESS)
    printf("Could not create kernel! Error codes: %d, %d. Exiting sInitRlips.\n",
           error, error2);
    if (*K→fullRotKernel)
        clReleaseKernel(*K→fullRotKernel);
    if (*K→partRotKernel)
        clReleaseKernel(*K→partRotKernel);
    if (*K→kernel_program)
        clReleaseProgram(*K→kernel_program);
    if (*K→commandqueue)
        clReleaseCommandQueue(*K→commandqueue);
    if (*K→context)
        clReleaseContext(*K→context);
    exit(1);

// Create OpenCL buffer for target matrix
K→dRmat = clCreateBuffer(*K→context,CL_MEM_READ_WRITE,
                          sizeof(float) * K→sizeRmat,
                          NULL,&error);
if (error != CL_SUCCESS || error2 != CL_SUCCESS)
{ 
  printf("Could not create OpenCL data buffers! Exiting
sInitRlips.\n");
  if(*K->fullRotKernel)
    clReleaseKernel(*K->fullRotKernel);
  if(*K->partRotKernel)
    clReleaseKernel(*K->partRotKernel);
  if (*K->kernel_program)
    clReleaseProgram(*K->kernel_program);
  if (*K->commandqueue)
    clReleaseCommandQueue(*K->commandqueue);
  if (*K->context)
    clReleaseContext(*K->context);
  exit(1);
}

// Construct address and return it
// in a integer 2-vector
addr D;
long *q;
q = (long *)K;
D.longValue = (long)q;
INTEGER(ref)[0] = D.II[0];
INTEGER(ref)[1] = D.II[1];
UNPROTECT(1);
return(ref);

/***************************************************************************/
// End of sInitRlips
/******************************************************************************/

/*/ 
sKillRlips 
======
Dispose and deallocate rlips structures and arrays
Arguments:
  REF    Integer 2-vector containing the 64bit
         address of the rlips C structure
 */
SEXP sKillRlips(SEXP REF)
{
    // Definitions required by R .Call functionality
    REF = coerceVector(REF, INTSXP);

    // Construct address
    addr D;
    D.I[0] = INTEGER(REF)[0];
    D.I[1] = INTEGER(REF)[1];

    sRlips *K;
    K = (sRlips *)D.longValue;

    // Free/release/deallocate all OpenCL structures associated
    // to this rlips structure
    if (*K->fullRotKernel)
        clReleaseKernel(*K->fullRotKernel);
    if (*K->partRotKernel)
        clReleaseKernel(*K->partRotKernel);
    if (*K->kernel_program)
        clReleaseProgram(*K->kernel_program);
    if (*K->commandqueue)
        clReleaseCommandQueue(*K->commandqueue);
    if (*K->context)
        clReleaseContext(*K->context);
    if (K->dRmat)
        clReleaseMemObject(K->dRmat);

    free(K);

    // Return nothing (can not use void with .Call)
    return R_NilValue;
}

_submenu
End of sKillRlips

/*
sRotateRlips

Takes dataBuffer containing theory matrix rows and
measurements, sends it to GPU device and makes
the rotations in GPU device

Arguments:

    REF                      Integer vector containing
                              the address of the RLIPS structure
    DOUBLE_DATABUFFER       Double vector containing the data
                              in row-major order

*/
366 BUFFERROWS: Integer containing the number of data rows in DOUBLE_DATABUFFER vector */
367
368 SEXP sRotateRlips(SEXP REF, SEXP DOUBLE_DATABUFFER, SEXP BUFFERROWS)
369 {
370 // Definitions required by R .Call system
371 REF = coerceVector(REF, INTSXP);
372 double *double_dataBuffer = REAL(DOUBLE_DATABUFFER);
373 BUFFERROWS = coerceVector(BUFFERROWS, INTSXP);
374 int bufferRows = INTEGER(BUFFERROWS)[0];
375
376 // Construct address
377 addr D;
378 D.II[0] = INTEGER(REF)[0];
379 D.II[1] = INTEGER(REF)[1];
380 sRlips *K;
381 K = (sRlips *)D.longValue;
382
383 // Check that the number of bufferRows does not exceed
384 // the device buffer size (This should not happen in
385 // any situation.)
386 if (bufferRows > K->sizeBuffer)
387 {
388 printf("Too many data rows to rotate! Buffer has %d rows
. You tried to rotate %d rows.\nRotations not done!\n", K->sizeBuffer, bufferRows);
389 return R_NilValue;
390 }
391
392 // Rotate only, if there is something to rotate
393 if (bufferRows > 0)
394 {
395 int i;
396 cl_int error, err1, error2;
397 int rowsToRotate, numColumns, fRow, fCol,
398 numRows1, numRows2;
399 int stage, totalStages, firstRow, firstCol,
400 numRotations, dRmatOffset, n;
401 size_t localSize, globalSize;
402
403 // Allocate data buffer (float)
404 float __attribute__((aligned (32))) *dataBuffer;
405 dataBuffer = malloc(sizeof(float)
406 * bufferRows * K->numRmatCols);
// Copy the given double values as float values to the new buffer.
// NB: This is actually faster than casting the data as floats in the R side!
for (i=0 ; i< bufferRows * K->numRmatCols ; i++)
{
    dataBuffer[i] = (float) double_dataBuffer[i];
}

// Move data buffer into device
// Note that we create and copy at the same time
K->dBufferMat =
    clCreateBuffer(*K->context,
        CL_MEM_READ_WRITE|CL_MEM_COPY_HOST_PTR,
        sizeof(float) * K->numRmatCols
        * bufferRows,
        dataBuffer,
        &error2);

// Are there any full rotations to be made, i.e. does the target matrix already have rotated rows in it?
if (K->numRmatRows > 0)
{
    // Is R matrix already full, i.e all consecutive rotations will be full rotations?
    if (K->numRmatRows >= K->numCols)
    {
        // Rotate the whole buffer at once
        rowsToRotate = bufferRows;
        // Rotate through all columns
        numColumns = K->numCols;
        // Set the rotation start row and column
        fRow = 0;
        fCol = 0;
        // Call full rotations
        sFullRotations(K,rowsToRotate,numColumns,
                    fRow,fCol);
    }
    // else: will R matrix become full with this buffer?
    // If so, divide the rotations in two parts:
    // 1. Rotate fully and partially enough rows to make the target matrix full
    // 2. Rotate the remaining rows fully into
    // the target matrix
else if (K->numRmatRows + bufferRows > K->numCols)
{
    // Divide the rows into two parts
    numRows1 = K->numCols - K->numRmatRows;
    numRows2 = bufferRows - numRows1;

    // Part 1.
    // Rotate first numRows1 rows
    rowsToRotate = numRows1;

    // Rotate numRmatRows columns
    numColumns = K->numRmatRows;

    // Set starting row and column
    fRow = 0;
    fCol = 0;

    // Rotate first numRows1 rows fully
    sFullRotations(K,rowsToRotate,numColumns,
                   fRow,fCol);

    // Part 2.
    // Rotate first numRows1 rows
    rowsToRotate = numRows1;

    // Rotate numRows1 columns
    numColumns = numRows1;

    // Set starting row and column
    fRow = 0;
    fCol = K->numRmatRows;

    // Rotate partially
    sPartialRotations(K,rowsToRotate,numColumns,
                      fRow,fCol);

    // Part 3.
    // Rotate the remaining rows fully
    rowsToRotate = numRows2;
    numColumns = K->numCols;
    fRow = numRows1;
    fCol = 0;

    // Full rotations
    sFullRotations(K,rowsToRotate,numColumns,
                   fRow,fCol);
}
// else: after rotating the buffer, the target
// matrix remains non-full, i.e. rotate fully
// first numRmatRows rows and then rotate partially
// the remaining rows.
else
{
    // Set arguments for full rotation
    rowsToRotate = bufferRows;
    numColumns = K->numRmatRows;
    fRow = 0;
    fCol = 0;

    // Full rotations
    sFullRotations(K,rowsToRotate,numColumns,
                   fRow,fCol);

    // Set arguments for partial rotation
    rowsToRotate = bufferRows;
    numColumns = bufferRows;
    fRow = 0;
    fCol = K->numRmatRows;

    // Partial rotations
    sPartialRotations(K,rowsToRotate,numColumns,
                      fRow,fCol);
}
// else: There are no rows in the target matrix, i.e.
// these are the first rotations.
else
{
    // If there are more buffer rows than there are rows
    // in the target matrix, first rotate numRmatCols
    // rows partially to form full target matrix and
    // then rotate the remaining rows fully into
    // the target matrix
    if (bufferRows > K->numCols)
    {
        // Rotate the first numRmatCols rows partially
        rowsToRotate = K->numCols;
        numColumns = K->numCols;
        fRow = 0;
        fCol = 0;

        // Partial rotations
        sPartialRotations(K,rowsToRotate,numColumns,
                          fRow,fCol);

        // Rotate the remaining rows fully
rowsToRotate = bufferRows - K->numCols;
numColumns = K->numCols;
fRow = K->numCols;
fCol = 0;

// Full rotations
sFullRotations(K,rowsToRotate,numColumns,
               fRow,fCol);

} else
  // Target matrix is empty and there are not enough
  // rows in the buffer to make the target matrix
  // full. Rotate the whole buffer into the target
  // matrix partially
  {
    rowsToRotate = bufferRows;
    numColumns = bufferRows;
    fRow = 0;
    fCol = 0;

    // Partial rotations
    sPartialRotations(K,rowsToRotate,numColumns,
                      fRow,fCol);
  }

// Update internal parameters
K->numTotRows += bufferRows;
K->numRmatRows += bufferRows;

// Number of rows in the target matrix can not be
// over numRmatCols
if (K->numRmatRows > K->numCols)
  K->numRmatRows = K->numCols;

// Release OpenCL buffer
clReleaseMemObject(K->dBufferMat);

// Free data buffer
free(dataBuffer);
}

// Return nothing to R
return R_NilValue;

// End of sRotateRlips
sGetDataRlips =============

Fetches R matrix from the GPU device and sends it back to R.

Arguments:

REF Integer 2-vector containing the 64bit address of the rlns C structure

SEXP sGetDataRlips(SEXP REF)
{
    // Definitions required by R .Call
    REF = coerceVector(REF,INTSXP);
    SEXP DOUBLE_DATABUFFER;

    // Construct address
    addr D;
    D.II[0] = INTEGER(REF)[0];
    D.II[1] = INTEGER(REF)[1];
    sRlips *K;
    K = (sRlips *)D.longValue;

    cl_int error;

    // Allocate array for data
    float __attribute__((aligned (32))) *dataBuffer;
    dataBuffer = malloc(sizeof(float) * K->sizeRmat);

    // This contains the data to be sent back to R.
    // Needs to be protected from R Garbage Collection
    PROTECT(DOUBLE_DATABUFFER = allocVector(REALSXP, K->sizeRmat));

    // Read dRmat from device to dataBuffer
    error = clEnqueueReadBuffer(*K->commandqueue,K->dRmat,
                                CL_TRUE,0,sizeof(float) * K->sizeRmat,
                                dataBuffer,0,NULL,NULL);
    if (error != CL_SUCCESS)
    { printf("Could not read buffer from device!\n");
      return R_NilValue;
    }
int i;
for (i=0 ; i < K->sizeRmat ; i ++)
{
    REAL(DOUBLE_DATABUFFER)[i] = (double) dataBuffer[i];
}

// Free allocated arrays and return
free(dataBuffer);

// Unprotect DOUBLE_DATABUFFER
UNPROTECT(1);

return DOUBLE_DATABUFFER;

/*
 Routines for single precision complex
 */

/*
cInitRlips
 =========
 Initialize a new RLIPS problem and structure, complex version
 Arguments:
   NCOLS     Number of columns in the theory matrix, i.e.
              number of unknowns
   NRHS      Number of columns in the measurement matrix
   NBUF      Rotation buffer size, i.e. number of data rows
              stored into a buffer matrix
   BLOCKSIZE OpenCL work group size, should currently be
              a multiple of 16. The optimal value depends
              on the GPU used.

 Returns a integer vector of length 2.
 The vector contains the (64bit) address of
 the initialized structure.
*/
SEXP cInitRlips(SEXP NCOLS, SEXP NRHS, SEXP NBUF,
    SEXP BLOCKSIZE)
{
    // Definitions required by R
    SEXP ref;
    PROTECT(ref = allocVector(INTSXP,2));
    NCOLS = coerceVector(NCOLS,INTSXP);
    NRHS = coerceVector(NRHS,INTSXP);
    NBUF = coerceVector(NBUF,INTSXP);
    BLOCKSIZE = coerceVector(BLOCKSIZE,INTSXP);
    int ncols = INTEGER(NCOLS)[0];
    int nrhs = INTEGER(NRHS)[0];
    int nbuf = INTEGER(NBUF)[0];
    int blocksize = INTEGER(BLOCKSIZE)[0];

    // Allocate new sRlips struct
    cRlips *K;
    K = (cRlips *)malloc(sizeof(cRlips));

    // Allocate OpenCL structures in K
    K->platform_id = malloc(sizeof(cl_platform_id));
    K->device_id = malloc(sizeof(cl_device_id));
    K->context = malloc(sizeof(cl_context));
    K->commandqueue = malloc(sizeof(cl_command_queue));
    K->kernel_program = malloc(sizeof(cl_program));
    K->fullRotKernel = malloc(sizeof(cl_kernel));
    K->partRotKernel = malloc(sizeof(cl_kernel));

    // Set the user provided parameters
    K->numCols = ncols;
    K->numRHS = nrhs;
    K->sizeBuffer = nbuf;
    K->sizeWorkgroup = blocksize;

    // Column size of OpenCL buffers (smallest multiple of
    // workgroup size that contains both theory matrix columns
    // and measurements, multiple of 32)
    K->numRmatCols = (ncols + nrhs + 32 - 1) / 32 * 32;

    // Numbers whose absolute value is smaller than zThreshold
    // are considered as zeroes
    K->zThreshold = 1.0E-8f;

    // Set initial values for status parameters
    K->numTotRows = 0;
    K->numBufferRows = 0;
    K->numRmatRows = 0;
    K->flops = 0L;
}
// Set OpenCL buffersizes
K->sizeRmat = K->numRmatCols * K->numCols;
K->sizeBufferMat = K->numRmatCols * K->sizeBuffer;

// Initialize OpenCL
cl_int error;

// Get first OpenCL platform
error = clGetPlatformIDs(1,K->platform_id,NULL);
if (error != CL_SUCCESS)
{
    printf("Did not get OpenCL platform! Error code %d. Exiting sInitRlips.\n", error);
    exit(1);
}

// Ask for one GPU
error = clGetDeviceIDs(*K->platform_id,CL_DEVICE_TYPE_GPU, 1,K->device_id,NULL);
if (error != CL_SUCCESS)
{
    printf("Did not get OpenCL device! Error code %d. Exiting sInitRlips.\n", error);
    exit(1);
}

// Create OpenCL context
*K->context = clCreateContext(0,1,K->device_id,NULL,NULL, &error);
if (error != CL_SUCCESS)
{
    printf("Did not create OpenCL context! Error code %d. Exiting sInitRlips.\n", error);
    if (*K->context) clReleaseContext(*K->context);
    exit(1);
}

// Create command queue
*K->commandqueue = clCreateCommandQueue(*K->context,*K->device_id, 0,&error);
if (error != CL_SUCCESS)
{
    printf("Did not create OpenCL command queue! Error code %d. Exiting sInitRlips.\n", error);
    if (*K->commandqueue) clReleaseCommandQueue(*K->commandqueue);
    if (*K->context) clReleaseCommandQueue(*K->commandqueue);
}
clReleaseContext(*K->context);
exit(1);
}

// Create kernel program (KernelSource in Rlips.h)
*K->kernel_program = clCreateProgramWithSource(*K->context,1,
(const char**)&cKernelSource,NULL,&error);
if (!*K->kernel_program)
{
    printf("Could not create compute program! Exiting sInitRlips.\n");
    if (*K->commandqueue)
        clReleaseCommandQueue(*K->commandqueue);
    if (*K->context)
        clReleaseContext(*K->context);
    exit(1);
}

// Build kernel executable
error = clBuildProgram(*K->kernel_program,0,
NULL,"-w",NULL,NULL);
if (error != CL_SUCCESS)
{
    printf("Error code: %d\n",error);
    size_t len;
    char buffer[1024*100];
    printf("Failed to build program executable! Exiting sInitRlips.\n");
    clGetProgramBuildInfo(*K->kernel_program,
        *K->device_id,
        CL_PROGRAM_BUILD_LOG,
        sizeof(buffer),
        buffer,
        &len);
    printf("Log length: %d\n%s\n",(int) len,buffer);
    if (*K->kernel_program)
        clReleaseProgram(*K->kernel_program);
    if (*K->commandqueue)
        clReleaseCommandQueue(*K->commandqueue);
    if (*K->context)
        clReleaseContext(*K->context);
    exit(1);
}
// Create the kernel functions
*K->fullRotKernel = clCreateKernel(*K->kernel_program,
"c_full_rotations", &error);
cl_int error2;
*K->partRotKernel = clCreateKernel(*K->kernel_program,
"c_partial_rotations", &error2);
if (error != CL_SUCCESS || error2 != CL_SUCCESS)
{
printf("Could not create kernel! Error codes: %d, %d.
Exiting sInitRlips.\n", error, error2);
if (*K->fullRotKernel)
clReleaseKernel(*K->fullRotKernel);
if (*K->partRotKernel)
clReleaseKernel(*K->partRotKernel);
if (*K->kernel_program)
clReleaseProgram(*K->kernel_program);
if (*K->commandqueue)
clReleaseCommandQueue(*K->commandqueue);
if (*K->context)
clReleaseContext(*K->context);
exit(1);
}

// Create OpenCL buffers
K->dRmat_r = clCreateBuffer(*K->context, CL_MEM_READ_WRITE,
sizeof(float) * K->sizeRmat, NULL, &error);
K->dRmat_i = clCreateBuffer(*K->context, CL_MEM_READ_WRITE,
sizeof(float) * K->sizeRmat, NULL, &error2);
if (error != CL_SUCCESS || error2 != CL_SUCCESS)
{
printf("Could not create OpenCL data buffers! Error
codes: %d, %d. Exiting sInitRlips.\n", error, error2);
if (*K->fullRotKernel)
clReleaseKernel(*K->fullRotKernel);
if (*K->partRotKernel)
clReleaseKernel(*K->partRotKernel);
if (*K->kernel_program)
clReleaseProgram(*K->kernel_program);
if (*K->commandqueue)
clReleaseCommandQueue(*K->commandqueue);
if (*K->context) clReleaseContext(*K->context);
exit(1);
}
895 // Construct address and return
896 addr D;
897 long *q;
898 q = (long *)K;
899 D.longValue = (long)q;
900
901 INTEGER(ref)[0] = D.II[0];
902 INTEGER(ref)[1] = D.II[1];
903
904 return(1);
905
906 }
907 ///////////////////////////////////////////////////////////////
908 // End of cInitRlips
909 ///////////////////////////////////////////////////////////////
910
911 /*
912 cKillRlips
913 ========
914 Dispose and deallocate rlips structures and arrays,
915 complex version
916 Arguments:
917 REF Integer 2-vector containing the 64bit
918 address of the rlips C structure
919 */
920 SEXP cKillRlips(SEXP REF)
921 {
922
923 REF = coerceVector(REF,INTSXP);
924
925 // Construct address
926 addr D;
927 D.II[0] = INTEGER(REF)[0];
928 D.II[1] = INTEGER(REF)[1];
929
cRlips *K;
930 K = (cRlips *)D.longValue;
931
932 // Release OpenCL structs
933 if(*K->fullRotKernel)
934 clReleaseKernel(*K->fullRotKernel);
935 if(*K->partRotKernel)
clReleaseKernel(*K->partRotKernel);
if (*K->kernel_program)
clReleaseProgram(*K->kernel_program);
if (*K->commandqueue)
clReleaseCommandQueue(*K->commandqueue);
if (*K->context)
clReleaseContext(*K->context);

// Release OpenCL memory objects
if (K->dRmat_r) clReleaseMemObject(K->dRmat_r);
if (K->dRmat_i) clReleaseMemObject(K->dRmat_i);

// Free RLIPS struct
free(K);

// Return nothing (Cannot use void for .Call)
return R_NilValue;

} // cKillRlips

/*/ 
cRotateRlips
Takes dataBuffer containing theory matrix rows and measurements, sends it to GPU device and makes the rotations in GPU device, complex version

Arguments:
REF Integer vector containing the address of the RLIPS structure
DOUBLE_DATABUFFER_R Double vector containing the data in row-major order, real part
DOUBLE_DATABUFFER_I Double vector containing the data in row-major order, imaginary part
BUFFERROWS Integer containing the number of data rows in DOUBLE_DATABUFFER vector

SEXP cRotateRlips(SEXP REF, SEXP DOUBLE_DATABUFFER_R,
SEXPR DOUBLE_DATABUFFER_I, SEXP BUFFERROWS)
{
  // Definitions required by R
  REF = coerceVector(REF, INTSXP);
double *double_dataBuffer_r = REAL(DOUBLE_DATABUFFER_R);
double *double_dataBuffer_i = REAL(DOUBLE_DATABUFFER_I);
BUFFERROWS = coerceVector(BUFFERROWS, INTSXP);
int bufferRows = INTEGER(BUFFERROWS)[0];

// Construct address
addr D;
D.II[0] = INTEGER(REF)[0];
D.II[1] = INTEGER(REF)[1];
cRlips *K;
K = (cRlips *)D.longValue;

// Check that the number of bufferRows does not exceed
// the device buffer size
if (bufferRows > K->sizeBuffer)
{
    printf("Too many data rows to rotate! Buffer has %d rows
    . You tried to rotate %d rows.\nRotations not done!\n",K->sizeBuffer,bufferRows);
    return R_NilValue;
}

// Make sure there is something to rotate
if (bufferRows > 0)
{
    int i;
    cl_int error;
    int rowsToRotate, numColumns, fRow,
    fCol, numRows1, numRows2;

    // Allocate data buffers for real and imag parts
    float __attribute__ ((aligned (32))) *dataBuffer_r;
    float __attribute__ ((aligned (32))) *dataBuffer_i;

dataBuffer_r
    = malloc(sizeof(float)
    * bufferRows
    * K->numRmatCols);

dataBuffer_i
    = malloc(sizeof(float)
    * bufferRows
    * K->numRmatCols);

