# Large-Scale Linear Computations With Dedicated Real-Time Architectures

Patrick Dewilde and Klaus Diepold
Technische Universität München

**Abstract**

Signal processing algorithms are intimately related with numerical linear algebra. The efficient implementation of such algorithms on dedicated real-time architectures requires a good understanding of the interplay between algorithms and architectures. The family of algorithms based on the QR decomposition of a coefficient matrix serves as a key example to review this interplay. We demonstrate how the technical requirements of efficient computing architectures influences the choice of algorithmic options as well as the algorithmic properties have an influence on the specifics of computing architectures. We discuss application examples, which benefit from the interplay between algorithms and real-time architectures. The presented QR-based computational framework also allows to present an elegant way to formulate the Kalman filtering approach starting with a computational viewpoint. Although the paper may present original or not so well known viewpoints, it is mostly of a tutorial nature.

## 1 Introduction

Understanding the connection between algorithms and solvers for large scale systems on the one hand, and appropriate architectures that execute them efficiently on the other is key to the effective design of modern signal processing applications. This trend has started with the emergence of digital media, large scale signal processing for image coding and analysis, digital mobile telephony and digital processing in medical imaging. In many cases dedicated, hardware or software dominated methods on a single processor have been used, only in recent times more generic methods based on general purpose array architectures, either dedicated to media processing or for general computing have become realizable. Massive use of parallelism becomes attractive and should, in the future, allow us to tackle large scale problems in a streamlined fashion. It is no exaggeration to state that this trend was started a long time ago when Georg Färber proposed parallel processing schemes for signal processing and control using coprocessors on standard busses and proceeded to prove his ideas in practice, thereby creating a company that set a standard in the field [7] - he was clearly many decades ahead of the field!

In this contribution to this volume to honor the attainment of the status of 'Professor Emeritus' by Georg Färber, we review a number of what we consider very attractive cases where the connection between algorithm and executing architecture proves to be very effective. Luckily, these cases handle

some of the most important algorithms in numerical linear algebra. The resulting combination algorithms-architectures brings the fields of signal processing and linear algebra together, a dream that has been somewhat elusive over the years, because programming paradigms and hardware design methods were not well adapted to each other. Work has to be done, both at the side of the choice of algorithm and at the side of the architectural design. Although we cannot go into the details of neither the most sophisticated algorithms nor the details of the design methodology, we can easily illustrate the principles. Essential is the combination of the choice of algorithm with the architectural consequences, the designer handles algorithm and architecture at the same time, to achieve a result in which both sides are optimally adapted on each other.

An important issue in signal processing is numerical stability and robustness of the algorithms used. The system designer should not deteriorate the numerical properties of his problem by introducing computational steps that degrade the conditioning and hence the quality of the computed results. The conditioning of a problem is defined as a measure of the sensitivity of the result to variations of the input data, or more precisely, how much errors in the input data are propagated to the output by the implemented mathematical function.

Jacobi's QR factorization is one of the numerical algorithms, which exhibits very favorable numerical properties, which it combines with great opportunities for parallelization [17]. The QR factorization sits at the core of solutions for many important technical problems: the solution of linear equations, channel estimation and signal identification in telecommunication, Kalman filtering (in the square root version) and H-infinity control. It also represents a core function in an iterative loop for computing eigenvalue and singular value decompositions (abbreviated as EVD and SVD, respectively). Although the QR factorization is used a lot, often hidden in embedded software, it is not as well known as it deserves. To honor Jacobi, we start the paper with a description of the algorithm, and an account of its main applications.

The matrices associated with real technical problems exhibit often special structural properties. These properties are exploited by an alternative class of algorithms for improved performance or for reduced computational complexity. Foremost for large system solvers are the iterative algorithms used to handle e.g. sparse matrices. We give an account of some of the major methods and the resulting architectures. Structure also plays a role in direct solvers. A very important type of structure is called 'semi-separable' or 'quasi-separable' (the terminology has not stabilized.) Here, both direct and iterative methods play a role, we give a brief account.

To connect algorithms to architectures, we make systematic use of 'data flow graphs', sometimes called dependence or precedence graphs. These are actually generalizations of the signal flow graphs classically used in the signal processing literature. They have been adopted in commercial design packages [2] and give the designer a convenient way to control the parallelization process without having to resort to complicated and often wieldy design tools.

# 2 Algorithms and Architectures

## 2.1 Technical Applications of Numerical Linear Algebra

Solving linear systems of equations or computing a least squares solution for an overdetermined system of equations belong to the most common computational tasks in science and engineering. Engineered products and services are relying on the capability of real-time systems to solve such problems fast, accurate and in a robust way. Examples for this statement are Kalman-filtering [20], rake-receivers in mobile communications [38], adaptive channel equalizers, adaptive beamforming [34], in stereo vision systems [19], to name just a few. Another set of applications using linear systems of equations originates from CAD tools and circuit simulators [9], structural analysis using finite element methods or partial differential equations [37].

Researchers in the domain of Numerical Linear Algebra have worked since the dawn of the modern computer on devising effective numerical methods for solving systems of equations of ever increasing dimension. Therefore, we suffer no shortage of published results and practical implementations of system solvers, which are readily available and widely used in terms of highly optimized software packages such as the LINPACK, LaPACK or IMSL libraries, as well as in software packages like Matlab, Mathematica, Octave, or Scilab.

## 2.2 Dense Matrices and Direct Algorithms

Direct solution schemes use factorizations of the coefficient matrix, which allow to map the original problem on a problem involving a triangular matrix. Examples are the LU factorization $T = L \cdot U$ for a square coefficient matrix $T$ and the lower and upper triangular factors $L$ and $U$, respectively. For a symmetric positive definite coefficient matrix (prime indicates transpose) $T = T', T > 0$ we can compute the Cholesky factorization $T = R' \cdot R$, where $R$ is upper triangular. Note that both factorizations require pivoting schemes to achieve numerical stability [17]. Pivoting is an effective method to improve numerics, but it destroys the regular data flow because of additional control structures and branching.

Factorization algorithms, which are based on orthogonal elementary operations, such as the QR decomposition, satisfy the need for numerically reliable computations without resorting to pivoting [17]. A solution strategy which computes the QR-decomposition of the coefficient matrix using elementary Jacobi (Givens) rotations has the added benefits to be amenable for parallelization on highly local and regular architectures.

Solving dense systems of linear equations takes $\mathcal{O}(n^3)$ operations [17], where $n$ denotes the size of the coefficient matrix. This computational effort may be overwhelming in case of large $n$. In many signal processing applications the matrices involved may have moderate values for $n$, for which the computational burden may be challenging for real-time application, but they comprise 'structure', that is, the matrices have only $\mathcal{O}(n)$ parameters. The structure in the matrix allows for solution algorithms, which require only $\mathcal{O}(n^2)$ operations. Typical examples for such structured matrices are diagonal matrices or Toeplitz or Hankel matrices [24].

### 2.3 Square-Root and Array Algorithms

Real-time computer systems gain additional advantages in terms of numerical robustness and reliability if the implemented algorithms operate directly on the data of the coefficient matrix instead of setting up normal equations. Computing the normal equation squares the condition number of the problem; this leads to a dramatic loss in precision for the result, if the coefficient matrix is badly conditioned and hence more sensitive to rounding errors and other imperfections of finite word length computations. Algorithms which work directly on the data are often referred to as 'square-root algorithms', because they avoid 'squaring' the data when determining the covariance matrices that come with approaches based on solving the normal equation. The combination of 'square-root' approaches and orthogonal elementary transformations leads to a family of so-called 'array' algorithms, which exploit an elementary identity known as Schur-complements and which are highly suitable for being mapped onto parallel computer architectures [31], [25].

### 2.4 Algorithms and Architectures

Applications running on real-time computer systems require that computations are completed with a pre-determined deadline, that the results are computed in a reliable way having an accuracy that is robust against perturbations and noise. Furthermore, favorable algorithms shall provide a high level of locality and parallelism [28]. For large scale real time architectures it would be very attractive to dispose of a generic high level algorithm that at the same time solves major problems, and at the other is amenable to real time realization, utilizing the available resources to a maximum. The requirements one can put on such an algorithm are:

- *parametrizable:* the algorithm should be able to handle problems of any size, even though the available resources are limited;
- *numerically accurate:* the algorithm should be numerically backward stable with errors close to the conditioning of the problem;
- *parallelizable:* there should be a natural way to partition the algorithm in chunks that operate in parallel and can be mapped to the underlaying architecture;
- *localizable:* both data transport and memory usage should be highly local, no massive storage needed during the algorithm nor massive data transport or data manipulation;
- *generic:* the algorithm should be able to handle many if not most large scale computations - much like multiplication and addition, which can handle most numerical problems as well;
- *incremental:* when additional data becomes available, the algorithm gracefully adapts.

## 3 The QR Algorithm As A Generic Method

The algorithm for computing the QR factorization was originally proposed by Jacobi and is an excellent candidate algorithm to satisfy the requirements list given in the previous section. We

start with a description of the algorithm, followed by a methodology to derive attractive, real-time architectures for it. We stop at the architectural level, but enough detail of the strategy will transpire to allow for attractive, concrete realizations.

### 3.1 The QR algorithm

The most attractive way to present QR is on a very common example. Suppose $T$ is a rectangular, tall matrix of dimension $m \times n$, $y$ a given vector of dimension $m$ and suppose we are interested in finding a vector $u$ of dimension $n$ such that $Tu$ is as close as possible to $y$ (we assume real arithmetic throughout. With slight modifications complex arithmetic or even finite field calculations are possible as well but beyond our present scope.) Numerical analysts call such a problem 'solving an overdetermined system'. It occurs in a situation where $u$ is a set of unknown parameters, row $i$ of $T$ consist of noisy data, which, when combined linearly with $u$ produce the measured result $y_i$. The situation occurs very often in measurement setups, where repeated experiments are done to reveal the unknown parameters, or in telecommunication where transmitted signals have to be estimated from the received signals (we omit the details.) The most common measure of accuracy is 'least squares', for a vector $u$ with components $u_i$ we write

$$\|u\|_2 = \sqrt{\sum_{i=1}^{n} |u_i|^2} \tag{1}$$

There may be more than one solution to the minimization of $\|Tu - y\|_2$, often one is interested in the least squares, so one tries to solve

$$u_{\min} = \text{argmin}_w \|(w = \text{argmin}_u \|Tu - y\|_2)\|_2 \tag{2}$$

the actual minimum being the estimation error. The strategy is to perform a QR decomposition of $T$. The matrix $Q$ has to be an orthogonal matrix (a generalized rotation), which keeps the norm of the vectors to which it is applied, while $R$ should be an upper triangular matrix. Let $Q'$ be the transpose of $Q$, then orthogonality means $Q'Q = QQ' = I$, $Q'$ is actually the reverse rotation. Let us just assume that $Q$ and $R$ can be found (see further), and let us apply $Q'$ to $y$, to obtain $\eta = Q'y$, then we have $QRu = y$ and hence $Ru = Q'y = \eta$. $R$ has the same dimensions as $T$, meaning that it is a tall matrix. It is also upper triangular, meaning that it has the form

$$R = \begin{bmatrix} R_u \\ 0 \end{bmatrix} \tag{3}$$

in which $R_u$ is now $n \times n$ square and upper triangular. We find that the solution $u$ must minimize

$$\|(\begin{bmatrix} R_u u \\ 0 \end{bmatrix} - \eta)\|_2. \tag{4}$$

If we partition $\eta = \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix}$ with $\eta_1$ of dimension $n$, we see that the minimal solutions must satisfy the square system $R_u u = \eta_1$ and that $\eta_2$ certainly contributes wholly to the error, there is nothing we can do about it. If $R_u$ is non-singular, i.e. if the original system has a full row basis, then the solution will be unique, i.e. $u = R_u^{-1}\eta_1$ and the error will be $\|\eta_2\|_2$. If that is not the case,

further analysis will be necessary, but the dimension of the problem is reduced to $n$, the number of parameters, from $m$, the number of measurements (usually much larger.) It may appear that the QR factorization step is not sufficient, it has to be followed by a 'back substitution' to solve $R_u u = \eta_1$, but this difficulty can be circumvented (see further.) Here we want to concentrate just on the QR step and its possible architectures.

### 3.2 The Basic Step: Jacobi Rotations

The elementary Jacobi matrix is a rotation over an angle $\theta$ in the 2D plane (for convenience we define $Q'$):

$$Q' = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{5}$$

Let's abbreviate to $Q' = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ and apply the rotation to two row vectors:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_1 & a_2 & \cdots & a_n \\ b_1 & b_2 & \cdots & b_n \end{bmatrix} = \begin{bmatrix} \sqrt{a_1^2+b_1^2} & ca_2+sb_2 & \cdots \\ 0 & -sa_2+cb_2 & \cdots \end{bmatrix} \tag{6}$$

which is achieved by choosing $c = \frac{a_1}{\sqrt{a_1^2+b_1^2}}$, $s = \frac{b_1}{\sqrt{a_1^2+b_1^2}}$ (and hence automatically $\tan\theta = \frac{b_1}{a_1}$.) In this way one can treat the entries of the original matrix $T$ row by row and create all the zeros below the main diagonal. With a $4 \times 3$ matrix this works as shown below. The only thing one must do is embed the $2 \times 2$ rotation matrices in the $4 \times 4$ schema, so that unaffected rows remain unchanged. We label the rotation matrices with the indices of the rows they affect - we indicate affected entries after each step with a $\star$:

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \xrightarrow{Q'_{1,2}} \begin{bmatrix} \star & \star & \star \\ 0 & \star & \star \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \xrightarrow{Q'_{1,3}} \begin{bmatrix} \star & \star & \star \\ 0 & \cdot & \cdot \\ 0 & \star & \star \\ \cdot & \cdot & \cdot \end{bmatrix} \xrightarrow{Q'_{1,4}} \begin{bmatrix} \star & \star & \star \\ 0 & \cdot & \cdot \\ 0 & \cdot & \cdot \\ 0 & \star & \star \end{bmatrix} \xrightarrow{Q'_{2,3}} \begin{bmatrix} \cdot & \cdot & \cdot \\ 0 & \star & \star \\ 0 & 0 & \star \\ 0 & \cdot & \cdot \end{bmatrix} \xrightarrow{Q'_{2,4}} \begin{bmatrix} \cdot & \cdot & \cdot \\ 0 & \star & \star \\ 0 & 0 & \cdot \\ 0 & 0 & \star \end{bmatrix} \tag{7}$$

and the final step is a $Q'_{3,4}$ which annihilates the $4,3$ entry. In each of these subsequent steps, the first operation determines the rotation matrix and then applies it to all the entries in the respective rows, skipping the already computed zero entries (which remain zero.) The overall rotation matrix is then $Q = Q_{1,2}Q_{1,3}Q_{1,4}Q_{2,3}Q_{2,4}Q_{3,4}$. In most cases it need not be put in memory (and if so there are tricks.) It turns out that this algorithm leads to a very regular computational schema, now known as the 'Gentleman-Kung array', which we discuss in the next section.

