Time-Variant and Quasi-separable Systems*
supplementary reading
.
- Computational Tasks -

Klaus Diepold, Patrick Dewilde

Spring 2025

1 Standard Computational Tasks

1.1 Introduction

Computational Science is an important field of research that has an increasing impact in all disciplines of science and engineering. Engineers have to study this more mathematical subject matter in order to solve their problems. The engineering approach to problem solving also introduces characteristic viewpoints and technically motivated questions onto computations, which may not be covered and answered in their entirety by Mathematics. So engineers have an opportunity to contribute their own native methods to the toolbox of computational methods. The present course is an attempt to establish such an engineering viewpoint on scientific computation by developing system theoretic concepts and frameworks to better understand computational issues and to find new methods and solutions, which are amenable to solving engineering problems.

We can see three major domains, where engineers employ numerical computations for problem solving. These three domains are

1. Numerical simulation of physical phenomena – bottom-up computations,
2. Analysis of empirical data sets – top-down computations,
3. Numerical computations embedded in a physical world – cyber-physical systems.

Each of these domains comes with its own particular mathematical structural characteristics. Also, the computations involved and their respective implementations need to satisfy technical requirements in order to be useful from an engineering point of view. I will discuss all three domains in a little more detail to clarify the particularities of each domain.

*P. Dewilde, K. Diepold, A.-J. v.d. Veen. Time-Variant ans Quasi-separable Systems, Cambridge University Press, 2024

1.2 Simulation

1.2.1 Characteristics

Simulation-based engineering work utilizes mathematical descriptions of physical phenomena, computing the static and dynamic properties of these physical phenomena by evaluating the corresponding formulas numerically. The mathematical descriptions come in the form of equations, which may be linear or non-linear, the may come as any kind of ordinary differential equations or as one of many variations of partial differential equations, depending on the problem at hand. For numerical computations these equations need to be linearized and discretized, solutions include numerical differentiation and integration techniques. For actually solving a numerical simulation problem we need real data for either setting the boundary conditions (boundary value problem) or for setting the initial conditions (initial value problem). Real data may come from application specific considerations or from real world measurements. Setting the equations for complex phenomena often is based on relatively simple equations to describe the behavior of elementary subsystems. A large number of such elementary subsystems are then put together to establish a large-scale simulation for the entire system. Take circuit-simulation as an example, where the relationship between the current and the voltage at an elementary electronic device is described by simple differential equations. All the electronic devices comprised in an complete circuit or even chip are then combined by Kirchhoff equations. the result is a large algebro-differential system of equations which needs to be solved numerically.

Since large-scale computations are compiled of many elementary computational models we also refer to this as a *bottom-up* approach to computational engineering.

1.2.2 Typical Application Examples

There exists a long list of application domains that employ a bottom-up strategy to numerical computations. This domain is characterized by the existence of mathematical formulas to describe the physical behavior of elementary building elements. Typical examples are: Circuit Simulation (analog and digital), Weather Forecast, Climate Modeling, Aerodynamics, Fluid Dynamics, Particle Physics, High-Energy Physics, Geology, Geography, Hydrology, Structural Design, Electromagnetic Wave Propagation (Antenna Design), Light and matter transport (Computer Graphics) and many more.

1.3 Data-driven Science and Engineering

1.3.1 Characteristics

Engineers and scientists analyze empirical measurement data to find out some sort of relationship or physical phenomena that is reflected in the data. This is a data-driven approach to science and engineering. The assumption is that a underlying physical law creates structures and patterns in the data, which needs to be identified and extracted by means of data analysis techniques. In many cases we assume that we can apply a linear model. We can use linear models even for non-linear relations if we linearize this relation around an operation point using a differential approach. Such an approach also requires techniques to estimate the complexity of the models, which is often expressed in terms of the rank of a matrix or the dimension of a dominant subspace spanned by the data samples. Georg Simon Ohm employed such an approach to find out about the relation between current and voltage, which lead to the discovery of the concept of resistance.

The matrices that we encounter in data analysis tasks are often very large, they are mostly dense matrices (i.e. all or nearly all matrix entries are different from zero) and they may not exhibit clearly visible structure. However, the data matrices often exhibit low-rank structures and some sort of hidden structure, i.e. some non-trivial dependencies between the matrix entries, which is not known a priori. While the corresponding numerical and statistical techniques have been around for some time, they are re-invented in recent times under the label of 'machine learning'. The effectiveness of data-analytical and statistical techniques is one aspect for research and development. The efficiency of the corresponding computations is another aspect with high impact for practical purposes.

1.3.2 Typical Application Examples

The list of application examples in this category is growing by the minute. Some of the key words in this context are summarized in the following list. Data Mining, Internet Search (e.g. Page Rank), Machine Learning (supervised and unsupervised), Deep Learning, Neural Nets, Reinforcement Learning, Statistical Data Analysis, User Studies, Analysis of Measurement Data, Predictive Data Analysis and Diagnostics and many more ...

1.4 Cyber-Physical Systems

1.4.1 Characteristics

Many technical systems that interact with the real world contain parts of the internal workings which are implemented in terms of a computer. Such a computer is interfacing with the real world through sensors and actuators and are also termed 'caber-physical systems'. The physical behavior is implemented by a numerical algorithm, which needs to be implemented on a computing platform that has restricted computational resources (limited computational capabilities and small memory capacity. As a consequence this type of applications requires efficient numerical algorithms, in particular if such an embedded system shall incorporate functionalities requiring increased levels of intelligence.

1.4.2 Typical Application Examples

Mobile Communications, Automotive Applications (Driver Assistance Systems), Process Control, Aeronautical Control, Robotics, Automation Systems, Autonomous Systems.

2 Linear Systems

2.1 Standard Representation

We can use the equivalence between Linear Systems on the one hand and Linear Algebra on the other hand if we want to represent such systems or if we are computing with input and output signals of linear systems. In Figure 1 depicts a symbolic representation for an system, that takes input sequence $[u_k]$ and produces the output sequence $[y_k]$, which is determined as the transfer operator $\mathcal{T}\{\cdot\}$ applied to the input sequence.

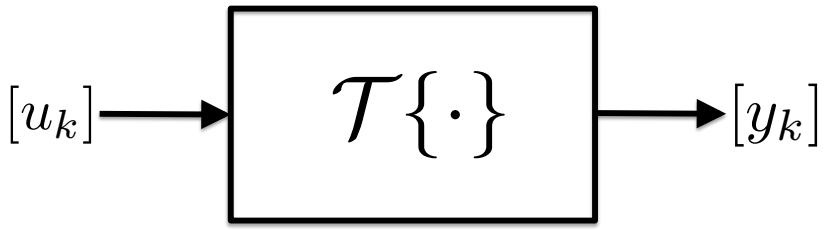

Figure 1: Standard symbolic representation of a linear input-output system

Once we are dealing with *linear* systems, the sequences $[u_k]$ and $[y_k]$ are mapped into vectors u and y of length m and n,

$$[u_k] \mapsto u = \begin{bmatrix} \vdots \\ u_{k-1} \\ u_k \\ u_{k+1} \\ \vdots \end{bmatrix}, \qquad [y_k] \mapsto y = \begin{bmatrix} \vdots \\ y_{k-1} \\ y_k \\ y_{k+1} \\ \vdots \end{bmatrix},$$

respectively and the transfer operator is represented as an a matrix with real- or complex-valued entries. The standard computational problem using a matrix-vector notation is then given as

$$T \cdot u = y. \tag{1}$$

Note that this representation of linear systems includes static as well as dynamic systems. Any linear system can be represented in terms of a matrix-vector product and hence the powerful toolbox of Linear Algebra is at our disposal.

2.2 Problem Formulations

Using equation 1 we can distinguish between three different standard problems, depending on which of the three variables are supposed to be known.

- Matrix T and vector u are known and we wish to compute the vector y. This can be seen as feeding a linear system, which is represented by the matrix T with an input signal u and we wish to determine the output signal y. This problem can be briefly represented as

  $$(T, u) \mapsto y.$$

  We denote this operation as *filtering*.