    // Copy the double values from R into
    // single float arrays
    // NB: This is actually faster than casting the data
    // as floats in the R side!
    for (i=0 ; i< bufferRows * K->numRmatCols ; i++)
    {
        dataBuffer_r[i] = (float) double_dataBuffer_r[i];
dataBuffer_i[i] = (float) double_dataBuffer_i[i];
}

cl_int error2;
// Move data buffers into device
// (Create and copy)
// Real part
K->dBuffMat_r = clCreateBuffer(*K->context,
  CL_MEM_READ_WRITE| CL_MEM_COPY_HOST_PTR,
  sizeof(float) * K->numRmatCols * bufferRows, dataBuffer_r,
  &error2);

// Imaginary part
K->dBuffMat_i = clCreateBuffer(*K->context,
  CL_MEM_READ_WRITE| CL_MEM_COPY_HOST_PTR,
  sizeof(float) * K->numRmatCols * bufferRows, dataBuffer_i,
  &error2);

// Are there any full rotations to be made, i.e. does
// the target matrix already have rotated rows in it?
if (K->numRmatRows > 0)
{
  // Is R matrix already full, i.e all consecutive
  // rotations will be full rotations?
  if (K->numRmatRows >= K->numCols)
  {
    // Rotate the whole buffer at once
    rowsToRotate = bufferRows;
    // Rotate through all columns
    numColumns = K->numCols;
    // Set the rotation start row and column
    fRow = 0;
    fCol = 0;
    // Call full rotations
    cFullRotations(K, rowsToRotate, numColumns, fRow, fCol);
  }
  // else: will R matrix become full with this buffer?
// If so, divide the rotations in two parts:
// 1. Rotate fully and partially enough rows to make
// the target matrix full
// 2. Rotate the remaining rows fully into
// the target matrix
else if (K->numRmatRows + bufferRows > K->numCols)
{
    // Divide the rows into two parts
    numRows1 = K->numCols - K->numRmatRows;
    numRows2 = bufferRows - numRows1;

    // Part 1.
    // Rotate first numRows1 rows
    rowsToRotate = numRows1;

    // Rotate numRmatRows columns
    numColumns = K->numRmatRows;

    // Set starting row and column
    fRow = 0;
    fCol = 0;

    // Rotate first numRows1 rows fully
cFullRotations(K,rowsToRotate,numColumns,
                   fRow,fCol);

    // Part 2.
    // Rotate first numRows1 rows
    rowsToRotate = numRows1;

    // Rotate numRows1 columns
    numColumns = numRows1;

    // Set starting row and column
    fRow = 0;
    fCol = K->numRmatRows;

    // Rotate partially
cPartialRotations(K,rowsToRotate,numColumns,
                       fRow,fCol);

    // Part 3.
    // Rotate the remaining rows fully
    rowsToRotate = numRows2;
    numColumns = K->numCols;
    fRow = numRows1;
    fCol = 0;
// Full rotations
cFullRotations(K, rowsToRotate, numColumns, fRow, fCol);

} // else: after rotating the buffer, the target
// matrix remains non-full, i.e. rotate fully
// first numRmatRows rows and then rotate partially
// the remaining rows.
else {

// Set arguments for full rotation
rowsToRotate = bufferRows;
numColumns = K->numRmatRows;
fRow = 0;
fCol = 0;

// Full rotations
cFullRotations(K, rowsToRotate, numColumns, fRow, fCol);

// Set arguments for partial rotation
rowsToRotate = bufferRows;
numColumns = bufferRows;
fRow = 0;
fCol = K->numRmatRows;

// Partial rotations
cPartialRotations(K, rowsToRotate, numColumns, fRow, fCol);

}

// else: There are no rows in the target matrix, i.e.
// these are the first rotations.
else {

// If there are more buffer rows than there are rows
// in the target matrix, first rotate numRmatCols
// rows partially to form full target matrix and
// then rotate the remaining rows fully into
// the target matrix
if (bufferRows > K->numCols) {

// Rotate the first numRmatCols rows partially
rowsToRotate = K->numCols;
numColumns = K->numCols;
fRow = 0;
fCol = 0;

}
// Partial rotations
cPartialRotations(K, rowsToRotate, numColumns, fRow, fCol);

// Rotate the remaining rows fully
rowsToRotate = bufferRows - K->numCols;
numColumns = K->numCols;
fRow = K->numCols;
fCol = 0;

// Full rotations
cFullRotations(K, rowsToRotate, numColumns, fRow, fCol);

else

// Target matrix is empty and there are not enough rows in the buffer to make the target matrix fully. Rotate the whole buffer into the target matrix partially
{
    rowsToRotate = bufferRows;
    numColumns = bufferRows;
    fRow = 0;
    fCol = K->numRmatRows;
    cPartialRotations(K, rowsToRotate, numColumns, fRow, fCol);
}

// Update internal parameters
K->numTotRows += bufferRows;
K->numRmatRows += bufferRows;

// Number of rows in the target matrix can not be over numRmatCols
if (K->numRmatRows > K->numCols)
    K->numRmatRows = K->numCols;

// Clean up
free(dataBuffer_r);
free(dataBuffer_i);
clReleaseMemObject(K->dBufferMat_r);
clReleaseMemObject(K->dBufferMat_i);

// Return nothing to R
return R_NilValue;
118

1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284

}
}
///////////////////////////////////////////////////////////////
// End of cRotateRlips
///////////////////////////////////////////////////////////////
/*
cGetDataRlips
=============
Fetches R matrix from the GPU device and sends it back to R.
Complex version
Arguments:
REF

Integer 2-vector containing the 64bit
address of the rlips C structure

*/
SEXP cGetDataRlips(SEXP REF)
{
// Definitions required by R
REF = coerceVector(REF,INTSXP);
SEXP DOUBLE_DATABUFFER;
// Construct address
addr D;
D.II[0] = INTEGER(REF)[0];
D.II[1] = INTEGER(REF)[1];
cRlips *K;
K = (cRlips *)D.longValue;
cl_int error;
// Allocate data arrays
float __attribute__ ((aligned (32))) *dataBuffer_r;
float __attribute__ ((aligned (32))) *dataBuffer_i;
dataBuffer_r = malloc(sizeof(float) * K->sizeRmat);
dataBuffer_i = malloc(sizeof(float) * K->sizeRmat);
// Allocate and protect R return array
PROTECT(DOUBLE_DATABUFFER
= allocVector(REALSXP, K->sizeRmat * 2));
// Read dRmat from device to dataBuffer
// Real part
error = clEnqueueReadBuffer(*K->commandqueue,
K->dRmat_r,
CL_TRUE,

67


0,
sizeof(float) * K->sizeRmat,
dataBuffer_r,
0,
NULL,
NULL);

// Imaginary part
error = clEnqueueReadBuffer(*K->commandqueue,
K->dRmat_i,
CL_TRUE,
0,
sizeof(float) * K->sizeRmat,
dataBuffer_i,
0,
dataBuffer_i,
NULL,
NULL);

if (error != CL_SUCCESS)
{
    printf("Could not read buffer from device!\n");

    return R_NilValue;
}

// Transfer (and re-cast) data from float arrays to
double array.
int i;
for (i=0 ; i < K->sizeRmat ; i++)
{
    REAL(DOUBLE_DATABUFFER)[2*i] = (double) dataBuffer_r[i];
    REAL(DOUBLE_DATABUFFER)[2*i+1] = (double) dataBuffer_i[i];
}

// Free data buffers and finish
free(dataBuffer_r);
free(dataBuffer_i);
UNPROTECT(1);

return DOUBLE_DATABUFFER;

}
3.2.3 Header file: rotations.h

Header file rotations.h contains function prototypes of file rotations.c.

```c
#ifndef __ROTATIONS_H
#define __ROTATIONS_H

#include "rlips.h"

void sFullRotations(sRlips *, int, int, int, int);
void sPartialRotations(sRlips *, int, int, int, int);
void cFullRotations(cRlips *, int, int, int, int);
void cPartialRotations(cRlips *, int, int, int, int);
#endif
```

3.2.4 File: rotations.c

File rotations.c contains the C/OpenCL routines that perform rotations for the data in the GPU.

```c
#include "rotations.h"
#include <stdio.h>

#define ridx(A,B,C) ((B)+(A)*(2*(C)-(A)-1)/2)
```

```c
// Create index for upper triangular matrix
// stored in row-major order
// A row
// B column
// C number of columns in matrix
#define ridx(A,B,C) ((B)+(A)*(2*(C)-(A)-1)/2)
```

```c
// Create index for rectangular matrix
// stored in row-major order
// A row
// B column
// C number of columns in matrix
```
```c
#define yidx(A, B, C) ((A)* (C)+(B))

// Standard definitions for min and max
#define min(A, B) (((A) <=(B))?(A) : (B))
#define max(A, B) (((A) > (B))?(A):(B))

/*
 * sFullRotations
 * =============
 * Performs full rotations for between target and buffer matrices
 * Arguments:
 * K Real RLIPS structure
 * rowsToRotate Number of rows to be rotated
 * numColumns Number of columns to be rotated
 * fRow First row to be rotated
 * fCol First column to be rotated
 */

void sFullRotations(sRlips *K, int rowsToRotate, int numColumns,
                    int fRow, int fCol)
{

    // Calculate rotation parameters
    // Stages
    int totalStages = rowsToRotate + numColumns - 1;

    // Loop through stages
    int stage;
    for (stage = 1; stage <= totalStages; stage++)
    {
        // Calculate number of rotations
        // in this stage
        int nn;
        if (rowsToRotate > numColumns)
        {
            nn = numColumns;
        }
        else
        {
            nn = rowsToRotate;
        }
        int numRotations;
        numRotations = min(stage, nn);
        numRotations = min(numRotations,
```

75 totalStages - stage + 1);
76
77 // Find first row and column for rotations
78 int firstRow, firstCol;
79 firstRow = min(stage, rowsToRotate) - 1 + fRow;
80 firstCol = max(1, stage - (rowsToRotate - 1))
81 - 1 + fCol;
82
83 // number of rotations/work-groups
84 // Every single rotation is made in its own
85 // work-group
86 size_t localSize = K->sizeWorkgroup;
87 size_t globalSize = localSize * numRotations;
88
89 int err1;
90
91 // Set up kernel arguments
92 int n = 0;
93 // R matrix array
94 err1 = clSetKernelArg (*K->fullRotKernel, 
95 n++,
96 sizeof(cl_mem),
97 &K->dRmat);
98
99 // Buffer array
100 err1 |= clSetKernelArg (*K->fullRotKernel, 
101 n++,
102 sizeof(cl_mem),
103 &K->dBufferMat);
104
105 // First row
106 err1 |= clSetKernelArg (*K->fullRotKernel, 
107 n++,
108 sizeof(int),
109 &firstRow);
110
111 // First column
112 err1 |= clSetKernelArg (*K->fullRotKernel, 
113 n++,
114 sizeof(int),
115 &firstCol);
116
117 // Number of columns in R matrix
118 err1 |= clSetKernelArg (*K->fullRotKernel, 
119 n++,
120 sizeof(int),
121 &K->numRmatCols);
122
123 // Handle possible errors
124 if ( err1 != CL_SUCCESS)
125 {
126 printf("Error in kernel arguments! Error code %d.\n", (int) err1);
// Launch full rotation kernel
err1 = clEnqueueNDRangeKernel(*K->commandqueue,
    *K->fullRotKernel,
    1,
    NULL,
    &globalSize,
    &localSize,
    0,
    NULL,
    NULL);

if (err1 != CL_SUCCESS)
    {
        printf("Error in kernel execution! Error code %d.\n", (int) err1);
    }

    // Finish
    return;
}

/***********************************************
// End of sFullRotations
department

DIRECTORY

* /
cFullRotations

===============
Perform full rotations for between target
and buffer matrices

Complex version

Arguments:

    K              Complex RLIPS structure
    rowsToRotate   Number of rows to be rotated
    numColumns     Number of columns to be rotated
    fRow           First row to be rotated
    fCol           First column to be rotated

* /
void cFullRotations(cRlips *K, int rowsToRotate, int numColumns,
    int fRow, int fCol)
{
    // Calculate rotation parameters
    // Stages
int totalStages = rowsToRotate + numColumns - 1;

// Loop through stages
int stage;
for (stage = 1; stage <= totalStages; stage++)
{
    // Calculate number of rotations
    // in this stage
    int nn;
    if (rowsToRotate > numColumns)
    {
        nn = numColumns;
    }
    else
    {
        nn = rowsToRotate;
    }

    int numRotations;
    numRotations = min(stage, nn);
    numRotations = min(numRotations,
                        totalStages - stage + 1);

    // Find first row and column for rotations
    int firstRow, firstCol;
    firstRow = min(stage, rowsToRotate) - 1 + fRow;
    firstCol = max(1, stage - (rowsToRotate - 1))
                - 1 + fCol;

    // number of rotations/work-groups
    // Every single rotation is made in its own
    // work-group
    size_t localSize = K->sizeWorkgroup;
    size_t globalSize = localSize * numRotations;

    int err1;

    // Set up kernel arguments
    int n = 0;
    // R matrix real part
    err1 = clSetKernelArg (*K->fullRotKernel,
                           n++,
                           sizeof(cl_mem),
                           &K->dRmat_r);
    // R matrix imaginary part
    err1 |= clSetKernelArg (*K->fullRotKernel,
                            n++,
                            sizeof(cl_mem),
                            &K->dRmat_i);
// Rotation buffer real part
err1 |= clSetKernelArg(*K->fullRotKernel,
n++,
sizeof(cl_mem),
&K->dBufferMat_r);

// Rotation buffer imaginary part
err1 |= clSetKernelArg(*K->fullRotKernel,
n++,
sizeof(cl_mem),
&K->dBufferMat_i);

// First row
err1 |= clSetKernelArg(*K->fullRotKernel,
n++,
sizeof(int),
&firstRow);

// First column
err1 |= clSetKernelArg(*K->fullRotKernel,
n++,
sizeof(int),
&firstCol);

// Number of columns in R matrix
err1 |= clSetKernelArg(*K->fullRotKernel,
n++,
sizeof(int),
&K->numRmatCols);

// Handle errors
if (err1 != CL_SUCCESS)
{
    printf("Error in kernel arguments! Error code %d.\n", err1);
}

// Launch full rotation kernel
err1 = clEnqueueNDRangeKernel(*K->commandqueue,
    *K->fullRotKernel,
    1,
    NULL,
    &globalSize,
    &localSize,
    0,
    NULL,
    NULL);

if (err1 != CL_SUCCESS)
{
    printf("Error in kernel execution! Error code %d.\n", err1);
}
// Finish
return;
}

/////////////////////////////////////////////////////////////
// End of cFullRotations
/////////////////////////////////////////////////////////////

/*/  

sPartialRotations
============

Performs partial rotations for between target
and buffer matrices

Arguments:

  K        Real RLIPS structure
rowsToRotate Number of rows to be rotated
numColumns Number of columns to be rotated
fRow      First row to be rotated
fCol      First column to be rotated
*/

void sPartialRotations(sRlips *K,
        int rowsToRotate,
        int numColumns,
        int fRow,
        int fCol)
{
    int err1, n;
    // Calculate rotation parameters
    // Stages
    int totalStages = 2 * rowsToRotate - 3;
    // Loop through stages
    int stage;
    for (stage = 1; stage <= totalStages; stage++)
    {
        // Calculate number of rotations
        // for this stage
        int numRotations =
            min((stage+1) / 2,
                (2 * rowsToRotate - 1 - stage) / 2 );
        // Find the first row and column for rotations

int firstRow =
    min(stage + 1, rowsToRotate)
    - 1 + fRow;
int firstCol =
    max(stage - rowsToRotate + 2 , 1)
    - 1;

// number of rotations/work-groups
// Every single rotation is made in its own
// work-group
size_t localSize = K->sizeWorkgroup;
size_t globalSize = localSize * numRotations;

// Set up kernel arguments
n = 0;
// Buffer matrix
er1 = clSetKernelArg (*K->partRotKernel,
    n++,
    sizeof(cl_mem),
    &K->dBufferMat);

// First rotation row
er1 |= clSetKernelArg (*K->partRotKernel,
    n++,
    sizeof(int),
    &firstRow);

// First rotation column
er1 |= clSetKernelArg (*K->partRotKernel,
    n++,
    sizeof(int),
    &firstCol);

// Number of R matrix columns
er1 |= clSetKernelArg (*K->partRotKernel,
    n++,
    sizeof(int),
    &K->numRmatCols);

// Column offset
er1 |= clSetKernelArg (*K->partRotKernel,
    n++,
    sizeof(int),
    &fCol);

if ( err1 != CL_SUCCESS)
{
    printf("Error in kernel arguments! Error code %d.\n",
        err1);
}

// Launch rotation kernel
er1 = clEnqueueNDRangeKernel(*K->commandqueue,
if ( err1 != CL_SUCCESS)
{
    printf("Error in partial rotation kernel execution! Error code %d.\n", err1);
}

// Rotations done. Move Rotated dBuffer in dRmat
// Calculate offset for copy
int dRmatOffset = K->numRmatRows * K->numRmatCols;

// Copy from one OpenCL buffer to another
err1 = clEnqueueCopyBuffer(*K->commandqueue,
    K->dBufferMat,
    K->dRmat,
    0,
    dRmatOffset * sizeof(float),
    sizeof(float) * rowsToRotate
    * K->numRmatCols,
    0,
    NULL,
    NULL);
if ( err1 != CL_SUCCESS)
{
    printf("Error in buffer copy! Error code %d.\n", err1);
    exit(1);
}

return;

}
and buffer matrices

Complex version

Arguments:

K Complex RLIPS structure
rowsToRotate Number of rows to be rotated
numColumns Number of columns to be rotated
fRow First row to be rotated
fCol First column to be rotated

*/

void cPartialRotations(cRlips *K,
    int rowsToRotate,
    int numColumns,
    int fRow,
    int fCol)
{
    int err1,n;
    // Calculate rotation parameters
    // Stages
    int totalStages = 2 * rowsToRotate - 3;
    // Loop through stages
    int stage;
    for (stage = 1; stage <= totalStages; stage++)
    {
        // Calculate number of rotations
        // for this stage
        int numRotations =
            min((stage+1) / 2,
                (2 * rowsToRotate - 1 - stage) / 2);
        // Find the first row and column for rotations
        int firstRow =
            min(stage + 1,
                rowsToRotate) - 1 + fRow;
        int firstCol =
            max(stage - rowsToRotate + 2 , 1) - 1;
        // number of rotations/work-groups
        // Every single rotation is made in its own
        // work-group
        size_t localSize = K->sizeWorkgroup;
        size_t globalSize = localSize * numRotations;
Set up kernel arguments
n = 0;

// Real part of buffer matrix
err1 = clSetKernelArg (*K->partRotKernel,
n++,
sizeof(cl_mem),
&K->dBufferMat_r);

// Imaginary part of buffer matrix
err1 = clSetKernelArg (*K->partRotKernel,
n++,
sizeof(cl_mem),
&K->dBufferMat_i);

// First rotation row
err1 |= clSetKernelArg (*K->partRotKernel,
n++,
sizeof(int),
&firstRow);

// First rotation column
err1 |= clSetKernelArg (*K->partRotKernel,
n++,
sizeof(int),
&firstCol);

// Number of columns in R matrix
err1 |= clSetKernelArg (*K->partRotKernel,
n++,
sizeof(int),
&K->numRmatCols);

// Column offset
err1 |= clSetKernelArg (*K->partRotKernel,
n++,
sizeof(int),
&fCol);

if ( err1 != CL_SUCCESS)
{
    printf("Error in kernel arguments! Error code %d.\n", err1);
}

// Launch rotation kernel
err1 = clEnqueueNDRangeKernel(*K->commandqueue,
*K->partRotKernel,
1,
NULL,
&globalSize,
&localSize,
0,
NULL,
NULL);
if (err1 != CL_SUCCESS) {
    printf("Error in partial rotation kernel execution!
            Error code %d.\n", err1);
}

// Rotations done. Move Rotated dBuffer in dRmat
// Calculate offset for copy
int dRmatOffset = K->numRmatCols * K->numRmatRows;

// Copy both real and imaginary parts of the buffer
// into R matrix buffers
err1 = clEnqueueCopyBuffer(*K->commandqueue, K->dBufferMat_r, K->dRmat_r, 0,
                           dRmatOffset * sizeof(float),
                           sizeof(float) * rowsToRotate * K->numRmatCols,
                           0, NULL, NULL);
err1 |= clEnqueueCopyBuffer(*K->commandqueue, K->dBufferMat_i, K->dRmat_i, 0,
                           dRmatOffset * sizeof(float),
                           sizeof(float) * rowsToRotate * K->numRmatCols,
                           0, NULL, NULL);
if (err1 != CL_SUCCESS) {
    printf("Error in buffer copy! Error code %d.\n", err1);
}
return;

// End of cPartialRotations

3.2.5 File: kernelsources.h

Header file kernelsources.h contains the code for the OpenCL kernel functions. In the file they are given as two character strings, namely sKernelSource for real valued kernels and cKernelSource for complex kernels. Because the C character strings are hard to read, they are represented below as C functions. The strings themselves are omitted from the code listing below.

```c
/* file: kernelsources.h */
/* OpenCL-LIPS compute kernel source codes */

(c) 2011- University of Oulu, Finland
Written by Mikko Orispaa <mikko.orispaa@oulu.fi>
Licensed under FreeBSD license. See file LICENSE for details.

#ifndef __KERNELSOURCES_H
#define __KERNELSOURCES_H

// Single precision real
const char *sKernelSource = /* String omitted, see below */
const char *cKernelSource = /* String omitted, see below */
#endif

sKernelSource

/* s_full_rotations */
============
OpenCL kernel function for performing full rotations.
Every instance of this function rotates one element.
The rows to rotate are given by the work-group id.
Every work-group is responsible of rotating one row.

Arguments
- Rmat OpenCL buffer for R matrix
- BufferMat OpenCL buffer for buffer matrix
- firstRow First row to rotate
- firstCol First column to rotate
```
nCols  Number of columns in R matrix

*/
__kernel void s_full_rotations( __global float* Rmat,
__global float* BufferMat,
int firstRow,
int firstCol,
int nCols)
{
    // Get the information of the current
    // work-item
    int currentRotation = get_group_id(0);
    int lid = get_local_id(0);
    int locSize = get_local_size(0);

    // Use local memory to store
    // rotation sine and cosine
    local float rotCos;
    local float rotSin;

    // Use private memory to store
    // the matrix elements that this
    // work-item rotates
    float locR1;
    float locR2;

    // Use local memory to store the skip and swap flags
    local int skip;
    local int swap;

    // Use local memory to store the indices of
    // rotated row and column
    local int rotRow, rotCol;

    // Initialize skip and swap flags
    skip = 0;
    swap = 0;

    // If this is work-item number 0, calculate
    // the rotation parameters. Otherwise, do
    // nothing.
    if (lid == 0)
    {
        // Find the row and starting column for
        // this work-group
        rotRow = firstRow - currentRotation;
        rotCol = firstCol + currentRotation;

        // Calculate the Givens rotation

// coefficients
float a, b, sqab;
// Fetch the first elements from R and buffer matrices
a = Rmat[rotCol * nCols + rotCol];
b = BufferMat[rotRow * nCols + rotCol];

// auxiliary variable
sqab = sqrt(a*a + b*b);

if (sqab < 0.00001f)
{
    // first elements are zeroes. No need to rotate
    rotCos = 1.0f;
    rotSin = 0.0f;
    skip = 1;
}
else if (fabs(a) < 0.00001f)
{
    // First R matrix element is zero. R and buffer lines are swapped.
    rotCos = 0.0f;
    rotSin = b/sqab;
    swap = 1;
}
else
{
    // Normal rotation sine and cosine calculation
    rotCos = a/sqab;
    rotSin = b/sqab;
}

// All other work-items wait here until work-item 0 has calculated the rotation parameters.
barrier(CLK_LOCAL_MEM_FENCE);

// If skip flag is set just return
if (skip)
{
    return;
}

int curBlock;

// Make rotations starting from element curBlock, jumping locSize elements forward till all done
for(curBlock = rotCol / 32 * 32 + lid;
    curBlock < nCols;
    curBlock += locSize)
{
    // Fetch the elements to be rotated
    locR1 = Rmat[rotCol * nCols + (curBlock)];
    locR2 = BufferMat[rotRow * nCols + (curBlock)];

    if (swap)
    {
        // If swap flag is set, swap the elements
        // Note that they still are multiplied
        // by rotSin
        Rmat[rotCol * nCols + (curBlock) ]
        = rotSin * locR2;
        BufferMat[rotRow * nCols + (curBlock)]
        = -rotSin * locR1;
    }
    else
    {
        // Make normal rotations
        Rmat[rotCol * nCols + (curBlock)]
        = rotCos * locR1 + rotSin * locR2;
        BufferMat[rotRow * nCols + (curBlock) ]
        = rotCos * locR2 - rotSin * locR1;
    }
}

////////////////////////////////////////////////////////////////////////
/// End of s_full_rotations
////////////////////////////////////////////////////////////////////////

/* s_partial_rotations
 ================

OpenCL kernel function for performing partial rotations.
The rows to rotate are given by the work-group id.
Every work-group is responsible of rotating one row.

