TR 3106

CORDIC for High Performance Numerical Computation

Gerben Johan Hekstra

CORDIC for High Performance Numerical Computation

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College voor Promoties aangewezen,
op maandag 23 maart 1998 te 16.00 uur

door

Gerben Johan HEKSTRA

informatica ingenieur
geboren te Rotterdam, Nederland

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. Patrick M. Dewilde, Technische Universiteit Delft

ISBN: 90-9011472-6
1. Computer arithmetic. 2. CORDIC. 3. Fast rotations. 4. Floating-point computations.

Printed in The Netherlands

To my grandfather, D.W.L. Milo

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

The earliest idea that I can trace in my own mind of calculating arithmetical Tables by machinery arose in this manner:—
One evening I was sitting in the rooms of the Analytical Society, at Cambridge, my head leaning forward on the Table in a kind of dreamy mood, with a Table of logarithms lying open before me. Another member, coming into the room, and seeing me half asleep, called out, "Well, Babbage, what are you dreaming about?" to which I replied, "I am thinking that all these Tables (pointing to the logarithms) might be calculated by machinery."

Charles Babbage *in*
'Passages from the life of a philosopher'
Longman, Green, Longman, Roberts, & Green (publishers)
London, 1864

1.1 Cordic for high performance numerical computation

Volder presented his "Cordic trigonometric computing technique" in [1] in 1959 as a means to solve trigonometric relationships involving plane rotations and conversions between polar and rectangular coordinate systems. The acronym "CORDIC" stands for "COordinate Rotation DIgital Computer". The precursor to this open publication was an internal report on Cordic at Convair [2, 3], in as early as 1956, in which it is shown how it can be used to compute the transcendental functions (sine, cosine, tangent, etc.), multiplication, division, square roots, logarithms, exponents, and hyperbolic coordinate transformations. The main focus of this thesis is algorithms and architectures for Cordic and Cordic-related arithmetic techniques that perform plane rotations in an efficient manner. We tailor these to suit their application to problems that require high performance numerical computations. The term "high performance" is used here in a dual context: that of very high speed, high throughput and low latency of the computations, and/or that of very high accuracy and large dynamic range of the data. Traditional Cordic implementations suffer from serious limitations in exactly this high performance context. Unless remedied, this makes Cordic less suited as a high performance numerical processor building block in computing hardware. The contribution of this thesis is twofold, focusing on the above limitations. On the one hand, we present a full floating-point Cordic, that overcomes the limitations in high accuracy and large dynamic range. On the other hand, we present a Cordic related technique that we have called "Fast rotation". This technique overcomes the limitations in high speed, high throughput and especially in low latency, by implementing rotations at a lower cost, without trading in numerical accuracy or dynamic range. In this thesis, we pay particular attention to fast rotations. We show their application to a number of diverse problems and show the advantage gained over traditional Cordic or other arithmetic techniques.

1.2 A short history of the Cordic

The earliest application of Cordic was that of solving trigonometric relationships of navigation equations, as appear in the problem of great circle navigation, at a high rate. Volder reports in [1, 2] of a special-purpose prototype computer, the "CORDIC I", being designed and constructed by Convair, especially for this purpose. The ultimate goal was to replace the Sperry navigation computer —which was based on analog resolvers— in the Convair B-58 transcontinental bomber airplanes, by an all-digital navigation computer; the CORDIC I. In this sense, the Cordic can be seen as the digital equivalent of the analog resolver. For navigation, the aircraft had a "stable platform" configuration. Like a gyroscope, a stable platform is suspended with gimbals, and keeps the same fixed orientation in space at all time. The three angles that determine the orientation of the aircraft, relative to the stable platform, were read out directly in digital form by means of digital rotary shaft encoders. With this information, and with the speed computed from inertial sensors, the CORDIC I would compute the current location of the aircraft. The incentive to develop the Cordic, was that the accuracy of the analog resolvers was not good enough, and with Cordic, being a digital technique, you could theoretically go to any accuracy you wanted. Only a single prototype of the CORDIC I was implemented [4]. Its successor, the CORDIC II [4] was fully transistorized. The logic functions were implemented on card-level: one flip-flop or other logical gate was put on one printed circuit board. The memory was implemented using a rotating magnetic drum. The total machine would take about 5 cubic feet of space [3, 5, 4].