The result of a QR factorization need not be in strict triangular form, the actual more general form is called an i.e., echelon form. It may happen that in the course of the algorithm, when the processor moves from one column to the next, an actual sub-column of zeros is discovered. In that case no rotation is necessary and the processor can move to the next sub-column, which again might be zero etc... until a sub-column is reached with non zero elements. The result will then have the form shown in fig. 1. The QR algorithm compresses the row data of the matrix in the North-East corner, leaving the norm of each relevant sub-column as leftmost non zero element. There is a dual version, called the LQ algorithm that compresses the columns in the South-West corner, and one can of course also construct versions for the other corners (but these will not bring

much additional information.) One can take care of the zero or kernel structure exemplified by the QR or LQ algorithms by testing on zero inputs when Jacobi rotations are applied, this can be arranged for automatically, we shall henceforth just assume that these provisions have been taken.

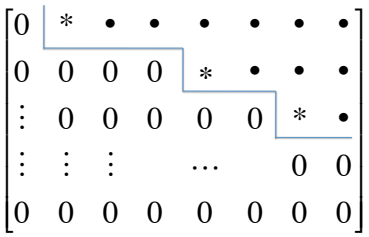

Figure 1: Example of an echelon form. Elements indicated with a '$\star$' are strictly positive. The QR algorithm compresses the rows to the North-Eastern corner of the matrix and generates a basis for the rows of the matrix.

### 3.3 The Gentleman-Kung Array

An important step for connecting the algorithmic information with potential architectures is the definition of a 'dependence graph', sometimes called a 'sequencing graph'. The graph exhibits operations as nodes and data transport as dependencies between nodes. In the case of the QR algorithm just defined, we have two types of operations: Type 1 consists in the computation of sine and cosine of an angle given two values (say $a$ and $b$) and the subsequent rotation of these two values to $\sqrt{a^2+b^2}$ and 0, while the second operation just applies the rotation to subsequent data on the same rows. The array is upper triangular, for each position $(i,j)$ in the upper part of the resulting matrix $R_u$ there is one processor, the processors on the diagonal are of the first type, while the others are of the second type. Angle information is propagated along the rows, while the data is inputed, row by row, along the columns. It gives a precise rendition of the operation-data dependencies in the original algorithm. From the view of a compiler, the graph represents a 'Single Assignment Code (SAC)' rendition of the original algorithm, there is no re-use neither of operations nor of memory elements. Actually, the local memory in this representation is reduced to the end result, the $r_{i,j}$ of the final matrix, one per processor, all other data is communicated either from the environment to the area, or from one processor to the next. In this algorithm, some data can actually be 'broadcast', namely the information about the angles $(c_i, s_i)$, along the rows, it might not be a bad idea to make special arrangements for this. The array shows what the architecture designer has to know about the algorithm. If only the type of the various data is specified and the operation in each node, he has enough information to design the architecture - we describe briefly what the architecture designer can do next in a further section, but before doing so we show how various standard problems can be solved with the array or interesting extensions of it.

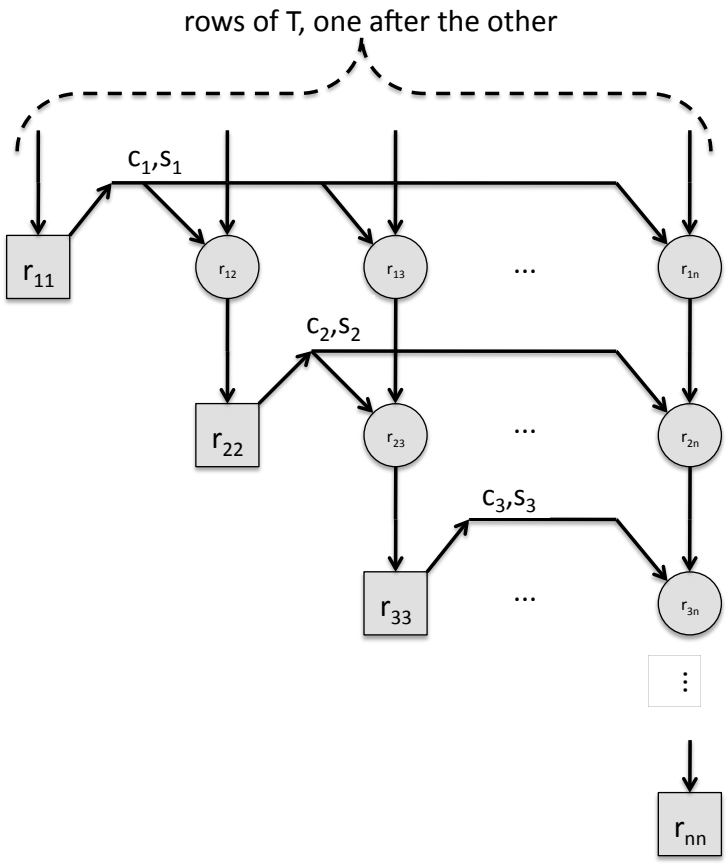


Figure 2: The Gentleman-Kung or Jacobi array. The squares represent 'vectorizing rotors', they compute the Jacobi angle information from the data and perform the first rotation, the circle are 'rotors', they rotate a two-dimensional vector over the given angle.

# 4 Architecture Design Strategies For Parallelization Of The QR Algorithm

Once a high level algorithm can be represented as a regular array, in which nodes represent operations and arrows data transport, the issue arises how these operations and data can best be mapped to an architecture consisting of actual processors, memories, busses and control. In the following sections we shall see that (extensions) of the Gentleman-Kung array succeeds in solving major problems of linear algebra, estimation and control. These problems may have any dimension, they are parametrized by their size, often indicated as $n$. It seems logical that the most attractive type of concrete architecture to map to would be an array itself, consisting of processing nodes that contain provisions for the necessary operations (in our case vectorization and rotation), local memory in each processor for intermediate results, and an infrastructure surrounding the array, whose task it is to provide the array with data and to collect results when they become available. Such a processor array will typically have a small dimension, e.g. $3 \times 3$ to mention a commercial

size. So the issue becomes the mapping of an arbitrarily (parametrized) SAC array onto potential, architecturally viable sizes. When the computational array is regular, this assignment can be done in a regular fashion as well. As data transport becomes a paramount issue in performance, it becomes important to use local memory as much as possible. The architectural problem becomes in the first place a problem of efficient memory usage, or, to put it more simply, to perform the partitioning and mapping to resources of the original array in such a way that local memory is fully used before data are sent to background memory. We can offer two general partitioning strategies to achieve this feat, whose combination allows to first exhaust local memory and then map well chosen remainder to background. They have been called with various names in the literature, here we call them 'LPGS' and 'LSGP' - for 'Local Parallel Global Sequential' and 'Local Sequential Global Parallel' [6]. We suffice here to describe these two strategies, there are many more, more detailed ones, but these would go beyond the scope of the present paper.

In each of the two strategies, the original array will first be 'tesselated', i.e. decomposed in tiles. Although not strictly necessary, we shall assume our tiles to be square, just to illustrate the methods. In a first approach, we have two choices: either we choose the tiles so big that the total array of tiles actually equals the processor array we envisage, and then we map each tile just to one processor, or we take tiles so large that they are exactly isomorphic to the processor array. The first strategy is LSGP - the operations within one tile will be executed sequentially on one processor. As a consequence of this strategy, the intermediary data that is produced by one operation either has to be mapped to local memory (if the operation that shall use the data belongs to the same tile) or it has to be shoveled to background memory (if the operation that shall use the data is in another tile.) Clearly, LSGP will use a lot of local memory and only be feasible if the size of the tiles are small enough so that all local intermediary results can also be stored locally. The opposite strategy is LPGS. Here the size of the tile would be chosen exactly equal to the size of the processor array, and the only local storage needed is what was already assigned to the individual nodes, all the rest goes to the background memory. The two strategies are illustrated in fig. 3 and fig. 4 respect.

What could then be an optimal strategy? The solution is almost obvious: choose tiles so big that LSGP is feasible on them, condensate the operations and then use LPGS on the result - LSGP by choice followed by LPGS by necessity.

# 5 Solving Problems With A QR Array

## 5.1 One Pass Equation Solver: The Faddeev/Faddeeva Array

To obtain a direct, one pass solution for the system of linear equations $Ty = u$ using a modification of the Gentleman-Kung array, one just has to perform QR on the following, extended matrix (we use $\cdot'$ to indicate the transpose of a matrix, namely $T'_{i,j} := T_{j,i}$) [22]:

$$F := \begin{bmatrix} T' & I & 0 \\ -u' & 0 & 1 \end{bmatrix} \tag{8}$$

Instead of having a tall matrix as in the original Gentleman-Kung array, we now have a flat one, the only thing we must do is extend the algorithmic array with a number of rotors to the right hand

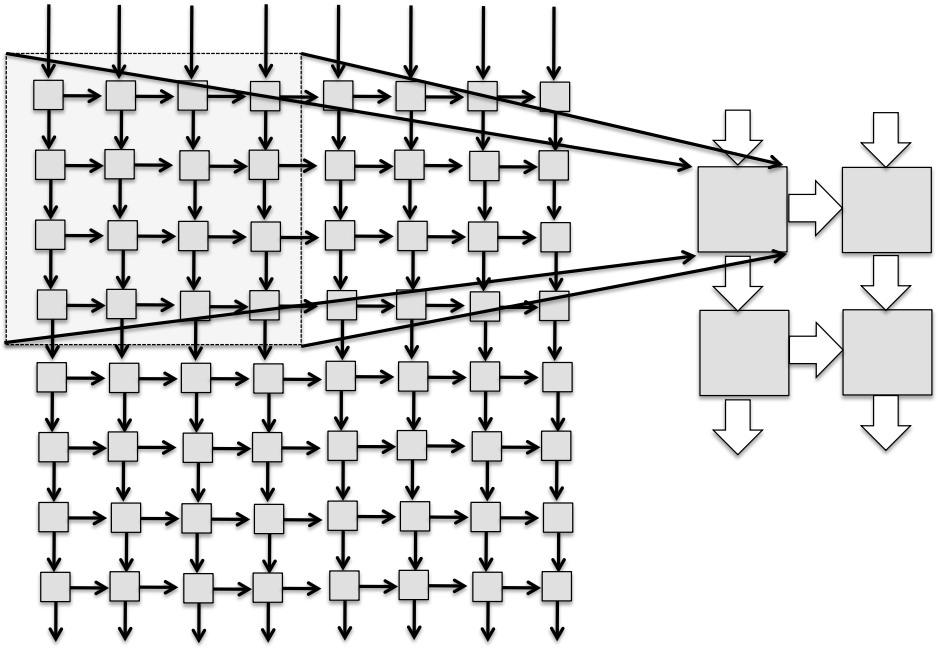

Figure 3: The 'Local Sequential, Global Parallel' strategy: the array is partitioned in subarrays, the partitions are mapped to the processor array.

side. The dimensions have also increased, it is now a processor array of dimensions $(n+1)\times(2n+1)$, with only $(n+1)$ vectorizing rotors on the main diagonal, but otherwise just a similar regular array as before, now rectangular. The $Q$ matrix will now have dimensions $(n+1)\times(n+1)$, and to amplify this we partition it accordingly:

$$Q = \begin{bmatrix} Q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix} \tag{9}$$

In this representation, $q_{12}$ is a tall vector of dimension $n$, $q_{21}$ a flat vector of dimension $n$ and $q_{22}$ just a scalar quantity. One can immediately verify that the QR factorization of this matrix produces:

$$\begin{bmatrix} T' & I & 0 \\ -u' & 0 & 1 \end{bmatrix} = Q \begin{bmatrix} R & Q'_{11} & q'_{21} \\ 0 & y'q_{22} & q_{22} \end{bmatrix} \tag{10}$$

in which $R$ is some matrix (which we do not use further), and the result appears as the pair $(y'q_{22}, q_{22})$. One interprets $q_{22}$ as a normalizing factor, it is actually equal to $1/\sqrt{1+\|y\|^2}$ flowing out of the array.

### 5.2 Channel And Signal Estimation In Telecommunications

The typical equation governing a telecommunication situation has the form

$$x = h * s + n. \tag{11}$$

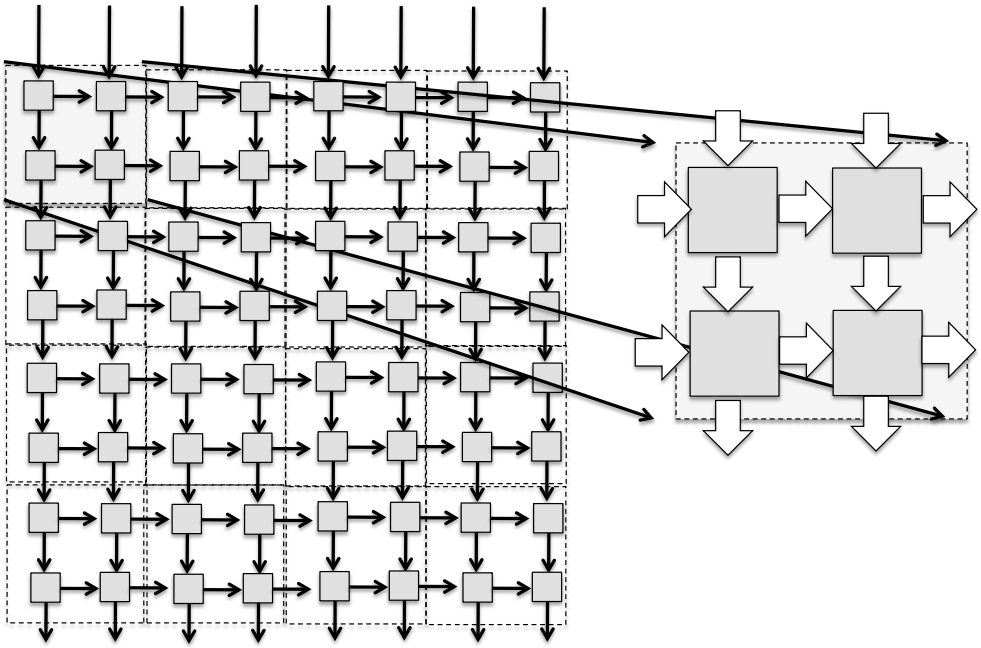

Figure 4: The 'Local Parallel, Global Sequential' strategy: tiles are mapped directly on the processor array and executed sequentially.