Arguments
BufferMat  OpenCL buffer for buffer matrix
firstRow   First row to rotate
firstCol   First column to rotate
nCols      Number of columns in R matrix
colOffset  Column offset
*/
__kernel void s_partial_rotations(__global float * BufferMat,
  int firstRow,
  int firstCol,
  int nCols,
  int colOffset)
{
  // Get the information of the current work-item
  int currentRotation = get_group_id(0);
  int lid = get_local_id(0);
  int locSize = get_local_size(0);

  // Use local memory to store rotation sine and cosine
  local float rotCos;
  local float rotSin;

  // Use private memory to store the matrix elements that this work-item rotates
  float locR1;
  float locR2;

  // Use local memory to store the skip and swap flags
  local int skip;
  local int swap;

  // Use local memory to store the indices of rotated row and column
  local int rotRow, rotCol;

  // Initialize skip and swap flags
  skip = 0;
  swap = 0;

  // If this is work-item number 0, calculate the rotation parameters. Otherwise, do nothing.
  if (lid == 0)
  {
    // Find the row and starting column for this work-group
    rotRow = firstRow - currentRotation;
    rotCol = firstCol + currentRotation;

    // Calculate the Givens rotation coefficients
    float a, b, sqab;

    // Fetch the first elements from buffer matrix rows
// Note the offset
a = BufferMat[rotCol * nCols + rotCol + colOffset];
b = BufferMat[rotRow * nCols + rotCol + colOffset];
// auxiliary variable
sqab = sqrt(a*a + b*b);

if (sqab < 0.00001f || fabs(b) < 0.00001f)
{
    // If b is zero, skip the rotation
    rotCos = 1.0f;
    rotSin = 0.0f;
    skip = 1;
}
else if (fabs(a) < 0.00001f)
{
    // If a is zero, swap the rows
    rotCos = 0.0f;
    rotSin = b/sqab;
    swap = 1;
}
else
{
    // Calculate normal
    // rotation coefficients
    rotCos = a/sqab;
    rotSin = b/sqab;
}

// All other work-items wait here until
// work-item 0 has calculated the rotation
// parameters.
barrier(CLK_LOCAL_MEM_FENCE);

// If skip flag is set just return
if (skip)
{
    // return;
}

int curBlock;

// Make rotations starting from element curBlock,
// jumping locSize elements forward till all done
for(curBlock = (rotCol + colOffset) / 32 * 32 + lid;
curBlock < nCols;
curBlock+=locSize)
{

// Fetch the elements to be rotated
locR1 = BufferMat[rotCol * nCols + (curBlock)];
locR2 = BufferMat[rotRow * nCols + (curBlock)];

if (swap)
{
    // If swap flag is set, swap the elements
    // Note that they still are multiplied by rotSin
    BufferMat[rotCol * nCols + (curBlock)]
        = rotSin * locR2;
    BufferMat[rotRow * nCols + (curBlock)]
        = -rotSin * locR1;
}
else
{
    // Make normal rotations
    BufferMat[rotCol * nCols + (curBlock)]
        = rotCos * locR1 + rotSin * locR2;
    BufferMat[rotRow * nCols + (curBlock)]
        = rotCos * locR2 - rotSin * locR1;
}

///////////////////////////////////////////////////////
///// End of s_partial_rotations
///////////////////////////////////////////////////////

cKernelSource

// Define macros for complex multiplication
// Real and imaginary parts
// (a + ib) * (x + iy)
#define cmult_r(a,b,x,y) ((a)*(x) - (b)*(y))
#define cmult_i(a,b,x,y) ((a)*(y) + (b)*(x))

/* c_full_rotations
 ==***********
 OpenCL kernel function for performing full rotations.
The rows to rotate are given by the work-group id.
Every work-group is responsible of rotating one row.
Arguments

# Handle rotations

```c
__kernel void c_full_rotations(__global float* Rmat_r, __global float* Rmat_i, __global float* BufferMat_r, __global float* BufferMat_i, int firstRow, int firstCol, int nCols)
{
  // Get the information of the current work-item
  int currentRotation = get_group_id(0);
  int lid = get_local_id(0);
  int locSize = get_local_size(0);

  // Use local memory to store rotation sine and cosine
  local float rotCos_r;
  local float rotCos_i;
  local float rotSin_r;
  local float rotSin_i;

  // Use private memory to store the matrix elements that this work-item rotates
  float locR1_r;
  float locR1_i;
  float locR2_r;
  float locR2_i;

  // Use local memory to store the skip and swap flags
  local int skip;
  local int swap;
  local int skiprot;

  // Use local memory to store the indices of rotated row and column
  local int rotRow, rotCol;

  // Initialize skip and swap flags
  skip = 0;
}```
swap = 0;
skiprot = 0;

// If this is work-item number 0, calculate
// the rotation parameters. Otherwise, do
// nothing.
if (lid == 0)
{
    // Find the row and starting column for
    // this work-group
    rotRow = firstRow - currentRotation;
    rotCol = firstCol + currentRotation;

    // Calculate the Givens rotation
    // coefficients
    float a_r, a_i, b_r, b_i, sqab, modA, modB, tmpA, tmpB;

    // Fetch the first elements from R and buffer matrices
    a_r = Rmat_r[rotCol * nCols + rotCol];
    a_i = Rmat_i[rotCol * nCols + rotCol];
    b_r = BufferMat_r[rotRow * nCols + rotCol];
    b_i = BufferMat_i[rotRow * nCols + rotCol];

    // auxiliary variables
    tmpA = a_r*a_r + a_i*a_i;
    tmpB = b_r*b_r + b_i*b_i;
    sqab = sqrt(tmpA + tmpB);
    modA = sqrt(tmpA);
    modB = sqrt(tmpB);

    if (sqab < 0.00001f)
    {
        // if both elements are zero, skip rotations
        skip = 1;
    }
    else if (modB < 0.00001f)
    {
        // If b is zero, no rotations,
        // but a must be scaled
        rotCos_r = a_r/modA;
        rotCos_i = -a_i/modA;
        rotSin_r = 0.0f;
        rotSin_i = 0.0f;
        skiprot = 1;
    }
    else if (modA < 0.00001f)
    {
        // If a is zero, swap rows
        // and scale b
rotCos_r = 0.0f;
rotCos_i = 0.0f;
rotSin_r = b_r/modB;
rotSin_i = -b_i/modB;
swap = 1;
else {
    // Calculate normal rotation coefficients
    rotCos_r = a_r/sqab;
    rotSin_r = b_r/sqab;
    rotCos_i = -a_i/sqab;
    rotSin_i = -b_i/sqab;
}

// All other work-items wait here until
// work-item 0 has calculated the rotation
// parameters.
barrier(CLK_LOCAL_MEM_FENCE);

// If skipped, just return
if (skip) {
    return;
}

int curBlock;

// Make rotations starting from element curBlock,
// jumping locSize elements forward till all done
for (curBlock = rotCol / 32 * 32 + lid;
    curBlock < nCols;
    curBlock += locSize)
{
    // Fetch the elements to be rotated
    locR1_r = Rmat_r[rotCol * nCols + curBlock];
    locR1_i = Rmat_i[rotCol * nCols + curBlock];
    locR2_r = BufferMat_r[rotRow * nCols + curBlock];
    locR2_i = BufferMat_i[rotRow * nCols + curBlock];

    if (swap) {
        // If swap flag is set, swap the elements
        // Note that they still are scaled
        // by rotSin
        Rmat_r[rotCol * nCols + curBlock]
            = cmult_r(rotSin_r, rotSin_i, locR2_r, locR2_i);
else if (skiprot)
{
    // No rotations but scale by rotCos
    Rmat_r[rotCol * nCols + curBlock]
        = cmult_r(rotCos_r,rotCos_i,locR1_r,locR1_i);
    Rmat_i[rotCol * nCols + curBlock]
        = cmult_i(rotCos_r,rotCos_i,locR1_r,locR1_i);

    BufferMat_r[rotRow * nCols + curBlock]
        = cmult_r(-rotSin_r,rotSin_i,locR2_r,locR2_i);
    BufferMat_i[rotRow * nCols + curBlock]
        = cmult_i(-rotSin_r,rotSin_i,locR2_r,locR2_i);
}
else
{
    // Make standard Givens rotations
    Rmat_r[rotCol * nCols + curBlock]
        = cmult_r(rotCos_r,rotCos_i,locR1_r,locR1_i)
        + cmult_r(rotSin_r,rotSin_i,locR2_r,locR2_i);
    Rmat_i[rotCol * nCols + curBlock]
        = cmult_i(rotCos_r,rotCos_i,locR1_r,locR1_i)
        + cmult_i(rotSin_r,rotSin_i,locR2_r,locR2_i);

    BufferMat_r[rotRow * nCols + curBlock]
        = cmult_r(rotCos_r,-rotCos_i,locR2_r,locR2_i)
        + cmult_r(-rotSin_r,rotSin_i,locR1_r,locR1_i);
    BufferMat_i[rotRow * nCols + curBlock]
        = cmult_i(rotCos_r,-rotCos_i,locR2_r,locR2_i)
        + cmult_i(-rotSin_r,rotSin_i,locR1_r,locR1_i);
}
}
OpenCL kernel function for performing full rotations. The rows to rotate are given by the work-group id. Every work-group is responsible of rotating one row.

Arguments

BufferMat_r OpenCL buffer for buffer matrix, real part
BufferMat_i OpenCL buffer for buffer matrix, imag part
firstRow First row to rotate
firstCol First column to rotate
nCols Number of columns in R matrix
colOffset Column offset

*/

__kernel void c_partial_rotations(__global float* BufferMat_r,
__global float* BufferMat_i,
int firstRow,
int firstCol,
int nCols,
int colOffset)
{
    // Get the information of the current work-item
    int currentRotation = get_group_id(0);
    int lid = get_local_id(0);
    int locSize = get_local_size(0);

    // Use local memory to store
    // rotation sine and cosine
    local float rotCos_r;
    local float rotCos_i;
    local float rotSin_r;
    local float rotSin_i;

    // Use private memory to store
    // the matrix elements that this work-item rotates
    float locR1_r;
    float locR1_i;
    float locR2_r;
    float locR2_i;

    // Use local memory to store the skip and swap flags
    local int skip;
    local int swap;
    local int skiprot;

    // Use local memory to store the indices of
    // rotated row and column
local int rotRow;

// Initialize skip and swap flags
skip = 0;
swap = 0;
skiprot = 0;

// If this is work-item number 0, calculate
// the rotation parameters. Otherwise, do
// nothing.
if (lid == 0)
{
    // Find the row and starting column for
    // this work-group
    rotRow = firstRow - currentRotation;
    rotCol = firstCol + currentRotation;

    // Calculate the Givens rotation
    // coefficients
    float a_r, a_i, b_r, b_i, sqab, modA, modB, tmpA, tmpB;
    // Fetch the first elements from buffer matrix rows
    // Note the offset
    a_r = BufferMat_r[rotCol * nCols + rotCol + colOffset];
    a_i = BufferMat_i[rotCol * nCols + rotCol + colOffset];
    b_r = BufferMat_r[rotRow * nCols + rotCol + colOffset];
    b_i = BufferMat_i[rotRow * nCols + rotCol + colOffset];

    // auxiliary variable
    tmpA = a_r*a_r + a_i*a_i;
    tmpB = b_r*b_r + b_i*b_i;
    sqab = sqrt(tmpA + tmpB);
    modA = sqrt(tmpA);
    modB = sqrt(tmpB);

    if (sqab < 0.00001f)
    {
        // If a and b are zeroes, skip rotation
        skip = 1;
    }
    else if (modB < 0.00001f)
    {
        // If b is zero, do not rotate, just
        // scale by rotCos
        rotCos_r = a_r/modA;
        rotCos_i = -a_i/modA;
        rotSin_r = 0.0f;
        rotSin_i = 0.0f;
        skiprot = 1;
    }
    else if (modA < 0.00001f)
{  // If a is zero, swap rows  // and scale by rotSin  rotCos_r = 0.0f;
  rotCos_i = 0.0f;
  
  rotSin_r = b_r/modB;
  rotSin_i = -b_i/modB;
  swap = 1;
}

else
{
  // Calculate normal  // rotation coefficients  rotCos_r = a_r/sqab;
  rotCos_i = -a_i/sqab;
  rotSin_r = b_r/sqab;
  rotSin_i = -b_i/sqab;
}

// All other work-items wait here until  // work-item 0 has calculated the rotation  // parameters.
barrier(CLK_LOCAL_MEM_FENCE);

if (skip)
{
  return;
}

int curBlock;

// Make rotations starting from element curBlock,  // jumping locSize elements forward till all done
for(curBlock = (rotCol + colOffset) / 32 * 32 + lid; curBlock < nCols; curBlock += locSize)
{
  // Fetch the elements to be rotated  locR1_r = BufferMat_r[rotCol * nCols + curBlock];
  locR1_i = BufferMat_i[rotCol * nCols + curBlock];
  locR2_r = BufferMat_r[rotRow * nCols + curBlock];
  locR2_i = BufferMat_i[rotRow * nCols + curBlock];

  if (swap)
  {
    // If swap flag is set, swap the elements
    // Note that they still are multiplied
// by rotSin
BufferMat_r[rotCol * nCols + curBlock]
  = cmult_r(rotSin_r, rotSin_i, locR2_r, locR2_i);
BufferMat_i[rotCol * nCols + curBlock]
  = cmult_i(rotSin_r, rotSin_i, locR2_r, locR2_i);
BufferMat_r[rotRow * nCols + curBlock]
  = cmult_r(-rotSin_r, rotSin_i, locR1_r, locR1_i);
BufferMat_i[rotRow * nCols + curBlock]
  = cmult_i(-rotSin_r, rotSin_i, locR1_r, locR1_i);
}
else if (skipRot)
{
  // Skip rotations, but scale by rotCos
BufferMat_r[rotCol * nCols + curBlock]
  = cmult_r(rotCos_r, rotCos_i, locR1_r, locR1_i);
BufferMat_i[rotCol * nCols + curBlock]
  = cmult_i(rotCos_r, rotCos_i, locR1_r, locR1_i);
BufferMat_r[rotRow * nCols + curBlock]
  = cmult_r(-rotCos_r, rotSin_i, locR2_r, locR2_i);
BufferMat_i[rotRow * nCols + curBlock]
  = cmult_i(-rotCos_r, rotSin_i, locR2_r, locR2_i);
}
else
{
  // Make normal rotations
BufferMat_r[rotCol * nCols + curBlock]
  = cmult_r(rotCos_r, rotCos_i, locR1_r, locR1_i)
          + cmult_r(rotSin_r, rotSin_i, locR2_r, locR2_i);
BufferMat_i[rotCol * nCols + curBlock]
  = cmult_i(rotCos_r, rotCos_i, locR1_r, locR1_i)
          + cmult_i(rotSin_r, rotSin_i, locR2_r, locR2_i);
BufferMat_r[rotRow * nCols + curBlock]
  = cmult_r(rotCos_r, -rotCos_i, locR2_r, locR2_i)
          + cmult_r(-rotSin_r, rotSin_i, locR1_r, locR1_i);
BufferMat_i[rotRow * nCols + curBlock]
  = cmult_i(rotCos_r, -rotCos_i, locR2_r, locR2_i)
          + cmult_i(-rotSin_r, rotSin_i, locR1_r, locR1_i);
}

////////////////////////////////////////////////////////////////////////
///// End of c_partial_rotations
////////////////////////////////////////////////////////////////////////
3.2.6 File: cbacksolve.c

File cbacksolve.c contains code fold complex back substitution solver. For some reason, this is missing in R. For speed, it is implemented in C using R’s .Call external function protocol.

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<R.h>
#include<Rinternals.h>

#include "rlips.h"

#define CPLX_MULT_R(ar,ai,br,bi) ((ar)*(br) - (ai)*(bi))
#define CPLX_MULT_I(ar,ai,br,bi) ((ar)*(bi) + (ai)*(br))

/*
cbacksolve
==========
Backsubstitution solver for complex data
Arguments
R Complex upper triangular matrix
Y Complex matrix
Returns
```
/*
SEXP cbacksolve(SEXP R, SEXP Y)
{
  int i,j,k;
  SEXP res;

  // Allocate R data in C side
  Rcomplex *rr = COMPLEX(R);
  Rcomplex *yy = COMPLEX(Y);

  // get the dimensions of R and Y
  SEXP Rdim = getAttrib(R,R_DimSymbol);
  SEXP Ydim = getAttrib(Y,R_DimSymbol);

  // I  number of rows in Y
  // J  number of cols in Y
  int I = INTEGER(Ydim)[0];
  int J = INTEGER(Ydim)[1];

  // Allocate and protect the result matrix
  PROTECT(res = allocMatrix(CPLXSXP,I,J));

  // Copy matrix Y to array res
  for (i = 0; i < I * J ; i++)
    {
      COMPLEX(res)[i].r = yy[i].r;
      COMPLEX(res)[i].i = yy[i].i;
    }

  // Auxiliary variables
  double Ur,Ui,tmp;

  // This is standard (complex)
  // backsubstitution algorithm
  // Starting from the last row, go up to
  // the second row
  for (i = I - 1 ; i >= 1; i--)
    {
      Ur = rr[i + i * I].r;
      Ui = rr[i + i * I].i;

      // Handle every column of Y separately
      for (k = 0 ; k < J ; k++)
        {
          tmp = COMPLEX(res)[i + k * I].r;

        }

      //...
COMPLEX(res)[i + k * I].r = (CPLX_MULT_R(
    COMPLEX(res)[i + k * I].r,
    COMPLEX(res)[i + k * I].i,
    Ur,
    -Ui
  ))/(Ur*Ur + Ui*Ui);

COMPLEX(res)[i + k * I].i = (CPLX_MULT_I(
  tmp,
  COMPLEX(res)[i + k * I].i,
  Ur,
  -Ui
 ))/(Ur*Ur + Ui*Ui);

for (j = 0 ; j < i ; j++)
{
  COMPLEX(res)[j + k * I].r
  = COMPLEX(res)[j + k * I].r 
  - (CPLX_MULT_R(
    COMPLEX(res)[i + k * I].r,
    COMPLEX(res)[i + k * I].i,
    rr[j + i * I].r,
    rr[j + i * I].i
  ));

  COMPLEX(res)[j + k * I].i
  = COMPLEX(res)[j + k * I].i 
  - (CPLX_MULT_I(
    COMPLEX(res)[i + k * I].r,
    COMPLEX(res)[i + k * I].i,
    rr[j + i * I].r,
    rr[j + i * I].i
  ));
}

// Handle the first row separately
for (k = 0 ; k < J ; k++)
{
  tmp = COMPLEX(res)[k * I].r;
  COMPLEX(res)[k * I].r = (CPLX_MULT_R(
    COMPLEX(res)[k * I].r,
    COMPLEX(res)[k * I].i,
    rr[0].r,
    -rr[0].i
  ))/(rr[0].r * rr[0].r 
    + rr[0].i * rr[0].i);
COMPLEX(res)[k * I].i = CPLX_MULT_I(
    tmp,
    COMPLEX(res)[k * I].i,
    rr[0].r,
    -rr[0].i
) / (rr[0].r * rr[0].r
    + rr[0].i * rr[0].i);

// Unprotect the result and return it
UNPROTECT(1);
return res;
Chapter 4

User manual

4.1 Installation

Rlips package is a standard R package but it requires that OpenCL libraries and header files are installed in the computer. If this is the case and the OpenCL libraries and header files are installed in the standard locations (In Mac OS X, the OpenCl libraries are installed by default. In Linux using NVIDIA provided OpenCL libraries (as part of CUDA distribution), the header files are in /usr/local/cuda/include), the installation is done by giving the standard command in a terminal window:

R CMD INSTALL rlips

Note that it might be required to add sudo to the above command or make the installation as root!

If NVIDIA's OpenCL is not installed, or it is installed in some non-standard location, it is necessary to modify the Makevars file in rlips/src directory. After that the installation is performed as above.

Unfortunately, Windows setups are not supported at the moment.

4.2 Loading package in R

After rlips package is installed, it is loaded into R using standard command(s):

library(rlips)
4.3 Using rlips package

All rlips related R commands have prefix rlips. All rlips commands are explained in this section.

Typical use of rlips is the following:

1. New inverse problem is initialized by rlips.init. This command creates a new R environment that contains all necessary data for solving the problem.

2. The actual data (theory matrix, measurements and error (co)variances) are fed into rlips by using the command rlips.add. Note that the data can be given in pieces.