- Vectors u and y are known and we wish to determine the matrix T from the observations of input and output signals. This problem can be briefly represented as

  $$(u, y) \mapsto T.$$

  We denote this operation as *system identification*. Looking at machine intelligence this task amounts to supervised training a machine intelligence model, where the given u and y denote the training

data for identifying the parameters pertaining to the model comprised in T. It is clear that for successfully identifying the values of all matrix entries in T we require a sufficient number of signal pairs (u, y), e.g. to set up a system of equations such as $T \cdot U = Y$, where the matrices U and Y contain the known input/output vectors, i.e.

$$U = \begin{bmatrix} | & | & \dots & | \\ u_1 & u_2 & \dots & u_n \\ | & | & \dots & | \end{bmatrix} \qquad Y = \begin{bmatrix} | & | & \dots & | \\ y_1 & y_2 & \dots & y_n \\ | & | & \dots & | \end{bmatrix}.$$

- Matrix T is known along with the vector y and we wish to compute the input signal that caused the output signal y. This problem can be briefly represented as

  $$(T, y) \mapsto u.$$

  We use the term *inversion problem* or *signal estimation* refer to this operation. Inversion problems are very common for many engineering problems, e.g. identifying the transmitted signal u having access to the received signal y and a model describing the transmission channel captured in the matrix T (channel model), or de-blurring images or removing reverberation from recorded audio signals.

3 Non-Linear Systems

Once we are considering non-linear systems we are no longer able to represent the mapping between input u and output y in terms of a matrix. We need a more general representation for the input-output operator $\mathcal{T}\{\cdot\}$. Most prominently, we will consider the non-linearities, which are introduced to neural nets by means for non-linear activation functions.

In Figure 3 a neural network is displayed, that has one input layer, two hidden layers and one output layer. The mapping of the input vector v_0 onto the output vector v_3 is represented by the nested map

$$F = F_3(F_2(F_1(x, v))).$$

Each individual layer is given by the function F_k as

$$v_k = F_k(v_{k-1}) = \mathrm{ReLU}(T_k v_{k-1} + b_k),$$

where we have $v_k \in \mathcal{R}^{n_k}$, $b_k \in \mathcal{R}^{n_k}$ and hence $T_k \in \mathcal{R}^{n_k \times n_{k-1}}$. Note that this function is composed of two parts, an affine transformation $T_k v_{k-1} + b_k$ followed by a non-linear activation function. For out short exposition we just use the *Rectified Linear Unit* or ReLU for short, which is shown in Figure 2.

Looking at Figure 3 we can identify the necessary computations as we step through the network. These are

$$\begin{aligned} T_1 v_0 = \widehat{v_1} \quad &\rightarrow \quad \mathrm{ReLU}(\widehat{v_1}) \quad \rightarrow \quad v_1 \\ T_2 v_1 + b_2 = \widehat{v_2} \quad &\rightarrow \quad \mathrm{ReLU}(\widehat{v_2}) \quad \rightarrow \quad v_2 \\ T_3 v_2 + b_3 = \widehat{v_3} \quad &\rightarrow \quad \mathrm{ReLU}(\widehat{v_3}) \quad \rightarrow \quad v_3 \end{aligned}$$

,

where we can observe that $T_1 \in \mathcal{R}^{8\times 4}$, $T_2 = \in \mathcal{R}^{4\times 8}$ and $T_3 = \in \mathcal{R}^{2\times 4}$, and $b_1 = 0$.

The computation of the affine transformation $T_k v_{k-1} + b_k$ requires $\mathcal{O}(n_k \cdot n_{k-1})$ operations, as n_k, n_{k-1} denote the number of neurons in the respective layers. For short we can summarize this a computational

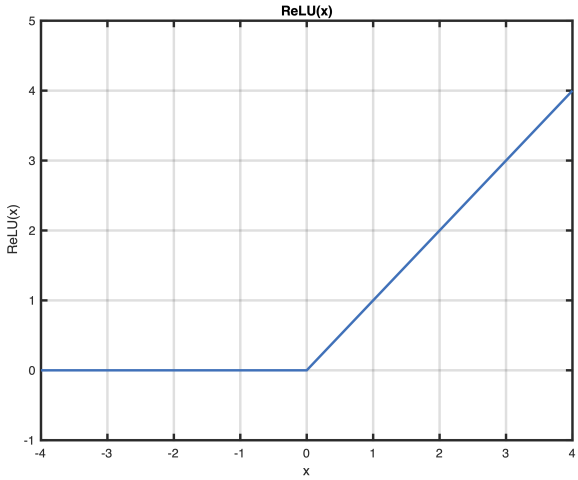

Figure 2: ReLU - Rectified Linear Unit

task of complexity $\mathcal{O}(n^2)$. The ReLU function needs only $\mathcal{O}(n)$ operations. This observation tells us that biggest junk of computation work is done by the affine transformation. It is also necessary to store the $\mathcal{O}(n^2)$ parameters for the matrix T_k and the bias vector b_k.

The little example shown in Figure 3 is a very modest neural net. You can easily imagine a state of the art neural net (Deep Learning) to comprise a much larger number of layers and nodes per layer. This is even more so as we consider convolutional neural nets (CNN), where the mapping between layers are consisting of convolutions.

In summary, for implementing Deep Neural Nets practically, it is of interest to look for ways how to reduce the computational requirements (arithmetic and memory).

3.1 Typical Computational Tasks

- Function F and vector v_0 are known and we wish to compute the vector v_N. This can be seen as feeding a Deep Neural Net, which implements the learned function F with an input signal v_0 and we wish to determine the inference (output) signal v_N. This problem can be briefly represented as

  $$(F, v_0) \mapsto v_N.$$

  Of course, this setting applies to any form of inference performed by any intelligent system, e.g. when deploying a trained Deep Neural Net, as it occurs in prediction or classification tasks.

- Vectors v_0 and v_N are known and we wish to determine the function F from the observations of input and output signals. This problem can be briefly represented as

  $$(v_0, v_N) \mapsto F.$$

  We denote this operation as *Learning*. Looking at machine intelligence this task amounts to supervised training a machine intelligence model F, where the given v_0 and v_N denote the training data for identifying the parameters pertaining to the model comprised in F. In order to determine the paramters making up the function F forces us to provide a large number of different pairs of v_0 and v_N. For this task, the *Backpropagation Algorithm* is the preferred choice. Using Backpropagation requires many function evaluations, each of which needing $\mathcal{O}(n^2)$ operations.