Herein $x$ is the received signal, $h$ represents the channel, $s$ the signal to be transmitted and $n$ the noise. All these can be viewed as 'signals', in the most current situation they are functions of time. $'*'$ is convolution, it represents the action of a channel on an input signal, convolution being typical for a linear, time-invariant (LTI) situation wherein the output is the sum of similar responses modulated by the signal and shifted according to their occurrence. The technical situation is often pretty complicated, but from a numerical point of view, relating original input sequences to the output, the overall model can be captured by two equivalent equations:

1. The *channel estimation model* $x = Sh + n$, in which

$$S = \begin{bmatrix} s_0 & & \\ \vdots & \ddots & \\ s_{K-1} & & s_0 \\ & \ddots & \vdots \\ & & s_{K-1} \end{bmatrix}, \quad h = \begin{bmatrix} h_0 \\ \vdots \\ h_{L-1} \end{bmatrix}, \tag{12}$$

2. The *signal estimation model* $x = Hs + n$, in which

$$H = \begin{bmatrix} h_0 & & \\ \vdots & \ddots & \\ h_{L-1} & & h_0 \\ & \ddots & \vdots \\ & & h_{L-1} \end{bmatrix}, \quad s = \begin{bmatrix} s_0 \\ \vdots \\ s_{K-1} \end{bmatrix}. \tag{13}$$

$s$ is now a sequence of original symbols that are being transmitted (we assume them to be real numbers), $h$ is the impulse response of the transmission medium and $n$ is the overall 'noise' being added at each reception (the noise is a combination of interferences and processing noise.) The first situation is the 'training' part, in which we assume that $s$ is known while $h$ has to be estimated, the system is adapting to the transmission situation by using a learning signal. In the second situation knowledge about the channel is assumed and it is used to transmit information $s$ that has to be estimated.

Classical estimation theory provides a number of techniques, called 'Best Linear Unbiased Estimator - BLUE', 'Minimum Variance Unbiased Estimator - MVU', 'Maximum Likelihood Estimator - MLE' and even Bayesian estimators, such as the 'Linear Minimum Mean Square Error Estimator - LMMSE'. Most popular are BLUE and LMMSE, which we briefly pursue. In the case of the channel estimator, assume the noise to have covariance $\sigma^2 I$ and the signal matrix $S$ to be sufficiently rich so as to have full row rank, then optimization theory shows that the BLUE and least square estimators are given by

$$\hat{h} = (S'S)^{-1}S'x. \tag{14}$$

$(S'S)^{-1}S'$ is the so called *Moore-Penrose inverse* of $S$. This is precisely the situation we have described in the section on QR-factorization. Indeed, when $S = QR$, then $(S'S)^{-1}S' = R^{-1}Q'$, we have called $Q'x = \eta_1$ and we find the result $\hat{h} = R^{-1}\eta_1$ - exactly the algorithm presented earlier. There is also a one-pass version, due to Jainandunsing and Deprettere [22].

The data situation is not much different. In the BLUE situation and with the white noise assumption in force, the signal estimator becomes

$$\hat{s} = (H'H)^{-1}H'x \tag{15}$$

and a QR factorization of $H$ will produce the result.

If a so-called Bayesian estimator is desired, then known covariance information on the the result has to be brought into play. The formulas get to be a little more complicated, for example

$$\hat{s} = (H'H + \sigma^2 C_s^{-1})H'x \tag{16}$$

for the data estimator (in which $C_s$ is the assumed known covariance of $s$ (this may purely be belief data!). Also in this case the QR factorization is the way to go, as $H'H = R'R$ and $H'x = R'\eta_1$ with $\eta_1$ defined as before. Typically, the $R$ matrix will be much smaller than the $H$ matrix, the former has the size of the data vector, while the size of the latter is determined by the number of experiments.

### 5.3 The Kalman Filter

The Kalman estimation filter attempts to estimate the actual state of an unknown discrete dynamical system, given noisy measurements of its output, for a general introduction see Kailath [23], see also Kailath, Sayed and Hassibi [25], here we give a brief account to connect up with the previous section on QR and the following section on structured matrices. The traditional set up makes a number of assumptions, which we summarize.

**Assumptions**

1. We assume that we have a reasonably accurate model for the system whose state evolution we try to estimate. Let $x_i$ be the evolving state at time point $i$ - it is a vector of dimension $\delta_i$. We further assume that the system is driven by an unknown noisy, zero mean, vectorial input $u_i$, whose second order statistical properties we know (as in the BLUE, we work only with zero mean processes and their variances.) We assume the dynamical system to be linear and given by three matrices $\{A_i, B_i, C_i\}$, which describe respectively the maps from state $x_i$ to next state $x_{i+1}$, from input $u_i$ to next state $x_{i+1}$ and from state $x_i$ to output $y_i$. The latter is contaminated by zero mean, vectorial *measurement noise* $\nu_i$, whose second order statistics we know also. The model has the form given by

$$\left\{ \begin{array}{rcl} x_{i+1} & = & A_i x_i + B_i u_i \\ y_i & = & C_i x_i + \nu_i \end{array} \right. \tag{17}$$

A data flow diagram of the state evolution is shown in Fig. 5. The *transition matrix* of this

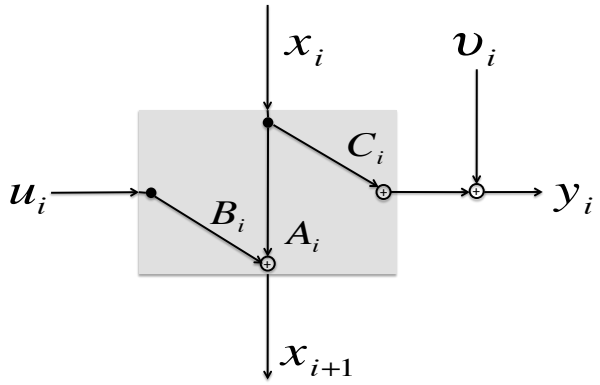


Figure 5: The model filter

filter is defined as the matrix $\left[ \begin{array}{cc} A_i & B_i \\ C_i & 0 \end{array} \right]$, it defines the map from all inputs $\left[ \begin{array}{c} x_i \\ u_i \end{array} \right]$ to all outputs $\left[ \begin{array}{c} x_{i+1} \\ y_i \end{array} \right]$.

2. Concerning the statistical properties of the driving process $u_i$ and the measurement noise $\nu_i$, we need only to define the second order statistics (the first order means is already assumed zero, and no further assumptions are made on the higher orders). We always work on the space of relevant, zero means stochastic variables, using "$\mathbf{E}$" as the expectation operator. In the present summary, we assume that $u_i$ and $\nu_i$ are uncorrelated with each other and with any other $u_k$, $\nu_k$, $k \neq i$, and that their covariances are given respectively by $\mathbf{E} u_i u_i' = Q_i$ and $\mathbf{E} \nu_i \nu_i' = R_i$, both non singular, positive definite matrices (this assumes that there is enough noise in the system), we assume again real matrices throughout, otherwise one must use Hermitian transposition.) On a space of scalar stochastic variables with zero mean we define a (second order statistical) inner product as $(x, y) = \mathbf{E}(xy)$. This can be extended to vectors

by using outer products such as

$$\mathbf{E}xy' = \mathbf{E}\left(\begin{bmatrix} u_1 \\ \vdots \\ u_m \end{bmatrix}\begin{bmatrix} y_1 \cdots y_n \end{bmatrix}\right) = \begin{bmatrix} \mathbf{E}u_1y_1 & \mathbf{E}u_1y_2 & \cdots & \mathbf{E}u_1y_n \\ \mathbf{E}u_2y_1 & \mathbf{E}u_2y_2 & \cdots & \mathbf{E}u_2y_n \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{E}u_my_1 & \mathbf{E}u_my_2 & \cdots & \mathbf{E}u_my_n \end{bmatrix}. \tag{18}$$

3. We also assume that the process whose state is to be estimated starts at the time point $i = 0$. The initial state $x_0$ has known covariance $\mathbf{E}x_0x_0^T = \Pi_0$.

**The Recursive Solution**

We start with summarizing the classical solution, based on the 'innovations model' pioneered by Kailath e.a., see [23]. Let us assume that we have been able to predict $x_i$ and attempt to predict $x_{i+1}$. The least squares predictor $\hat{x}_i$ is the one that minimizes the prediction error $e_{x,i} = x_i - \hat{x}_i$ in the covariance sense, assuming linear dependence on the data (the same assumptions will hold for the next predictor). The Wiener property asserts that $\hat{x}_i$ is a linear combination of the known data (in our case all the $y_k$ for $k = 0 \cdots i-1$) and that the error, also called the state innovation, $e_i$ is orthogonal on all the known data so far. These properties will of course be propagated to the next stage, given the (noise contaminated) new information $y_i$. It turns out that the only information needed from the past of the process is precisely the estimated state $\hat{x}_i$, the new estimate being given by

$$\hat{x}_{i+1} = A_i\hat{x}_i + K_{p,i}(y_i - C_i\hat{x}_i). \tag{19}$$

In this formula $K_{p,i}$ denotes the 'Kalman gain', which has to be specified, and which is given by

$$K_{p,i} = K_iR_{e,i}^{-1},\ R_{e,i} = R_i + C_iP_iC_i',\ K_i = A_iP_iC_i'. \tag{20}$$

In these formulas, the covariances $P_i = \mathbf{E}e_{x,i}e_{x,i}^T$ and $R_{e,i}$ are used, in view of the formula for the latter, only $P_i$ has to be updated to the next step, and is given by

$$P_{i+1} = A_iP_iA_i' + B_iQ_iB_i' - K_{p,i}R_{e,i}K_{p,i}' \tag{21}$$

The covariance $P_{i+1}$ is supposed to be positive definite for all values of $i$, a constraint which may be violated at times because of numerical errors caused be the subtraction in the formula. In the next paragraph we shall introduce the square root version of the Kalman filter, which cannot create this type of numerically caused problems. So far we have only given summaries the known results, we give a simple direct proof in the next paragraph. Starting values have to be determined since this solution is recursive, and they are given by

$$\hat{x}_0 = 0,\ P_0 = \Pi_0. \tag{22}$$

**Proof**

We give a recursive proof based on the parameters of the model at time point $i$. We assume recursively that the error $e_i = x_i - \hat{x}_i$ at that time point is orthogonal on the previously recorded

data, and that the new estimate $\hat{x}_{i+1}$ is a linear combination of the data recorded up to that point. We first relax the orthogonality condition, and only ask $e_{i+1}$ to be orthogonal on $\hat{x}_i$ (a linear combination of already recorded previous data) and $y_i$, the newly recorded data at time point $i$. We show that this estimator already produces an estimate that is orthogonal (of course in the second order statistical sense) on all the previously recorded data.) From our model we know that $x_{i+1} = A_i x_i + B_i u_i$. We next ask that $\hat{x}_{i+1}$ be a linear combination of the known data $\hat{x}_i$ and $y_i$, i.e. there exist matrices $X_i$ and $Y_i$, to be determined, such that

$$\hat{x}_{i+1} = X_i \hat{x}_i + Y_i y_i. \tag{23}$$

Requesting second order statistical orthogonality of $e_{i+1}$ on $\hat{x}_i$ we obtain

$$\mathbf{E}(x_{i+1} - \hat{x}_{i+1})\hat{x}_i' = \mathbf{E}(A_i x_i + B_i u_i - X_i \hat{x}_i - Y_i y_i)\hat{x}_i' = 0. \tag{24}$$

We now observe that $\mathbf{E}u_i \hat{x}_i' = 0$ by assumption and that $\mathbf{E}\hat{x}_i \hat{x}_i' = \mathbf{E}x_i \hat{x}_i'$ because $\mathbf{E}e_i \hat{x}_i = 0$ through the recursive assertion. The previous equation then reduces to

$$(A_i - X_i - Y_i C_i)\mathbf{E}(x_i \hat{x}_i') = 0, \tag{25}$$

which shall certainly be satisfied when $X_i = A_i - Y_i C_i$.
Next we request orthogonality on the most recent data, i.e.

$$\mathbf{E}e_{i+1} y_i' = 0. \tag{26}$$

In fact, we can ask a little less, by using the notion of 'innovation'. The optimal predictor for $y_i$ is simply $\hat{y}_i = C_i \hat{x}_i$, and its innovation, defined as $e_{y,i} = y_i - \hat{y}_i$, is $e_{y,i} = C_i e_i + \nu_i$. We now just require that $e_{i+1}$ is orthogonal on $e_{y,i}$, as it is already orthogonal on $\hat{x}_i$ and $\mathbf{E}e_{i+1} y_i' = \mathbf{E}[e_{i+1}(x_i' C_i' + \nu_i' - \hat{x}_i' C_i')]$. We now obtain an expression for the innovation $e_{i+1}$ in term of past innovations and the data of section $i$

$$e_{i+1} = A_i e_i + B_i u_i - (A_i - Y_i C_i)\hat{x}_i - Y_i y_i = A_i e_i + B_i u_i - Y_i e_{y,i}, \tag{27}$$

which we now require to be orthogonal on $e_{y,i}$. With $P_i = \mathbf{E}e_i e_i'$, we have $\mathbf{E}e_i e_{y,i}' = P_i C_i'$ and $\mathbf{E}e_{y,i} e_{y,i}' = C_i P_i C_i' + R_i$. The orthogonality condition becomes therefore

$$Y_i(R_i + C_i P_i C_i') = A_i P_i A_i'. \tag{28}$$

Hence the formulas given for the Kalman filter, after identifying $K_{p.i} = Y_i$ and $R_{e,i} = R_i + C_i P_i C_i'$ (actually the covariance of $e_{y,i}$.)