3. Data is then rotated using Givens rotations into the target matrix and vector using rlips.rotate. Note that this command is not normally needed to be used by the user. Rlips is capable of handling the necessary rotations automatically.

4. After all the data is fed in, the problem is solved using rlips.solve. The solution vector and (optionally) the posterior covariance matrix are written in the rlips environment that was used for the problem.

5. Finally, the used rlips environment is disposed using rlips.dispose.
4.3.1 r lips.init

Description:

Initializes a new r lips environment.

Usage:

```r
rlips.init(ncols, nrhs, type='s', nbuf=ncols, workgroup.size=128)
```

Arguments:

- `ncols`: Integer giving the number of unknowns, i.e. the number of columns in the theory matrix.
- `nrhs`: Integer giving the number of columns in the measurement matrix.
- `type`: Problem numerical type. One of the following character strings:
  - 's' Single precision real
  - 'c' Single precision complex
- `nbuf`: Size (number of rows) of the rotation buffer. Default is the number of unknowns, i.e. default rotation buffer is of the same size as the problem after rotations. Size of the buffer can have a significant impact on performance. The optimal size of the buffer depends on the problem and also on computer hardware. Small buffers conserve computer memory but tend to slow r lips down.
- `workgroup.size`: Size of the OpenCL workgroup. Optimal size depends on GPU hardware. Usually, This should be of form $2^n$, where $n = 2,3,...,10$. Typical optimal value for NVIDIA GPU’s is 128 and for AMD GPU’s 256. Your mileage may vary.

Value:

R environment. This environment contains a number of internal variables needed by ‘r lips’. After the r lips problem is solved (see ‘r lips.solve’) it also contains the solution and (if calculated) posteriori covariance matrix.
4.3.2  rlips.dispose

Description:

Deletes the rlips enviroment’s internal variables and the files used by the eniron-
ment.

Usage:

rlipsdispose(e)

Arguments:

e:  rlips environment to be deleted

Details:

'rlips.dispose' will delete all variables in rlips environment 'h'.

Value:

None. ‘e’ will be an empty environment afterwards.
4.3.3  rllips.add

Description:

Add data (theory matrix, measurements and errors) into the RLIPS system row(s)-by-row(s).

Usage:

rlips.add(e,A.data,M.data,E.data=1)

Arguments:

e:  Initialized RLIPS environment
A.data:  Matrix or vector containing the theory matrix rows.
M.data:  Matrix or vector containing the measurement matrix rows.
E.data:  Vector or scalar containing the measurement errors (variances), or the full error covariance matrix. See details.

Details:

Theory matrix rows (‘A.data’) can be given as a vector (row-wise) or as a matrix.

If given as a vector, the size of the vector must be ‘num.rows+ncols’, where ‘ncols’ is the number of unknowns in the problem (i.e. the number of columns in the theory matrix) and ‘num.rows’ is the number of rows in the theory matrix ‘A.data’.

If given as a matrix, its dimensions must be ‘c(num.rows,ncols)’.

In the same way, the measurements can be given as a vector or a matrix. If given as a vector, its size must be ‘num.rows+nrhs’, where ‘nrhs’ is the number of columns in the measurement vector. If it is given as a matrix its size must be ‘c(num.rows,nrhs)’.

The error variances can be given as a vector (size ‘num.rows’) or if the error variance is constant for all measurements added into RLIPS, it can also be given as a single scalar value. If error is omitted scalar value 1.0 will be used.
If the full error covariance matrix is given, it must be a symmetric positive-definite square matrix with size ‘num.rows’-by-‘num.rows’.

**Value:**

None.
### 4.3.4 rlips.solve

**Description:**

Solve the problem fed into rlips environment.

**Usage:**

rlips.solve(e, calculate.covariance=FALSE, full.covariance=FALSE)

**Arguments:**

- **e**: An existing rlips environment with enough data fed in.
- **calculate.covariance**: Flag for calculating the posteriori covariance matrix.
- **full.covariance**: Flag for calculating the full posteriori covariance matrix.

**Details:**

The calculation of the posteriori covariance matrix is controlled by the two flags 'calculate.covariance' and 'full.covariance'.

If 'calculate.covariance=FALSE' the posteriori covariance matrix is not calculated at all.

If 'calculate.covariance=TRUE', then if 'full.covariance=TRUE' the full a posteriori covariance matrix is calculated.

If 'full.covariance=FALSE', only the a posteriori variances (i.e. the diagonal of the a posteriori covariance matrix) is calculated. This is much faster than calculating the full matrix!

**Value:**

None. The solution is written into rlips environment variable ‘e$solution’. The a posteriori covariance matrix (if calculated) is written into environment variable ‘e$covariance’.
Note:

Rlips does not check that there exists enough data in the problem, nor that the problem is solvable (i.e. there exists enough linearly independent data rows in the theory matrix).

This routine calls first rlips.get.data, which fetches the rotated system from the GPU. The rotated upper triangular target matrix $R$ is put into `$eR.mat$’ and the corresponding rotated target vector into `$eY.mat$’.
4.3.5 rlips.rotate

Description:

Rotates all unrotated data of a rlips environment

Usage:

rlips.rotate(e)

Arguments:

e: Existing rlips environment with unrotated data

Details:

'rlips.rotate' forces the Givens rotations on unrotated rlips data. Especially for large problems this can reduce the memory consumption and solution time remarkably. On the other hand, overusing this command will decrease the performance.

The size of the rotation buffer can be controlled by 'buffersize'. The default 'ncols' should be good for most situations.

Value:

None. The unrotated data is rotated into rlips system.

Note:

This routine is called automatically by RLIPS when necessary. It is provided as a user command for special and debugging purposes.
rlips.get.data

Description:

Fetches the target matrix R and the target vector Y from the GPU.

Usage:

rlips.get.data(e)

Arguments:

e:  Initialized RLIPS environment

Details:

The rotated system matrices are fetched from the GPU. The upper triangular target matrix R is put into ‘e$R.mat’ and the corresponding target vector Y is put into ‘e$Y.mat’.

Note that this routine is called every time the command ‘rlips.solve’ is used, so there is no reason to use this in normal operation.

Value:

None.
4.3.6  rlips.copy

Description:
Copy RLIPS environment into a new environment

Usage:
rlips.copy(oenv)

Arguments:

\texttt{oenv}: RLIPS environment

Value:
RLIPS environment which is a copy of ‘\texttt{oenv}’.
4.3.7 rlips.test

Description:

Simple test to check the performance of RLIPS.

Usage:

rlips.test(type, size, buffersize=size[2], loop=1,
            wg.size=128, return.data=FALSE,
            averaging.fun=mean)

Arguments:

type: Type of the problem. ‘s’ for single precision real, or ‘c’ for single precision complex.

size: 2-vector holding the size (rows,columns) of the test problem.

buffersize: RLIPS rotation buffersize. The default is the number of column-unknowns of the problem.

loop: Number of tests performed. Default is 1. If ‘loop’ is larger than 1, the results are averaged using the function ‘averaging.fun’.

wg.size: OpenCL workgroup size.

return.data: If ‘TRUE’, the rotated system matrices are returned.

averaging.fun: Function used in averaging the results, if ‘loop’ is greater than 1. Default is arithmetic mean.

Details:

This test routine can (and should) be used to trim the rotation buffer and OpenCL workgroup sizes for the used hardware.

Value:

Returns a list with components:
time Elapsed time for solving the problem using RLIPS.

accuracy Maximum absolute error.

Gflops Approximate Gigaflops (floating point operations per second).

R If ‘return.data=TRUE’, the rotated upper triangular target matrix.

Y If ‘return.data=TRUE’, the rotated target vector.

Examples:

Test the performance by solving a real-valued overdetermined problem of size 5000 × 1000 using 2500-row rotation buffer and OpenCL workgroup size 256:

```r
rlips.test('s',c(5000,1000),2500,1,256)
```

Test the performance by solving a complex-valued problem of size 1000 × 1000 using 1000-row buffer and 128 workgroup size. Perform the test 10 times and return the median of the results:

```r
rlips.test('c',c(1000,1000),1000,10,256,
           averaging.fun=median)
```
4.4 Example of use

In this section we give a simple example how to solve a linear inverse problem using RLIPS.

Let us consider a deconvolution problem where the measurement data \( m \) is a noisy convolution of an unknown signal \( s \) with a known convolution kernel \( k \), in other words,

\[
m = k * s + \varepsilon,
\]

where the noise term \( \varepsilon \) is Gaussian, i.e. \( \varepsilon \sim \mathcal{N}(0, \Sigma) \).

For this example, we set the signal \( s \) to be a vector of length 500 presenting the discrete values of a function plotted in Figure 4.1.

As for the convolution kernel \( k \) we take a vector of length 100, \( k = (k_1, k_2, \ldots, k_{100}) \) such that \( k_i = 1/100, \ i = 1, \ldots, 100 \). In other words, convolution of any vector with the kernel \( k \) is a running average of length 100. Moreover, we set that the noise covariance \( \Sigma = \sigma^2 I \), where \( \sigma^2 = 0.0001 \).

Now we can create the noisy data for the example problem by

\[
m = \text{convolve}(s, k, \text{type}='o') + \sqrt{0.0001} \ast \text{rnorm}(599).
\]

Measurement vector \( m \) has length 599.

One realization of noisy data is plotted in Figure 4.2.
In order to use RLIPS for the deconvolution, we need a matrix equation of the convolution. It is fairly easy to see that the convolution equation (4.1) can be written as

\[ m = Ax + \varepsilon, \]

where the convolution matrix \( A \) is

\[
A = \begin{pmatrix}
  k_1 & 0 & \cdots & \cdots & 0 \\
  k_2 & k_1 & 0 & \cdots & 0 \\
  \vdots & k_2 & \ddots & \vdots & \vdots \\
  k_{100} & \vdots & \ddots & \ddots & 0 \\
  0 & k_{100} & \ddots & k_1 & \vdots \\
  \vdots & 0 & \ddots & k_2 & \vdots \\
  \vdots & \vdots & \ddots & \vdots & \vdots \\
  0 & 0 & \cdots & 0 & k_{100}
\end{pmatrix}.
\]

The convolution matrix \( A \in \mathbb{R}^{599 \times 500} \), so the matrix equation is overdetermined. The matrix can be constructed in R using the following code:

```r
k <- rep(0.01,100)
A <- matrix(0,599,500)
for (i in 1:500) A[i:(i+99),i] <- k
```

Now we can try to solve the deconvolution problem using RLIPS. First, make sure that RLIPS is loaded in R:
require(rlips)

Next, initialize a new RLIPS problem. Now we have real problem with 500 unknowns and one measurement vector:

h <- rlips.init(500,1,’s’)

We could also have defined rotation buffer and OpenCL workgroup sizes here, but we left them in their defaults. Note that it is not necessary to define the number of the rows in the theory matrix \( A \) or the length of the measurement vector \( m \! \)!

Next we need to feed the data in the problem. Note that this can be done by feeding all the data at once or by feeding it in parts. Here we feed all the data at once:

rlips.add(h,A,m,0.0001)

In above \( h \) is the handle for the problem we initialized earlier, \( A \) is the theory matrix, \( m \) is the (noisy) measurement vector and 0.01 is the error variance. Note that in practise the exact error variance is not known, but it must be estimated.

An alternative is to feed the data in chunks (although for such a small and simple problem it really doesn not make much sense). Here we feed the data so that first the first 100 rows are fed in, then the next 400 rows and finally the remaining 99 rows are fed one-by-one. Again, this is just an example. There is no real need to do this for this kind of problem!

rlips.add(h,A[1:100,,]m[1:100],0.0001)
rlips.add(h,A[101:500,,]m[101:500],0.0001)
for (i in 501:599) rlips.add(h,A[i,,]m[i],0.0001)

Now all the data is fed in and we can try to solve the problem. This is done simply by

rlips.solve(h).

In above, only the solution (in the least square sense) is calculated and it is put into \( h\$solution \). However, it is well-known that deconvolution problems are severely ill-posed. In Figure 4.3 is plotted the solution given by RLIPS (blue) and the true solution, i.e. the signal \( s \), (in red). As can be seen, the solution is just rubbish. It is clear that some kind of regularization is necessary. Note that even that we have calculated the solution, all data that is fed in the system are still available (in rotated form).

Here we first try the most simple regularization scheme available, namely using

require(rlips)

Next, initialize a new RLIPS problem. Now we have real problem with 500 unknowns and one measurement vector:

h <- rlips.init(500,1,’s’)

We could also have defined rotation buffer and OpenCL workgroup sizes here, but we left them in their defaults. Note that it is not necessary to define the number of the rows in the theory matrix \( A \) or the length of the measurement vector \( m \! \)!

Next we need to feed the data in the problem. Note that this can be done by feeding all the data at once or by feeding it in parts. Here we feed all the data at once:

rlips.add(h,A,m,0.0001)

In above \( h \) is the handle for the problem we initialized earlier, \( A \) is the theory matrix, \( m \) is the (noisy) measurement vector and 0.01 is the error variance. Note that in practise the exact error variance is not known, but it must be estimated.

An alternative is to feed the data in chunks (although for such a small and simple problem it really doesn not make much sense). Here we feed the data so that first the first 100 rows are fed in, then the next 400 rows and finally the remaining 99 rows are fed one-by-one. Again, this is just an example. There is no real need to do this for this kind of problem!

rlips.add(h,A[1:100,,]m[1:100],0.0001)
rlips.add(h,A[101:500,,]m[101:500],0.0001)
for (i in 501:599) rlips.add(h,A[i,,]m[i],0.0001)

Now all the data is fed in and we can try to solve the problem. This is done simply by

rlips.solve(h).

In above, only the solution (in the least square sense) is calculated and it is put into \( h\$solution \). However, it is well-known that deconvolution problems are severely ill-posed. In Figure 4.3 is plotted the solution given by RLIPS (blue) and the true solution, i.e. the signal \( s \), (in red). As can be seen, the solution is just rubbish. It is clear that some kind of regularization is necessary. Note that even that we have calculated the solution, all data that is fed in the system are still available (in rotated form).

Here we first try the most simple regularization scheme available, namely using
White Noise prior, i.e. using prior

\[ X \sim \mathcal{N}(0, \gamma^2 I). \]

This can be modelled as a matrix equation

\[ 0 = Ix + \eta, \quad \eta \sim \mathcal{N}(0, \gamma^2 I). \]

Now we can add this prior data to the previous problem. Before that let us make a copy of the problem, so we can try different priors, if necessary:

\[ \text{h.original} \leftarrow \text{rlips.copy(h)} \]

Now the R environment \text{h.original} contains all the data the environment \text{h} contains, so if we ever have to come back to the problem at this stage, we can just make new copies of \text{h.original}.

Now, fed in the prior. Set (guess) the prior model variance \( \gamma^2 = 0.1 \). We need also first construct the identity matrix \( I \in \mathbb{R}^{500 \times 500} \):

\[ I \leftarrow \text{diag(rep(1, 500))} \]
\[ \text{rlips.add(h, I, rep(0, 500), 0.1)} \]

Note that we also needed to construct the zero vector of length 500 as the measurement vector in above RLIPS command. Next, let us solve the problem again:

\[ \text{rlips.solve(h)}. \]
In Figure 4.4 is plotted the solution using White Noise prior (blue) and the true solution (red). The solution is still not very good, but it is a huge improvement to the original one.

Next we are going to try a more smoothing prior, namely the so called first order difference prior. This prior can be represented as a stochastic model

$$0 = L_1 X + \eta, \quad \eta \sim \mathcal{N}(0, \gamma^2 I),$$

where the matrix $L_1$ is given as

$$L_1 = \begin{pmatrix}
1 & 1 \\
-1 & 1 \\
-1 & 1 \\
\vdots & \vdots \\
-1 & 1 \\
\end{pmatrix},$$

i.e. the diagonal consists of 1’s and the first subdiagonal consists of -1’s.

In order to not having to start all over again, we first copy the problem containing the original measurement data into a new one:

```r
h2 <- rlips.copy(h.orig)
```

Then we add the difference prior model with variance $\gamma^2 = 0.005$ and solve:

```r
rlips.add(h2, L1, rep(0, 500), 0.005)
rlips.solve(h2)
```
This solution is plotted in Figure 4.5.

If we knew a priori that the unknown signal has discontinuities at certain points, we could insert that information in our prior model. Namely, we can give those points a much larger variance in the prior model. Let us assume that we happen to know that the discontinuities are at points 201, 301 and 401. Then we can give the prior model variances as a vector representing the diagonal of the prior model covariance matrix and set the variances of the discontinuation points much larger:

```r
gamma = rep(0.005,500)
gamma[c(201,301,401)] <- 10000
```

Here the discontinuity points have prior variance of 10000, and all the other points have variance of 0.005. Again, we first copy the problem containing the original data into a new one. Then we add the prior model and solve:

```r
h3 <- rlips.copy(h.orig)
rlips.add(h3,L1,rep(0,500),gamma) rlips.solve(h3)
```

The result is plotted in the Figure 4.6.

To finish this example, we dispose the used RLIPS environments:

```r
rlips.dispose(h)
rlips.dispose(h.orig)
rlips.dispose(h2)
rlips.dispose(h3)
```
Figure 4.6: Solution using First Order Difference prior taking account of the discontinuity points.
Appendix B

R routines for beam-forming
R routines for beamforming

EISCAT_3D, Work Package 11, Task 11.2

Mikko Orispää and Markku Lehtinen
Sodankylä Geophysical Observatory
University of Oulu, Finland
Licenses

File ML_filter.r

Copyright 2013–2014 Markku Lehtinen. All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

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

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

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of Markku Lehtinen.
**Files** `beamformer.r` and `bf_simu.r`

Copyright 2014 University of Oulu, Finland. All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

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

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

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of University of Oulu.
Chapter 1

Beamforming routines

This document lists R source code which can be used to perform beamforming as described in the Final Report of the Work Package 11 of EISCAT_3D project and the Radar Handbook written by Markku Lehtinen.

1.1 File ML_filter.r

This file contains code for constructing beamforming filters described in the Radar Handbook written by Markku Lehtinen.

```r
# Filter design routine by searching a minimum of total
# energy of filter coeffs longer than a specified
# half-length. Pass band response required to be exactly
# what is specified as well as stop band exactly 0 if all
# coeffs retained. Frequency response freely chosen in
# ramp intervals to find the minimum of long filter coeffs.
#
# Copyright Markku S. Lehtinen 2013. All rights reserved.
# Licensed under freeBSD license.

# Auxillary functions

# circular distance of p and q modulo len
Cdist <- function(p,q,len){
  return( Mod(((p-q+len/2) %% len) - len/2) )
}

# circular position of p from q modulo len
Crel <- function(p,q,len){
```
return( ((p-q+len/2) %% len) - len/2)
#
boxcar for fft'ing
bcarv <- function(bcar,N){
  return(pmax(pmin(bcar/2+0.5-Cdist(1:N,1,N),1),0)/bcar)
}
#
Main filter routine
makefilter <- function(
  smp = 84, # input data sample frequency (in MHz -- or anything, but then GMAC figures not correct)
  cnt=233, # input center frequency - aliasing and undersampling allowed and taken care of internally
  hBW=15, # half bandwidth of pass band or band of required proper operation
  ramp= 6, # free-form band around pass band. Outside # cnt+-(hBW+|ramp|) response put to 0.
  # Negative value implies real input and response forced to 0 at mirror of pass band.
  # In this case GMAC calculation assumes real
  # input and decimation by 2
  sd=10, # frequency step subdivision, may be necessary
  # to increase
  hlen=1:1000, # half length of ‘short’ filter
  # coefficients.
  # Total no 2*hlen+1. If a vector, each value is tried until total power of long filter
  # coefficients goes below goal
  tdfrac=0, # time delay fraction (of sample step)
  goal=-90, # goal below which to put long filter coeffhs
  # relative total power (dB)
  dec=smp/floor(smp/ifelse(2*hBW+Mod(ramp)>smp,
    smp,
    2*hBW+Mod(ramp))));
  # decimated output frequency
  bcar=1 # boxcar integration
}{
  # keep all parameters and function locals in filter$xxx
  filter <- new.env()
  filter <- parent.env(filter)
  with(filter,{
    # length of fft:s
    N <- smp*sd
    # time axis
    f <- (1:N-1) / sd; b <-NA *f
    # stop band
A0 <- Cdist(f,cnt%%smp,smp)>=hBW+Mod(ramp)
# If real data, add mirror of pb to stop band
if (ramp<=0) A0 <- A0 | (Cdist(f,-cnt%%smp,smp)<=hBW)
# pass band
A1 <- Cdist(f,cnt%%smp,smp)<=hBW
# pass band freq response
b[A1] <- exp(2i*pi*(Crel(f[A1],cnt%%smp,smp)+cnt) * tdfrac/smp)
# stop band response
b[A0] <- 0
# boxcar shape impulse response
b <- b*fft(bcarv(bcar,N)*exp(2i*pi*f*cnt*sd/smp))
# band where freq response free
B <- !(A0 | A1)
# band of fixed values
A <- (A0 | A1)

respow <- NULL
# increase filter halflength until goal achieved
# for total power of long coeffs
for (i in hlen) {
    # support of short filter coeffs
    C <- rep(FALSE,N)
    C[1:(i+1)] <- TRUE
    C[(N-i+1):N] <- TRUE
    # fft of support of long coeffs
    fftCfun <- function(p,r){ fft((!C)[(p-r)%%N+1]) }
    # matrix of minimum condition
    Amat <- outer(which(B),which(B),fftCfun)
    # rhs of minimum condition
    rhs <- -outer(which(B),which(A1),fftCfun) %*%
    b[which(A1)]
    # f response solution minimizing long coeffs
    b[B] <- solve(Amat,rhs)
    # filter coeffs (filter taps)
    c <- fft(b,inverse=TRUE)/N
    # power of long coeffs
    respow <- c(respow,
                sum(Mod(c[!C])^2)/sum(Mod(c)^2))
    # goal for power of long coeffs achieved
    if (respow[length(respow)] < 10^(goal/10)) break
}
# truncated freq responce
bt <- fft( ifelse(C,c,0) )
# aliased response
ba<-rep(NA,N)
for (i in 1:N){
    if(Cdist(f[i],cnt,smp)<=dec/2){
        bt[i] <- bt[i] * ifelse(C,c,0)
        ba[i] <- NA
    }
}
### 1.2 File beamformer.r

This file contains routines to perform beamforming for given (simulated) data that is residing in computer memory. For real data beamforming one must modify the code to handle the data streams containing the real samples from antennas.

```r
# One-stage beamformer
# Part of WP11 of EISCAT_3D project
# Copyright (c) University of Oulu, 2014.
# Written by Mikko Orispaa <mikko.orispaa@oulu.fi>
# Licensed under FreeBSD license.

library(signal, warn.conflicts = FALSE)
```

```r
ba[i] <- sum(bt[Cdist(f,f[i],dec)<1e-9])
```
# NB: signal package will mask objects from package:stats (filter, poly)

# Auxiliary routines

# Calculates the delays in meters
# ant.loc - matrix(n.ant,3) giving the locations
# (x,y & z in meters) of the antennas
# Azimuth - given in degrees, calculated
# counterclockwise from the x axis
# Inclination - given in degrees calculated from z axis
distance.delay <- function(ant.loc,azimuth,inclination){
  # Number of antennas
  n.ant <- nrow(ant.loc)
  # Azimuth in radians
  azimuth.rad <- azimuth/360*2*pi
  # Inclination in radians
  inclination.rad <- inclination/360*2*pi

  # Calculate the inner product between the antenna coordinates and the beam vector
  delays <-
    ant.loc[,1]*sin(inclination.rad)*cos(azimuth.rad) +
    ant.loc[,2]*sin(inclination.rad)*sin(azimuth.rad) +
    ant.loc[,3]*cos(inclination.rad)

  # Find the minimum delay and translate that to zero
  delays <- delays-min(delays)

  # Return delay distances as a vector
  return(delays)
}

# Calculates the delay in sample units
# given the sample frequency (in Hz) and the delay
# distance (in m)
sample.delay <- function(sample.freq,distance){
  # distance between samples
  sample.length <- 299792458/sample.freq
  # delays in sample units
  sample.delay <- distance/sample.length
  return(sample.delay)
}

# Beamformer for a given data matrix
# Intended to be used for simulation and testing purposes
# (i.e. no reading data files, parsing data etc.)