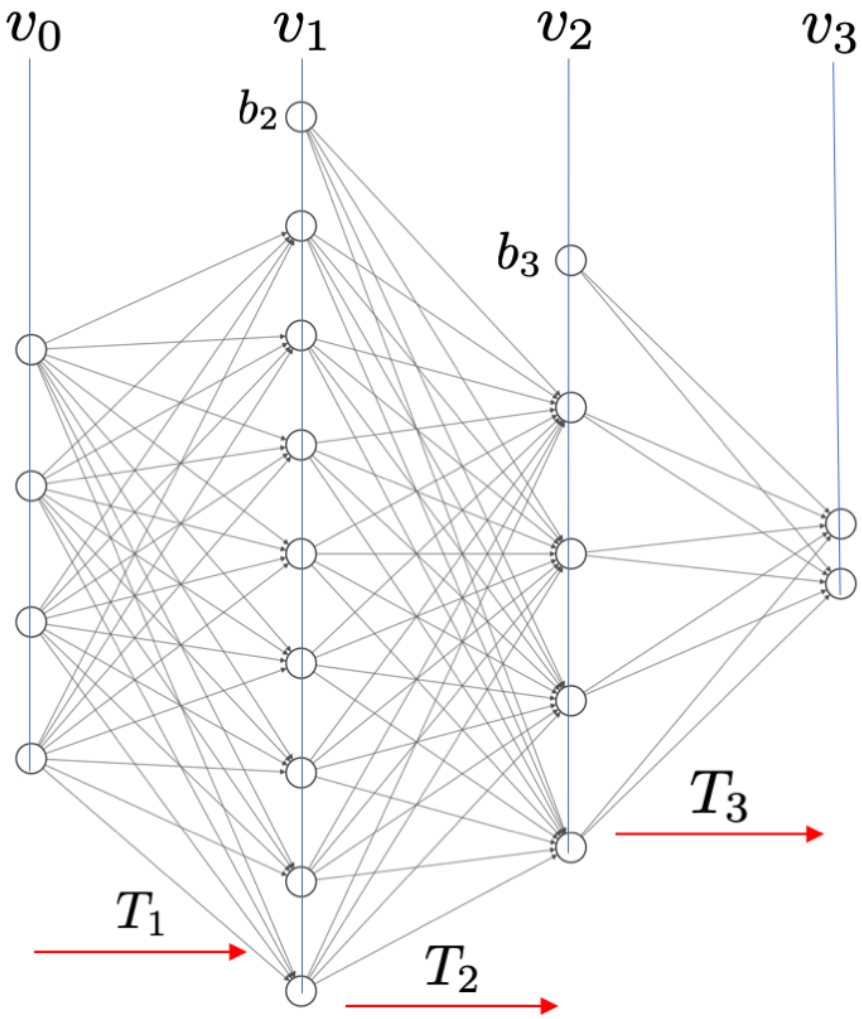

Figure 3: Neural Network

4 Numerical Linear Algebra

4.1 Cost of Linear Algebra

Numerical linear algebra is composed of many elementary computations, which require arithmetic operations. The number of elementary arithmetic operations required to compute such elementary results determine their computational complexity. The number of arithmetic operations is one measure to assess the efficiency of a given algorithm. It is a costly effort to determine the exact operation count for a given computation. For engineers it is often helpful to know the asymptotic complexity of an algorithm, that is, to know how the number of operations grows with the size of the problem. The asymptotic behaviour amounts to look for very large values of the parameter n, which describes the size of the problem to be solved. For elementary linear algebra computations the following complexities are known, where we assume the vector x, y to be of length n and the matrices A, B to be of size $n \times n$:

- Inner product $x'y$ – $\mathcal{O}(n)$ operations
- Matrix-vector product Ax – $\mathcal{O}(n^2)$ operations
- Matrix addition $A + B$ – $\mathcal{O}(n^2)$ operations
- Matrix-matrix product AB – $\mathcal{O}(n^3)$ operations
- Matrix inversion A^{-1} – $\mathcal{O}(n^3)$ operations

For those asymptotic values we assume that the matrices involved have no particular structure that we could exploit to reduce the amount of computations. We further assume that the matrices are of size $n \times n$ and the vectors have corresponding sizes.

4.2 Data Sparse and Structured Matrices

In general, for a given $n \times n$-matrices we require memory for storing n^2 matrix entries. However, in engineering applications we often encounter matrices, which are data sparse, that is, where we need much less memory to store these matrices. Such matrices have 'structure' or their degrees of freedom are otherwise reduced. Simple examples for matrices with structure are symmetric or triangular matrices, which require to store only $n(n+1)/2$ values or rotation matrices, which require only $n(n-1)/2$ values. However, these matrices still consist of $\mathcal{O}(n^2)$ free values.

For our discussion we are considering matrices which have $\mathcal{O}(n)$ free parameters. While such matrices require a reduced amount of storage space we want to find computational schemes for such matrices, which also need only $\mathcal{O}(n)$ computations for e.g. matrix-vector multiplication, or $\mathcal{O}(n^2)$ for matrix-matrix multiplication or inversion.

Some matrices have many zero-entries such that there are $\mathcal{O}(n)$ non-zero entries left, which need to be stored. There are other matrices where the matrix entries exhibit inner relationships such that only $\mathcal{O}(n)$ entries need to be stored.

Diagonal matrices

$$D = \begin{bmatrix} d_1 & & & & \\ & d_2 & & & \\ & & d_3 & & \\ & & & \ddots & \\ & & & & d_n \end{bmatrix}$$

Block-diagonal matrices

$$D = \begin{bmatrix} \boxed{D_1} & & & & \\ & \boxed{D_2} & & & \\ & & \boxed{D_3} & & \\ & & & \ddots & \\ & & & & \boxed{D_n} \end{bmatrix},$$

where each block entry D_i has the dimension $m_i \times n_i$.

Sparse matrices

$$D = \begin{bmatrix} d_0 & & d_1 & & d_2 \\ & & & d_3 & \\ d_4 & & & & \\ & d_5 & d_6 & & d_7 \\ d_8 & & & & d_9 \end{bmatrix},$$

with $MathcalO(n)$ non-zero matrix entries d_i.

Toeplitz-, and Hankel-, Vandermonde-matrices

$$T = \begin{bmatrix} t_0 & t_1 & t_2 & t_3 & t_4 \\ t_{-1} & t_0 & t_1 & t_2 & t_3 \\ t_{-2} & t_{-1} & t_0 & t_1 & t_2 \\ t_{-3} & t_{-2} & t_{-1} & t_0 & t_1 \\ t_{-4} & t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix}, \qquad H = \begin{bmatrix} h_0 & h_1 & h_2 & h_3 & h_4 \\ h_1 & h_2 & h_3 & h_4 & h_5 \\ h_2 & h_3 & h_4 & h_5 & h_6 \\ h_3 & h_4 & h_5 & h_6 & h_7 \\ h_4 & h_5 & h_6 & h_7 & h_8 \end{bmatrix}$$

$$V = \begin{bmatrix} 1 & v_1 & v_1^2 & v_1^3 & v_1^4 \\ 1 & v_2 & v_2^2 & v_2^3 & v_2^4 \\ 1 & v_3 & v_3^2 & v_3^3 & v_3^4 \\ 1 & v_4 & v_4^2 & v_4^3 & v_4^4 \\ 1 & v_5 & v_5^2 & v_5^3 & v_5^4 \end{bmatrix}$$

All three types of structured matrices have $MathcalO(n)$ matrix-entries, which may take on any value.

Low-rank matrices

$$A = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} \cdot \begin{bmatrix} v_1 & v_2 & v_3 & v_4 & v_5 \end{bmatrix} = \begin{bmatrix} u_1v_1 & u_1v_2 & u_1v_3 & u_1v_4 & u_1v_5 \\ u_2v_1 & u_2v_2 & u_2v_3 & u_2v_4 & u_2v_5 \\ u_3v_1 & u_3v_2 & u_3v_3 & u_3v_4 & u_3v_5 \\ u_4v_1 & u_4v_2 & u_4v_3 & u_4v_4 & u_4v_5 \\ u_5v_1 & u_5v_2 & u_5v_3 & u_5v_4 & u_5v_5 \end{bmatrix}$$

Here the 5×5 matrix A has rank one., while the 4×5 matrix B

$$B = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \\ u_{31} & u_{32} \\ u_{41} & u_{42} \end{bmatrix} \cdot \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} & v_{15} \\ v_{11} & v_{12} & v_{13} & v_{14} & v_{15} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} & B_{15} \\ B_{21} & B_{22} & B_{23} & B_{24} & B_{25} \\ B_{31} & B_{32} & B_{33} & B_{34} & B_{35} \\ B_{41} & B_{42} & B_{43} & B_{44} & B_{45} \end{bmatrix}$$

has rank 2.

Band matrices

$$B = \begin{bmatrix} b_{11} & b_{12} & & & \\ b_{21} & b_{22} & b_{23} & & \\ & b_{32} & b_{33} & b_{34} & \\ & & b_{43} & b_{44} & b_{45} \\ & & & b_{54} & b_{55} \end{bmatrix}$$

. . . and many more types of structured matrices, including block versions such as block-Toeplitz and Toeplitz-block matrices

$$B_T = \begin{bmatrix} B_1 & B_2 & B_3 & & \\ B_2 & B_1 & B_2 & B_3 & \\ B_3 & B_2 & B_1 & B_2 & B_3 \\ & B_3 & B_2 & B_1 & B_2 \\ & & B_3 & B_2 & B_1 \end{bmatrix}, \qquad T_B = \begin{bmatrix} T_{11} & T_{12} & & & \\ T_{21} & T_{22} & T_{23} & & \\ & T_{32} & T_{33} & T_{34} & \\ & & T_{43} & T_{44} & T_{45} \\ & & & T_{54} & T_{55} \end{bmatrix},$$

where T_{ij} denote a Toeplitz matrix.