Concerning the propagation of the covariance of the innovation $P_i$, we rewrite the formula for $e_{i+1}$ as (reverting back to the notation in the previous paragraph)

$$e_{i+1} + K_{p,i} e_{y,i} = A_i e_i + B_i u_i. \tag{29}$$

Remarking that the terms of the left hand side are orthogonal to each other, and those of the right hand side as well, we obtain the equality

$$P_{i+1} + K_{p,i} R_{e,i} K_{p,i}' = A_i P_i A_i' + B_i Q_i B_i', \tag{30}$$

which shows the propagation formula for the innovation covariance.

Finally, when $y_k$ is some data collected at a time point $k < i$, we see that $\mathbf{E}e_{i+1} y_k^T = \mathbf{E}[(A_i e_i + B_i u_i - K_{p,i}\hat{y}_i) y_k']$. The recursion hypothesis states that $e_i$ is orthogonal to all past collected data, in particular to $y_k$. Hence we see that the expression is equal to zero, after working out the individual terms.

**The Square Root (LQ) Algorithm**

The square root algorithm solves the Kalman estimation problem efficiently and in a numerical stable way, avoiding the Riccati equation of the original formulation. It computes an *LQ factorization* on the known data to produce the unknown data. An LQ factorization is the dual of the QR factorization, rows are replaced by columns and the order of the matrices inverted, but otherwise it is exactly the same and is done on the same architecture. Not to overload the symbol '$Q$', already defined as a covariance, we call the orthogonal transformation matrix at step $i$, $U_i$, acting on a so called *pre-array* and producing a *post-array*

$$\begin{bmatrix} C_i P_i^{1/2} & R_i^{1/2} & 0 \\ A_i P_i^{1/2} & 0 & B_i Q_i^{1/2} \end{bmatrix} U_i = \begin{bmatrix} R_{e,i}^{1/2} & 0 & 0 \\ \bar{K}_{p,i} & P_{i+1}^{1/2} & 0 \end{bmatrix}. \tag{31}$$

The square root algorithm gets it name because it does not handle the covariance matrices $P_i$ and $R_{e,i}$ directly, but their so called square roots, actually their Cholesky factors, where one writes, e.g. $P_i = P_i^{1/2} P_i'^{/2}$ assuming $P_i^{1/2}$ to be lower triangular, and then $P_i'^{/2}$ is its upper triangular transpose (this notational convention is in the benefit of reducing the number of symbols used, the exact mathematical square root is actually not used in this context.) The matrix on the left hand side is known from the previous step, applying $U_i$ reduces it to a lower triangular form and hence defines all the matrices on the right hand side. Because of the assumptions on the non singularity of $R_i$, $R_{e,i}$ shall also be a square matrix, the non-singularity of $P_{i+1}$ is not directly visible from the equation and is in fact a more delicate affair, the discussion of which we skip here.

The right hand side of the square root algorithm actually defines a new filter with transition matrix

$$\begin{bmatrix} A_i & \bar{K}_{p,i} \\ C_i & R_{e,i}^{1/2} \end{bmatrix} \tag{32}$$

One obtains the original formulas in the recursion just by *squaring* the square root equations (multiplying to the right with the respective transposes). In particular this yields $A_i P_i A_i' = \bar{K}_{p,i} R_{e,i}'$ and hence

$$\bar{K}_{p,i} = K_{p,i} R_{e,i}^{1/2} = K_i R_{e,i}^{-\prime/2} \tag{33}$$

(different versions of the *Kalman gain*.) This form is called an *outer filter*, i.e. a filter that has a causal inverse. The inverse can be found by arrow reversal (see Fig. 6 and it can rightfully be called both the Kalman filter (as it produces $\hat{x}_{i+1}$) and the (normalized) innovations filter, as it produces $\epsilon_i = R_i^{-1/2}(y_i - C_i \hat{x}_i)$, the normalized innovation of $y_i$ given the preceding data summarized in $\hat{x}_i$.

## 5.4 Solving Symmetric Positive Definite Systems With The Schur Algorithm

We consider the solution of a linear least-squares problem via the normal equation

$$T'Tu = T'y,$$

where the coefficient matrix $C = T'T$ is a symmetric positive definite matrix. We split up the symmetric matrix according to $C = U + D + U' \in \mathbb{R}^{m \times m}$, and define the intermediate matrices

$$V = \frac{1}{2}\left(D + 2U + \mathbf{1}_n\right), \quad W = \frac{1}{2}\left(D + 2U - \mathbf{1}_n\right)$$

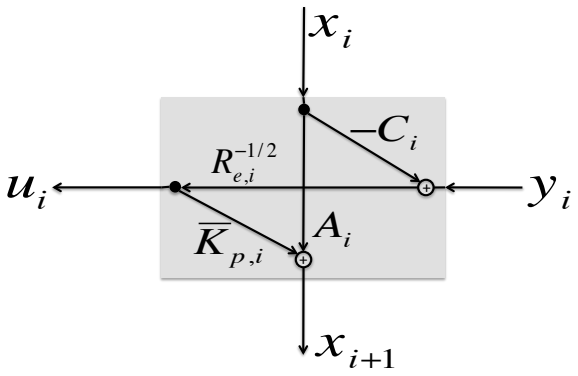


Figure 6: The Kalman filter, alias innovations filter

such that $C$ can be represented as

$$C = V'V - W'W = \begin{bmatrix} V \\ W \end{bmatrix}' J \begin{bmatrix} V \\ W \end{bmatrix}, \qquad J = \begin{bmatrix} \mathbf{1}_n & \\ & -\mathbf{1}_n \end{bmatrix}.$$

The Schur-Cholesky algorithm, as presented in [8] and [21] solves this symmetric linear system determines a $J$-orthogonal $\Theta$ satisfying

$$\Theta \begin{bmatrix} V \\ W \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}, \quad \Theta' J \Theta = J.$$

The matrix $\Theta$ being $J$-orthogonal results in the identity

$$\begin{bmatrix} V \\ W \end{bmatrix}' \Theta' J \Theta \begin{bmatrix} V \\ W \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}' J \begin{bmatrix} R \\ 0 \end{bmatrix} = V'V - W'W = R'R,$$

where the matrix $R$ is an upper triangular matrix, hence, a triangular factor of $C$. If we complete this map by extending it to the orthogonal space we arrive at the full equation

$$\Theta \begin{bmatrix} V & W' \\ W & V' \end{bmatrix} = \begin{bmatrix} R & 0 \\ 0 & L \end{bmatrix}, \quad C = LL',$$

where $L$ is supposed to be a lower triangular factor of $C$. Assuming that all necessary matrices are invertible, then we can devise an expression for the transformation

$$\Theta = \begin{bmatrix} (R')^{-1}V' & -(R')^{-1}W' \\ -(L')^{-1}W & (L')^{-1}V \end{bmatrix}. \tag{34}$$

Similar to the approach for computing the QR decomposition using Jacobi rotations (compare with section 3.2) we can compute the overall transformation $\Theta$ using elementary $J$-orthogonal or hyperbolic rotations. The elementary hyperbolic matrix is a rotation over an angle $\theta$ in the 2D plane (for convenience we define $H'$):

$$H' = \begin{bmatrix} c_h & s_h \\ -s_h & c_h \end{bmatrix}, \quad \text{using} \quad \begin{matrix} c_h := \cosh\theta \\ s_h := \sinh\theta \end{matrix} \tag{35}$$

Let's apply such a rotation $H'$ to two row vectors such as

$$\begin{bmatrix} c_h & s_h \\ -s_h & c_h \end{bmatrix} \begin{bmatrix} a_1 & a_2 & \cdots & a_n \\ b_1 & b_2 & \cdots & b_n \end{bmatrix} = \begin{bmatrix} \sqrt{a_1^2 - b_1^2} & c_h a_2 + s_h b_2 & \cdots & c_h a_n + s_h b_n \\ 0 & -s_h a_2 + c_h b_2 & \cdots & -s_h a_n + c_h b_n \end{bmatrix} \tag{36}$$

which is achieved by choosing

$$c_h = \frac{a_1}{\sqrt{a_1^2 - b_1^2}}, \quad s_h = \frac{b_1}{\sqrt{a_1^2 - b_1^2}},$$

and hence automatically $\tanh\theta = \frac{b_1}{a_1}$.

In this way one can treat the entries of the original matrix $T$ row by row and create all the zeros below the main diagonal. With a $3 \times 3$ matrix $C$ this works as shown below. The only thing one must do is embed the $2 \times 2$ rotation matrices in the $4 \times 4$ schema, so that unaffected rows remain unchanged. We label the hyperbolic rotation matrices with the indices of the rows they affect

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \cdot \\ \hline \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \cdot \end{bmatrix} \xrightarrow{H'_{1,4}} \begin{bmatrix} \star & \star & \star \\ & \cdot & \cdot \\ & & \cdot \\ \hline 0 & \star & \star \\ & \cdot & \cdot \\ & & \cdot \end{bmatrix} \xrightarrow{H'_{2,5}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \star & \star \\ & & \cdot \\ \hline 0 & \cdot & \cdot \\ & 0 & \star \\ & & \cdot \end{bmatrix} \xrightarrow{H'_{3,6}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \star \\ \hline 0 & \cdot & \cdot \\ & 0 & \cdot \\ & & 0 \end{bmatrix} \xrightarrow{H'_{2,4}} \tag{37}$$

$$\xrightarrow{H'_{2,4}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \star & \star \\ & & \cdot \\ \hline 0 & 0 & \star \\ & 0 & \cdot \\ & & 0 \end{bmatrix} \xrightarrow{H'_{3,5}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \star \\ \hline 0 & 0 & \cdot \\ & 0 & 0 \\ & & 0 \end{bmatrix} \xrightarrow{H'_{3,4}} \begin{bmatrix} \cdot & \cdot & \cdot \\ & \cdot & \cdot \\ & & \star \\ \hline 0 & 0 & 0 \\ & 0 & 0 \\ & & 0 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ & r_{22} & r_{23} \\ & & r_{33} \\ \hline 0 & 0 & 0 \\ & 0 & 0 \\ & & 0 \end{bmatrix} \tag{38}$$

The final step is a $H'_{3,4}$ which annihilates the $4,3$ entry. In each of these subsequent steps, the first operation determines the rotation matrix and then applies it to all the entries in the respective rows, skipping the already computed zero entries (which remain zero hence avoiding fill-ins.) The overall $J$-orthogonal matrix is then determined as the product of the elementary hyperbolic rotation $\Theta = H_{1,4}H_{2,5}H_{3,6}H_{2,4}H_{3,5}H_{3,4}$. Note that rotation steps with non-overlapping row indices can be carried out in parallel, i.e. the sequence of rotations $H_{1,4}$, $H_{2,5}$, and $H_{3,6}$ can be executed simultaneously on a dedicated processor array as well as the rotations $H_{2,4}$ and $H_{3,5}$.

After having computed the $J$-orthogonal matrix $\Theta$ by a sequence of hyperbolic rotations we can take the representation of the $J$-orthogonal transformation as given in equation 34, and determine the effect of applying $\Theta$ to two tacked identity matrices, i.e. we can compute

$$\Theta \begin{bmatrix} \mathbf{1}_n \\ \mathbf{1}_n \end{bmatrix} = \begin{bmatrix} (R')^{-1}(V' - W') \\ (L')^{-1}(V - W) \end{bmatrix} = \begin{bmatrix} (R')^{-1} \\ (L')^{-1} \end{bmatrix}.$$

This expression shows that the process of elementary elimination steps implicitly creates the inverses of the triangular factors of the matrix $C$. The symmetric system of equation can be used in a straight forward manner with appropriate back-substitution steps. However, back-substitution destroys the

homogenous data flow and is therefore not desirable for parallel processing hardware. Applying the Fadeeva approach to this algorithm [21], similar to the approach used for the QR solver, allows us to devise a purely feed forward algorithm that avoids all back-substitution steps. This version of the algorithm is based on performing the following two stages in an elimination process that employs hyperbolic rotations. Stage 1 executes the elimination of the matrix $W$ by hyperbolic rotations, as explained previously creating the matrix $\Theta_1$ according to

$$\Theta_1 \begin{bmatrix} V & \mathbf{1}_n & 0 \\ W & \mathbf{1}_n & 0 \\ -b' & 0 & 1 \end{bmatrix} = \begin{bmatrix} R & (R')^{-1} & 0 \\ 0 & (L')^{-1} & 0 \\ -b' & 0 & 1 \end{bmatrix}.$$

Stage 2 of the elimination process annihilates the entries of the vector $-b'$ while creating the matrix $\Theta_2$

$$\Theta_2 \begin{bmatrix} R & (R')^{-1} & 0 \\ 0 & (L')^{-1} & 0 \\ -b' & 0 & 1 \end{bmatrix} = \begin{bmatrix} \tilde{R} & \tilde{(R')}^{-1} & \star \\ 0 & (L')^{-1} & 0 \\ 0 & ku' & k \end{bmatrix}.$$

After both stages of elimination are completed, the solution vector $k \cdot u'$ can be read off from the resulting array as well as scalar parameter $k$. Figure 5.4 depicts the processing array to implement the Schur-Cholsky algorithm for solving symmetric positive systems of equations in a highly parallel and regular way and without a need to perform back-substitution [21].