# All data must be in data.matrix

# Forms a beam in direction (azimuth,inclination)
# from REAL or COMPLEX samples given in data.matrix

# One stage beamforming, i.e. single filter used

# FIR Filter taps are calculated using routine written by
# Markku Lehtinen. For details, see EISCAT_3D Radar Handbook
# written by M. Lehtinen.

# Arguments:

# azimuth - beam azimuth in degrees
# inclination - beam inclination in degrees
# data.matrix - matrix(data.len,n.ant)
# ant.loc.matrix - matrix(n.ant,3)
# sample.freq - sampling frequency (in MHz)
# center.freq - signal center frequency (in MHz)
# hBW - filter half bandwidth (in MHz)
# ramp - filter ramp (in MHz)
# ... - other arguments passed to makefilter routine
# (see ML_filter.R for details)
# decimate - If TRUE, the resulting data is decimated
# Decimation amount depends on the used
# filter parameters

beamformer.one.stage <- function(azimuth=0, inclination=0,
                                   data.matrix, ant.loc,
                                   sample.freq, center.freq,
                                   hBW, ramp,
                                   ...
                                   decimate=FALSE){

  # number of antennas
  n.ant <- nrow(ant.loc)

  # Data length
  data.len <- nrow(data.matrix)

  # Calculate delays (in sample units)
  dd <- distance.delay(ant.loc,azimuth,inclination)
  ss <- sample.delay(sample.freq*1e6,dd)

  # Full sample delays
  idx <- floor(ss)

  # fractional sample delays
frac.del <- ss-idx

# Calculate filters for each antenna
# See ML_filter.R for details
# Filter taps are stored in list "filters".
filters <- list()
for (i in 1:n.ant){
  filters[[i]] <- create.filter.taps(smp = sample.freq,
                                 cnt=center.freq,
                                 hBW=hBW,
                                 ramp=ramp,
                                 tdfrac=frac.del[i],
                                 goal=-48,
                                 sd=40,
                                 ...
)
}

# Run the data through the filters
# Pad the data with zeros to compensate
# the full sample delays
data.matrix <- rbind(data.matrix,
                      matrix(0,max(idx),n.ant))

# Decimation amount
if (decimate){
  dec <- sample.freq/filters[[1]]$dec
} else {
  dec <- 1
}

# Storage for filtered data
data.out <- matrix(0,data.len,n.ant)

# Filter data using fftfilt from "signal" package
# Note that full sample delays shift the data
# starting point
for ( i in 1:n.ant){
  data.out[,i] <-
  fftfilt(filters[[i]]$taps,
          data.matrix[1:data.len+idx[i],i])
}

# Sum up and decimate the filtered data
return(rowSums(data.out[seq(1,data.len,by=dec),]))
1.3 File `bf_simu.r`

This file contains simple routines to construct simulated data for demonstrating the beamformer.

```r
# Beamformer simulator routines
# Part of WP11 od EISCAT_3D project.
# Copyright (c) University of Oulu, 2014.
# Written by Mikko Orispaa <mikko.orispaa@oulu.fi>

# Creates simulated sinusoidal signal with
# given source azimuth and inclination.
# data.len - number of samples to be calculated
# fsample - sampling frequency (in Hz)
# fsignal - signal frequency
# az - signal source azimuth angle (in degrees)
# inc - signal source inclination angle (in degrees)
# ant.loc - antenna coordinates (x,y,z in meters)
# given as matrix (n.ant,3), i.e.
# each row gives the coordinates for
# single antenna
make.real.data.matrix <- function(data.len, fsample, fsignal, az, inc, ant.loc){

  # Number of antennas
  n.ant <- nrow(ant.loc)
  
  # Delay distances in meters
  dd <- distance.delay(ant.loc, az, inc)
  
  # Time delays in seconds
  time.delay <- dd/299792458
  
  # Storage for the result
  res <- matrix(0,n.ant,data.len)

  # Calculate the signal samples for each antenna
  # Samples from a single antenna are stored in the
  # res matrix row-wise.
```

10
for (i in 1:n.ant){
    res[,i] <- make.real.data(data.len,
                              fsample,
                              fsignal,
                              time.delay[i])
}
return(res)

# Calculate real signal samples
# data.len - number of samples to be calculated
# fsample - sampling frequency (in Hz)
# fsignal - Signal frequency (in Hz)
# t.delay - time delay of the signal (in seconds)
make.real.data <- function(data.len,
                            fsample,
                            fsignal,
                            t.delay){

    # Sampling points
    xx <- seq(0, data.len/fsample, len=data.len+1)[1:data.len]

    # Calculate the samples
    dd <- cos( 2 * pi * fsignal * (xx - t.delay) )

    return(dd)
}
Appendix C

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

Ilkka Virtanen
Department of Physics
University of Oulu, Finland

Version 0.3-5
October 7, 2014
License

LPI (Lag Profile Inversion)

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

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

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

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

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

1 Introduction 5
  1.1 Purpose of this document ............................... 5
  1.2 Lag Profile Inversion ................................. 5
  1.3 Installation .......................................... 6
  1.4 Help ................................................. 6

2 Lag profile inversion 7
  2.1 Transmitter and receiver signals ....................... 7
  2.2 Scattering from a target .............................. 8
  2.3 Target covariance functions ........................... 9
  2.4 Lag profiles ......................................... 9
  2.5 Range ambiguity functions ............................ 10
  2.6 Radar measurement as a linear inverse problem ........ 10
  2.7 Additional analysis steps ............................. 11
    2.7.1 Ground clutter suppression ....................... 11
    2.7.2 Voltage level decoding ........................... 12

3 LPI implementation 13
  3.1 Resampling and filtering .............................. 13
  3.2 Ground clutter suppression ........................... 13
  3.3 Voltage level decoding filters ....................... 14
  3.4 Data correlation ..................................... 14
  3.5 Theory matrix ....................................... 15
  3.6 Lag profile inversion solvers ....................... 15
  3.7 Input and output control ............................. 16

4 LPI in practice 17
  4.1 Examples with simulated data ......................... 17
    4.1.1 A coherent point target ......................... 18
    4.1.2 Ground clutter suppression ...................... 22
4.2 Examples with real data ........................................... 27
  4.2.1 Autocovariance function measurement with a monostatic radar ........................................... 27
  4.2.2 Bistatic measurements and crosscovariance 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 LPlimitCluster.R .......................................... 40
    5.1.4 LPlimitRemoteNode.R ....................................... 45
    5.1.5 LPlimitComputingSlaves.R ................................ 47
    5.1.6 currentTime.R ............................................. 48
    5.1.7 nextIntegrationPeriods.R .................................. 49
    5.1.8 LPlive.acf.R ............................................... 51
    5.1.9 noUpdate.R ................................................ 53
    5.1.10 LPlrun.remote.R ........................................... 54
    5.1.11 LPlrun.R .................................................. 55
    5.1.12 initLPLEnv.R ............................................. 59
    5.1.13 LPlsolve.R ............................................... 61
  5.2 Signal pre- and post-processing ................................ 66
    5.2.1 prepareLPIdata.R ........................................... 66
    5.2.2 clutterSuppress.R ......................................... 76
    5.2.3 decoFilter.R ............................................... 79
    5.2.4 LPlaveragePower.R .......................................... 82
    5.2.5 LPlsaveACF.R ............................................. 83
    5.2.6 stripACF.R ................................................ 86
  5.3 Correlation and inverse problem formulation .................. 88
    5.3.1 laggedProducts.R ........................................... 88
    5.3.2 lagprodVar.R ............................................... 89
    5.3.3 rangeAmbiguity.R .......................................... 90
    5.3.4 averageProfiles.R .......................................... 92
    5.3.5 theoryRows.R ............................................... 93
  5.4 Inverse problem solvers ......................................... 95
    5.4.1 fishs.init.R .............................................. 95
    5.4.2 fishs.add.R ................................................ 96
    5.4.3 fishs.solve.R ............................................. 97
    5.4.4 deco.init.R ................................................. 99
    5.4.5 deco.add.R ............................................... 100
    5.4.6 deco.solve.R ............................................. 101
    5.4.7 dummy.init.R ............................................. 103
5.4.8  dummy.add.R  ........................................... 105
5.4.9  dummy.solve.R ........................................... 106
5.4.10 ffts.init.R .............................................. 108
5.4.11 ffts.add.R ............................................... 110
5.4.12 ffts.solve.R .............................................. 113
5.4.13 rlips.solve2.R ........................................... 115

5.5  C functions and headers ................................... 117

5.5.1  src/LPI.h .................................................. 117
5.5.2  register.c .................................................. 120
5.5.3  clutter_meas.c ............................................ 122
5.5.4  clutter_subtract.c ....................................... 125
5.5.5  dummy_add.c ............................................... 128
5.5.6  deco_add.c ................................................ 131
5.5.7  fishes_add.c .............................................. 134
5.5.8  fishes_add_clutter.c ..................................... 137
5.5.9  index_adjust.c ............................................ 139
5.5.10 average_power.c ......................................... 144
5.5.11 lagged_products.c ....................................... 152
5.5.12 average_profile.c ....................................... 158
5.5.13 mix_frequency.c ......................................... 162
5.5.14 resample.c ............................................... 166
5.5.15 prepare_data.c .......................................... 172
5.5.16 theory_rows.c ........................................... 176
5.5.17 range_ambiguity.c ....................................... 185
Chapter 1

Introduction

1.1 Purpose of this document

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

1.2 Lag Profile Inversion

LPI is an 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.

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

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

$$z'(t) = (s * p)(t).$$

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

$$z'_r(t) = (\text{env} * p)(t).$$

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

$$z'_i = z'(t_i)$$

$$z'_r = z'_r(t_i)$$
where $t_i = i\Delta t$.

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

### 2.2 Scattering from a target

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

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

(2.5)

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

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

(2.6)

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

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

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

(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 $\xi$ was defined as function of the total signal travel time $S$, and it will thus be different for two physically separated receivers.

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

---

1Monostatic incoherent scatter radars typically have duty-cycles from 5 to 25 %
write the received signal as

\[ s(t) = \int_{S_1}^{S_2} \xi(S,t) \ast \text{env}(t - S) dS \]  \hspace{1cm} (2.8)

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

2.4 Lag profiles

The autocovariance function as function of range \( \sigma(S, \tau) \) was previously considered. The formulation essentially deals with fixing a range \( S \) and defining the autocovariance function of the scattering process at the given range.
Instead of fixing a range one can fix a time lag and deal with the resulting range profiles as well. These fixed time lags of covariance function as function of range are called lag profiles. Denoting the lag profile at lag $\tau_i$ with $\rho_i(S)$ we will have

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

### 2.5 Range ambiguity functions

Expectation value of the product

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

can be written as

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

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

$$W(t, t', S) = z'(t - S)z'(t' - S). \quad (2.12)$$

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

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

where the last term is random noise.

### 2.6 Radar measurement as a linear inverse problem

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

$$m_{i,j} = \sum_{k=1}^{k_1} W_{i,j,k} \sigma_{k,i-j} + \epsilon_{i,j} \quad (2.14)$$

where the coefficients $W_{i,j,k}$ can be calculated from the continuous range ambiguity function.
In real measurements only discrete signal samples are available and the coefficients must be approximated from products of these samples. Oversampling or interpolation is generally needed in order to achieve sufficient accuracy.

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

$$\mathbf{m}_l = \mathbf{W}_l \rho_l + \mathbf{\epsilon}_l$$

(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 $W_{i,i+l,k}$, $\rho_l$ is the unknown discrete lag profile and $\mathbf{\epsilon}_l$ is random noise. If the noise is zero-mean and gaussian the Maximum Aposteriori (MAP) estimate of the lag profile is

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

(2.16)

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

(2.17)

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

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

### 2.7 Additional analysis steps

#### 2.7.1 Ground clutter suppression

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

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

Incoherent scatter spectrum in the ionospheric D region is rather narrow and it becomes possible to decode the received data at voltage level prior to lag profile inversion. After voltage level decoding the signal will correspond to a measurement with short pulses matched to the measurement range resolution. It is thus possible to calculate only a rather small number of time lags and, furthermore, to replace the general lag profile inversion with simple averaging of lagged products. Voltage level decoding thus provides a possibility for very fast inversion of D region lag profiles. However, one should be very careful when measuring short time lags with this technique, because it may allow F region echoes to alias on top of the true D region signal.
Chapter 3

LPI implementation

3.1 Resampling and filtering

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

3.2 Ground clutter suppression

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

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

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

3.4 Data correlation

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

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

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

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

3.6 Lag profile inversion solvers

Altogether five different inverse problem solvers are supported.

- **rlips** R Linear Inverse Problem 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 r lips, fishs, and deco require explicit theory matrix rows. When other solver are used the theory rows are not produced but the solvers operate directly on the correlated data vectors.

\(^1\)http://www.sgo.fi/ m/pages/rlips.html
3.7 Input and output control

In order to make LPI suitable for wide range of data formats, the package allows the user to define a set of functions used for data input and output. These functions can be collected in separate packages that can be maintained independently from LPI. Names of the I/O functions and packages including them are given as input arguments to the main solver function. It is usually practical to include also a simple wrapper function that generates a call to the main analysis loop of LPI. A raw data input function is mandatory, whereas a few other routines have defaults, see the LPI manual for details. Currently available I/O packages are `LPI.gdf` and `LPI.KAIRA`. See LPI-manual.pdf for detailed descriptions of the I/O functions.
Chapter 4

LPI in practice

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

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

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

> help(package=LPI)

and the help page of the main analysis function

> ?LPI

4.1 Examples with simulated data

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

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

> library(LPI)

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

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

# Add the ndata element
outlist[['TX1'][['ndata']] <- as.integer(nd)

# TX2 is identical with TX1
outlist[['TX2']] <- outlist[['TX1']]

# Our coherent target is assumed to be at 200 km range,
# convert to sample intervals
rtarg <- floor(200e3 / 2.9972458e8 * 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']]
> 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
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.

```r
> 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 4.1: LPI detection of a simulated coherent point target.

```r
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
```
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

```r
> 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
+ + plen <- floor( 1e-3 * srate )
+ + # Pulse lengths counted as data samples
+ + 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']]
We have now everything needed for the simulation run, let us call LPI, first without clutter suppression

```r
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
)
```

**Input Packages:**
- dataInputFunction: `datafun`
- dataEndTimeFunction: `currentTimes`
- clusterNodes: `NA`
- nup: `RX1:1 RX2:1 TX1:1 TX2:1`

**Lag Limitation:**
- 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`

**Other Parameters:**
- 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 RX2:0.00000000 TX1:0.00000000 TX2:0.00000000`
- indexShifts: `RX1:0 0 RX2:0 0 TX1:0 0 TX2:0 0`
- solver: `fishs`

```
24
```
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
maxRanges: Inf

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: fishes

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)

```r
> 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))
```
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,

```r
> 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
```

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.
+ odata$TX1 <- list(cdata=beata20130313$cdata,
+ idata=beata20130313$itx,
+ ndata=beata20130313$ndata)
+ odata$TX2 <- odata$TX1
+ odata$RX1 <- list(cdata=beata20130313$cdata,
+ idata=beata20130313$irx,
+ ndata=beata20130313$ndata)
+ odata$RX2 <- odata$RX1
+ odata$success <- TRUE
+
+ return(odata)
+
> 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: fishes
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 crosscovariance 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 designed 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.

```r
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[["rangeLimits"]]) <- "integer"
  storage.mode( LPIparam[["rangeLimits"]]) <- "integer"
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

for(n in inputPackages){cat(n," ")}
cat('
')
cat(sprintf("%20s %s
","startTime :",startTime,
  format(as.POSIXlt(startTime,origin='1970-01-01',tz='ut'),"%Y-%m-%d %H:%M:%OS6")))
cat(sprintf("%20s %s
","stopTime :",stopTime,
  format(as.POSIXlt(stopTime,origin='1970-01-01',tz='ut'),"%Y-%m-%d %H:%M:%OS6")))
cat(sprintf("%20s %s
","inputPackages :",inputPackages))
cat(sprintf("%20s %s
","dataInputFunction :",dataInputFunction))
cat(sprintf("%20s %s
","dataEndTimeFunction :",dataEndTimeFunction))
cat(sprintf("%20s %s
","clusterNodes :",clusterNodes))
if(is.list(clusterNodes)){
  for(n in names(clusterNodes)){cat(sprintf("%s :",n));
    cat(clusterNodes[[n]],"");cat('
')}
} else{
  cat(clusterNodes );
  cat('
')
}
cat(sprintf("%20s %s
","nup :",nup));for(dType in c("RX1","RX2","TX1","TX2"))
  cat(' ',dType,':',LPIparam[['nup']][[dType]],sep='');cat('
')
cat(sprintf("%20s %s
","filterLength :",filterLength));for(dType in c("RX1","RX2","TX1","TX2"))
  cat(' ',dType,':',LPIparam[['filterLength']][[dType]],sep='');cat('
')
cat(sprintf("%20s %s
","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 %s\n", "maxWait.s:", maxWait.s))
cat(sprintf("%20s %f %f TX1:%f TX2:%f\n", "freqOffset:", LPIparam$freqOffset["RX1"], LPIparam$freqOffset["RX2"], LPIparam$freqOffset["TX1"], LPIparam$freqOffset["TX2"]))
cat(sprintf("%20s %s\n", "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 %s\n", "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"]]) )

  # 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
waitSum <- waitSum + 10

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

# 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

```r
# 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"]
```

## 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".
# 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 unsuccessful, stop the whole analysis
stop("Cannot parse the input vector ", paste(substitute(parvec)))
}

5.1.3 LPIinitCluster.R

```r
## 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 processes 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 significantly 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
sequentally 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)){
    nnames <- rep(0, length(nodes))
  } else{
    nnames <- nchar(names(nodes))
  }
}
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 packages 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 packages 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

```r
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)
}
```

47
48 | }

231
### LPIinitComputingSlaves.R

```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)
## }
```

5.1.5 LPIinitComputingSlaves.R
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**

```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.ready ) {

  # Truly available periods
  intPer.available <- intPer.ready[ which( intPer.ready <= 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"])
```

---

234
# 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

```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 successful
    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(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(LPIdparam[['resultSaveFunction']]))(
    LPIdparam, intPeriod, ACF
  )
}

# Return the integration period
# number to the main process
return(intPeriod)
5.1.9  noUpdate.R

```r
noUpdate <- function ( LPIparam , intPeriod ) {
  if(is.null(LPIparam[["callN"]])){
    LPIparam[["callN"]]<-1
    return(LPIparam)
  }
  return(NULL)
}
```
5.1.10  LPIrun.remote.R

```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 ) )
}
```

LPIrun <- function ( LPIenv.name )
{
  nlags <- eval(LPIenv.name)["nLags"]
  x <- seq( nlags )

  ngates <- eval(LPIenv.name)['nGates']
  maxgates <- max(ngates)

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

  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 slower to transfer
ACFreturn <- list()
ACFreturn["ACF"] <- ACFmat
ACFreturn["var"] <- VARmat
ACFreturn["covariance"] <- COVARmat
ACFreturn["lag"] <- lgates
ACFreturn["range"] <- rgates
ACFreturn["nGates"] <- ngates

return(ACFreturn)
# 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 updated 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 )

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

  # Laged 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

```r
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"]
```

---

5.1.13 LPIsolve.R

```r
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"]
```
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"]][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 )
}
else{

  # Other solvers need theory matrix rows
  # 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)]) ,
                   A.ambig = LPIenv[['ambig']][1:(LPIenv[['nrows']]*(LPIenv[['nGates']][lag]+1)]) ,
                   I.ambig = LPIenv[['iamb']][1:(LPIenv[['nrows']]*(LPIenv[['nGates']][lag]+1)]) ,
                   I.prod = LPIenv[['iprod']][1:(LPIenv[['nrows']]*(LPIenv[['nGates']][lag]+1)]) ,
                   E.data = LPIenv[['var']][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

```r
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 first 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
  pulseStarts
```

---

5.2 Signal pre- and post-processing

5.2.1 prepareLPIdata.R

```r
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 first 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
  pulseStarts
```
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"]) / LPIdparam["filterLength"]["TX1"] * LPIdParam["nup"]["TX1"] * LPIdparam["filterLength"])[XXN] / LPIdparam["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(LPIdparam["nup"]==LPIdparam["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]]) %/% (LPIdParam["filterLength"])[XXN] / LPIdparam["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 <- 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"]]
}
for( XXN in dTypes ){
  firstSample[[XXN]] <- round( firstSample["TX1"] *
    LPIparam[ ["filterLength"]][[XXN]] / LPIparam[ [" filterLength"]][["TX1"] ] / LPIparam[ ["nup "]][[XXN]]
    * LPIparam[ ["nup"]][["TX1"]])
}

# Conversions to integer mode
storage.mode( firstSample ) <- "integer"
storage.mode(LPIparam["filterLength"][]) <- "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],
    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""] <- LPI:::decoFilter.cdata( LPIdatalist.final[]["RX1"]["cdata"][1:nd], LPIdatalist.final[]["TX1"]["cdata"][1:nd], LPIparam["decodingFilter"][1] )
    LPIdatalist.final[]["TX1"]["cdata""] <- LPI:::decoFilter.cdata( LPIdatalist.final[]["TX1"]["cdata"][1:nd], LPIdatalist.final[]["TX1"]["cdata"][1:nd], LPIparam["decodingFilter"][1] )
    nd <- LPIdatalist.final[]["nData"]
}
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']][['idata']][1:nd] , LPIdatalist.final[['TX2']][['ndata']] , LPParam[['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(LPParam[['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[['RX1']][['idata']] , LPIdatalist.final[['RX1']][['ndata']] , maxr )

  # Average power in signal vector RX2
  LPIdatalist.final[['RX2']][['power']] <- LPIaveragePower( LPIdatalist.final[['RX2']][['cdata']] , LPIdatalist.final[['RX2']][['idata']] , LPIdatalist.final[['RX2']][['ndata']] , maxr )

  # Flag data points whose power is more than three times the average at a given height
  itx1 <- which( abs(LPIdatalist.final[['RX1']][['cdata']][1:LPIdatalist.final[['nData']]]) > (sqrt(LPIdatalist.final[['RX1']][['power']])*3) )
  LPIdatalist.final[['RX1']][['idata']][itx1] <- FALSE
  itx2 <- which( abs(LPIdatalist.final[['RX2']][['cdata']][1:LPIdatalist.final[['nData']]]) > (sqrt(LPIdatalist.final[['RX2']][['power']])*3) )
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

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

# Maximum ranges, repeat the last value as necessary
LPIdatalist.final["maxRanges"] <- LPIdatalist.final["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"] <- LPIdatalist.final["rangeLimits"]
LPIdatalist.final["nGates"] <- rep(length(LPIdatalist.final["rangeLimits"])) - 1, LPIdatalist.final["nLags"]
for( k in seq(LPIdatalist.final["nLags"])) {
  LPIdatalist.final["nGates"][k] <- length(LPIdatalist.final["rangeLimits"])[ LPIdatalist.final["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: clutter Suppress .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 contains 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
```

76
rmin <- max(rmin, 0)
rmax <- max(rmax, 0)

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

# Initialize a fishes object
e <- fishes.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"] <- 0 + 0i

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


5.2.3 decoFilter.R

```r
## 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))
    }
  } 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))
    }
  } else{
    stop("Unknown decoding filter")
  }
} else{
  stop("Unknown decoding filter")
}
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)
}
## 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

```r
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] ) )
  }
  else if( pow[1] > .1 ){
    pow[] <- mean( abs( cdata[idatarx] ) )
  }

  return( pow )
}
```
5.2.5 LPIsaveACF.R

```r
## 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='’)
}
```
# 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
invisble(resFile)
}

5.2.6 stripACF.R

```r
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"]]

# Update 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

```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
## 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),
    }
```

---

**5.3.3 rangeAmbiguity.R**

```r
## 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),
    }
```
# Simple lagged products of decimated data, works with strong codes.

```r
LPIenv[['nData']],
lag
}

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

```r
## 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 accuracy and flexibility is acceptable.