Inverses of band matrices If we consider the class of band matrices such as

$$B = \begin{bmatrix} \cdot & \cdot & & & \\ \cdot & \cdot & \cdot & & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & \cdot \\ & & & \cdot & \cdot \end{bmatrix} \longrightarrow \quad B^{-1} = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

we can easily identify the $\mathcal{O}(n)$ free parameters of B. This number of free parameters does not change upon inversion of the matrix, i.e. B^{-1} still has the same $\mathcal{O}(n)$ free parameters. This is true even though the inverse of a band matrix in general is no longer a band matrix but rather a dense matrix. Just looking at a matrix, which happens to be the inverse of a band matrix does not immediately reveal this reduced number of parameters. In other words, the inverse of a band matrix is also a data sparse matrix.

Summary It is fair to say that in engineering applications we mostly encounter either dense matrices, which exhibit exploitable structure (data sparse) or we encounter sparse matrices. Most of the standard algorithms engineering students learn in the context of signal processing are efficient versions of linear algebra algorithms. For sparse matrices mathematicians and numerical analysts have developed a whole suite of specialized algorithms that allow to take advantage of the sparsity of matrices.

5 Evaluation of Numerical Computations

5.1 Evaluation Criteria

When solving a problem we are faced frequently with a choice among algorithms. On what basis should we choose? There are two often contradictory goals.

1. We would like an algorithm that is easy to understand, code, and debug.
2. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

When we are writing a program to be used once or a few times, goal (1) is most important. The cost of the programmer's time will most likely exceed by far the cost of running the program, so the cost to optimize is the cost of writing the program. When presented with a problem whose solution is to be used many times, the cost of running the program may far exceed the cost of writing it, especially, if many of the program runs are given large amounts of input. Then it is financially sound to implement a fairly complicated algorithm, provided that the resulting program will run significantly faster than a more obvious program. Even in these situations it may be wise first to implement a simple algorithm, to determine the actual benefit to be had by writing a more complicated program. In building a complex system it is often desirable to implement a simple prototype on which measurements and simulations can be performed, before committing oneself to the final design. It follows that programmers must not only be aware of ways of making programs run fast, but must know when to apply these techniques and when not to bother.

5.2 Runtime of a program

The running time of a program depends on factors such as:

1. the input to the program,
2. the quality of code generated by the compiler used to create the object program,
3. the nature and speed of the instructions on the machine used to execute the program,
4. the level of parallelism supported by the computational platform used for implementation, and
5. the time complexity of the algorithm underlying the program.

The fact that running time depends on the input tells us that the running time of a program should be defined as a function of the input. Often, the running time depends not on the exact input but only on the ßizeöf the input. A good example is the process known as sorting, which we shall discuss in Chapter 8. In a sorting problem, we are given as input a list of items to be sorted, and we are to produce as output the same items, but smallest (or largest) first. For example, given 2, 1, 3, 1, 5, 8 as input we might wish to produce 1, 1, 2, 3, 5, 8 as output. The latter list is said to be sorted smallest first. The natural size

measure for inputs to a sorting program is the number of items to be sorted, or in other words, the length of the input list. In general, the length of the input is an appropriate size measure, and we shall assume that measure of size unless we specifically state otherwise.

5.3 Numerical stability

(This section has been adopted from Nicholas Higham's Blog, See also [6]))

Numerical stability concerns how errors introduced during the execution of an algorithm affect the result. It is a property of an algorithm rather than the problem being solved. I will assume that the errors under consideration are rounding errors, but in principle the errors can be from any source.

Consider a scalar function $y = f(x)$ of a scalar x. We regard x as the input data and y as the output. The forward error of a computed approximation $\widehat{y}$ to y is the relative error $|y - \widehat{y}|/|y|$. The backward error of $\widehat{y}$ is

$$\min \left\{ \frac{|\Delta x|}{|x|} : \widehat{y} = f(x + \Delta x) \right\}.$$

If $\widehat{y}$ has a small backward error then it is the exact answer for a slightly perturbed input. Here, 'small' is interpreted relative to the floating-point arithmetic, so a small backward error means one of the form cu for a modest constant c, where u is the unit roundoff.

An algorithm that always produces a small backward error is called backward stable. In a backward stable algorithm the errors introduced during the algorithm have the same effect as a small perturbation in the data. If the backward error is the same size as any uncertainty in the data then the algorithm produces as good a result as we can expect.

If x undergoes a relative perturbation of size u then $y = f(x)$ can change by as much as $\mathrm{cond}_f(x)u$, where

$$\mathrm{cond}_c f(x) = \lim_{\epsilon \to 0} \sup_{|\Delta x| \le \epsilon |x|} \frac{|f(x + \Delta x) - f(x)|}{\epsilon |f(x)|}$$

is the condition number of f at x. An algorithm that always produces $\widehat{y}$ with a forward error of order $\mathrm{cond}_f(x)u$ is called forward stable.

The definition of $\mathrm{cond}_f(x)$ implies that a backward stable algorithm is automatically forward stable. The converse is not true. An example of an algorithm that is forward stable but not backward stable is Gauss-Jordan elimination for solving a linear system.

If $\widehat{y}$ satisfies

$$\widehat{y} + \Delta y = f(x + \Delta x), \quad |\Delta y| \le \epsilon |\widehat{y}|, \quad |\Delta x| \le \epsilon |x|,$$

with ϵ small in the sense described above, then the algorithm for computing y is mixed forward stable. Such an algorithm produces almost the right answer for a slightly perturbed input. The following diagram illustrates the previous equation.

With these definitions in hand, we can turn to the meaning of the term numerically stable. Depending on the context, numerical stability can mean that an algorithm is (a) backward stable, (b) forward stable, or (c) mixed backward-forward stable.

For some problems, backward stability is difficult or impossible to achieve, so numerical stability has meaning (b) or (c). For example, let $Z = xy'$, where x and y are n-vectors. Backward stability would

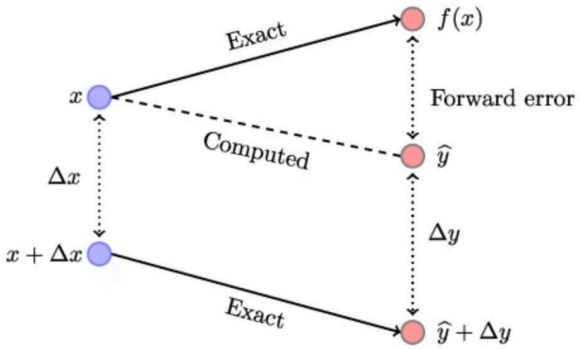

Figure 4: Forward-Backward Error

require the computed $\widehat{Z}$ to satisfy $\widehat{Z} = (x + \Delta x)(y + \Delta y)'$ for some small Δx and Δy, meaning that $\widehat{Z}$ is a rank-1 matrix. But the computed $\widehat{Z}$ contains n^2 independent rounding errors and is very unlikely to have rank 1.

5.4 Sensitivity - Condition Number

(This section has been adopted from Nicholas Higham's Blog. See also [6])

A condition number of a problem measures the sensitivity of the solution to small perturbations in the input data. The condition number depends on the problem and the input data, on the norm used to measure size, and on whether perturbations are measured in an absolute or a relative sense. The problem is defined by a function, which may be known explicitly or may be only implicitly defined (as when the problem is to solve an equation).

The most well known example of a condition number is the condition number of a nonsingular square matrix A, which is $\kappa(A) = \|A\|\|A^{-1}\|$. More correctly, this is the condition number with respect to inversion, because a relative change to A of norm ϵ can change A^{-1} by a relative amount as much as, but no more than, about $\kappa(A)\epsilon$ for small ϵ. The same quantity $\kappa(A)$ is also the condition number for a linear system $Ax = b$ (exactly if A is the data, but only approximately if both A and b are the data).

It is easy to see that $\kappa(A) \geq 1$ for any norm for which $\|I\| = 1$ (most common norms, but not the Frobenius norm, have this property) and that $\kappa(A)$ tends to infinity as A tends to singularity.