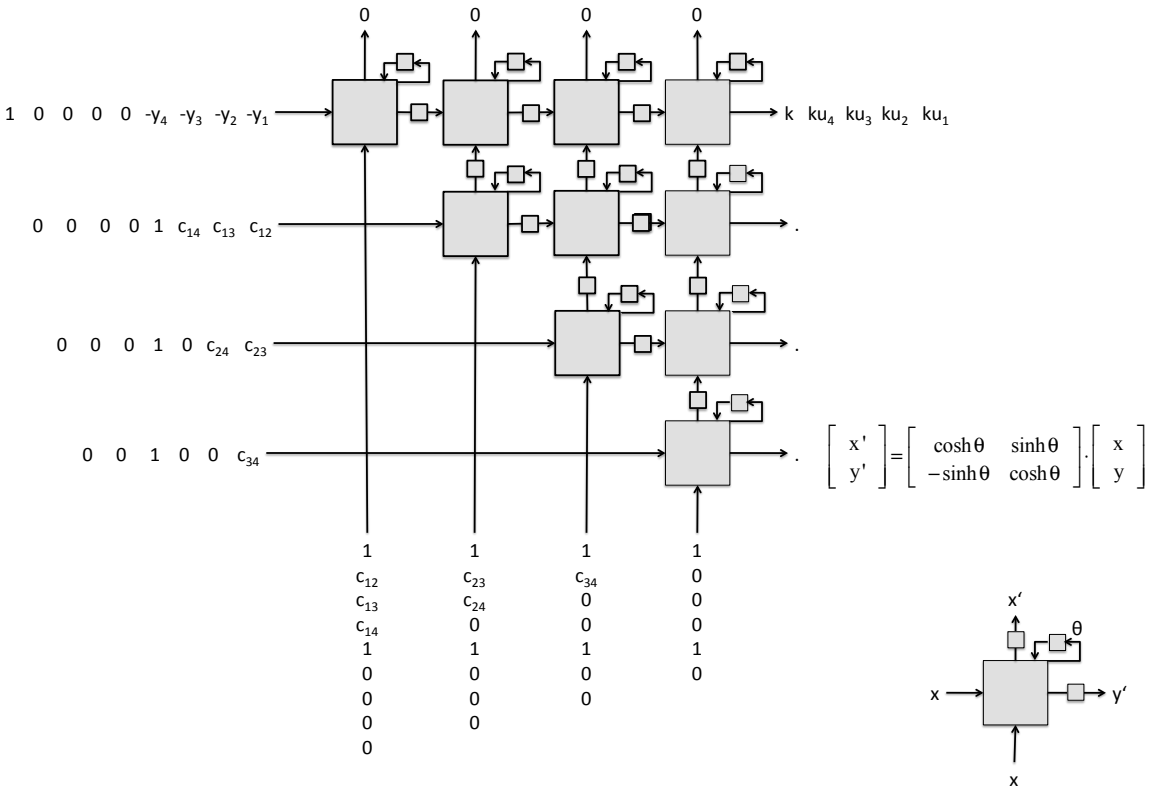


Figure 7: The array for the Schur-Cholesky Algorithm

# 6 Sparse Matrices and Iterative Algorithms

Large scale linear systems, which are characterized by a coefficient matrix of enormous size are a topic of particular interest from a practical point of view. Luckily, such matrices are mostly sparse, which means that only a small fraction of the matrix entries are different from zero. Sparse matrices exhibit very large values for $n$, but only $\mathcal{O}(n)$ matrix entries are different from zero. The sparsity allows the matrices to be stored with $\mathcal{O}(n)$ memory locations [36]. This type of matrices originate for example from finite-element computations, from solving partial differential or Euler-Lagrange equations to name a few examples [36]. Applying standard matrix algebra does not preserve the sparsity pattern, that is, it destroys the sparsity pattern when adding, multiplying and inverting sparse matrices. For example, the inverse of a tri-band matrix is in general not a tri-band matrix, but a full matrix. Similar statements hold if matrix factorizations and dense solvers are applied to sparse matrices; elementary transformations tend to fill up the matrix by creating fill-ins, i.e. by overwriting zero matrix entries with non-zero values [36],[3].

Algorithms to efficiently solve systems of equations with a sparse coefficient matrix use iterative approaches, which are using a sequence of matrix vector multiplications of the form $u_{k+1} = T \cdot u_k$. This way, the sparsity pattern of the coefficient matrix is preserved and the sequence of vectors $u_k$ converges, under certain conditions to the solution vector $u$. Matrix-vector multiplication with a sparse matrix $T \in \mathbb{R}^{m \times n}$ amounts to a computational load of $\mathcal{O}(m+n)$ operations and $\mathcal{O}(m+n)$ memory requirement. Iterative schemes such as a Conjugate Gradient algorithm require $\mathcal{O}(n)$ iterations, resulting in an solution method with a computational complexity of $\mathcal{O}(n^2)$ operations overall. For a comprehensive coverage of iterative solution methods we refer the interested reader to [36].

## 6.1 Sparse Matrices in Motion Analysis

For many applications in the domain of computer vision or in the field of digital video signal processing the task of estimating the apparent motion of objects or pixels throughout a video sequence is a fundamental task. Motion estimation is an expensive calculation, in particular when considering to deal with standard definition resolution images ($576 \times 720$ pixels per image) or moving on to even handle High Definition resolutions ($1080 \times 1920$ pixels per image) [10].

### Optic Flow Constraint

We discuss how to compute the optical flow according to the approach proposed by Horn & Schunck [4]. The brightness of a pixel at point $(x, y)$ in an image plane at time $t$ is denoted by $I(x, y, t)$. Let $I(x, y, t)$ and $I(x, y, t+1)$ be two successive images of a video sequence. Each image is comprised of a rectangular lattice of $N = m \times n$ pixels.Optic flow computation is based on the assumption that the brightness of a pixel remains constant in time and that all apparent variations of the brightness throughout a video sequence are due to spatial displacements of the pixels, which again are caused

by motion of objects. We denote this brightness conservation assumption as

$$\frac{dI}{dt} = 0.$$

This equation is called the optical flow constraint. Using the chain rule for differentiation the optic flow constraint is expanded into

$$\frac{\partial I}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial I}{\partial y} \cdot \frac{dy}{dt} + \frac{\partial I}{\partial t} = 0.$$

Using the shorthand notation $v_x = \frac{dx}{dt} \quad v_y = \frac{dy}{dt}, I_x = \frac{\partial I}{\partial x}, I_y = \frac{\partial I}{\partial y}, I_t = \frac{\partial I}{\partial t}$, the optic flow constraint can be written as

$$E_{of} = I_x \cdot v_x + I_y \cdot v_y + I_t = 0. \tag{39}$$

Equation (39) is only one equation for determining the two unknowns $v_x$ and $v_y$, which denote the horizontal and the vertical component of the motion vector at each pixel position. Hence the optical flow equation is an underdetermined system of equation. Solving this equation in a least squares sense only produces the motion vector component in direction of the strongest gradient for the texture. Therefore a second constraint has to be found to regularize this ill-posed problem.

**Smoothness Constraint**

To overcome the underdetermined nature of the optic flow constraint, Horn & Schunck introduced an additional smoothness constraint. Neighboring pixels of an object in a video sequence are likely to move in a similar way. The motion vectors $v_x$ and $v_y$ are varying spatially in a smooth way. Spatial discontinuities in the motion vector field occur only at motion boundaries between objects, which move in different directions and which are occluding each other. Therefore, the motion vector field to be computed is supposed to be spatially smooth. This smoothness constraint can be formulated using the Laplacian of the motion vector field $v_x$ and $v_y$

$$E_{sc} = \nabla^2 v_x + \nabla^2 v_y = \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_y}{\partial x^2} + \frac{\partial^2 v_y}{\partial y^2}. \tag{40}$$

The Laplacians of $v_x$ and $v_y$ can be calculated by the approximation

$$\nabla^2 v_x \approx \overline{v}_x - v_x \quad \text{and} \quad \nabla^2 v_y \approx \overline{v}_y - v_y.$$

The term $\overline{v}_{x,y} - v_{x,y}$ can be computed numerically as the difference between the central pixel $v_{x,y}$ and a weighted average of the values in a 2-neighborhood of the central pixel. The corresponding 2-dimensional convolution for performing this filtering operation is given as

$$\overline{v}_x(x,y) - v_x(x,y) = \mathcal{L}(x,y) * v_x(x,y)$$

$$\overline{v}_y(x,y) - v_y(x,y) = \mathcal{L}(x,y) * v_x(x,y),$$

where the convolution kernel $\mathcal{L}(x,y)$ is given by a 2D-filtering mask, such as

$$\mathcal{L}(x,y) = \begin{bmatrix} 1/12 & 1/6 & 1/12 \\ 1/6 & -1 & 1/6 \\ 1/12 & 1/6 & 1/12 \end{bmatrix}. \tag{41}$$

The Horn & Schunck approach uses the optic flow equation (39) along with the smoothness constraint (40) to express the optic flow computation as the optimization problem for the cost function

$$E^2 = E_{of}^2 + \alpha^2 \cdot E_{sc}^2,$$

which needs to be minimized in terms of the motion vector $[v_x v_y]^T$. The parameter $\alpha$ is a scalar regularization parameter, which controls the contribution of the smoothness constraint (40). The optimization problem finally expands into the equation

$$E^2 = (I_x v_x + I_y v_y + I_t)^2 + \alpha^2 \left( (\overline{v}_x - v_x)^2 + (\overline{v}_y - v_y)^2 \right). \tag{42}$$

Applying the calculus of variations to42 results in the following two equations

$$\begin{aligned} I_x^2 v_x + I_x I_y v_y + I_x I_t - \alpha^2(\overline{v}_x - v_x) = 0 \\ I_x I_y v_x + I_y^2 v_y + I_y I_t - \alpha^2(\overline{v}_y - v_y) = 0, \end{aligned}$$

which need to be solved for the motion vector components $v_x(x, y)$ and $v_y(x, y)$.

**Linear System of Equations**

Based on (39), the optic flow equation for all pixels can be written as a matrix equation [10]

$$\begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix} = -\mathbf{I}_t, \tag{43}$$

where $\mathbf{I}_x$ and $\mathbf{I}_y$ are diagonal matrices of size $N \times N$ and $\mathbf{I}_t$ is a vector of length $N$. The $x$- and $y$-components of the motion vector field are given as the vectors $\mathbf{v}_x$ and $\mathbf{v}_y$, each with the dimension $N$. The effect of the convolution kernel $\mathcal{L}(x, y)$ on the motion vector field can be represented by a constant and sparse $N \times N$ band matrix $\mathbf{L}$, which has Toeplitz structure. We will use the symbol $\boldsymbol{C}$ to denote the negative Laplacian, i.e. $\boldsymbol{C} = -\mathcal{L}$. The specific banded structure of $C$ is depicted in Figure 8,which can be symbolically denoted as

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{M} & \boldsymbol{U} & & & & & \\ \boldsymbol{U} & \boldsymbol{M} & \boldsymbol{U} & & & & \\ & \boldsymbol{U} & \boldsymbol{M} & \boldsymbol{U} & & & \\ & & \boldsymbol{U} & \ddots & \ddots & & \\ & & & \ddots & \ddots & \boldsymbol{U} & \\ & & & & \boldsymbol{U} & \boldsymbol{M} \end{bmatrix}.$$

The minimization of the cost function (42) for all pixels in the image leads to the set of regularized linear equations,

$$\left( \alpha^2 \begin{bmatrix} C & 0 \\ 0 & C \end{bmatrix} + \begin{bmatrix} \mathbf{I}_x \\ \mathbf{I}_y \end{bmatrix} \cdot \begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix} \right) \cdot \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix} = \begin{bmatrix} \mathbf{I}_x \\ \mathbf{I}_y \end{bmatrix} \cdot \mathbf{I}_t. \tag{44}$$

The term in brackets of equation (44) represents a $2N \times 2N$ band matrix, the structure of which can be seen on the left hand side of Figure 9. This structured matrix needs to be solved efficiently

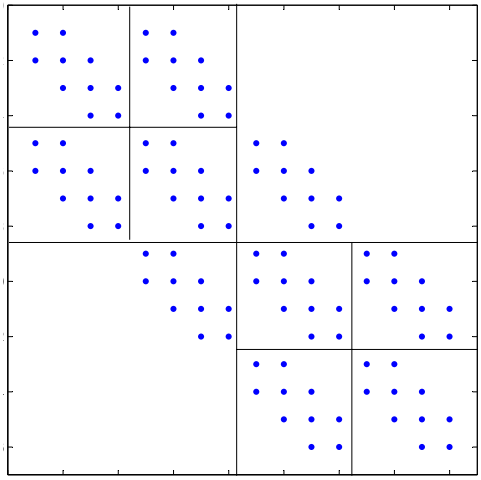

Figure 8: Structure for the Laplace Operator

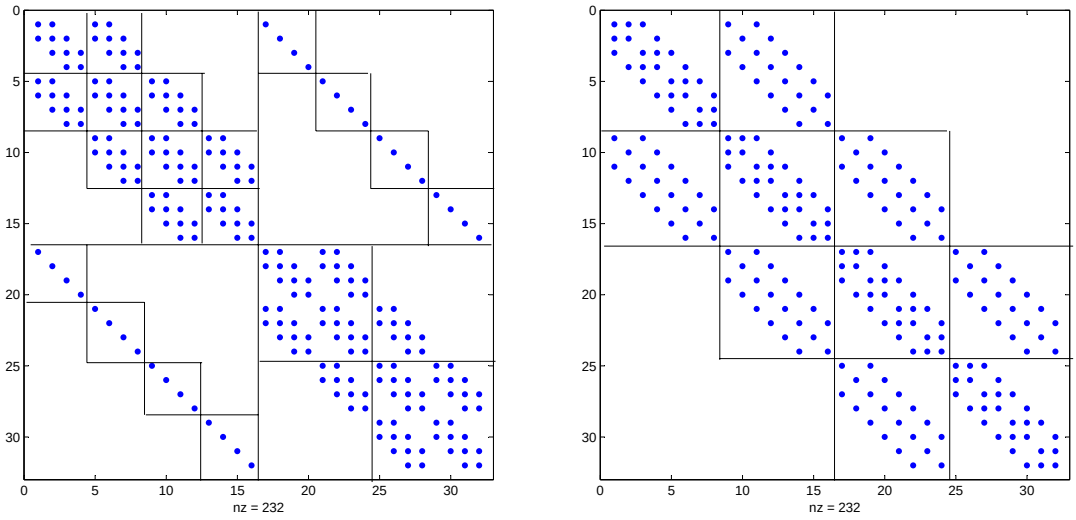


Figure 9: Structure for original matrix (left) and re-ordered matrix (right)

for computing the motion vector fields $\mathbf{v}_x$ and $\mathbf{v}_y$. We can modify the structure of the matrix by re-ordering the variables and hence the matrix entries. In the right hand side of Figure 9 the resulting matrix structure is shown if the the variables $v_x$ and $v_y$ are re-ordered by interleaving them. Such a change of structure for the matrix entries has influence on the efficiency of sparse linear system solvers [36],[3].

Lucas and Kanade proposed an alternative scheme for computing optical flow [29], which is better suitable for implementation on parallel processors such as a Graphic Processor Unit (GPU) [14].