## 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"]]
                               - l )
               , as.integer( LPIenv[["nCode"]]) )

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

  invisible( ( s1 & s2 ) )
}
```
5.3.5 theoryRows.R

```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 produced, FALSE otherwise.
##
## The rows are written to LPIenv[['arows']],
## the corresponding measurements to LPIenv[['meas']],
## variance to LPIenv[['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']])
}
```

93
47 |
48 |
49 |
5.4 Inverse problem solvers

5.4.1 fishes.init.R

```r
## file: fishes.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 fishes solver environment
#
fishes.init <- function ( ncols , ... )
{
  s <- new.env()

  assign( 'ncol' , ncols , s )

  assign( 'Qvec' , rep(0+0i,(ncols*(ncols+1)/2)) , s )

  assign( 'y' , rep(0+0i,ncols) , s )

  storage.mode(s$Qvec) <- storage.mode(s$y) <- "complex"
  storage.mode(s$ncol) <- "integer"

  return(s)
}
```

```
5.4.2 fishs.add.R

```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 calculation of Fisher information matrix.
### Data accumulation function.
###
### Arguments:
### e A fishs solver environment
### 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 fishes.solve.R

```r
## file: fishes.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 fishes 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.

```R
defines a function `fishs.solve` that calculates the Fisher information matrix for a given fishes solver environment. The function takes an environment `e` and an optional argument `full.covariance` which, if set to `TRUE`, calculates the full covariance matrix. Otherwise, only the variances are returned. The function also accepts additional arguments, denoted by `...`.

The Fisher information matrix calculation involves the following steps:

1. Allocate a matrix for the full Fisher information matrix.
2. Copy the upper triangular part from the environment's `Qvec`.
3. The lower triangular part is the complex conjugate of the upper one.
4. The above row multiplies the diagonal with 2, then divides accordingly.
5. Select points at which the diagonal of the matrix is zero.

The function returns nothing; it assigns the calculated solution to the `e` environment.
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( '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

```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 environment
## 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
# 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

```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
  s$rmin <- rmin
  s$rmax <- rmax

  s$msum <- msum
  s$vsum <- vsum

  s$rrange <- rrange

  return(s)
}
```

103
assign( 'msum', msum, s )
assign( 'vsum', vsum, s )
assign( 'rmin', rmin, s )
assign( 'rmax', rmax, s )

# Return the environment

return(s)
## Dummy inverse problem solver that calculates simple averages.
## Data accumulation function.

### Arguments:
- `e`: A dummy solver environment
- `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

```r
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) 
}
```
## Dummy inverse problem solver that calculates simple averages.

## Arguments:
- **e**: A dummy solver environment
- **rlims**: Range gate limits

## Returns:
- Nothing, the solution is assigned to the solver environment

```r
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$vs 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()
## 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

```r
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

```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 environment
## 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

```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

```r
## 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

```c
#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

// Data types and function prototypes

// 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 cdata, SEXP idatap, SEXP ndata1, SEXP ndata2, SEXP lag);
SEXP lagged_products_r(SEXP rdata1, SEXP rdata2, SEXP prdata, SEXP ndata1, SEXP ndata2, SEXP lag);
```
35 // Theory matrix construction
36 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 );
37 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 );
38
39 // Inverse problem solvers
40 SEXP fishs_add ( const SEXP Qvec , const SEXP yvec , const SEXP
   arows , const SEXP meas , const SEXP var , const SEXP nx
   , const SEXP nrow );
41 SEXP deco_add ( const SEXP Qvec , const SEXP yvec , const
   SEXP arows , const SEXP meas , const SEXP var , const SEXP
   nx , const SEXP nrow );
42 SEXP dummy_add ( SEXP msum , SEXP vsum , SEXP rmin , SEXP rmax
   , SEXP mdata , SEXP mambig , SEXP iamb , SEXP iprod ,
   SEXP edata , SEXP ndata );
43
44 // All data preparations collected together
45 SEXP prepare_data ( SEXP cdata , SEXP idata , SEXP ndata ,
   SEXP frequency , SEXP shifts , SEXP nup , SEXP nfilter ,
   SEXP nfirst , SEXP ipartial );
46
47 // Average signal power in points with identical IPPs and
48 // pulse lengths
49 SEXP average_power ( SEXP cdata , SEXP idatatx , SEXP idatarx
   , SEXP ndata , SEXP maxrange );
50
51 // Average lag profile
52 SEXP average_profile ( SEXP cdata , SEXP idata , SEXP ndata ,
   SEXP N_CODE );
53
54 // Resampling
55 SEXP resample ( SEXP cdata , SEXP idata , SEXP ndata , SEXP
   nup , SEXP nfilter , SEXP nfirst , SEXP ipartial );
56 SEXP resample_R ( SEXP cdata , SEXP idata , SEXP ndata , SEXP
   nup , SEXP nfilter , SEXP nfirst , SEXP ipartial );
57
58 // Range ambiguity function calculation with optional
59 // interpolation
60 SEXP range_ambiguity ( SEXP cdata1 , SEXP cdata2 , SEXP idata1
   , SEXP idata2 , SEXP cdatap , SEXP idatap , SEXP ndata1 ,
   SEXP ndata2 , SEXP lag );
61
// 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.

#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 , 9 },
    { "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 , 7 },
    { "resample_R" , (DL_FUNC) &
      resample_R , 7 },
    { "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

// 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
cdata 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;
xn = 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
    fishes_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 );
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

```c
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

1 // file: dummy_add.c
2 // (c) 2010 - University of Oulu, Finland
3 // Written by Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
4 // Licensed under FreeBSD license.
5
6 #include "LPI.h"

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

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;

// 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

#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 successful, 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 diagonal)
        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

#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, const SEXP arow, const SEXP 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

#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 adjustments
 *   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) ) ) );

287
139
// 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

#include "LPI.h"

/*
 * Average power vector for variance estimation
 * 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 pi;
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 )
        {
            // 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;
        }
    }
}
pinds[ p1 - 1 ] = pinds[ i ];
break;
}
}

// Give a new index for the pulse p1-1 if it did not
// match
// with any of the existing 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 )
        {
            // 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.;
        }
    }
}
282 // Go through the indices again and copy the power
283 // values to appropriate places Set pinds to -1 at
284 // points that have already been visited
285 pindcur = pinds[k];
286 for( j = k ; j < nedges ; ++j )
287 {
288     if( pinds[j] == pindcur )
289     {
290         if( ( j + 1 ) >= nedges )
291             ippend = nd - pedges[j];
292         else
293             ippend = pedges[j + 1] - pedges[j];
294         for( i = 0 ; i < ippend ; ++i )
295             r = pedges[j] + i;
296             pd[r] = ptmp[i];
297         pinds[j] = -1;
298     }
299 }
300
301 // Put the grand average power to points that did not have
302 // enough averaged samples (they are set to -1 at this
303 // point)
304 ptot /= (float)ntot;
305 for( i = 0 ; i < nd ; ++i )
306     if( pd[i] < 0.) pd[i] = ptot;
307 }
308
309 // Store the ratio pindmax / nedges to the first data point
310 // If the ratio is large the power estimation will not
311 // perform
312 // very well.
313 // The power value in this point cannot ever be needed in
314 // LPI.
315 pd[0] = (float)pindmax / (float)nedges;
316
317 // Free the temporary allocations
318 Free(ptmp);
Free(nsamp);
Free(pinds);
Free(pedges);
UNPROTECT(1);
return(pdata);
}
5.5.11 lagged_products.c

/*
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]);
}
if(lid[k]){
    lcd[k].r = cd1[k].r * cd2[k+1].r + cd1[k].i * cd2[k+1].i;
    lcd[k].i = -cd1[k].r * cd2[k+1].i + cd1[k].i * cd2[k+1].r;
}

// Collect the return values under the list "ans"
SET_VECTOR_ELT( ans , 0 , lddata );
SET_VECTOR_ELT( ans , 1 , idata );
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
*/
nda1  Number of samples in cdata1 and idata1
nda2  Number of samples in cdata2 and idata2
lag   Lag

Returns:
  success 1 if processing was successful, 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 ){
    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+1].r + cd1[k].i * cd2[k+1].i;

cdp[k].i = cd1[k].r * cd2[k+1].i - cd1[k].i * cd2[k+1].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
#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
 * measurements 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 * 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 begining 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
  // 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 values to make further 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;
}

// 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;
}

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

// 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

#include "LPI.h"

/*
Frequency mixing for IQ data

This function allocates new vectors

Arguments:
- 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

Arguments:
cdata      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

#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     Upsampling 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:
 * success 1 if resampling was successful, 0 otherwise
 *
 */

SEXP resample(SEXP cdata, SEXP idata, SEXP ndata, SEXP nup, SEXP nfilter, SEXP nfirst, 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 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
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
                      // Increment the sample counter of the original data vector
        ++k;
    } // 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;
}

// 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
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 Upsampling 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
* /

SEXP resample_R(SEXP cdata, SEXP idata, SEXP ndata, SEXP nup, SEXP nfilter, SEXP nfirst, 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 ){
\[
\text{inew}[k] = \text{iold}[k];
\]

// Use the same pointers to copy old ndata to new ndata
\[
\text{inew} = \text{INTEGER}( \text{ndata}_\text{new} );
\]
\[
\text{iold} = \text{INTEGER}( \text{ndata} );
\]
\[
* \text{inew} = * \text{iold};
\]

// The success logical
\[
\text{PROTECT}( s = \text{allocVector}( \text{LGLSXP} , 1 ));
\]

// The actual resampling
\[
s = \text{resample}( \text{cdata}_\text{new} , \text{idata}_\text{new} , \text{ndata}_\text{new} , \text{nup} , \text{nfILTER} , \text{nfirst} , \text{ipartial} );
\]

// Reallocate the vectors to match with the new data length
\[
\text{SET\_LENGTH}( \text{cdata}_\text{new} , \text{INTEGER}(\text{ndata}_\text{new}) );
\]
\[
\text{REPROTECT}( \text{cdata}_\text{new} , \text{cpind} );
\]
\[
\text{SET\_LENGTH}( \text{idata}_\text{new} , \text{INTEGER}(\text{ndata}_\text{new}) );
\]
\[
\text{REPROTECT}( \text{idata}_\text{new} , \text{ipind} );
\]

// Collect the data into the return list
\[
\text{SET\_VECTOR\_ELT}( \text{ans} , 0 , \text{cdata}_\text{new} );
\]
\[
\text{SET\_VECTOR\_ELT}( \text{ans} , 1 , \text{idata}_\text{new} );
\]
\[
\text{SET\_VECTOR\_ELT}( \text{ans} , 2 , \text{ndata}_\text{new} );
\]
\[
\text{SET\_VECTOR\_ELT}( \text{ans} , 3 , s );
\]

// Set the name attributes
\[
\text{PROTECT}( \text{names} = \text{allocVector}( \text{STRSXP} , 4 ));
\]
\[
\text{SET\_STRING\_ELT}( \text{names} , 0 , \text{mkChar}( \text{cnames}[0] ));
\]
\[
\text{SET\_STRING\_ELT}( \text{names} , 1 , \text{mkChar}( \text{cnames}[1] ));
\]
\[
\text{SET\_STRING\_ELT}( \text{names} , 2 , \text{mkChar}( \text{cnames}[2] ));
\]
\[
\text{SET\_STRING\_ELT}( \text{names} , 3 , \text{mkChar}( \text{cnames}[3] ));
\]
\[
\text{setAttrib}( \text{ans} , \text{R\_NamesSymbol} , \text{names} );
\]
\[
\text{UNPROTECT}(6);
\]
\[
\text{return}(\text{ans});
\]
\[
\text{)}
\]
5.5.15    prepare_data.c

#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 Upsampling 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 successful
*/

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

#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);

    // code
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 ,

177
\begin{verbatim}

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 * (nranges + 1)));
    REPROTECT(arows, arind);
    SET_LENGTH(irows, (n_rows * (nranges + 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);
}

/*

\end{verbatim}
Make theory matrix rows and measurement vectors.

This function overwrites existing data vectors.

Arguments:
camb Complex 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

SEXPT 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 limit - 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 accepted) 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;
        } 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

/*
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 Second 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 npri;
    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 - 1;
if( nd1 > nd2 ) npr = nd2 - 1;

// 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+1]);
  // 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 ) * (1.0 - (double)i / (double)( 2 * ninterp ));
        tmpi1[i] = cd1[k-1].i + ( cd1[k].i - cd1[k-1].i ) * (1.0 - (double)i / (double)( 2 * ninterp ));
        tmpr2[i] = cd2[k-1+1].r + ( cd2[k+1].r - cd2[k-1+1].r ) * (1.0 - (double)i / (double)( 2 * ninterp ));
        tmpi2[i] = cd2[k-1+1].i + ( cd2[k+1].i - cd2[k-1+1].i ) * (1.0 - (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+1].r + ( cd2[k+1+1].r - cd2[k+1].r ) * ( (double)i / (double)( 2 * ninterp ) );
        tmpi2[i+ninterp] = cd2[k+1].i + ( cd2[k+1+1].i - cd2[k+1].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);
Appendix D

LPI manuals
D.1: LPI package manual
Package ‘LPI’

September 5, 2014

Version 0.3-5

Date 2014-02-17

Title Lag Profile Inversion

Author Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>

Maintainer Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>

Description Lag Profile Inversion

License FreeBSD

Depends R (>= 2.14.0), parallel

Suggests rlips

Copyright University of Oulu, Finland

R topics documented:

LPI-package .......................................................... 1
LPI ................................................................. 2
LPIexpand.input ................................................... 8
stripACF ........................................................... 9

Index 11

LPI-package       Lag Profile Inversion

Description

Lag Profile Inversion with support for computer clusters

Details

Package: LPI
Version: 0.3-5
Date: 2014-02-17
License: FreeBSD
Depends: R ()
Built:
Index:

LPI Lag Profile Inversion

Author(s)

Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>

Description

Lag Profile Inversion

Usage

LPI(dataInputFunction, inputPackages=c(), startTime = 0, stopTime = 4000000000, nup = LPIexpand.input(1), filtersLength = 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, nodeMultiplicity = 1, useXDR = FALSE, maxWait.s = -1, freqOffset = LPIexpand.input(0), indexShifts = LPIexpand.input(list(c(0,0))), solver = "fish" , nBuf = 10000, fullCovar = FALSE, rLips.options = list(type="c", nBuf=10000, workgroup.size=128), remoteRX = FALSE, normTX = FALSE, nCode = NA, ambInterp = FALSE, resultDir = paste(format(Sys.time("%Y-%m-%d_%H:%M"),"%Y-%m-%d_%H:%M"),LP,sep=""), dataEndTimeFunction="currentTimes", resultSaveFunction = "LPIsaveACF", paramUpdateFunction="noUpdate" , ... )

Arguments

dataInputFunction
Raw data input function name as a string. The function will get a list of all LPI input arguments as input and it must return a valid list of raw data vectors. See details.

inputPackages
Additional package(s) that contain functions for raw data input, as strings.

startTime
Analysis start time in POSIX format.
Default: 0 (1st Jan 1970 00:00 UT)

stopTime
Analysis end time in POSIX format.
Default: 4000000000 (2nd Oct 2096 07:00 UT)

clusterNodes
A list defining the computer cluster, see details.
Default: NA (sequential analysis)
	nodeMultiplicity
Number of integration periods solved in each cluster node before returning to the main node for file list update.
Default: 1

useXDR
Should serialization in cluster communication use XDR? See help of the function `makeCluster` from package `parallel`.
Default: FALSE
timeRes.s Analysis time resolution (integration time) in seconds.
Default: 10

maxWait.s Maximum time to wait for new data before stopping the analysis, in seconds.
Default: -1 (Stop immediately at end of data)

freqOffset Frequency offsets from baseband. A named vector with elements "RX1", "RX2", "TX1", "TX2" or anything that can be converted into this format with LPIexpand.input.
Default: 0.0

nup Upsampling factors. The data are upsampled by factor 'nup' before applying boxcar filter of length 'filterLength'. A named vector with elements "RX1", "RX2", "TX1", "TX2" or anything that can be converted into this format with LPIexpand.input.
Default: 1

filterLength Length of the boxcar-shaped post-detection filters. The actual filter lengths are filterLength / nup. A named vector with elements "RX1", "RX2", "TX1", "TX2" or anything that can be converted into this format by the function LPIexpand.input. The ratio filterLength / nup must be identical for all four data vectors.
Default: 1

lagLimits Limits of time-lag gates.
Default: c(1,2) (The first lag)

rangeLimits Limits of range gates.
Default: c(1,2) (One range gate very close to ground)

maxRanges Maximum range for each lag profile. Allows the number of range gates to be limited at selected lags. See details.
Default: Inf

nCode Number of pulses in a full modulation cycle. If positive, the value is used for lag profile pre-averaging before the actual inversion. Use NA to disable the averaging.
Default: NA

backgroundEstimate Logical, if TRUE, an additional unknown is modeled in each lag profile in order to absorb time-stationary background noise.
Default: TRUE

fullCovar If TRUE, full covariance matrices of each lag profile are stored, if FALSE, only variances are stored.
Default: FALSE

decodingFilter Amplitude domain decoding filter selection, accepted values are a complex vector of filter coefficients and character strings "none", "matched", and "inverse". See details.
Default: "none"

maxClutterRange Maximum range at which ground clutter will be subtracted. Set below min(rangeLimits) in to disable the clutter suppression. See details.
Default: 0
clutterFraction
The fraction of a full integration period used for the coherent ground clutter profile estimation. A float from interval (0,1]. Values smaller than 1 should be used carefully.
Default: 1.0

remoteRX
Logical, if TRUE, all receiver samples flagged as usable by 'dataInputFunction' are used in analysis. If FALSE, only those lagged products whose range ambiguity function is zero in the range interval [0 min(rangeLimits.km)] are used. Should usually be FALSE in monostatic operation and TRUE in bistatic measurements. Notice that zero range ambiguity requirement does not completely cut off signal from small ranges, use indexShifts for this purpose.
Default: FALSE

ambInterp
Logical, if TRUE, the range ambiguity functions are calculated from transmitter samples that are oversampled by factor 11. The oversampling is performed by means of linear interpolation of decimated transmitter samples. If FALSE, the range ambiguity functions are calculated as simple products of the decimated data.
Default: FALSE

indexShifts
Additional adjustments to TX and RX indices, see details.
Default: list( TX1=c(0,0) , TX2=c(0,0) , RX1=c(0,0) , RX2=c(0,0) )

normTX
Logical, if TRUE, the transmitter sample amplitudes are normalized to their mean value after filtering and decimation. May be useful with low-quality transmitter samples, but must be used with care.
Default: FALSE

solver
Inverse problem solver selection, accepted values are "rlips", "fishs", "deco", "ffts", and "dummy". See details.
Default: "fishs"

nBuf
Number of theory matrix rows to buffer before calling the solver function.
Default: 10000

rlips.options
Additional options to the 'rlips' solver. See rlips help for details.
Default: list( type="c" , nbuf=1000 , workgroup.size=128)

resultDir
Output directory. Use non-character value (e.g NA) if 'resultSaveFunction' does not write to files.
Default: paste(format(Sys.time(),

resultSaveFunction
Function that is used for saving the analysis results. The function gets as input the full LPI input argument list ("LPIparam"), the integration period number ("intPeriod"), and the resolved ACF ("ACF"). The function may output anything but the output will not be used anywhere.
The resolved ACF list (argument 'ACF') has the following components:
'range' A vector of range gate centres
'lag' A vector of lag gate centres
'ACF' length(range) x length(lag) complex lag profile matrix
'var' length(range) x length(lag) real matrix of lag profile variances
'backgroundACF' length(lag) complex vector of background ACF
'backgroundvar' length(lag) real vector of background ACF variances
'covariance' (length(range) + 1) x (length(range) + 1) x length(lag) complex array of lag profile covariances, or NULL if LPIparam$fullCovar=FALSE. The
extra elements are from the additional background ACF estimates that are added to the end of each lag profile.