Walther, in his seminal paper [6] in 1972, unified the Cordic algorithm for the circular, linear, and hyperbolic coordinate systems[1] and presented the computational

[1]The Cordic algorithms for hyperbolic rotation and computation of exponents and logarithms were originally discovered by Volder, as published in [2].

schemes to compute the elementary functions, using only Cordic arithmetic. Aside from this, he laid the foundation for Cordic arithmetic for when the arguments, both vector *and* angle, are in floating-point number representation. The applications mentioned by Walther are those in the field of computer arithmetic, as the computation of the elementary functions in some of the earliest scientific desk calculators, such as the Hewlett-Packard HP-9100, and in a special purpose floating-point numerical co-processor to the HP-2116 computer[2]. Also the first hand-held calculators, such as the HP-35, were equipped with the Cordic algorithm [9], albeit a decimal arithmetic version of the algorithm.

As (V)LSI area became more readily available, and integration densities grew, implementations of the Cordic transitioned from a discrete-component implementation [1, 3] via micro-programmed architectures [10, 6] to single-chip implementations [11] in 1980. The first single-chip implementations, such as the one by Haviland and Tuszynski [11], were necessarily sequential (word-serial or even bit-serial) by nature. Parallel, pipelined, implementations followed, such as by Deprettere et al. [12], and the commercially available TMC2330 and TMC2340 chips [13, 14] from TRW.

As mentioned above, the initial fields of application of the Cordic were that of navigation and arithmetic. To date it is widely applied in the fields of computer arithmetic itself, numerical analysis (matrix algebra), computer graphics, but most notably in the field of digital signal processing. As early as 1974, Despain presented in [15] an application to compute the Fast Fourier Transform (FFT) employing Cordic arithmetic. Digital synthesis of waveforms (modulation and demodulation) [14, 16] is another main application.

1.3 Advantages and disadvantages

Without doubt, we can state that the Cordic algorithm is versatile when we look at the large and diverse group[3] of functions that can be implemented with it. From the same point of view, we also argue that it is very efficient in implementation, requiring little hardware. From a viewpoint of numerical stability in computations, Cordic is attractive as it guarantees orthonormality and robust computation (no catastrophic exceptions) at the level of the individual operations.

However, when looking at a single specific function only, more efficient implementations exist, as is the case for multiplication, division, square root, exponentiation, and taking logarithms. When only one, or a few of the functions are required it may well pay off to select other, more specific and less versatile, algorithms than

[2]The Cordic algorithm has even been applied in the INTEL line of numeric co-processors up to the 80387 [7, 8].

[3]In Appendix 1.A, we present a number of schemes that compute commonly used functions in Cordic arithmetic.

Cordic. The same reasoning holds true for actual Cordic implementations: almost none of these incorporate both the rotation and vectoring modes, *and* work in all three coordinate systems (circular, hyperbolic, and linear), unless this diversity is required by the application. In practice, only a selection of these operation modes is sufficient.

Still, the cost of an already "stripped" Cordic implementation can be prohibitive. For some applications, alternatives based on non-Cordic arithmetic might prove advantageous. As an illustration, we take the application of QR-decomposition based Recursive Least Squares (QRD-RLS) and refer the reader to [17]. We would consider this as a good candidate for an implementation with Cordic arithmetic, as the basic operations are embedded 2×2 Givens rotations, which match to the Cordic's vectoring and rotation modes for the circular coordinate system. Ercegovac and Lang, in [18], purposefully avoid the use of Cordic, and present an implementation which uses separate multipliers, dividers, and square-rooters. Whereas it is likely to have a better speed performance, the operations are no longer guaranteed to be orthonormal, and there is a loss of robustness in the operations due to the possibility of a divide-by-zero exception. Taking things a step further, Gentleman, in [19], reformulates the algorithm, to arrive at a square-root-free version. This lowers the number and the diversity of the operations, and makes it more attractive to implement with more "conventional" multiply-add and divide arithmetic. Similarly, Parhi et al., in [20, 21], present an alternative square-root free version, which permits higher factors of pipelining, at the cost of giving up orthogonality.

Other than just the prohibitive cost factor, there are other factors that influence the choice whether or not to choose other techniques over Cordic. We sum these up below.

- **Restricted scalability.** Regarding the large amount of literature on Cordic implementations, only a few architectural alternatives appear. These are the *bit-serial* [1], the *on-line* [22, 23, 24], the *word-serial* [11], and the *word-parallel* [12] architectures. There are to our knowledge no intermediate forms. This greatly limits the choice of area-time trade-off in implementation. This becomes an essentially serious problem as the word-size increases. On the one side there are the small but slow sequential architectures, and on the other side the fast, but large, parallel architectures. An architecture for Cordic which lies in between the word-serial and word-parallel alternatives, as is commonplace for multipliers and dividers, is only hinted upon by a few researchers, but is not considered a serious, cost effective alternative.