In [11] an approach is presented to solve the sparse linear system arising from optical flow computations in an efficient way by exploiting the structure of the matrix. This structure is called *hierarchically semi-separable*, a notion that will be explained in more detail in section 7.

## 6.2 Iterative Matrix Solvers

Iterative methods take an initial approximation of the solution vector and successively improve the solution by continued matrix vector multiplications until an acceptable solution has been reached [3]. Since matrix-vector multiplications can be computed efficiently for sparse matrices iterative matrix solvers are attractive. However, iterative methods may have poor robustness and often are only applicable for a rather narrow range of applications. With view to implementing such schemes on real-time computer systems we need to understand the convergence properties of iterative schemes, which strongly depend on the input data. Therefore it is difficult to determine worst case deadlines for the iterations to be completed [3].

### General Approach

Iterative algorithms do not explicitly compute the term $T'T$, a term that appears in the context of normal equation, but only propagate the effects and $T'$ and $T$ in a factored form, and hence exploit the sparsity of the coefficient matrix $T$. This leads to the following representation of the standard least-squares problem

$$T' \cdot (Tu - b) = 0.$$

Stationary iterative methods for solving linear systems of equations run the iteration of the form

$$Mu_{k+1} = Nu_k + b, \quad k = 1, 2, 3, \ldots,$$

where $u_0$ is an initial approximation for the solution vector. In this context we have the 'splitting' of the positive definite matrix $T'T = M - N$ where $M$ is assumed to be non-singular. The matrix $M$ should be chosen such, that solving the linear system with coefficient matrix $M$ is easy to do. Analyzing the equation

$$u_{k+1} = M^{-1}Nu_k + M^{-1}b = Gu_k + c, \quad k = 1, 2, 3, \ldots,$$

reveals that the iterative scheme is convergent if the spectral radius of the matrix $G \in \mathbb{R}^{m \times m}$

$$\rho(G) = \max |\lambda_i(G)|, \quad 1 \leq i \leq m$$

satisfies $\rho(G) < 1$. Taking the additive splitting of the coefficient matrix as

$$T'T = L + D + L', \quad D \geq 0,$$

then we get the Jacobi method if we choose $M = D$, whereas the choice $M = L + D$ will lead us to the Gauss-Seidel iterative scheme. The Gauss-Seidel has better convergence properties when compared with Jacobi-methods (one Gauss-Seidel iteration step corresponds to two Jacobi iterations). However, Jacobi-type methods are better suited for parallel implementation [3].

For executing iterative matrix solvers on parallel computing architectures it is essential to implement an efficient way of matrix vector multiplication. This is possible only for a predetermined sparsity pattern. See [18] for a corresponding array to execute iterative matrix solvers.

**LSQR Algorithm**

An attractive alternative to Jacobi- or Gauss-Seidel-iterations is the family of quasi-iterative methods such as Conjugate Gradient techniques. One very interesting algorithm from this family is Paige and Saunder's LSQR algorithm for solving least squares problems [32]. The LSQR technique combines matrix-vector multiplication based steps (Lanczos and Arnoldi methods) with a QR decomposition step. The starting point for LSQR is to compute the factorization

$$T = V \begin{bmatrix} B \\ 0 \end{bmatrix} W', \quad V'V = \mathbf{1}_m, \quad W'W = \mathbf{1}_n,$$

where

$$B = B_n = \begin{bmatrix} \alpha_1 & & & \\ \beta_2 & \alpha_2 & & \\ & \beta_3 & \ddots & \\ & & \ddots & \alpha_n \\ & & & \beta_{n+1} \end{bmatrix} \in \mathbb{R}^{(n+1)\times n}$$

is a lower bi-diagonal matrix. The orthogonal matrices $V = (v_1, v_2, \ldots, v_m)$ and $W = (w_1, w_2, \ldots, w_n)$ can be computed as a product of Householder reflections or Jacobi rotations for the elimination of the matrix entries in $T$. However, this process will destroy the sparsity pattern of the coefficient matrix. For the case of sparse matrices, Golub and Kahan suggested an alternative procedure to compute this factorization, which is based on a Lanczos process. This process is again using matrix-vector multiplications to propagate vectors and to leave the coefficient matrix $T$ unchanged, hence preserving its sparsity pattern. The Lanczos process starts by setting $\beta_1 w_0 = 0$ and $\alpha_{n+1} w_{n+1} = 0$. If we initialize the process with the vector $\beta_1 v_1 = u \in \mathbb{R}^m$ and $\alpha_1 w_1 = T'v_1$ then, for $k = 1, 2, \ldots$ the recurrence relations

$$\beta_{k+1} v_{k+1} = T w_k - \alpha_k v_k, \quad \alpha_{k+1} w_{k+1} = T' v_{k+1} - \beta_{k+1} w_k,$$

continue to produce the sequence of vectors $w_1, v_2, w_2, \ldots, v_{m+1}$ and the corresponding entries in the bi-diagonal matrix $B$. Note that the parameters $\alpha_{k+1} \geq 0$ and $\beta_{k+1} \geq 0$ are determined such that $\|v_{k+1}\|_2 = \|w_{k+1}\|_2 = 1$. After $k$ steps the algorithm produces the matrices $V = (v_1, v_2, \ldots, v_k)$ and $W = (w_1, w_2, \ldots, w_{k+1})$ as well as the bi-diagonal matrix $B_k$. We now can go for an approximate solution vector $u_k \in \mathcal{K}_k$ lying in the Krylov subspace $\mathcal{K}_k = \mathcal{K}_k(T'T, T'y)$, where a Krylov subspace

is defined as $\mathcal{K}_k(T,u) = span\left[u, Tu, T^2u, \ldots, T^{k-1}u\right]$. Since in the present case we have $\mathcal{K}_k = span(W_k)$ it is possible to write $u_k = W_k\psi_k$.

For computing the vector $\psi_k$ we have to determine a solution of the least-square problem

$$\min_{\psi_k} \|B_k\psi_k - \beta_1 e_1\|_2.$$

The LSQR algorithm calculates this solution via the QR decomposition of $B_k$

$$Q_k B_k = \begin{bmatrix} R_k \\ 0 \end{bmatrix}, \quad Q_k(\beta_1 e_1) = \begin{bmatrix} f_k \\ \phi_{k+1} \end{bmatrix},$$

where $R_k$ is upper bi-diagonal

$$R_k = \begin{bmatrix} \rho_1 & \theta_1 & & & \\ & \rho_2 & \theta_2 & & \\ & & \ddots & \ddots & \\ & & & \rho_{k-1} & \theta_{k-1} \\ & & & & \rho_k \end{bmatrix} \in \mathbb{R}^{k\times k}, \quad f_k = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \\ \phi_k \end{bmatrix}.$$

Notice the similarity of this approach with the algorithm described in section 3.1. The matrix $Q_k$ is computed as a product of Jacobi rotations parametrized to eliminate the subdiagonal elements of $B_k$. The solution vector $\psi_k$ and the residual vector $r_{k+1}$ can be determined from

$$R_k\psi_k = f_k, \quad r_{k+1} = Q_k' \begin{bmatrix} 0 \\ \phi_{k+1} \end{bmatrix}.$$

The iterative solution $u_k$ is then computed via

$$u_k = \left(W_k R_k^{-1}\right) f_k = Z_k f_k,$$

where the matrix $Z_k$ satisfies the lower triangular system $R_k' Z_k' = V_k'$ such that we can computed the column vectors $(z_1, z_2, \ldots, z_k)$ by forward substitution. With $z_0 = u_0 = 0$ the process proceeds as

$$z_k = \frac{1}{\rho_k}\left(w_k - \theta_k z_{k-1}\right), \quad u_k = u_{k-1} + \phi_k z_k.$$

The LSQR algorithm produces the same sequence of intermediate solution vectors $u_k$ as the Conjugate Gradient Least Squares algorithm [3]. The algorithm only accesses the coefficient matrix $T$ only to produce the matrix-vector products $Tw_k$ and $T'v_k$ and exhibits preferable numerical properties, in particular for ill-conditioned matrices $T$ [32].

### Iterative Algorithms and Memory Bandwidth

An efficient design of an numerical algorithm strives to minimize the amount of arithmetic operations along with the amount of memory space needed to store the data. Besides those two important design criteria we have to consider the amount of data that needs to be moved in and out of main memory during the execution of an algorithm; this is denoted as memory bandwidth.

Even a modern real-time computer system may be challenged to provide excessive sustained memory bandwidth if it executes an iterative matrix solver for very large and sparse matrices. The challenge originates from iterating the solution vector, that is, from moving a potentially very large vector in and out of main memory for each iteration as it will not fit into the cache memory anymore.

For calculating a simple example we consider a problem where the solution vector $u$ has length $n$. Each vector entry is represented by a double precision floating point number using a word length of $B = 64$ bits. We assume that the iterative algorithm requires $N$ iterations until convergence. Hence, the algorithm needs to move $n \cdot N \cdot B$ bits of data between the CPU and Memory. In a real-time application such as in video signal processing (deconvolution, motion estimation, scan conversion, etc.) the algorithm has to complete its calculations within a time interval of $T$ seconds, which leads to a required memory bandwidth of $\frac{n \cdot N \cdot B}{T}$ bits per second. For a modern HDTV application ($1920 \times 1080$ pixels per frame) with a frame rate of 25 Hz, the necessary memory bandwidth amounts to over 41 GByte/sec, if we assume that the computation for one system of equation converges after $N = 100$ iterations. Even under these optimistic assumptions for the number of iterations the memory bandwidth becomes the critical element [33].

If the real-time computer system employs modern Graphical Processing Units (GPU), which offer high computational performance, this big amount of data needs to be transported over a bus between CPU and GPU [27],[5]. Hence, the data transport becomes a bottleneck to achieve the required system performance. Furthermore, the internal structure of a GPU, i.e. the shader does not support iterative computations very well. Taking all this together leads us to reconsider the use of direct solution methods for large scale systems as the pure operation count is not the most restricting factor [14]. If there are direct solution methods, which can take advantage of special structures in the coefficient matrix, such as sparsity, then this is beneficial.

# 7 Structure In Matrices

The classical QR algorithm on an $n \times n$ matrix $T$ has computational complexity of order $\mathcal{O}(n^3)$. Better computational complexity, at the cost of numerical stability, is offered by Strassens' method, but is certainly not advisable in the context of embedded processing, where numerical complexity plays only a minor role as compared to data transport and where good numerical properties are of paramount importance. There is, however, another much more promising possibility, which consists in exploiting intrinsic structure of the matrix. In the literature, many diverse types of matrix structure have been considered, such as Toeplitz or Hankel, in the present discussion we shall only consider the semi-separable or time-varying structure as it connects up nicely with QR factorization and is very important from an applications point of view. In our discussion on the Kalman filter, we were given a model for the system to be considered. That meant that the overall covariance matrix of the output data is not arbitrary, although the system parameters vary from one time point to the next. As a result, the Kalman filter in its square root version allowed state estimation on the basis not of the overall covariance data, but just on the basis of properties local to each point in time. The computational complexity was therefore determined, not by the size of the overall system, but by the dimension of the local state representation (called $\delta_i$.)

The structure we encountered implicitly in the Kalman filter is known by various terms, depending on the relevant literature, semi-separable systems, time-varying systems or quasi-separable systems. They appeared for the first time in [16], where it was shown that LU factorization of such a system would have $\mathcal{O}(n\overline{\delta_i^3})$ complexity instead of $\mathcal{O}(n^3)$. Later this idea was generalized to QR and other factorizations in [12]. We summarize the main results. It should be remarked that the complexity of the method we shall describe can be considerably better than $\mathcal{O}(n\overline{\delta_i^3})$, actually $\mathcal{O}(n\overline{\delta_i^2})$ when state space representations are judiciously chosen, but this topic goes far beyond our present purpose.

To work comfortably with time-varying systems, we need the use of sequences of indices and then indexed sequences. When $\mathcal{M} = [m_k]_{k=-\infty}^{\infty}$ is a sequence of indices, then each $m_k$ is either a positive integer or zero, and a corresponding indexed sequence $[u_k] \in \ell_2^{\mathcal{M}}$ will be a sequence of vectors such that each $u_k$ has dimension $m_k$ and the overall sum

$$\sum_{k=-\infty}^{\infty} \|u_k\|^2 \tag{45}$$

is finite, the square root of which is then the quadratic norm of the sequence. When $m_k = 0$, then the corresponding entry just disappears (it is indicated as a mere 'place holder'). A regular n-dimensional finite vector can so be considered as embedded in an infinite sequence, whereby the entries from $-\infty$ to zero and $n+1$ to $\infty$ disappear, leaving just $n$ entries indexed by $1 \cdots n$, corresponding e.g. to the time points where they are fed into the system. On such sequences we may define a generic shift operator $Z$. It is also convenient to represent sequences in row form, underlying the zero'th element for orientation purposes, taking transposes of the original vectors if they are in column form. Hence:

$$[\cdots, u'_{-2}, u'_{-1}, \underline{u'_0}, u'_1, u'_2, \cdots]Z = [\cdots, u'_{-2}, \underline{u'_{-1}}, u'_0, u'_1, \cdots] \tag{46}$$

$Z$ is then a unitary shift represented as a block upper unit matrix, whose inverse $Z'$ is then a lower matrix with first lower block diagonal consisting of unit matrices (notice that the indexing of the rows is shifted w.r. to the indexing of the columns.) Typically we handle only finite sequences of vectors, but the embedding in infinite ones allows us to apply delays as desired and not worry about the precise time points. Similarly, we handle henceforth matrices in which the entries are matrices themselves. For example, $T_{i,j}$ is a block of dimensions $m_i \times n_j$ with $[m_i] = \mathcal{M}$ and $[n_j] = \mathcal{N}$, and, again, unnecessary indices are just placeholder, with the corresponding block entries disappearing as well - also consisting of place holders (interesting enough, MATLAB now allows for such matrices, the lack of which was a major problem in previous versions. Place holders are very common in computer science, here they make their entry in linear algebra.)