A general definition of (relative) condition number, for a function f from $\mathbb{R}^n \mapsto \mathbb{R}^n$, is

$$\text{cond}(f, x) = \lim_{\epsilon \to 0} \sup_{\|\Delta x\| \leq \epsilon \|x\|} \frac{\|f(x + \Delta x) - f(x)\|}{\epsilon \|f(x)\|}.$$

Taking a small, nonzero ϵ, we have

$$\frac{\|f(x + \Delta x) - f(x)\|}{\|f(x)\|} \lesssim \text{cond}(f, x) \frac{\|\Delta x\|}{\|x\|}$$

for small $\|\Delta x\|$, with approximate equality for some Δx.

An explicit expression for $\text{cond}(f, x)$ can be given in terms of the Jacobian matrix, $J(x) = (\partial f_i / \partial x_j)$

$$\text{cond}(f, x) = \frac{\|x\|\|J(x)\|}{\|f(x)\|}.$$

We give two examples.

If f is a scalar function then $J(x) = f'(x)$, so $\mathrm{cond}(f, x) = |xf'(x)/f(x)|$. Hence, for example, $\mathrm{cond}(\log, x) = 1/|\log x|$. If z is a simple (non-repeated) root of the polynomial $p(t) = a_n t^n + \cdots + a_1 t + a_0$ then the data is the vector of coefficients $a = [a_n, \ldots, a_0]'$. It can be shown that the condition number of the root z is, for the ∞-norm,

$$\mathrm{cond}(z, a) = \frac{\max_i |a_i| \sum_{i=0}^{n} |z|x^i}{|zp'(z)|}.$$

A problem is said to be well conditioned if the condition number is small and ill conditioned if the condition number is large. The meaning of 'small' and 'large' depends on the problem and the context. The diagram in Figure 5 illustrates a well-conditioned function f: small changes in x produce small changes in f.

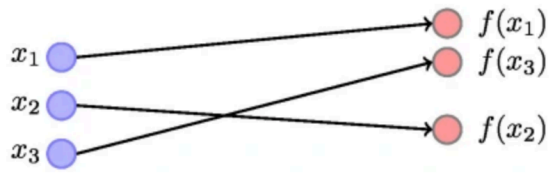

Figure 5: Well conditioned

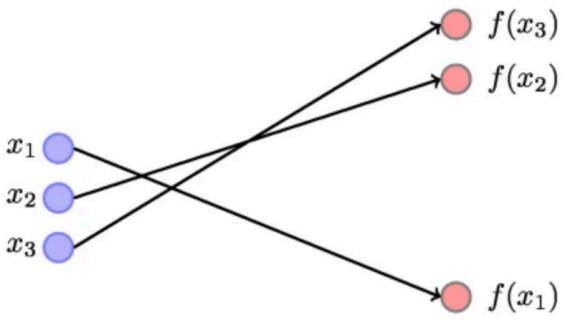

Figure 6: Ill conditioned

The diagram in Figure 6 depicts an ill-conditioned function f: small changes in x can produce large changes in f (but do not necessarily do so, as the closeness of $f(x_2)$ and $f(x_3)$ illustrates).

Here are a few key points about condition numbers.

- Even though an explicit expression may be available for it, computing $\mathrm{cond}(f, x)$ is usually as expensive as computing $f(x)$, so a lot of research has focused on obtaining inexpensive estimates of the condition number or bounds for it.
- While $\kappa(A) \geq 1$, it is not true for all functions that $\mathrm{cond}(f, x)$ is bounded below by 1.
- For a range of functions that includes the matrix inverse, matrix eigenvalues, and a root of a polynomial, it is known that the condition number is the reciprocal of the relative distance to the nearest singular problem (one with an infinite condition number).

6 Examples of Large-Scale Matrix Computations

6.1 Motion Analysis in Video

6.1.1 Description of Technical Task

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 (576×720 pixels per image) or moving on to even handle High Definition resolutions (1080×1920 pixels per image).

Optic Flow Constraint We discuss how to compute the optical flow according to the approach proposed by Horn & Schunck. 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

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

Using the shorthand notation

$$v_x = \frac{dx}{dt} \quad v_y = \frac{dy}{dt} \quad I_x = \frac{\partial I}{\partial x} \quad I_y = \frac{\partial I}{\partial y}, \quad I_t = \frac{\partial I}{\partial t},$$

the optic flow constraint can be written as

$$E_{of} = I_x \cdot v_x + I_y \cdot v_y + I_t = 0. \tag{2}$$

Equation (2) 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

$$E_{sc} = \nabla^2 v_x + \nabla^2 v_y = \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_y}{\partial x^2} + \frac{\partial^2 v_y}{\partial y^2}. \tag{3}$$

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 the filtering mask

$$\mathcal{L}(x,y) = \begin{bmatrix} 1/12 & 1/6 & 1/12 \\ 1/6 & -1 & 1/6 \\ 1/12 & 1/6 & 1/12 \end{bmatrix}. \tag{4}$$

Optimization of Euler-Lagrange Equations The Horn & Schunck approach uses the optic flow equation (2) along with the smoothness constraint (3) 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 α is a scalar regularization parameter, which controls the contribution of the smoothness constraint (3). The optimization problem finally expands into the equation

$$E^2 = (I_x v_x + I_y v_y + I_t)^2 + \alpha^2 \left((\overline{v}_x - v_x)^2 + (\overline{v}_y - v_y)^2 \right). \tag{5}$$

Applying the calculus of variations results in the following two equations

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

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

Computing Partial Derivatives When dealing with sampled or digital images, the partial derivatives showing up in equations (2) need to be calculated based on using finite differences. One simple though effective way of doing this is to compute

$$I_x \approx \frac{1}{4}\left(I[x,y+1,t] - I[x,y,t] + I[x+1,y+1,t] - I[x+1,y,t] + I[x,y+1,t+1]\right.$$
$$\left. -I[x,y,t+1] + I[x+1,y+1,t+1] - I[x+1,y,t+1]\right)$$
$$I_y \approx \frac{1}{4}\left(I[x+1,y,t] - I[x,y,t] + I[x+1,y+1,t] - I[x,y+1,t] + I[x+1,y,t+1]\right.$$

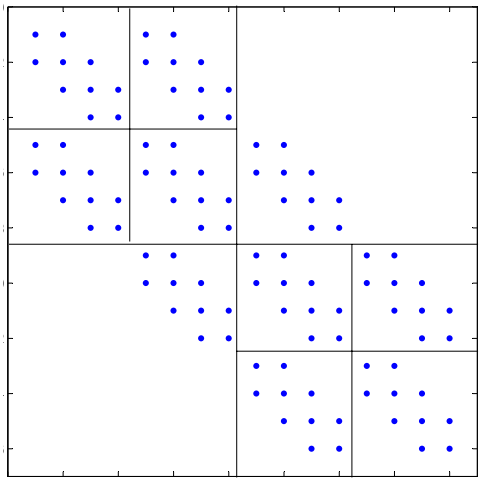

Figure 7: Structure for the Laplace Operator

$$
\begin{aligned}
&-I[x,y,t+1] + I[x+1,y+1,t+1] - I[x,y+1,t+1])\\
I_t \approx \frac{1}{4}\,&(I[x,y,t1] - I[x,y,t] + I[x+1,y,t+1] - I[x+1,y,t] + I[x,y+1,t+1]\\
&-I[x,y+1,t] + I[x+1,y+1,t+1] - I[x+1,y+1,t])\,.
\end{aligned}
$$

This amounts to compute simple differences of intensity values along the x-, the y- and the t-direction and taking the average of four neighboring values to arrive at the result.