The integration period number (argument ‘intPeriod’) starts from one for the first period, and is incremented by one for each successive period.

The LPI input argument list (argument 'LPIparam') contains all LPI input arguments, including an expansion of the optional ones (‘...’). Any information needed in the resultSaveFunction can thus be passed via the ‘...’ argument of LPI.

Default: LPIsaveACF

dataEndTimeFunction

Function that returns sampling times of last available sample in each data vector. The function will get the full LPI input argument list as argument and it must return a named numeric vector with elements "RX1", "RX2", "TX1", and "TX2".

Default: currentTimes (5s ago)

paramUpdateFunction

An optional function for updating the parameter list for each integration period. The function gets the full LPI input argument list as input and must return a valid LPI parameter list. Within each integration period, the analysis is continued until NULL is returned by paramUpdateFunction, subsequent calls to paramUpdateFunction will get the *modified* LPIparam from the previous call as input.

Default: "noUpdate"

... Additional arguments to be collected in the LPI parameter list. All input arguments of LPI are collected in an "LPI parameter list", which is passed to 'dataInputFunction', 'dataEndTimeFunction', 'resultSaveFunction', and 'paramUpdateFunction'.

Details

'clusterNodes' Definition of the computer cluster. Following inputs are accepted:

1. clusterNodes = NA No parallelism, everything is run sequentially.
2. clusterNodes = n, n = 1, 2, 3, ... Run n integration periods in parallel on the local computer.
3. clusterNodes = list( computer1=n1 , computer2=n2 , ... , computerM=nM ) Run M integration periods in parallel in machines "computer1", "computer2", ... . The lag profiles in each integration period are divided in between n1 processes in computer1, in between n2 processes in computer2, etc. The list may also contain an entry "localControl", if localControl=FALSE the data is read from files at the remote computers, otherwise at the local machine and then transferred to other computers. If localControl=FALSE, all computers in the clusterNodes list must have exactly identical directory hierarchies. This option requires passwordless ssh connection in between the machines computer1, computer2, etc., i.e. generation of ssh keys.

As an example, the default cluster definition is list( tesla1=8 , tesla2=8 , tesla3=8 , tesla4=8 , tesla5=8 ) and it is used in a computer cluster that contains computers tesla1 ... tesla5, each of which has 8 cores. The analysis first starts length(clusterNodes) control nodes in the same machine where the main LPI process is running. Each of these control nodes then connects to nodes running in the remote computers, in this case to tesla1 ... tesla5. Finally, each of the remote nodes connects to a number of 'calculation slave' nodes running in their own local computers. In the default case, 8 nodes would be running on each computer. The main process that the user controls from the command line generates input argument lists for length(clusterNodes) integration periods. Each local control node receives one set of arguments, reads in the corresponding raw data, performs necessary filtering, decimation, etc. and...
passes the data to the remote computers. In each remote node, the data is copied to all calculation
slaves. Each remote node then requests its calculation nodes to solve lag profiles until all
time lags are solved. The remote nodes then collect the profiles together and send the result
back to the control nodes, which call `resultSaveFunction` and signal the main process about
a completed job. When all control nodes are done, a new set of integration periods is selected.
In principle, a remote node could have computing slaves in computers other than its own as
well. This is done simply by replacing the number of computing slaves with a list of computer
names.

`lagLimits` A vector of time lag limits, lagLimits = c( l1, l2, l3, ... , lN ). The analysis integrates
lags l1, l1+1, ... , l2-1 (of decimated data) into lag profile 1, lags l2, l2+1, ... , l3-1 into lag
profile 2, etc. The steps in lagLimits do not need to be uniform, it is possible to use coarser
lag resolution at longer lags by selecting e.g. lagLimits = c(1,2,3,5,7,9,11).

`maxRanges` Allows one to reduce the number of range-gates at selected lags. If the vector is
shorter than lagLimits its last value is repeated as necessary. Defaults to Inf, which means that
all lags are solved at all range gates defined in rangeLimits.

maxRanges may be used in combined D/E/F-region experiments when correlations longer
than certain limit are known to exist only in D-region. As an example, in order to solve first
30 lags at all range gates, but the longer ones only below 100 samplese range, one can use
maxRanges = c( rep( Inf , 30 ) , 100 ).

Another use case is a dedicated D region experiment that makes use of voltage level decoding.
Most time lags will then have zero range ambiguity functions and can be safely skipped.
Assuming that modulation bit length is 1 decimated sample and inter-pulse period is 1000
samples, one could use e.g. the following combination: decodingFilter='matched', lagLimits=c(1000,1001,2000,2001,3000,3001), maxRanges=c(Inf,0,Inf,0,Inf) which would solve three
pulse-to-pulse lags and save the analysis from the work of inspecting all time lags (only 1 in
1000 of them can actually be measured).

`endTime` Analysis end time in POSIX format. The analysis will be stopped without reaching
endTime if all existing data is analysed and the process has waited maxWait.s seconds for new
data.

`freqOffset` Frequency offset from baseband to the signal centre frequency. Currently only one
frequency per data vector, i.e. either a single offset for all data, or different shifts for "RX1", "RX2", "TX1", and "TX2". The frequencies are not in SI units, but time is measured in
decimated sample intervals and the frequecies should be scaled accordingly.

`indexShifts` Additional adjustments to the TX (and) RX indices returned by `dataInputFunction`.
The user input is converted into a list with entries "RX1", "RX2", "TX1", and "TX2". Each
entry is a two-element vector c( shift1, shift2 ) where, in case of TX, shift1 is the adjustment
at rising edge of each pulse, and shift2 is the adjustment at falling edge of each pulse. For RX,
shift1 is the adjustment at each start of reception, and shift2 at end of reception. All shifts are
positive forwards, use negative values to adjust towards earlier times. As an example, if the
recorded TX bit for "TX1" starts 10 samples *before* the actual pulse, and ends 20 samples
*after* the actual end of the pulse, one should have indexShifts["TX1"] == c(10,-20).

`decodingFilter` Voltage level coherent decoding with given filter coefficients or by means of
matched or inverse filtering using measured samples of transmitted waveform. The decoding
is performed for a single inter-pulse period at a time. Combining the "matched" or "inverse"
filters with range gates extending above the range corresponding the shortest inter-pulse pe-
riod will thus usually lead to meaningless results. With the "matched" and "inverse" filters data
index vectors are modified in the decoding as if the decoding would perfectly suppress code
sidelobes. Setting decodingFilter="matched" will thus generally generate range sidelobes.
The "matched" and "inverse" filters are intended to be used for D-region pulse-to-pulse cor-
relations together with solver="dummy", which enables fast calculation of lag profiles when
high lag resolution is not required. More complicated decoding filters that generally require specific information about the modulation should be implemented in dataInputFunction

'solver' LPI allows the user to select an inverse problem solver that best suits for the problem at hand from among the following options:

1. "rlips", R Linear Inverse Problem Solver, is an R package for solving large linear inverse problems. The software exploits GPUs providing massive parallelism. The solver is most suitable for solving large problems (large number of range gates). The rlips package must be installed in all computers of the analysis cluster.
2. "fishs" is a simple inverse problem solver provided as an integral part of the LPI package. It can be used as an CPU-based alternative for "rlips". Relative speed in between "rlips" and "fishs" depends on several factors, such as size of the inverse problem and the computer hardware.
3. "deco" is a modification of "fishs" that essentially performs matched filter decoding of lag profiles. It can thus be used as a faster alternative when alternating codes or long cycles of random codes are used as transmitter modulation.
4. "ffts" is a fast solver that gains its speed from exploiting FFT. The FFT-based solution is not reliable if the received signal is not continuous. The solver is thus mainly intended for bistatic operations. Background noise subtraction cannot be combined with "ffts".
5. "dummy" is a dummy solver that calculates simple averages of lag profiles without actually decoding them. The solver is intended to be used together with voltage level decoding in D-region measurements.

'inputPackages' A list of packages that contain the data input functions. LPI does not need to be listed. The packages must be installed on all computers of the analysis cluster.

'dataInputFunction' Name of a function that returns one integration period of a raw voltage level data. The package containing this function and all functions that this function calls must be installed in all computers of the analysis cluster and listed in 'inputPackages'. All LPI input arguments are collected in a list and passed to this function.

The function is called by means of

```
eval(as.name(LPIparam["dataInputFunction"]))(LPIparam, intPeriod)
```

and it must return a list with elements "RX1", "RX2", "TX1", "TX2", and "success". The first four elements are lists themselves, consisting of a complex data vector 'cdata', a logical "index vector" 'idata', and integer "ndata". Denoting complex samples with 'cn' and logical samples with 'in' the output list is of the form

```
list( RX1=list( cdata=c(c1,c2,c3,...,cn1) , idata=c(i1,i2,i3,...,in1) , ndata=n1) , RX2=list( ... ) , TX1=list( ... ) , TX2=list( ... ) , success=TRUE/FALSE )
```

where '...' denotes a set of vectors similar to that in 'RX1'. The input argument 'intPeriod' is the integration period, counted in steps of 'timeRes.s' from 'startTime'. The period starting exactly at 'startTime' is period number 1.

If success=FALSE in the output list the analysis will completely skip the integration period. E.g. in case of missing data one can ignore the other elements and simply return list(success=FALSE). The integration period will be skipped also if any of the 'idata' vectors has only FALSE elements.

The input argument list 'LPIparam' is the full LPI input argument list, containing also the additional arguments, with the following modifications:

1. The entries 'nup', 'filterLength', 'maxClutterRange', 'clutterFraction', 'freqOffset', and 'indexShifts' are expanded to the LPI internal format with named elements "RX1", "RX2", "TX1", and "TX2" using LPIexpand.input.
2. Storage mode of 'nup', 'filterLength', 'lagLimits', 'rangeLimits', 'maxClutterRange', 'indexShifts', and 'nCode' is set to "integer".
Elements 'lastIntPeriod', 'iscluster', 'dataEndTimes', and 'maxIntPeriod' are added. 'dataEndTimes' is the return value of 'dataEndTimeFunction' and 'maxIntPeriod' is the corresponding integration period number. 'lastIntPeriod' is the integration period number corresponding 'stopTime'. The main analysis loop will update 'dataEndTimes' and 'maxIntPeriod' repeatedly. In addition, the optional 'paramUpdateFunction' may modify any other element of the list as well.

All additional arguments to LPI are collected as is to the parameter list.

'dataEndTimeFunction' Name of a function that returns sampling times of latest recorded samples. The function gets a list of all LPI input arguments as input and must return a named vector of POSIX times ("TX1", "TX2", "RX1", and "RX2"). The function should be reasonably fast, because it is called regularly.

Default: "currentTime"

Because LPI does not have its own I/O routines it cannot actually check availability of data, but it assumes that all integration periods from 'startTime' to 'dataEndTimes' are available. It will stop only if everything from 'startTime' to 'stopTime' is already analysed, or if everything from 'startTime' to 'dataEndTimes' is analysed and 'dataEndTimes' has not progressed despite waiting for 'maxWait.s' seconds. In real-time analysis it is extremely important that 'dataEndTimes' is never ahead of the actual sampling time of the latest sample that is available for 'dataInputFunction'.

Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>

Description
Expand vectors or lists to the LPI internal format with names "RX1", "RX2", "TX1", and "TX2"

Usage
LPIexpand.input( parvec)

Arguments
parvec A vector or list. See details.

Details
The input may be in several formats:
1. If parvec is readily in the internal format it is returned as such.
2. If parvec does not have named elements, it is repeated / truncated to length 4 and the elements are named in order "RX1", "RX2", "TX1", "TX2".
3. If parvec does not contain all the internally used names, but it contains also unnamed elements, the unnamed ones are repeated and named to fill the output.
4. Names 'RX' and 'TX' are expanded in an obvious way.
5. Names 'TR1' and 'TR2' are expanded in an obvious way.
stripACF

Value
A named vector or list of the internally used format

Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>

Examples

# Expansion of a single unnamed value
LPIexpand.input( 4 )

# A combination of named and unnamed elements is accepted
LPIexpand.input( c( RX1=2, 4 ) )

# The RX and TX entries
LPIexpand.input(c(RX=2,TX=3))

# The TR1 and TR2 entries
LPIexpand.input(c(TR1=2,TR2=3))

stripACF

Description
Strip unwanted ranges and lags from ACFs

Usage
stripACF(ACFlist, rgates, lags, fullCovar = FALSE)

Arguments
ACFlist An ACF list read from an LPI output file or returned by LPIrun.
rgates Indices of range gates to preserve
lags Indices of lags to preserve
fullCovar Logical, TRUE if the ACF list contains full covariance matrices.

Value
An ACF list with only the selected ranges and lags maintained.

Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>
Examples

## Not run:

# Load an LPI result file
load( "13455418100000LP.Rdata" )

# There will now be an ACF list on the workspace.
# Check if it contains a full covariance matrix
fullCov <- ifelse( is.null(ACF$covariance) , FALSE , TRUE )
# Select range gates 30 - 40 and lags 1, 4, and 5.
ACFstrip <- stripACF( ACF , seq(30,40) , c(1,4,5) , fullCov )

## End(Not run)
Index

*Topic package
  LPI-package, 1

LPI, 2, 2, 7, 8
LPI-package, 1
LPIexpand.input, 3, 7, 8

stripACF, 9
D.2: LPI.gdf package manual
Package ‘LPI.gdf’

October 29, 2013

Version 0.1-1

Date 2013-06-04

Title Lag Profile Inversion with gdf files

Author Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>

Maintainer Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>

Description gdf format I/O routines and wrapper function to LPI

License FreeBSD

Depends LPI

Copyright University of Oulu, Finland

R topics documented:

LPI.gdf-package .......................................................... 2
getDataEndTimes.gdf ................................................. 2
getFileSizeLengths.gdf ............................................. 3
getSamplingFrequencies.gdf ......................................... 4
getSamplingStartTimes.gdf .......................................... 5
LPI.gdf ................................................................. 6
LPIParam.default.gdf .................................................. 10
LPIsaveACF.gdf ........................................................ 10
readData.gdf ............................................................ 11
readLPIdata.gdf ........................................................ 12

Index 14
**getDataEndTimes.gdf**

---

**LPI.gdf-package**

*Lag Profile Inversion with gdf files*

**Description**

gdf format I/O routines and wrapper function to LPI

**Details**

<table>
<thead>
<tr>
<th>Package:</th>
<th>LPI.gdf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version:</td>
<td>0.1-1</td>
</tr>
<tr>
<td>Date:</td>
<td>2013-06-04</td>
</tr>
<tr>
<td>License:</td>
<td>FreeBSD</td>
</tr>
<tr>
<td>Depends:</td>
<td>LPI</td>
</tr>
<tr>
<td>Built:</td>
<td></td>
</tr>
</tbody>
</table>

Index:

- **LPI.gdf** Wrapper to LPI::LPI with gdf format specific parameter parsing.
- **readData.gdf** gdf data input
- **getDataEndTimes.gdf** gdf data end time function
- **getSamplingFrequencies.gdf** gdf format sample rate function
- **LPIparam.default.gdf** default gdf format input
- **readLPIdata.gdf** gdf data input function for LPI
- **getFileLengths.gdf** gdf format file lengths
- **getSamplingStartTimes.gdf** gdf data sampling start times

**Author(s)**

Ilkka Virtanen (University of Oulu, Finland)

<ilkka.i.virtanen@oulu.fi>

---

**getDataEndTimes.gdf**

**Description**

Read sampling times of latest data samples

**Usage**

```
getDataEndTimes.gdf( LPIparam )
```
getFileLengths.gdf

Arguments

LPIparam An LPI parameter list from LPI.gdf

Details

Following elements of the LPI parameter list are used in this function.

'dataDir' A named character vector that contains the data directories for each data type.

'fileNamePrefix' A named character vector that contains the file name prefixes for each data type.

'fileNameExtension' A named character vector that contains the file name extensions for each data type.

'dataFileLengths' A named numeric vector that contains number of samples in a single data file of each data type.

'dataSampleFreqs' A named numeric vector that contains sample rates of each data type in Hz.

'dataStartTimes' A named numeric vector that contains sampling times of the first samples of each data type in POSIX format.

Value

A named vector with elements "RX1","RX2","TX1","TX2", each of which is the sampling time of the latest recorded data sample of the corresponding type in POSIX format.

Author(s)

Ilkka Virtanen (University of Oulu, Finland)
<i>ilka.i.virtanen@oulu.fi</i>

Description

Read file lengths of gdf format data

Usage

getFileLengths.gdf( LPIparam )

Arguments

LPIparam An LPI parameter list from LPI.gdf
getSamplingFrequencies.gdf

Details
Following elements of the LPI parameter list are used in this function.

'dataDir’ A named character vector that contains the data directories for each data type.

'fileNamePrefix’ A named character vector that contains the file name prefixes for each data type.

'fileNameExtension’ A named character vector that contains the file name extensions for each data type.

Value
A named vector with elements "RX1","RX2","TX1","TX2", each of which is the number of samples in data files of the corresponding data type.

Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>

Description
Read sample rates of gdf format data

Usage
getSamplingFrequencies.gdf( LPIparam )

Arguments
LPIparam An LPI parameter list from LPI.gdf

Details
Following elements of the LPI parameter list are used in this function.

'dataDir’ A named character vector that contains the data directories for each data type.

Value
A named vector with elements "RX1","RX2","TX1","TX2", each of which is the sample rate of the corresponding data type in Hz.

Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>
Description
Read sampling start times from gdf format data directories

Usage
getSamplingStartTimes.gdf( LPIparam )

Arguments
LPIparam An LPI parameter list from LPI.gdf

Details
Following elements of the LPI parameter list are used in this function.

'dataDir' A named character vector that contains the data directories for each data type.

'timeFileNamePrefix' A named character vector that contains the file name prefixes for each data type.

'timeFileNameExtension' A named character vector that contains the file name extensions for each data type.

'dataFileLengths' A named numeric vector that contains number of samples in a single data file of each data type.

'dataSampleFrequents' A named numeric vector that contains sample rates of each data type in Hz.

Value
A named vector with elements "RX1","RX2","TX1","TX2", each of which is the sampling time of the first data sample of the corresponding type in POSIX format.

Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>
LPI.gdf

Description

Lag Profile Inversion with gdf files

Usage

LPI.gdf( ... )

Arguments

Arguments to be passed for functions that parse the final LPI input argument list. Accepts a reference to a stored input argument file and / or direct command line arguments. All parameters have default values which are used if the parameter is not found elsewhere.

Following arguments are effective:

'paramFile' A file containing any subset of the accepted input arguments. In conflict situations values from command line will override those read from file. See details.
Default: NULL

'clusterNodes' A list defining the computer cluster, see help of the main LPI package.
Default: list(tesla1=8,tesla2=8,tesla3=8,tesla4=8,tesla5=8,localControl=TRUE)

'beginTime' Analysis start time, c( year , month , day , hour , minute , seconds), see details.
Default: c(1970,1,1,0,0,0)

'endTime' Analysis end time, see details. c( year , month , day , hour , minute , seconds), see details.
Default: c(3000,1,1,0,0,0)

'timeRes.s' Analysis time-resolution (integration time) in seconds.
Default: 10

'maxWait.s' Maximum time to wait for new data before stopping the analysis, in seconds.
Default: -1

# Basic definitions of the analysis

'freqOffset.Hz' Frequency offset from baseband in Hz, see details.
Default: 0.0

'filterLength.us' Length of the boxcar-shaped post-detection filter in us.
Default: 10

'lagLimits.us' Limits of time-lag gates in us, see details.
Default: seq(50)*10

'rangeLimits.km' Limits of range gates in km, see details.
Default: c(seq(50,149.9,by=.9),seq(154.4,298.4,by=4.5),seq(343.4,793.5,by=45))
'maxRanges.km' Maximum range for each lag profile. Allows the number of range gates to be limited at selected lags. See details.
    Default: Inf

'nCode' Code cycle length for the optional lag profile pre-averaging, see help of the main LPI package.
    Default: NA

'backgroundEstimate' Optional background ACF suppression. See help of the main LPI package.
    Default: TRUE

'fullCovar' Full covariance matrix / variance calculation selection. See help of the main LPI package.
    Default: FALSE

'decodingFilter' Amplitude domain decoding filter selection. See help of the main LPI package.
    Default: "none"

'maxClutterRange.km' Maximum range at which ground clutter will be subtracted. Set below min(rangeLimits.km) in order to disable the clutter suppression. See details.
    Default: 60

'clutterFraction' The fraction of a full integration period used for the coherent ground clutter profile estimation. A float from between (0,1]. Values smaller than 1 should be used carefully. See help of the main LPI package.
    Default: 1.0

'remoteRX' Monostatic / bistatic analysis selection. See help of the main LPI package.
    Default: FALSE

'ambInterp' Range ambiguity function oversampling on / off switch. See help of the main LPI package.
    Default: FALSE

'indexShifts.us' Additional adjustments to TX and RX indices in us, see details.
    Default: list( TX=c(0,0) , RX=c(0,0) )

'normTX' Optional TX amplitude normalization. See help of the main LPI package
    Default: FALSE

'llhT' Latitude [deg], longitude [deg], and height [m] of the transmitter.
    Default: c( 69.58 , 19.23 , 86.00 ) # EISCAT Tromso

'llhR' Latitude [deg], longitude [deg], and height [m] of the receiver.
    Default: c( 69.58 , 19.23 , 86.00 ) # EISCAT Tromso

'azelT' Azimuth and elevation of the transmitter beam [deg]. Azimuth 0 ... 360, north=0, east=90.
    Default: c(0,90)

'azelR' Azimuth and elevation of the receiver beam.
    Default: c(0,90)

'radarFreq' Transmitter carrier frequency [Hz].
    Default: 224e6
'solver'  Inverse problem solver selection. See help of the main LPI package.
    Default: "fishs"

'nBuf'  Number of theory matrix rows to buffer before calling the solver function.
    Default: 10000

'rlips.options'  Additional options to the rlips solver. See rlips help for details.
    Default: list( type="c" , nbuf=1000 , workgroup.size=128)

'dataDir'  Raw voltage data directory / directories, see details.
    Default: "."  

'resultDir'  Output directory.
    Default: paste(format(Sys.time()," 

Details

The final argument list is constructed from default values, values read from the optional parameter file, and command line arguments. Priorities of these, from lowest to highest, are defaults, values from a file, and command line arguments. Name collisions are solved by selecting the option with highest priority. In other words, command line arguments will override both defaults and file input, and file input will override the defaults. Defaults are used only if nothing else is available.

'paramFile'  Usually a file that the analysis has automatically stored in a previous LPI run. The file is stored in the output directory ('resultDir') under the name "LPIparam.Rdata". Giving such a file as an input argument is the simplest way to restart an analysis with identical parameters. It is also possible to e.g. store parameters specific to an experiment in a file in order to avoid long input argument lists.