In this convention we define a causal system by the set of equations

$$\left\{ \begin{array}{rcl} x_{i+1} & = & A_i x_i + B_i u_i \\ y_i & = & C_i x_i + D_i u_i \end{array} \right. \tag{47}$$

very much as before, but now with a direct term $D_i u_i$ added to the output equation (in the Kalman filter this term is zero, because the prediction is done strictly on past values.) $\left[ \begin{array}{cc} A_i & B_i \\ C_i & D_i \end{array} \right]$ is called the *system transition matrix* at time point $i$ ($A_i$ being the state transition matrix.) What is the corresponding input/output matrix $T$? As is tradition in system theory, we replace the local

equations above with global equations on the (embedded) sequences $u = [u_i]$, $y = [y_i]$ and $x = [x_i]$, and define 'global' block diagonal matrices $A = \mathrm{diag}(A_i)$, $B = \mathrm{diag}B_i$, etc... to obtain

$$\left\{ \begin{array}{rcl} Zx & = & Ax + Bu \\ y & = & Cx + Du \end{array} \right. \tag{48}$$

and for the input-output matrix

$$T = D + CZ'(I - AZ')^{-1}B. \tag{49}$$

This represents a block lower matrix in semi-separable form. A block upper matrix would have a similar representation, now with $Z$ replacing $Z'$:

$$T = D + CZ(I - AZ)^{-1}B \tag{50}$$

**Remark:** it is not necessary to change to a row convention to make the theory work as in [12]. Instead of defining $Z$ as pushing forward on a row of data, we have defined $Z'$ as pushing forward on a column of data. Hence, $Z' = Z^{-1}$ is the causal shift, and a causal matrix is lower (block) triangular, rather than upper.

Such representations, often called *realizations*, produce in a nutshell the special structure of an upper, semi-separable system. When $T$ is block banded upper with two bands, then $A = 0$ and $B = I$ will do, the central band is represented by $D$ and the first off band by $C$. With a block three band, one can choose $A = \begin{bmatrix} 0 & 0 \\ I & 0 \end{bmatrix}$, $C = \begin{bmatrix} C_1 & C_2 \end{bmatrix}$ and $B = \begin{bmatrix} I \\ 0 \end{bmatrix}$, with $Z := \begin{bmatrix} Z & \\ & Z \end{bmatrix}$ because the state splits in two components. We find, indeed, $Z(I - AZ)^{-1} := \begin{bmatrix} Z & 0 \\ Z^2 & Z \end{bmatrix}$, and hence $T = D + C_1 Z + C_2 Z^2$. This principle can easily be extended to yield representations for multi-band matrices or matrix polynomials in $Z$.

State space representations are not unique. The dimension chosen for $x_i$ at time point $i$ may be larger than necessary, in which case one would call the representation 'non minimal' - we shall not consider this case further. Assuming a minimal representation, one could also introduce a non singular state transformation $R_i$ at each time point, defining a transformed state $\hat{x}_i = R_i^{-1} x_i$. The transformed system transition matrix now becomes

$$\begin{bmatrix} \hat{A}_i & \hat{B}_i \\ \hat{C}_i & D_i \end{bmatrix} := \begin{bmatrix} R_{i+1}^{-1} A_i R_i & R_{i+1}^{-1} B_i \\ C_i R_i & D_i \end{bmatrix}. \tag{51}$$

for a lower system, and a similar, dual representation for the upper.

Given a block upper matrix $T$, what is a minimal system representation for it? This problem is known as the *system realization problem*, and was solved for the first time by Kronecker (for representations of rational functions [26]), and then later by various authors, for the semi-separable case, see [12] for a complete treatment. An essential role in realization theory is played by the so called $i^{\mathrm{th}}$ Hankel matrix $H_i$ defined as

$$H_i = \begin{bmatrix} \vdots & \vdots & \ddots \\ T_{i-1,i+1} & T_{i-1,i+2} & \cdots \\ T_{i,i+1} & T_{i,i+2} & \cdots \end{bmatrix} \tag{52}$$

i.e. a right-upper corner matrix just right of the diagonal element $T_{i,i}$. It turns out that any minimal factorization of each $H_i$ yields a minimal realization, we have indeed

$$H_i = \begin{bmatrix} \vdots \\ C_{i-2}A_{i-1}A_i \\ C_{i-1}A_i \\ C_i \end{bmatrix} \begin{bmatrix} B_{i+1} & A_{i+1}B_{i+2} & A_{i+1}A_{i+2}B_{i+3} & \cdots \end{bmatrix} \tag{53}$$

where, as explained before, entries may disappear when they reach the border of the matrix e.g. This decomposition has an attractive physical meaning. We recognize

$$\mathcal{O}_i = \begin{bmatrix} \vdots \\ C_{i-2}A_{i-1}A_i \\ C_{i-1}A_i \\ C_i \end{bmatrix} \tag{54}$$

as the $i^{\text{th}}$ *observability operator*, and

$$\mathcal{R}_i = \begin{bmatrix} B_{i+1} & A_{i+1}B_{i+2} & A_{i+1}A_{i+2}B_{i+3} & \cdots \end{bmatrix} \tag{55}$$

as the $i^{\text{th}}$ *reachability operator* - all these related to the (anti-causal) upper operator we assumed. $\mathcal{R}_i$ maps inputs after the time point $i$ to the state $x_i$, while $\mathcal{O}_i$ maps state $x_i$ to actual and outputs before index point $i$, giving its linear contribution to them. The rows of $\mathcal{R}_i$ form a basis for the rows of $H_i$, while the columns of $\mathcal{O}_i$ form a basis for the columns of $H_i$ in a minimal representation. When e.g. the rows are chosen as an orthonormal basis for all the $H_i$, then a realization will result for which $A_iA_i' + B_iB_i' = I$ for all $i$. We call a realization in which $\begin{bmatrix} A_i & B_i \end{bmatrix}$ has this property of being part of an orthogonal or unitary matrix, in *input normal form*.

It may seem laborious to find realizations for common systems. Luckily, this is not the case. In many instances, realizations come with the physics of the problem. Very common are, besides block banded matrices, so called smooth matrices [35], in which the Hankel matrices have natural low-rank approximations, and ratios of block banded matrices (which are in general full matrices), and, of course, systems derived from linearly couples subsystems.

## Solving Semi-Separable Systems With QR: The URV Method

The goal of an URV factorization is a little more ambitious than the QR factorization presented as the beginning. As we saw, the factorization only works well when $T$ is non-singular, otherwise we end up with an $R$ factor that is not strictly triangular but only upper and in so called 'echelon' or staircase form. Clearly, row operations are not sufficient. To remedy the situation, one also needs column operations that reduce the staircase form to purely triangular. This is in a nutshell the URV factorization, in which U is a set of columns of an orthogonal matrix, V a set of rows of another, and R is strictly upper triangular and invertible. When $T = URV$ and $T$ is invertible, then $U$ and $V$ will be unitary, and $T^{-1} = V'R^{-1}U'$. However, when $T$ is general, then the solution of the least squares solution for $y = Tu$ is given by $u = T^{\dagger}y$ with $T^{\dagger} = V'R^{-1}U'$ (the same would be true for $y = uT$, now with $u = yV'R^{-1}U'$!) $T^{\dagger}$ is called the 'Moore-Penrose inverse' of $T$.

The URV recursion would start with orthogonal operations on (block) columns, transforming the mixed matrix $T$ to the upper form - actually one may alternate (block) column with (block) row operations to achieve a one pass solution. However, the block column operations are completely independent from the row operations, hence we can treat them first and then complete with row operations. We assume a semi-separable representation for $T$ where the lower and upper parts use different state space realizations (all matrices shown are block diagonal and consisting typically of blocks of low dimensions):

$$T = C_\ell Z'(I - A_\ell Z')^{-1} B_\ell + D + C_u Z(I - A_u Z)^{-1} B_u \tag{56}$$

This corresponds to a 'model of computation' shown in fig. 10. The URV factorization starts with

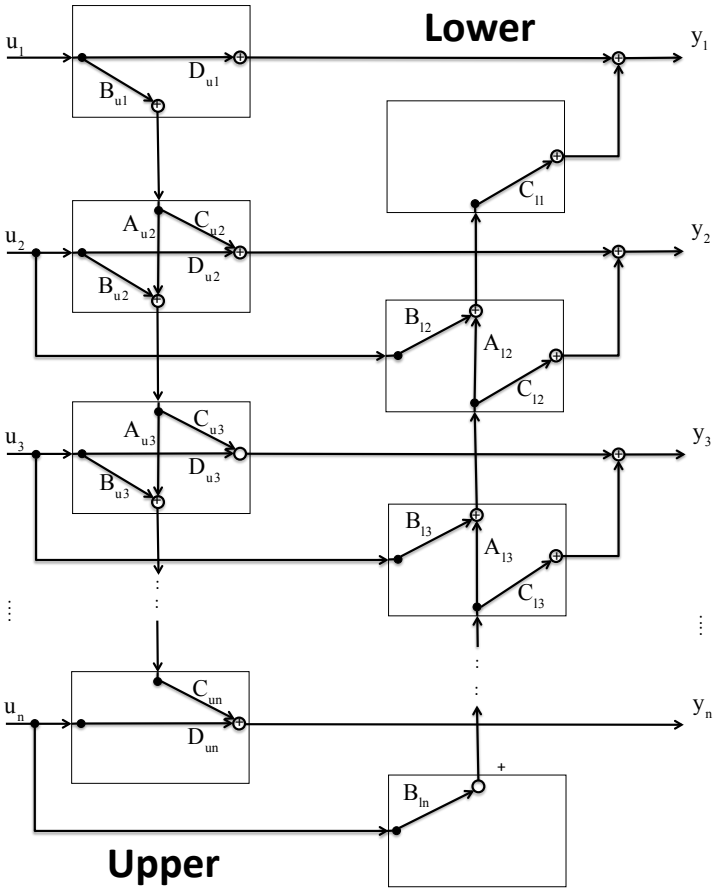


Figure 10: The semi-separable model of computation

getting rid of the lower or anticausal part in $T$ by post-multiplication with a unitary matrix, like in the traditional LQ factorization, but now working on the semi separable representation instead of on the original data. If one takes the lower part in input normal form, i.e. $\hat{C}_\ell Z'(I - \hat{A}_\ell Z')^{-1}\hat{B}_\ell = C_\ell Z'(I - A_\ell Z')^{-1} B_\ell$ such that $\hat{A}_\ell \hat{A}'_\ell + \hat{B}_\ell \hat{B}'_\ell = I$, then the realization for (upper) $V$ is given by

$$V \approx \begin{bmatrix} \hat{A}_\ell & \hat{B}_\ell \\ C_V & D_V \end{bmatrix} \tag{57}$$

where $C_V$ and $D_V$ are formed by unitary completion of the isometric $\left[\begin{array}{cc} \hat{A}_\ell & \hat{B}_\ell \end{array}\right]$ (for an approach familiar to numerical analysts see [35].) $V$ is a minimal causal unitary operator, which pushes $T$ to upper: $\left[\begin{array}{cc} T_u & 0 \end{array}\right] := TV$ can be checked to be upper (we shall do so further on where we show the validity of the operation) and a realization for $T_u$ follows from the preceding as

$$T_u \approx \left[\begin{array}{cc|c} \hat{A}'_\ell & 0 & C'_V \\ B_u\hat{B}'_\ell & A_u & B_uD'_V \\ \hline \hat{C}_\ell\hat{A}'_\ell + D\hat{B}'_\ell & C_u & \hat{C}_\ell\hat{C}'_V + DD'_V \end{array}\right]. \tag{58}$$

As expected, the new transition matrix combines lower and upper parts and has become bigger, but $T_u$ is now upper. Numerically, this step is executed as an LQ factorization as follows. Let $x_k = R_k\hat{x}_k$ and let us assume we know $R_k$ at step $k$, then

$$\left[\begin{array}{cc} A_{\ell,k}R_k & B_{\ell,k} \\ C_{\ell,k}R_k & D_k \end{array}\right] = \left[\begin{array}{ccc} R_{k+1} & 0 & 0 \\ \hat{C}_{\ell,k}\hat{A}'_{\ell,k} + D_k\hat{B}'_{\ell,k} & \hat{C}_{\ell,k}\hat{C}'_{V,k} + D_kD'_{V,k} & 0 \end{array}\right] \left[\begin{array}{cc} \hat{A}_{\ell,k} & \hat{B}_{\ell,k} \\ C_{V,k} & D_{V,k} \end{array}\right] \tag{59}$$

The LQ factorization of the left handed matrix computes everything that is needed, the transformation matrix, the data for the upper factor $T_u$ and the new state transition matrix $R_{k+1}$, all in terms of the original data. Because we have not assumed $T$ to be invertible, we have to allow for an LQ factorization that produces an echelon form rather than a strictly lower triangular form, and allows for a kernel as well, represented by a block column of zeros.

The next step is what is called an inner/outer factorization on the upper operator $T_u$ to produce an upper and upper invertible operator $T_o$ and an upper orthogonal operator $U$ such that $T_u = UT_o$. The idea is to find an as large as possible upper and orthogonal operator $U$ such that $U'T_u$ is still upper - $U'$ tries to push $T_u$ back to lower, when it does so as much as possible, an upper and upper invertible factor $T_o$ should result. There is a difficulty here that $T_u$ might not be invertible, already in the original QR case one may end up with an embedded $R_u$ matrix. This difficulty is not hard to surmount, but in order to avoid a too technical discussion, we just assume invertibility at this point and we shall see that the procedure actually produces the general formula needed. If the entries of $T_u$ would be scalar, then we would already have reached our goal. However, the operation of transforming $T$ to a block upper matrix $T_u$ will destroy the scalar property of the entries, and the inverse of $T_u$ may now have a lower part, which will be captured by the inner operator $U$ that we shall now determine.