6.1.2 Linear System of Equations

Based on (2), the optic flow equation for all pixels can be written as a matrix equation

$$
\begin{bmatrix} I_x & I_y \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \end{bmatrix} = -I_t, \tag{6}
$$

where I_x and I_y are diagonal matrices of size $N \times N$ and I_t is a vector of length N. The x- and y-components of the motion vector field are given as the vectors v_x and 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 $\mathcal{L}$, which has Toeplitz structure. We will use the symbol C to denote the negative Laplacian, i.e. $C = -\mathcal{L}$. The specific banded structure of C is depicted in Figure 7,which can be symbolically denoted as

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

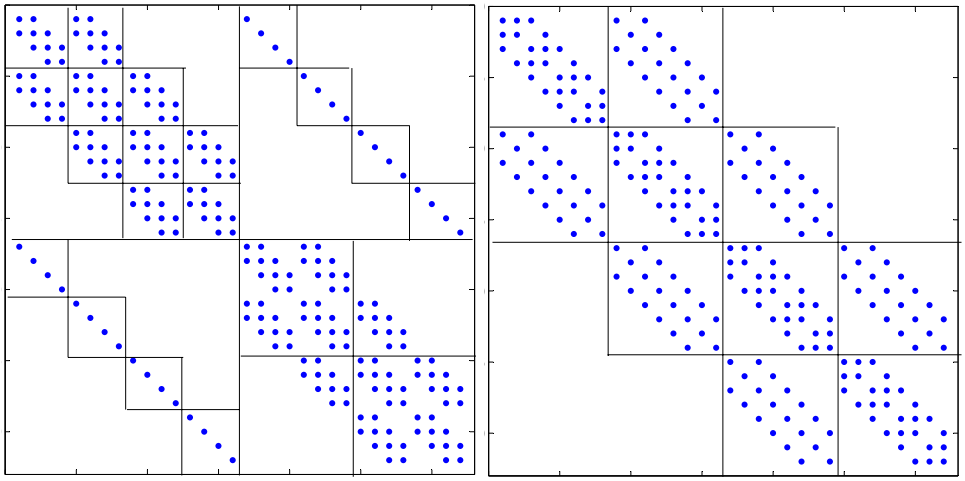

Figure 8: Structure for original (sparse) matrix (left) and re-ordered matrix (right)

The minimization of the cost function 5 for all pixels in the image leads to the set of regularized linear equations,

$$\left(\alpha^2 \begin{bmatrix} C & 0 \\ 0 & C \end{bmatrix} + \begin{bmatrix} I_x \\ I_y \end{bmatrix} \cdot \begin{bmatrix} I_x & I_y \end{bmatrix}\right) \cdot \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} I_x \\ I_y \end{bmatrix} \cdot I_t. \tag{7}$$

The term in brackets of equation (7) represents a $2N \times 2N$ band matrix, the structure of which can be seen on the left hand side of Figure 8. This structured matrix needs to be solved efficiently for computing the motion vector fields v_x and 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 8 the resulting matrix structure is shown if 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.

6.2 W-CDMA Rake Receiver

6.2.1 Problem description

We consider the problem of joint channel estimation and symbol detection in a long-code wideband code division multiple access (CDMA) system that has features of third-generation wireless. The scrambling sequences are aperiodic, data and control information may be modulated separately onto the in-phase and quadrature parts of the signal using different channelization codes with different spreading gains, pilots are often part of the control symbols, users may have different spreading gains, or multiple channelization codes may be assigned to the same user. For uplink applications, users are asynchronous, and their multipath channels may have delays longer than the symbol period. Multiple antennas may be used.

RAKE receivers are widely used in both uplink and downlink CDMA systems. If the spreading codes have good crossand auto-correlation properties, the matched filter front-end suppresses multi-access interference, and the RAKE receiver captures multi-path diversity through its diversity branches (or the RAKE fingers). For high-rate CDMA under frequency selective fading, however, code orthogonality can not be guaranteed, and the conventional RAKE receiver that uses a bank of matched filters may perform

poorly. The loss of code orthogonality has adverse effects on both channel estimation and symbol detection, and the performance degradation is especially pronounced when the network is heavily loaded and power control imperfect.

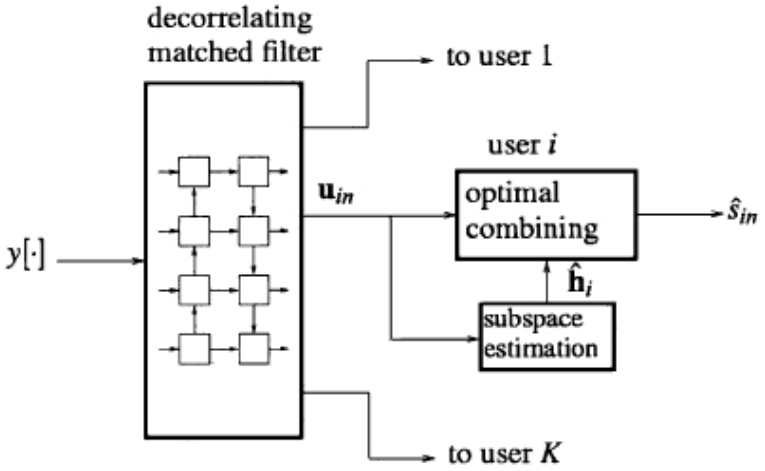

Figure 9: receiver structure. The decorrelating matched filter is implemented by an efficient matrix inversion method. Taken from [3].

In this section, we introduce the problem of a joint channel and symbol estimation scheme for RAKE receivers. As illustrated in Figure 9, a decorrelating matched filter projects the received chip-rate sequence into the signal space of each user whose channel and data sequence can be estimated jointly and independent of other users by least squares via a rank-one decomposition. The decorrelating matched filter does not depend on channel coefficients and may be pre-computed for certain applications. The scheme imposes no conditions on channel parameters and is capable of dealing with rapid multi-path fading.

The key point here is to find an efficient implementation of the decorrelating matched filter. The idea of using the decorrelating matched filter for short-code CDMA is known, but applying it to long-code CDMA presents a daunting task in terms of both computational complexity and storage requirements. A direct implementation of a ten-user system each has three multi-path fingers with a 100-symbol slot and a spreading gain of 64 amounts to inverting a code matrix of size around 6400 3000. The code matrix, fortunately, is highly structured and sparse; only 1% of its entries are nonzero. The inverse of the code matrix, however, will in general lose the structure and the sparsity.

6.2.2 Matrix Model

We introduce a matrix model for long-code CDMA. We assume that K asynchronous users transmit linearly modulated symbols. The transmission is slotted, and user i transmits M_i symbols $\{s_{in}, n = 1, 2, \ldots, M_i\}$ in each slot. The symbol sequence from user i is represented by the vector $s_i = [s_{i1}, \ldots, s_{i,M_i}]$. At the transmitter, each symbol s_{in} is spread by an aperiodic code vector c_{in} with spreading gain (length) G_i, followed by a chip pulse-shaping filter. The propagation channel of user i can be modeled by an equivalent chip-rate finite impulse response $h_{ij}, j = 0, \ldots, L_i - 1$, where h_{ij} can be viewed as the gain of the j^{th} finger of the user i's multi-path channel.

Because the channel is linear, we can first focus on symbol s_{in} from user i transmitted in the n^{th} symbol interval and set all other symbols and noise to zero. Let the received signal corresponding to symbol s_{in}

be passed through a chip-matched filter and sampled at the chip rate. All samples are put in a vector y_{in}. As shown in Figure 10, y_{in} is a linear combination of shifted (delayed) code vectors c_{in}, where c_{in} is the segment of G_i chips of user iÂ's spreading code corresponding to the n^{th} symbol. Each shifted code vector is multiplied by the j^{th} fading coefficient h_{ij}, and the channel response to s_{in} is given by

$$y_{in} = T_{in} h_i s_{in}, \quad h_i = [h_{i0}, \ldots, h_{i,L_i-1}]'.$$

Here, T_{in} is the code matrix of user and symbol [see the top part of Figure 10], and h_i is the multi-path fading channel for user i. We assume that user i has a relative delay of D_i chips with respect to the reference at the receiver. One can view that each column of T_{in} corresponds to a discrete multi-path component. For example, the first column of T_{in} is made of $(n-1)G_i + D_i$ zeros that model the relative delay of the first path with respect to the reference followed by the code vector c_{in} and additional zeros that make the size of the total number of chips of the entire slot. The second column of T_{in} models the second multi-path component similarly. Note that for sparse channels, the shifting of the code vectors does not have to be consecutive. For user i, the total received noiseless signal is given by

$$y_i = \sum_{n=1}^{M_i} T_{in} h_i s_{in} = T_i (I_{M_i} \otimes h_i) s_i$$

$$T_i = [T_{i1}, \ldots, T_{i,M_i}].$$

Matrix T_i is the code matrix of user i , and it does not depend on the gains and phases of the multi-path channel. Now including all users and the noise, we have

$$y = THs + w$$

$$T = [T_1, \ldots, T_K]$$

$$H = diag\,\{I_{M_i} \otimes h_1, \ldots, I_{M_K} \otimes h_K\}$$

where matrix H is block diagonal with $I_{M_i} \otimes h_i$ as the i^{th} block, vector s is a stacking of all symbol vectors, and w is a vector representing the additive Gaussian noise. The structure of the code matrix T is illustrated in the bottom part of Figure 10. Note that by allowing to have different sizes for different users, we include cases where variable spreading gains are used. We will impose the following assumptions.