'lagLimits.us'  A vector of time lag limits. The vector is first divided with filterLength.us, then rounded to nearest integer, and finally possible duplicate values are removed. Output of this process is a vector lagLimits = c( l1, l2, l3, IN ). The analysis then integrates lags l1, l1+1, ... l2-1 (of decimated data) into lag profile 1, lags l2, l2+1, ... l3-1 into lag profile 2, etc. As an example, in order to decimate data to 10 us sample rate and to calculate all lags up to 100 us separately, one would use filterLength.us=10 and lagLimits.us = c(10,20,30,40,50,60,70,80,90,100,110). The steps in lagLimits.us do not need to be of equal length. It is possible to use coarser lag resolution at longer lags by selecting e.g. lagLimits.us = c(10,20,30,50,70,90,110).

'rangeLimits.km'  A vector of range-gate limits. The vector is first converted into time-delays assuming that signal propagates at speed of light, then decimated to filterLength.us sample rate, and possible duplicate values are removed. Range-gates are then defined in exactly the same way as lag-gates (see above).

'maxRanges.km'  Allows one to reduce the number of range-gates at selected lags. If the vector is shorter than lagLimits.us its last value is repeated as necessary. Defaults to Inf, which means that all lags are solved at all range gates defined in rangeLimits.km. maxRanges.km may be used in combined D/E/F-region experiments when correlations longer than certain limit are known to exist only in D-region. As an example, in order to solve first 30 lags at all range gates, but the longer ones only below 100 km, one can use maxRanges.km = c( rep( Inf , 30 ) , 100 ). Another use case is a dedicated D region experiment that makes use of voltage level decoding. Most time lags will then have zero range ambiguity functions and can be safely
skipped. Assuming that modulation bit length is 2 us and inter-pulse period is 1 ms, one could use e.g. the following combination: filterLength.us=2, decodingFilter='matched', lagLimits.us=c(1000,1002,2000,2002,3000,3002), maxRanges=c(Inf,0,Inf,0,Inf) which would solve three pulse-to-pulse lags and save the analysis from inspecting all time lags (only 1 in 500 of them can actually be measured).

'beginTime’ A vector c( year, month, day, hour, minute, seconds ) of the desired analysis start time. If data is available from the given time, this is the start time of the first integration period. If data is not available from this time, the first integration period is still aligned with 'beginTime'; and the analysis is started from the first available data sample that was recorded after 'beginTime'. Notice that, in order to make the software more suitable for real-time analysis, the *latest* available data is always used first, and the first integration period may thus be the last to actually analyse.

'endTime' A vector c( year, month, day, hour, minute, seconds ) of the analysis end time. Data samples recorded after "endTime" are not used.

'freqOffset.Hz' Frequency offset from baseband to the signal centre frequency. Currently only one frequency per data vector, i.e. either a single offset for all data, or different shifts for "RX1", "RX2", "TX1", and "TX2". See the details for "dataDir" for instructions.

'indexShifts.us' Additional adjustments to the TX (and) RX indices returned by 'dataInputFunction'. The user input is converted into a list with entries "RX1", "RX2", "TX1", and "TX2" in the way explained in details for "dataDir". Each entry is a two-element vector c( shift1, shift2 ) where, in case of TX, shift1 is the adjustment at rising edge of each pulse, and shift2 is the adjustment at falling edge of each pulse. For RX, shift1 is the adjustment at each start of reception, and shift2 at end of reception. All shifts are positive forwards, use negative values to adjust towards earlier times. As an example, if the recorded TX bit for "TX1" starts 10 us *before* the actual pulse, and ends 20 us *after* the actual end of the pulse, one should have indexShifts.us["TX1"] == c(10,-20).

'dataDir' Either a character string of a single data directory, or a named vector of several data directories. The internally used "dataDir" is always a vector with entries "RX1", "RX2", "TX1", and "TX2", some or all of which may be equal. The analysis calculates lagged products in between data samples from "RX1" and "RX2", and range ambiguity functions as lagged products in between data samples from "TX1" and "TX2". The internally used list is formed from user input as follows:

1. If the input vector contains any of the entries "RX1", "RX2", "TX1", "TX2", they are copied as such.
2. If any of the four entries is missing after step 1, the input is searched for "RX", which is used to replace missing "RX1" and / or "RX2". A similar search is performed for "TX".
3. If any of the four entries is missig after steps 1 and 2, the input is searched for "TX1", which is used to replace "RX1" and / or "TX1". A similar search is performed for "TR2"
4. If any of the four entries is missing after steps 1 to 3, the *unnamed* elements of the input vector are repeated until a vector of length 4 is formed, and its elements are named "RX1", "RX2", "TX1", and "TX2".
5. If still unsuccessfull, i.e. the input vector was either empty or it contained only named entries with unknown names, the analysis will stop with an error message.
Description

LPI default parameters

Usage

LPIparam.default.gdf()

Arguments

None

Value

A named list of LPI.gdf default arguments. See LPI.gdf for details.

Author(s)

Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>

Description

Save LPI results to files

Usage

LPIsaveACF.gdf( LPIparam , intPeriod , ACF )

Arguments

LPIparam  An LPI parameter list from LPI.gdf
intPeriod  Integration period number.
ACF        An ACF list. See LPIsaveACF from the main LPI package for details.
**readData.gdf**

**Value**

Result file name (invisibly)

**Author(s)**

Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>

**Description**

Read gdf data files

**Usage**

`readData.gdf(files, n=1e6, istart=1, bigEndian=TRUE)`

**Arguments**

- `files` A vector of data file names. The files are read in the same order in which they are in the vector.
- `n` Number of samples to read from each file, the vector is repeated as necessary to match its length with the "files" vector.
- `istart` Indices from which to start the reading in each file, repeated as necessary to match its length with the "files" vector.
- `bigEndian` Logical telling whether the files are in big endian (default) or little endian format.

**Value**

A list with the following entries:

- `cdata` A vector of complex data values.
- `idatar` Lowest bit in real part of the data. Should contain the PPS bits in radar measurements.
- `idatai` Lowest bit in imaginary part of the data. Should contain the TX bits in radar measurements.
- `ndata` Total number of data points read. This is always the sum of the number of samples *intended* to be read from each file. If `n` is larger than file length for any of the input files, vectors `cdata`, `idatar`, and `idatai` will contain arbitrary values correspondingly.
- `success` A logical indicating whether `ndata` points were successfully read (success = TRUE), or not (success = FALSE). e.g. any `(n + istart - 1)` larger than file length will lead to `success=FALSE`.
Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>

Examples

```r
## Not run:
# read all data from file "file1.gdf"
d <- readData.gdf( files = "file1.gdf" , n = file.info( "file1.gdf" )$size / 4 )

# read samples 1001 ... 10000 from "file1.gdf", samples 1 ... 10000 from
# "file2.gdf", and samples 1 ... 200 from "file3.gdf".
d <- readData.gdf( files = c("file1.gdf", "file2.gdf", "file3.gdf"),
n = c(9000, 10000, 200),
istart = c(1001, 1, 1) )
## End(Not run)
```

Description
Read one integration period of voltage level data stored in gdf format.

Usage

```
readLPIdata.gdf( LPIparam , intPeriod )
```

Arguments

- **LPIparam**: An LPI parameter list from `LPI.gdf`
- **intPeriod**: Integration period number. Integration periods are counted in steps of LPI-param$timeres.s, the period number 1 starting at LPIparam$beginTime.

Details

'**LPIparam contents**' Following components of the LPI parameter list are used for selecting the correct signal samples

- **startTime** 'beginTime' converted into POSIX format, i.e. second count from 1970-01-01 00:00:00.
- **dataStartTimes** A named vector with components 'RX1', 'RX2', 'TX1', and 'TX2'. Each element is sampling time of the first sample of corresponding data type. The times are in seconds in POSIX format.
`'dataSampleFreqs'` A named vector with components 'RX1', 'RX2', 'TX1', and 'TX2'. Each element is the sample rate of the corresponding data type in Hz.

`'timeRes.s'` Analysis time resolution (incoherent integration period) in seconds.

`'dataFileLengths'` A named vector with components 'RX1', 'RX2', 'TX1', and 'TX2'. Each element is the number of complex samples in one data file of the corresponding data type.

`'fileNamePrefix'` A named vector with components 'RX1', 'RX2', 'TX1', and 'TX2'. Each element contains the file name prefix of the corresponding data type as a string.

**Value**

A list with the following contents

- `'RX1'` First receiver samples.
- `'RX2'` Second receiver samples. Will be identical with 'RX1' in autocovariance function estimation.
- `'TX1'` First transmitter samples.
- `'TX2'` Second transmitter samples. Will usually be identical with 'TX1', but may be different e.g. in orthogonal polarization experiments.

`success` TRUE if all requested data was successfully read, FALSE otherwise.

The elements "RX1", "RX2", "TX1", and "TX2" are lists themselves. Their elements are

- `'cdata'` Complex sample vector
- `'idata'` Logical vector, TRUE if the sample should be used in LPI.
- `'ndata'` Number of samples in vectors cdata and idata

**Author(s)**

Ilkka Virtanen (University of Oulu, Finland)

<ilkka.i.virtanen@oulu.fi>
Index

*Topic package

  LPI.gdf-package, 2

getDataEndTimes.gdf, 2, 2
getFileLengths.gdf, 2, 3
getSamplingFrequencies.gdf, 2, 4
getSamplingStartTimes.gdf, 2, 5

LPI.gdf, 2–5, 6, 10, 12
LPI.gdf-package, 2
LPIparam.default.gdf, 2, 10
LPIsaveACF.gdf, 10

readData.gdf, 2, 11
readLPIdata.gdf, 2, 12
D.3: LPI.KAIRA package manual
Package ‘LPI.KAIRA’

October 29, 2013

Version 0.1-3
Date 2013-10-22
Title Lag Profile Inversion with KAIRA data
Author Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
Maintainer Ilkka Virtanen <ilkka.i.virtanen@oulu.fi>
Description Lag Profile Inversion using KAIRA klp data
License FreeBSD
Depends LPI.gdf

Copyright University of Oulu, Finland

R topics documented:

LPI.KAIRA-package .................................................. 1
LPI.KAIRA ............................................................. 2
LPIparam.default.KAIRA ........................................... 6
readLPIdata.KAIRA .................................................. 7

Index

LPI.KAIRA-package  Lag Profile Inversion for KAIRA

Description
Lag Profile Inversion with KAIRA klp files
Details

Package: LPI.KAIRA
Version: 0.1-3
Date: 2013-10-22
License: FreeBSD
Depends: LPI.gdf

Index:

LPI.KAIRA Lag Profile Inversion for KAIRA
readLPIdata.KAIRA Raw data input function for KAIRA
LPIparam.default.KAIRA Default LPI.KAIRA parameters

Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>

Description
Lag Profile Inversion for KAIRA

Usage
LPI.KAIRA( ... )

Arguments
... Arguments to be passed for functions that parse the final LPI input argument list. Accepts a reference to a stored input argument file and / or direct command line arguments. All parameters have default values which are used if the parameter is not found elsewhere.

Following arguments are effective:

'paramFile' A file containing any subset of the accepted input arguments. In conflict situations values from command line will override those read from file. See details.
Default: NULL

'clusterNodes' A list defining the computer cluster. See help of the main LPI package.
Default: list(tesla1=8,tesla2=8,tesla3=8,tesla4=8,tesla5=8,localControl=TRUE)
'begintime'  Analysis start time, c( year , month , day , hour , minute , seconds), see details.
  Default: c(1970,1,1,0,0,0)

'endtime'  Analysis end time, c( year , month , day , hour , minute , seconds),
  see details.
  Default: c(3000,1,1,0,0,0)

'timeRes.s'  Analysis time-resolution (integration time) in seconds.
  Default: 10

'maxWait.s'  Maximum time to wait for new data before stopping the analysis,
  in seconds.
  Default: -1

'freqOffset.Hz'  Frequency offset from baseband in Hz.
  Default: 0.0

'filterLength.us'  Length of the boxcar-shaped post-detection filter in us.
  Default: 10

'lagLimits.us'  Limits of time-lag gates in us, see details.
  Default: seq(50)*10

'rangeLimits.km'  Limits of range gates in km, see details.
  Default: c(seq(50,149.9,by=.9),seq(154.4,298.4,by=4.5),seq(343.4,793.5,by=45))

'maxRanges.km'  Maximum range for each lag profile. Allows the number of
  range gates to be limited at selected lags. See details.
  Default: Inf

'nCode'  Code cycle length for the optional lag profile pre-averaging, see help
  of the main LPI package.
  Default: NA

'backgroundEstimate'  Optional background ACF suppression. See help of
  the main LPI package.
  Default: TRUE

'fullCovar'  Full covariance matrix / variance calculation selection. See help of
  the main LPI package.
  Default: FALSE

'decodingFilter'  Amplitude domain decoding filter selection. See help of the
  main LPI package.
  Default: "none"

'maxClutterRange.km'  Maximum range at which ground clutter will be sub-
  tracted. Set below min(rangeLimits.km) in order to disable the clutter sup-
  pression. See details.
  Default: 60

'clutterFraction'  The fraction of a full integration period used for the coher-
  ent ground clutter profile estimation. A float from between (0,1]. Values
  smaller than 1 should be used carefully. See help of the main LPI package.
  Default: 1.0

'amblInterp'  Optional oversampling of range ambiguity functions. See help of
  the main LPI package.
  Default: FALSE

'indexShifts.us'  Additional adjustments to TX and RX indices in us, see de-
  tails.
Default: list( TX=c(0,0) , RX=c(0,0) )

'normTX' Optional TX amplitude normalization. See help of the main LPI package.
Default: FALSE

'azelT' Azimuth and elevation of the transmitter beam [deg]. Azimuth 0 ... 360, north=0, east=90.
Default: c(0,90)

'azelR' Azimuth and elevation of the receiver beam.
Default: c(313.95,45)

'radarFreq' Transmitter carrier frequency [Hz].
Default: 224e6

'solver' Inverse problem solver selection. See help of the main LPI package.
Default: "fishs"

'nBuf' Number of theory matrix rows to buffer before calling the solver function.
Default: 10000

'rlips.options' Additional options to the rlips solver. See rlips help for details.
Default: list( type="c" , nbuf=1000 , workgroup.size=128)

# Data I/O
'dataDir' Raw voltage data directory / directories, see details.
Default: "."

'beamlets' KAIRA beamlet numbers to combine in data vector, see details.
Default: list(RX1=0,RX2=0,TX1=c(),TX2=c())

'polarization' KAIRA x/y polarization selection.
Default: c(RX1="y",RX2="y",TX1="",TX2="")

'resultDir' Output directory.
Default: paste(format(Sys.time(),""

Details

This version is only for bistatic ACF estimation with TX samples from Tromso and RX samples from KAIRA. The way sample rates and timestamps are read does not currently allow one to cross-correlate data from Tromso and KAIRA (but does anyone ever need that?).

The final argument list is constructed from default values, values read from the optional parameter file, and command line arguments. Priorities of these, from lowest to highest, are defaults, values from a file, and command line arguments. Name collisions are solved by selecting the option with highest priority. In other words, command line arguments will override both defaults and file input, and file input will override the defaults. Defaults are used only if nothing else is available.

'paramFile' is usually a file that the analysis has automatically stored in a previous LPI run. The file is stored in the output directory ('resultDir') under the name "LPIparam.Rdata". Giving such a file as an input argument is the simplest way to restart an analysis with identical parameters. It is also possible to e.g. store parameters specific to a radar facility in a file in order to avoid long input argument lists.

'lagLimits.us' A vector of time lag limits. The vector is first divided with filterLength.us, then rounded to nearest integer, and finally possible duplicate values are removed. Output of
this process is a vector lagLimits = c( l1, l2, l3, lN ). The analysis then integrates lags l1, l1+1, ... l2-1 (of decimated data) into lag profile 1, lags l2, l2+1, ... l3-1 into lag profile 2, etc. As an example, in order to decimate data to 10 us sample rate and to calculate all lags up to 100 us separately, one would use filterLength.us=10 and lagLimits.us = c(10,20,30,40,50,60,70,80,90,100). The steps in lagLimits.us do not need to be of equal length. It is possible to use coarser lag resolution at longer lags by selecting e.g. lagLimits.us = c(10,20,30,50,70,90,110).

'rangeLimits.km' A vector of range-gate limits. The vector is first converted into time-delays assuming that signal propagates at speed of light, then decimated to filterLength.us sample rate, and possible duplicate values are removed. Range-gates are then defined in exactly the same way as lag-gates (see above).

'maxRanges.km' Allows one to reduce the number of range-gates at selected lags. If the vector is shorter than lagLimits.us its last value is repeated as necessary. Defaults to Inf, which means that all lags are solved at all range gates defined in rangeLimits.km. maxRanges.km may be used in combined D/E/F-region experiments when correlations longer than certain limit are known to exist only in D-region. As an example, in order to solve first 30 lags at all range gates, but the longer ones only below 100 km, one can use maxRanges.km = c( rep( Inf , 30 ) , 100 ).

Another use case is a dedicated D region experiment that makes use of voltage level decoding. Most time lags will then have zero range ambiguity functions and can be safely skipped. Assuming that modulation bit length is 2 us and inter-pulse period is 1 ms, one could use e.g. the following combination: filterLength.us=2, decodingFilter='matched', lagLimits.us=c(1000,1002,2000,2002,3000,3002), maxRanges=c(Inf,0,Inf,0,Inf) which would solve three pulse-to-pulse lags and save the analysis from inspecting all time lags (only 1 in 500 of them can actually be measured).

'beginTime' A vector c( year, month, day, hour, minute, seconds ) of the desired analysis start time. If data is available from the given time, this is the start time of the first integration period. If data is not available from this time, the first integration period is still aligned with "beginTime", and the analysis is started from the first available data sample that was recorded after "beginTime". Notice that, in order to make the software more suitable for real-time analysis, the *latest* available data is always used first, and the first integration period may thus be the last to actually analyse.

'endTime' A vector c( year, month, day, hour, minute, seconds ) of the analysis end time. Data samples recorded after "endTime" are not used.

'freqOffset.Hz' Frequency offset from baseband to the signal centre frequency. Currently only one frequency per data vector, i.e. either a single offset for all data, or different shifts for "RX1", "RX2", "TX1", and "TX2". See the details for "dataDir" for instructions.

'indexShifts.us' Additional adjustments to the TX (and) RX indices returned by 'dataInputFunction'. The user input is converted into a list with entries "RX1", "RX2", "TX1", and "TX2" in the way explained in details for "dataDir". Each entry is a two-element vector c( shift1, shift2 ) where, in case of TX, shift1 is the adjustment at rising edge of each pulse, and shift2 is the adjustment at falling edge of each pulse. For RX, shift1 is the adjustment at each start of reception, and shift2 at end of reception. All shifts are positive forwards, use negative values to adjust towards earlier times. As an example, if the recorded TX bit for "TX1" starts 10 us *before* the actual pulse, and ends 20 us *after* the actual end of the pulse, one should have indexShifts.us["TX1"] == c(10,-20).
'dataDir' Either a character string of a single data directory, or a named vector of several data directories. The internally used "dataDir" is always a vector with entries "RX1", "RX2", "TX1", and "TX2", some or all of which may be equal. The analysis calculates lagged products in between data samples from "RX1" and "RX2", or from their subdirectories if 'beamlets' and 'polarization' are defined, and range ambiguity functions as lagged products in between data samples from "TX1" and "TX2". The internally used list is formed from user input as follows:
1. If the input vector contains any of the entries "RX1", "RX2", "TX1", "TX2", they are copied as such.
2. If any of the four entries is missing after step 1, the input is searched for "RX", which is used to replace missing "RX1" and / or "RX2". A similar search is performed for "TX".
3. If any of the four entries is missing after steps 1 and 2, the input is searched for "TR1", which is used to replace "RX1" and / or "TX1". A similar search is performed for "TR2".
4. If any of the four entries is missing after steps 1 to 3, the "unnamed" elements of the input vector are repeated until a vector of length 4 is formed, and its elements are named "RX1", "RX2", "TX1", and "TX2".
5. If still unsuccessful, i.e. the input vector was either empty or it contained only named entries with unknown names, the analysis will stop with an error message.

Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>
Description

Read one integration period of voltage level data. Transmitter samples are recorded with USRP at Tromso and receiver samples with KLP at Kilpisjarvi.

Usage

```
readLPIdata.KAIRA( LPIparam, intPeriod )
```

Arguments

- **LPIparam**: An LPI parameter list from LPI.KAIRA
- **intPeriod**: Integration period number. Integration periods are counted in steps of LPI-param$timeres.s, the period number 1 starting at LPIparam$startTime.

Details

- **LPIparam contents**: Following components of the LPI parameter list are used for selecting the correct signal samples
  - **startTime**: 'beginTime' converted into POSIX format, i.e. second count from 1970-01-01 00:00:00.
  - **dataStartTimes**: A named vector with components 'RX1', 'RX2', 'TX1', and 'TX2'. Each element is sampling time of the first sample of corresponding data type. The times are in seconds in POSIX format.
  - **dataSampleFreqs**: A named vector with components 'RX1', 'RX2', 'TX1', and 'TX2'. Each element is the sample rate of the corresponding data type in Hz.
  - **timeRes.s**: Analysis time resolution (incoherent integration period) in seconds.
  - **dataFileLengths**: A named vector with components 'RX1', 'RX2', 'TX1', and 'TX2'. Each element is the number of complex samples in one data file of the corresponding data type.
  - **fileNamePrefix**: A named vector with components 'RX1', 'RX2', 'TX1', and 'TX2'. Each element contains the file name prefix of the corresponding data type as a string.

Value

A list with the following contents

- **'RX1'**: First receiver samples
- **'RX2'**: Second receiver samples. Will be identical with 'RX1' in autocovariance function estimation
- **'TX1'**: First transmitter samples
- **'TX2'**: Second transmitter samples. Will usually be identical with 'TX1', but may be different e.g. in orthogonal polarization experiments.
success TRUE if all requested data was successfully read, FALSE otherwise.

The elements "RX1", "RX2", "TX1", and "TX2" are lists themselves with elements

'cdata' Complex data vector.
'idata' Logical vector, TRUE for samples that should be used in LPI.
'ndata' Number of samples in the data vectors.

Author(s)
Ilkka Virtanen (University of Oulu, Finland)
<ilkka.i.virtanen@oulu.fi>

See Also
LPI.gdf, LPI
Index

*Topic package
  LPI.KAIRA-package, 1
  LPI.KAIRA, 2, 2, 6, 7
  LPI.KAIRA-package, 1
  LPIparam.default.KAIRA, 2, 6
  readLPIdata.KAIRA, 2, 7