When $T_u = UT_o$ with $U$ upper and orthogonal, then we also have $T_o = U'T_u$. Writing out the factorization in terms of the realization, and redefining for brevity $T_u := D + CZ(I - AZ)^{-1}B$ we obtain

$$\begin{array}{rcl} T_o & = & \left[D'_U + B'_U(I - Z'A'_U)^{-1}Z'C'_U\right]\left[D + CZ(I - AZ)^{-1}B\right] \\ & = & D'_UD + B'_U(I - Z'A'_U)^{-1}Z'C'_UD + D'_UCZ(I - AZ)^{-1}B \\ & & +B'_U\{(I - Z'A'_U)^{-1}Z'C'_UCZ(I - AZ)^{-1}\}B \end{array} \tag{60}$$

This expression has the form: 'direct term' + 'strictly lower term' + 'strictly upper term' + 'mixed product'. The last term has what is called 'dichotomy', what stands between $\{\cdot\}$ can again be split in three terms:

$$(I - Z'A'_U)^{-1}Z'C'_UCZ(I - AZ)^{-1} = (I - Z'A'_U)^{-1}Z'A'_UY + Y + YAZ(I - AZ)^{-1} \tag{61}$$

with $Y$ satisfying the 'Lyapunov-Stein equation'

$$ZYZ' = C'_UC + A'_UYA \tag{62}$$

or, with indices: $Y_{k+1} = C'_{U,k}C_k + A'_{U,k}Y_kA_k$. The resulting strictly lower term has to be annihilated, hence we require $C'_UD + A'_UYB = 0$, in fact $U$ should be chosen maximal with respect to this property (beware: Y depends on U!) Once these two equations are satisfied, the realization for $T_o$ results as $T_o = (D'_UD + B'_UYB) + (D'_UC + B'_UYA)Z(I - AZ)^{-1}B$ - we see that $T_o$ inherits $A$ and $B$ from $T$ and gets new values for the other constituents $C_o$ and $D_o$. Putting everything together in one matrix equation and in a somewhat special order, we obtain

$$\begin{bmatrix} YB & YA \\ D & C \end{bmatrix} = \begin{bmatrix} B_U & A_U \\ D_U & C_U \end{bmatrix} \begin{bmatrix} D_o & C_o \\ 0 & ZYZ' \end{bmatrix}. \tag{63}$$

Let us interpret this result without going into motivating theory (as in done in [12, 35]). We have a pure QR factorization of the left hand side. At stage $k$ one must assume knowledge of $Y_k$, and then perform a regular QR factorization of $\begin{bmatrix} Y_kB_k & Y_kA_k \\ D_k & C_k \end{bmatrix}$. $D_{o,k}$ will be an invertible, upper triangular matrix, so its dimensions are fixed by the row dimension of $Y_k$. The remainder of the factorization produces $C_{o,k}$ and $Y_{k+1}$, and, of course, the Q factor that gives a complete realization of $U_k$. What if $T$ is actually singular? It turns out that then the QR factorization will produce just an upper staircase form with a number of zero rows. The precise result is

$$\begin{bmatrix} Y_kB_k & Y_kA_k \\ D_k & C_k \end{bmatrix} = \begin{bmatrix} B_{U,k} & A_{U,k} & B_{W,k} \\ D_{U,k} & C_{U,k} & D_{W,k} \end{bmatrix} \begin{bmatrix} D_{o,k} & C_{o,k} \\ 0 & Y_{k+1} \\ 0 & 0 \end{bmatrix}, \tag{64}$$

in which the extra columns represented by $B_W$ and $D_W$ define an isometric operator $W = D_W + C_WZ(I - A_WZ)^{-1}B_W$ so that

$$T_u = \begin{bmatrix} U & W \end{bmatrix} \begin{bmatrix} T_o \\ 0 \end{bmatrix}. \tag{65}$$

In other words, $W$ characterizes the row kernel of $T$.

Remarkably, the operations work on the rows of $T_u$ in ascending index order, just as the earlier factorization worked in ascending index order on the columns. That means that the URV algorithm can be executed completely in ascending index order. The reader may wonder at this point (1) how to start the recursion and (2) whether the proposed algorithm is numerically stable. On the first point and with our convention of empty matrices, there is no problem starting out at the upper left corner of the matrix, both $A_1$ and $Y_0$ are just empty, the first QR is done on $\begin{bmatrix} D_1 & C_1 \end{bmatrix}$. In case the original system does not start at index 1, but has a system part that runs from $-\infty$ onwards, then one must introduce knowledge of the initial condition on $Y$. This is provided, e.g., by an analysis of the LTI system running from $-\infty$ to 0 if that is indeed the case, see [13] for more details. On the matter of numerical stability, we offer two remarks. First, propagating $Y_k$ is numerically stable, one can show that a perturbation on any $Y_k$ will die out exponentially if the propagating system is assumed exponentially stable. Second, one can show that the transition matrix $\Delta$ of the inverse of the outer part will be exponentially stable as well, when certain conditions on the original system are satisfied [12].

To obtain the Moore-Penrose inverse (or the actual inverse when $T$ is invertible) one only needs to specify the inverse of $T_o$, as the inverses of $U$ and $V$ are already known (they are just $U'$ and $V'$ with primed realizations as well.) By straightforward elimination, and with the knowledge that $T_o$ is upper invertible, we find with $\Delta = A - BD_o^{-1}C_o$,

$$T_o^{-1} = D_o^{-1} - D_o^{-1}C_oZ(I - \Delta Z)^{-1}BD_o^{-1}. \tag{66}$$

One does not need to compute these matrices, the inverse filter can easily be realized directly on the realization of $T_u$ by arrow reversal, as shown in fig. 11.

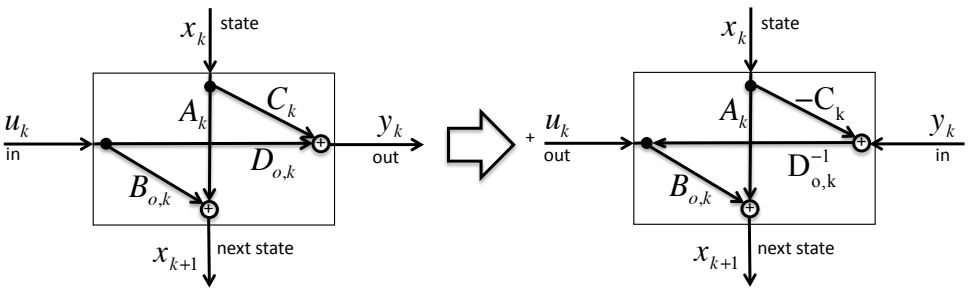


Figure 11: Realization of the inverse of an outer filter in terms of the original

With a bit of good will, one recognizes the square root algorithm for the Kalman filter as a (very) special case of inner-outer factorization. Now, we actually have the dual case, due to the fact that the anti-causal part first had to be eliminated, forcing the inner-outer factorization on the rows rather than on the columns (we would have found the exact same formula as for the Kalman filter, if we had started with an upper matrix and then had done an outer-inner rather than an inner-outer factorization.) There is an alternative to the URV algorithm presented here, namely working first on the rows and then on the columns, but that would necessitate a descending rather than an ascending recursion.

# 8 Concluding remarks

The interplay of signal processing applications, numerical linear algebra algorithms and real-time computing architectures is a fascinating cross-road of interdisciplinary research, which is under constant change due to technological progress in all associated fields. Besides all the research aspects this is also an essential aspect for designing engineering curricula - students need to learn about this interplay, to command a good understanding of the associated domains. This understanding is essential for them to successfully design real-time computer systems implementing state of the art products in terms of highly integrated embedded systems.

We discussed the QR decomposition of a coefficient matrix as a versatile computational tool that is central to many such algorithms because of its superior numerical properties matching the requirements for implementation on parallel real-time architectures. Besides its preferable properties, the QR decomposition provides for an elegant computational framework that allows for an improved understanding of many related concepts, including Kalman filtering and time-varying system theory.

Many of the mentioned arguments and features of algorithms for solving linear systems have been investigated and discussed in the context of algorithm specific systolic VLSI arrays for signal pro-

cessing applications [28]. Even though the hey-days of systolic arrays are gone, the topics are still relevant, since programmable Graphical Processing Units (GPU) are relevant parallel data processing targets for implementing number crunching algorithms in real-time computer systems [5],[27], [14].

Fragment shaders of programmable GPUs provide for a high performance parallel computing platform, but algorithms have to be able to fully exploit the performance offered by GPUs. Besides the costs for arithmetic operations and static memory designers need to consider the cost associated with data transport (memory bandwidth) as another important design criterium. Iterative matrix solvers for large-scale sparse matrices are attractive for many large-scale computational problems running on mainframe computers, however, for real-time applications running on embedded systems it may in fact be more interesting to use other algebraic means to exploit the structure of matrices. The advent of concepts to exploit the semi-separable or hierarchically semi-separable matrix structure holds new promise for attacking large-scale computational problems using embedded real-time computer systems.

## References


[1] H.M. Ahmed, J.-M. Delosme, and M. Morf. Highly concurrent computing structures for matrix arithmetic and signal processing. *IEEE Computer*, 15(1):65–86, 1982.

[2] Bart Kienhuis e.a. *Hotspot Parallelizer*. Compaan Design, 2010.

[3] A. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, Pennsylvania, 1996.

[4] B.G. Schunck B.K.P. Horn. Determining optical flow. *Aritifiial Intelligence*, 17(1-3):185–203, 1981.

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In *SIGGRAPH 2003*, pages 917–924. ACM, 2003.

[6] Jichun Bu, Ed.F. Deprettere, and P. Dewilde. A design methodology for fixed-size systolic arrays. In *Proceedings of the International Conference on Application Specific Array Processors*, 1990.

[7] D. Larsson and P. Schinner e.a. *The CADMUS 9230 ICD Graphic Workstation 1*, volume The Integrated Design Handbook, chapter 8, pages 8.1–8.29. Delft University Press, 1986.

[8] Jean-Marc Delosme and Ilse C.F. Ipsen. Parallel solution of symmetric positive definite systems with hyperbolic rotations. *Linear Algebra and its Applications*, 77:75 – 111, 1986.

[9] P. Dewilde. New algebraic methods for modelling large-scale integrated circuits. *International Journal of Circuit Theory and Applications*, 16(4):473–503, 1988.

[10] P. Dewilde, K. Diepold, and W. Bamberger. Optic flow computation and time-varying system theory. In *Proceedings of the International Symposium on Mathematical Theory of Networks and Systems (MTNS)*. Katholieke Universiteit Leuven, Belgium, July 2004.

[11] P. Dewilde, K. Diepold, and W. Bamberger. A semi-separable approach to a tridiagonal hierarchy of matrices with applications to image analysis. In *Proceedings of the International Symposium on Mathematical Theory of Networks and Systems (MTNS)*. Katholieke Universiteit Leuven, Belgium, July 2004.

[12] P. Dewilde and A.-J. van der Veen. *Time-varying Systems and Computations*. Kluwer, 1998.

[13] P. Dewilde and A.-J. van der Veen. Inner-outer factorization and the inversion of locally finite systems of equations. *Linear Algebra and its Applications*, 313:53–100, 2000.

[14] M. Durkovic, M. Zwick, F. Obermeier, and K. Diepold. Performance of optical flow techniques on graphics hardware. In *IEEE International Conference on Multimedia and Expo (ICME)*, 2006.

[15] V.N. Fadeeva. *Computational Methods of Linear Algebra*. Dover Publications, New York, New York, 1959.

[16] I. Gohberg, T. Kailath, and I. Koltracht. Linear complexity algorithms for semiseparable matrices. *Integral Equations and Operator Theory*, 8:780–804, 1985.

[17] G. Golub and Ch. van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, Maryland, 1989.

[18] J. Götze and U. Schwiegelshohn. Sparse matrix-vector multiplication on a systolic array. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 2061 – 2064 vol.4. IEEE, 1988.

[19] R.I. Hartley. In defence of the 8-point algorithm. *IEEE Transctions on Pattern Analysis and Machine Intelligence*, 19(6):580 – 593, 1997.

[20] F. M. F. Gastona; G. W. Irwina. Systolic approach to square root information kalman filtering. *International Journal of Control*, 50(1):225–248, 1989.

[21] K. Jainandunsing and E. F. Deprettere. A new class of parallel algorithms for solving systems of linear equations. *SIAM Journal on Scientific Computing*, 10(5):880–912, 1989.

[22] K. Jainandunsing and Ed.F. Deprettere. A new class of parallel algorithms for solving, systems of linear equations. *SIAM J. Sct. Stat. Comput.*, 10(5):880–912, September 1989.

[23] T. Kailath. *Lectures on Wiener and Kalman Filtering*. Springer Verlag, CISM Courses and Lectures No. 140, Wien, New York, 1981.

[24] T. Kailath and A. Sayed. *Fast Reliable Algorithms for Matrices with Structure*. SIAM, Philadelphia, Pennsylvania, 1999.

[25] T. Kailath, A. Sayed, and B. Hasibi. *Linear Esimtation*. Prentice Hall, Upper Saddle River, New Jersey, 2000.

[26] L. Kronecker. Algebraische Reduction der schaaren bilinearer Formen. *S.B. Akad. Berlin*, pages 663–776, 1890.

[27] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH 2005*. ACM, 2005.

[28] S.Y. Kung. *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[29] B.D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of Imaging Understanding Workshop*, pages 121–130, 1981.

[30] M. Misraa, D. Nassimib, and V. K. Prasannaa. Efficient vlsi implementation of iterative solutions to sparse linear systems. *Parallel Computing*, 19(5):525–544, May 1993.

[31] J. G. Nash and S. Hansen. Modified faddeeva algorithm for concurrent execution of linear algebraic operations. *IEEE TRANSACTIONS ON COMPUTERS*, 37(2):129–137, February 1988.

[32] Ch.C. Paige and M.A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Softw.*, 8:43–71, March 1982.

[33] L. A. Polka, H. Kalyanam, G. Hu, and S. Krishnamoorthy. Package technology to address the memory bandwidth challenge for tera-scale computing. *Intel Technology Journal*, 11(3):197–206, 2007.

[34] I.K. Proudler, J.G. McWhirter, and T.J. Shepherd. Computationally efficient qr decomposition approach to least squares adaptive filtering. *IEE Proceedings F, Radar and Signal Processing*, 138(4):341 – 353, 1991.

[35] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, A.-J. van der Veen and J. Xia. *A fast backward stable solver for sequentially semi-separable matrices*, volume HiPC202 of *Lecture Notes in Computer Science*, pages 545–554. Springer Verlag, Berlin, 2002.

[36] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, Pennsylvania, 2003.

[37] G. Strang. *Computational Science and Engineering*. Wellesley-Cambridge Press, Wellesley, MA, 2007.

[38] L. Tong, A.-J. van der Veen, and P. Dewilde. A new decorrelating rake receiver for long-code wcdma. In *Proceedings 2002 Conference on Information Sciences Systems*. Princeton University, March 2002.

[39] R. Vanderbril, M. van Barel, and N. Mastronardi. *Matrix Computations and Semi-Separable Matrices*. John Hopkins University Press, Baltimore, Maryland, 2008.