A1 The code matrix is known.

A2 The code matrix has full column rank.

A3 The noise vector is complex Gaussian with possibly unknown .

Assumption A1) implies that the receiver knows the codes, the delay offsets D_i , and the number of channel coefficients L_i of all users. If D_i is unknown, we may set it to 0 and model all paths. L_i is a model parameter, and its choice is often left to algorithm designers. Since any channel coefficient is allowed to be zero, one can over-parameterize the channel to accommodate channel length and delay uncertainties and pay a price for the lack of modeling details. If we know that the channel is sparse, it is more efficient to model the channel as separate clusters of fingers. In that case, we assume that the approximate locations of these clusters are known.

Assumption A2) is sufficient but not necessary for the channel to be identifiable. When A2) fails, the channel may still be identifiable. In practice, one may only include a limited number of dominant interferers and significant fingers in the data model.

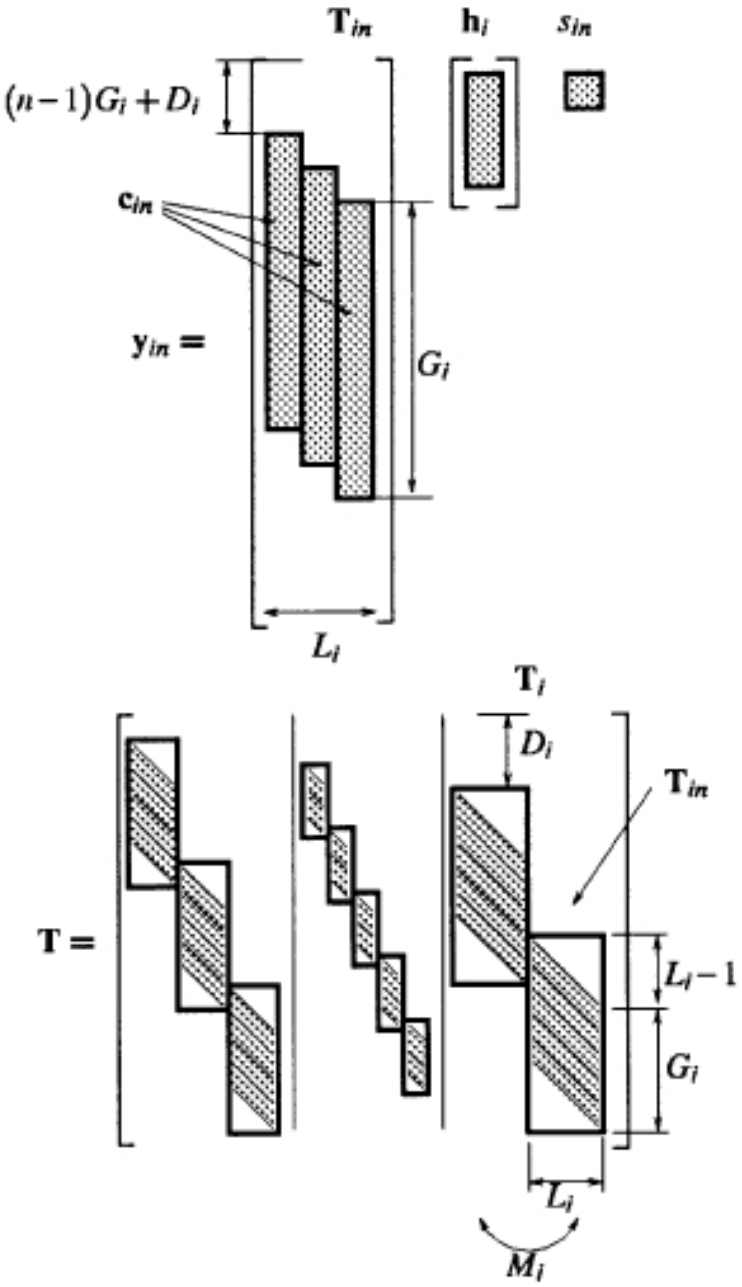

Figure 10: (Top) structure of T_{in}. (Bottom) Structure of the code matrix T. Taken from rom [3].

6.2.3 Blind Channel estimation via Least Squares

The output of the decorrelating matched filter is given by

$$u = T^{\dagger} y = diag\left\{I_{M_i} \otimes h_1, \ldots, I_{M_K} \otimes h_K\right\} s + n$$

where $n = T^{\dagger} w$ is the (colored) noise vector. Partition u into segments of length L_i with u_{in} as the $\left(\sum_{j=1}^{i-1} M_j\right) + n^{th}$ subvector. The structure of u in the equation above implies that u_{in} corresponds to symbol n of user i and satisfies

$$u_{in} = h_i s_{in} + n_{in}, \quad n = 1, \ldots, M_i.$$

Collecting all data for user i gives

$$U_i = [u_{i1}, \ldots, u_{i,M_i}] = h_i s_i' + N_i.$$

Treating h_i and s_i as deterministic parameters, we can define the least squares problem

$$\{h_i, s_i\} = Min h, s \| U_i - h s' \|_F^2$$

and estimates of h_i and s_i (with an unknown scaling factor) are found from a rank-one approximation of U_i . In other words, denoting

$$\hat{R}_i = \frac{1}{M_i} \sum_{n=1}^{M_i} u_{in} u_{in}'$$

we obtain the least squares estimates

$$\hat{h}_i = argmax_{\|g\|=1} g' \hat{R}_i g, \quad \hat{s}_{in} = \hat{h}_i' u_{in},$$

The solution $\hat{h}_i$ is given as the dominant eigenvector of $\hat{R}_i$. The scaling ambiguity in the above estimates must be removed by either incorporating prior knowledge of the symbol, using pilot symbols, or employing differential encoding of s_{in}.

6.3 Non-causal prediction for image compression

An image is modelled as a noncausal Gaussian Markov Random Field (GMRF). Basically, this says that the image is represented by a bidirectional autoregressive linear model driven by a correlated input noise. The image is defined on a $M \times N$ lattice $I(i, j)$, where i, j represent the row and column index of a pixel, respectively. We stack the intensity levels of the pixels in column i in the column vector x^i and then stack these in a NM-dimensional vector x

$$x^i = \begin{bmatrix} I(1,i) \\ I(2,i) \\ \vdots \\ I(M,i) \end{bmatrix}, \qquad x = \begin{bmatrix} x^1 \\ x^2 \\ \vdots \\ x^N \end{bmatrix}$$

Representing an image by a first-order GMRF leads to the expression

$$\hat{I}(i,j) = \beta_v (I(i-1,j) + I(i+1,j)) + \beta_h (I(i,j-1) + I(i,j+1))$$

for the prediction of the pixel value at position (i, j) and quantities β_v and β_h are the vertical and horizontal model parameters. The prediction error is defined as

$$e(i,j) = I(i,j) - \hat{I}(i,j)$$

Using vector notation and collecting the coefficients β_v and β_h in an $NM \times NM$ matrix A , the MMSE representation of the finite lattice GMRF is written as

$$Ax = e$$

where e denotes the MN-dimensional vector containing the noise samples $e(i, j)$ using the same stacking principle as we used for the image. The set of pixels belonging to to the neighbourhood $Mathcal N(p)$ of order p is indicated in the following scheme:

$$\mathcal{N}(p)_{p=1,2,\ldots,6}: \begin{array}{|ccccccc|} \hline \cdot & \cdot & \cdot & 6 & \cdot & \cdot & \cdot \\ \cdot & 5 & 4 & 3 & 4 & 5 & \cdot \\ \cdot & 4 & 2 & 1 & 2 & 4 & \cdot \\ 6 & 3 & 1 & 0 & 1 & 3 & 6 \\ \cdot & 4 & 2 & 1 & 2 & 4 & \cdot \\ \cdot & 5 & 4 & 3 & 4 & 5 & \cdot \\ \cdot & \cdot & \cdot & 6 & \cdot & \cdot & \cdot \\ \hline \end{array}.$$