- **Limited absolute accuracy, low dynamic range.** The classical Cordic algorithm, as given by Volder, employs a fixed-point representation for the angle of rotation, which implies a certain absolute accuracy. If the vector data *is* in a floating-point representation, the absolute accuracy affects the precision of computations, and may even completely mask the results itself. This argument is not entirely true, as Cordic algorithms with a floating-point representation of the angle *do* exist [6, 25, 26], but are significantly more complex.

- **High latency, low throughput** The Cordic algorithm typically belongs to the class of digit-recurrence [27] algorithms, like division and square-root. At every step, a decision is made, based on results of the previous step. Schemes to reduce the latency using feed-forward techniques, such as exist for the Wallace multiplier [28], do not exist for Cordic.

1.4 Focus of this thesis

This thesis focuses on what we see as the two major disadvantages of the Cordic, namely the problem of the limited accuracy and low dynamic range, and the problem of a relatively high cost of implementation combined with the lack of scalability. As our solution to the former problem, we present the floating-point Cordic arithmetic in Chapter 2. As we mentioned earlier, floating-point Cordic arithmetic was already founded by Walther [6] in as early as 1971, but his solution is suitable only for efficient implementation as a sequential architecture, and is practically unworkable for a parallel pipelined architecture. Our approach results in efficient implementations for both types of architecture.

As for the latter of the problems, we present "fast rotation" techniques in Chapter 3 to intend to solve the issues of cost and scalability. Fast rotations [29], also called orthonormal micro-rotations [30], are closely related to Cordic arithmetic and perform orthonormal rotation over a certain angle at a very low cost in implementation. Although they only exist for certain "nice" angles only, this set can be made sufficiently large by a whole range of techniques. One can say that what fast rotations are for Cordic is the analog of what Canonic Signed Digit (CSD) optimization [31], and the work of Magenheimer et al. [32], are for multiplication. The lower cost of implementation also result in an increase of throughput and a reduction of the latency.

We have chosen to incorporate *both* topics together in this thesis, as they are closely related, and fit in the frame of a "bigger picture".

1.5 Pre-requisites and Conventions

We briefly describe some of the conventions that we adopt for this thesis.

1.5.1 Rotations

Unless otherwise specified, when we speak about a rotation we inherently assume a circular rotation in 2-D space. In this thesis, we treat hyperbolic rotations too, and will explicitly mention when this is the case.

We define a rotation over the angle α as given by the 2×2 rotation matrix **R** in:

$$\mathbf{R} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}. \tag{1.1}$$

The rotation adheres to the convention that a rotation is counterclockwise (CCW) for positive angles α, and clockwise (CW) for negative angles.

1.5.2 Micro-rotations

Let us define a cosine/sine approximation pair as a tuple (c, s) in which s plays the role of the sine and c that of the cosine. In the remainder of this thesis, we speak simply of an **approximation pair** when we refer to a cosine/sine approximation pair. We impose the constraint that the approximation pair c, s must be exact representable in the number system which we use for the computations.

We define the magnification factor m of an approximation pair as:

$$m = \sqrt{c^2 + s^2}. \tag{1.2}$$

Note that the magnification factor is the 2-norm of the vector with components c and s.

We define the angle α of the approximation pair as the angle between the positive axis and the vector $[c\,s]^T$, measured in a positive sense (counterclockwise). The relationship between the approximation pair (c, s), magnification factor m and angle α is given below in Equation (1.3).

$$\begin{aligned} c &= m\cos(\alpha) \\ s &= m\sin(\alpha) \end{aligned} \tag{1.3}$$

If $c > 0$, then the following formula for the angle α is valid.

$$\alpha = \arctan(\frac{s}{c}) \tag{1.4}$$

We define a generalized micro-rotation over the angle α as given by the 2×2 matrix **F** in:

$$\mathbf{F} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}. \tag{1.5}$$

Often, we impose the extra constraint that the implementation of **F** is cheap in terms of computational/hardware complexity. This constraint restricts the possible values of α for the angle of rotation. The micro-rotation need not be orthonormal. Substituting Equation (1.3) for the approximation pair in Equation (1.5) results in the alternative formulation of **F** in:

$$\mathbf{F} = m \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} . \tag{1.6}$$

Let $\mathbf{F}_i$ be generalized micro-rotations with corresponding approximation pairs $c_i.s_i$, magnification factors m_i and angles of rotation α_i.