For the most simple case, i.e. for $p = 1$ we can write out the corresponding matrices to be

$$A = A' = \begin{bmatrix} B_1 & C_1 & \underline{0} & \cdot & \cdot & \cdot \\ C_1 & B & C & \underline{0} & \cdot & \cdot \\ \underline{0} & C & B & C & \underline{0}\cdot & \\ \underline{0} & & \ddots & \ddots & \ddots & \underline{0} \\ \cdot & \cdot & \underline{0} & C & B & C_1 \\ \cdot & \cdot & \cdot & \underline{0} & C_1 & B_1 \end{bmatrix}$$

using the block matrix

$$B = \begin{bmatrix} 1-\beta_v & -\beta_h & 0 & \cdot & \cdot & \cdot \\ -\beta_h & 1 & -\beta_h & 0 & \cdot & \cdot \\ 0 & -\beta_h & 1 & -\beta_h & 0 & \cdot \\ 0 & & \ddots & \ddots & \ddots & 0 \\ \cdot & \cdot & 0 & -\beta_h & 1 & -\beta_h \\ \cdot & \cdot & \cdot & 0 & -\beta_h & 1-\beta_v \end{bmatrix}$$

and

$$C = \begin{bmatrix} -\beta_v & & & & & \\ & -\beta_v & & & & \\ & & -\beta_v & & & \\ & & & \ddots & & \\ & & & & -\beta_v & \end{bmatrix}.$$

The exact form for the matrices B and C also depends on how the borders of an image are treated, i.e. with boundary conditions are applied. We can choose from various alternatives such as Dirichlet or Neumann boundary conditions. Asymmetric Neumann boundary conditions amount to assuming that a

boundary pixel outside of the image has the same value as the next boundary pixel inside the image. For this boundary treatment the matrices B_1 and C_1 are determined as

$$B_1 = B + C \quad \text{and} \quad C_1 = C.$$

For higher orders of the neighbourhood, the matrices A, B and C will contain additional diagonals. In any case, the matrices involved are all sparse containing mostly zero-entries. Since A is symmetric positive definite, we can apply a block Cholesky factorization $A = U'U$ with

$$U = \begin{bmatrix} U_1 & \Theta_1 & \underline{0} & \cdot & \cdot \\ \underline{0} & U_2 & \Theta_2 & \underline{0} & \cdot \\ \cdot & \ddots & \ddots & \ddots & \cdot \\ \cdot & \cdot & \underline{0} & U_{N-1} & \Theta_{N-1} \\ \cdot & \cdot & \cdot & \underline{0} & U_N \end{bmatrix},$$

where the matrices U_i and Θ_i are upper-triangular. With this factorization we rewrite the non-causal prediction process of order $p = 1$ as

$$Ux = (U^{-1})'e = w$$

where the vector w contains uncorrelated noise samples. In detail, we compute

$$Ux = \begin{bmatrix} U_1 & \Theta_1 & \underline{0} & \cdot & \cdot \\ \underline{0} & U_2 & \Theta_2 & \underline{0} & \cdot \\ \cdot & \ddots & \ddots & \ddots & \cdot \\ \cdot & \cdot & \underline{0} & U_{N-1} & \Theta_{N-1} \\ \cdot & \cdot & \cdot & \underline{0} & U_N \end{bmatrix} \begin{bmatrix} x^1 \\ x^2 \\ x^3 \\ \vdots \\ x^N \end{bmatrix} = \begin{bmatrix} U_1x^1 + \Theta_1x^2 \\ U_2x^2 + \Theta_2x^3 \\ \vdots \\ U_{N-1}x^{N-1} + \Theta_{N-1}x^N \\ U_Nx^N \end{bmatrix}.$$

$$U'U = \begin{bmatrix} U_1' & \underline{0} & \cdot & \cdot & \cdot \\ \Theta_1' & U_2' & \underline{0} & \cdot & \cdot \\ \cdot & \ddots & \cdots & \ddots & \cdot \\ \cdot & \cdot & \Theta_{N-2}' & U_{N-1}' & \underline{0} \\ \cdot & \cdot & \cdot & \Theta_{N-1}' & U_N' \end{bmatrix} \begin{bmatrix} U_1 & \Theta_1 & \underline{0} & \cdot & \cdot \\ \underline{0} & U_2 & \Theta_2 & \cdot & \cdot \\ \cdot & \ddots & \ddots & \ddots & \cdot \\ \cdot & \cdot & \underline{0} & U_{N-1} & \Theta_{N-1} \\ \cdot & \cdot & \cdot & \underline{0} & U_N \end{bmatrix} =$$

$$= \begin{bmatrix} B_1 & C_1 & \underline{0} & \cdot & \cdot & \cdot \\ C_1 & B & C & \underline{0} & \cdot & \cdot \\ \underline{0} & C & B & C & \underline{0} & \cdot \\ \underline{0} & & \ddots & \ddots & \ddots & \underline{0} \\ \cdot & \cdot & \underline{0} & C & B & C_1 \\ \cdot & \cdot & \cdot & \underline{0} & C_1 & B_1 \end{bmatrix}$$

Based on this, we can devise a recursive scheme for computing the individual block entries U_i and Θ_i of the matrix U as

$$U_i'U_i = S_i, \qquad U_i'\Theta_i = C$$

using the Riccati-type recursion

$$S_1 = B, \qquad S_i = B - C'S_{i-1}^{-1}C, \qquad 2 \le i \le N.$$

For the purpose of image compression this scheme can be utilized to compute the prediction error vector w from a given image I. The coder will apply further compression techniques on w, such as vector

quantization and entropy coding (e.g. Huffmann codes) to determine a bit efficient representation. The decoder then needs to receive the parameters β_v and β_h along with the compressed vector w to invert the operations and reconstruct the image I. The non-causal prediction process has the purpose to compute a vector w, which requires significantly less bits to be represented than the original image I.

The decompression could be seen to just simply invert the filtering process by computing

$$A^{-1}e = x.$$

Here the issue is that while A is a sparse matrix the matrix A^{-1} will in general be a full matrix. For practical purposes this is unfeasible to determine. Numerical linear algebra offers a range of computational methods for solving linear systems of equations with a sparse coefficient matrix in an efficient manner, i.e. without explicitly determining the inverse matrix. However, these methods are iterative schemes, like Gauss-Seidl- or Jacobi-Iterations, which are not well suited for a technical implementation in hardware. The inversion of the non-causal prediction filter can be computed recursively as starting out from the filter recursion

$$w^i = U_i x^i + \Theta_i x^{i+1},$$

which we can invert to generate the backward recursion

$$x^{i+1} = -\Theta_i^{-1} U_i x^i + w^i$$

$$x^i = U_i^{-1}(w^i - \Theta_i x^{i+1}),$$

where the indexing starts with $i = N$ running backwards. Notice that Θ_i^{-1} is a full upper triangular matrix

Literatur

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

[2] P. Dewilde, K. Diepold, W. Bamberger. *Optic Flow Computation and Time.Varying System Theory*. Proc. MTNS, Leuven, 2004.

[3] L. Tong, A.J. van der Veen, P. Dewilde. *Blind decorrelating RAKE receivers for long-code WCDMA*, IEEE Trans. Signal Processing, Volume 51, Issue 6, 2003.

[4] A. Asif. J. Moura. *Image codec by noncausal prediction, residual mean removal, and cascaded VQ*. IEEE Trans. Circuits and Systems for Video Technology, Vol.6, Issue 1, 1996.

[5] N. Balram, J.M.F. Moura, *Noncausal predictive image codec*, Image Processing IEEE Transactions on, vol. 5, pp. 1229-1242, 1996, ISSN 1057-7149.

[6] N. Higham, *Accuracy and Stability of Numerical Algorithms*, Second Edition, SIAM, 2002, xxx+680 pp, hardcover, ISBN 0-89871-521-0.