Let **P** be the product of two generalized micro-rotations $\mathbf{F}_1$ and $\mathbf{F}_2$, as in:

$$\mathbf{P} = \mathbf{F}_2 \cdot \mathbf{F}_1 . \tag{1.7}$$

Substituting Equation (1.5) in the above, we can write out the product **P** in terms of the approximation pairs as:

$$\begin{aligned} \mathbf{P} &= \mathbf{F}_2 \cdot \mathbf{F}_1 \\ &= \begin{bmatrix} c_2 & -s_2 \\ s_2 & c_2 \end{bmatrix} \begin{bmatrix} c_1 & -s_1 \\ s_1 & c_1 \end{bmatrix} \\ &= \begin{bmatrix} c_1c_2 - s_1s_2 & -s_1c_2 - c_1s_2 \\ s_1c_2 + c_1s_2 & c_1c_2 - s_1s_2 \end{bmatrix} . \end{aligned} \tag{1.8}$$

Alternatively, we can write out this product, substituting the $\mathbf{F}_i$ according to Equation (1.6), resulting in:

$$\begin{aligned} \mathbf{P} &= \mathbf{F}_2 \cdot \mathbf{F}_1 \\ &= m_1 \begin{bmatrix} \cos(\alpha_1) & -\sin(\alpha_1) \\ \sin(\alpha_1) & \cos(\alpha_1) \end{bmatrix} \cdot m_2 \begin{bmatrix} \cos(\alpha_2) & -\sin(\alpha_2) \\ \sin(\alpha_2) & \cos(\alpha_2) \end{bmatrix} \\ &= m_1m_2 \begin{bmatrix} \cos(\alpha_1 + \alpha_2) & -\sin(\alpha_1 + \alpha_2) \\ \sin(\alpha_1 + \alpha_2) & \cos(\alpha_1 + \alpha_2) \end{bmatrix} . \end{aligned} \tag{1.9}$$

Referring to the results in Equations (1.8) and (1.9), we can state the following about the product:

1. The product is commutative, that is $\mathbf{P} = \mathbf{F}_2 \cdot \mathbf{F}_1 = \mathbf{F}_1 \cdot \mathbf{F}_2$. This is easy to prove, given the alternative representation of Equation (1.6), and the fact that rotations are commutative.

2. The matrix $\mathbf{P}$ has the same structure as a generalized micro-rotation, and is given by:

$$\mathbf{P} = \begin{bmatrix} c_{1,2} & -s_{1,2} \\ s_{1,2} & c_{1,2} \end{bmatrix}, \tag{1.10}$$

where the coefficients $c_{1,2}, s_{1,2}$ follow as:

$$\begin{aligned} c_{1,2} &= c_1c_2 - s_1s_2 \\ s_{1,2} &= c_1s_2 + s_1c_2 \;. \end{aligned} \tag{1.11}$$

3. The overall angle of rotation $\alpha_{1,2}$, as effected by application of $\mathbf{P}$, is given as the sum of the individual angles of rotation α_i in:

$$\alpha_{1,2} = \alpha_1 + \alpha_2 \,. \tag{1.12}$$

4. The overall magnification factor $m_{1,2}$ of the product $\mathbf{P}$ is given by the product of the individual magnification factors m_i in:

$$m_{1,2} = m_1 m_2 \,. \tag{1.13}$$

Note that $\mathbf{P}$ is not necessarily a generalized micro-rotation by itself, since there is no inherent guarantee that the $c_{1,2}, s_{1,2}$ are exact representable in the number system.

1.5.3 Realization of a rotation with micro-rotations

The Cordic algorithm relies on the property that a rotation over the angle α can be decomposed into a sequence of rotations over the angles α_i. Hence we can approximate the angle of rotation α by $\tilde{\alpha}$ with an error ε_α, as given by:

$$\alpha = \tilde{\alpha} + \varepsilon_\alpha \,, \tag{1.14}$$

where the approximation $\tilde{\alpha}$ is given the sum of N angles α_i

$$\tilde{\alpha} = \sum_{i=1}^{N} \alpha_i \,. \tag{1.15}$$

As for now, we do not put any restrictions on the choice of the angles α_i nor on the number N of them, nor on the error in approximation ε_α. Likewise, we define $\tilde{\mathbf{R}}$ and $\mathbf{E}_\alpha$ to be rotation matrices that perform rotations over $\tilde{\alpha}$ and ε_α respectively, $\tilde{\mathbf{R}}$ and $\mathbf{E}_\alpha$ being of the form of Equation (1.1). Conform the approximation of the angle in Equation (1.14), we can express the rotation $\mathbf{R}$ as:

$$\mathbf{R} = \tilde{\mathbf{R}}\mathbf{E}_\alpha \,. \tag{1.16}$$

We implement the rotations over α_i by means of generalized micro-rotations $\mathbf{F}_i$, each with approximation pair (c_i, s_i), and magnification factor m_i. Let us define $\mathbf{F}$ as the sequence of the N micro-rotations $\mathbf{F}_i$, as given by:

$$\mathbf{F} = \mathbf{F}_N \cdots \mathbf{F}_2 \mathbf{F}_1 = \prod_{i=1}^{N} \mathbf{F}_i . \tag{1.17}$$

Based on the results gathered in Equation (1.9), we state that $\mathbf{F}$ performs a rotation and magnification, where the overall angle of rotation is equal to $\tilde{\alpha}$ as in Equation (1.15), while the overall magnification factor K introduced is given by[4]:

$$K = \prod_{i=1}^{N} m_i . \tag{1.18}$$

Combining the above, we can write $\mathbf{F}$ as the product of the magnification factor K and the rotation $\tilde{\mathbf{R}}$, as in:

$$\mathbf{F} = K\tilde{\mathbf{R}} . \tag{1.19}$$

We can now make a prior approximation of the rotation $\mathbf{R}$ by the sequence of micro-rotations $\mathbf{F}$ in:

$$\mathbf{R} \approx \tilde{\mathbf{R}} = K^{-1}\mathbf{F} . \tag{1.20}$$

Note that we have to perform either a division with K or a multiplication with its (pre-calculated) inverse, which is cheaper in implementation. In finite wordlength computation, we approximate the inverse of the magnification factor K^{-1} by $\acute{K}$, such that:

$$K\acute{K} = 1 + \varepsilon_K . \tag{1.21}$$

where ε_K is the error in approximation, and is designed to be well below the error in the representation of numbers. In many practical Cordic implementations, the magnification factor is constant and known beforehand, and an efficient implementation of the multiplication with $\acute{K}$ can be constructed. We show examples of this in Appendix 1.B.

We approximate the rotation $\mathbf{R}$ in both the angle of rotation and the magnification factor, as the product of the sequence of micro-rotations $\mathbf{F}$ and the approximate inverse magnification factor $\acute{K}$:

$$\mathbf{R} \approx \tilde{\mathbf{R}} = K^{-1}\mathbf{F} \approx \acute{K}\mathbf{F} . \tag{1.22}$$

[4]We prefer to use the symbol K instead of m here to be consistent with general Cordic literature

The entire operation is **contractive** if $||\mathbf{v}'|| \leq ||\mathbf{v}||$, which implies $\varepsilon_K \leq 0$. The errors introduced in this approximation follow from the successive substitution of Equations (1.19), (1.21) and (1.16) in the approximation of Equation (1.22).

$$\begin{aligned} \dot{K}\mathbf{F} &= \dot{K}K\tilde{\mathbf{R}} \\ &= (1+\varepsilon_K)\tilde{\mathbf{R}} \\ &= (1+\varepsilon_K)\mathbf{E}_\alpha^{-1}\mathbf{R} \\ &= \mathbf{ER} \quad , \end{aligned} \tag{1.23}$$

where the error matrix $\mathbf{E}$ is given by:

$$\begin{aligned} \mathbf{E} &= (1+\varepsilon_K)\mathbf{E}_\alpha^{-1} \\ &= \begin{bmatrix} (1+\varepsilon_K)\cos(\varepsilon_\alpha) & (1+\varepsilon_K)\sin(\varepsilon_\alpha) \\ -(1+\varepsilon_K)\sin(\varepsilon_\alpha) & (1+\varepsilon_K)\cos(\varepsilon_\alpha) \end{bmatrix} . \end{aligned} \tag{1.24}$$

For small $\varepsilon_\alpha, \varepsilon_K$, this matrix approaches the identity matrix.

1.6 Organization of this thesis

The organization of this thesis is as follows: In Chapter 2, we address the problem of the poor relative accuracy the Cordic algorithm. We present a full floating-point Cordic algorithm as the solution to this problem. This is when, apart from the input data, the angle is in also a floating-point representation. In this chapter, we mainly follow the work on floating-point Cordic as presented by Hekstra and Deprettere in [25, 33, 34, 26]. We present the background, algorithm, and two architectures: a sequential one and a parallel one, for floating-point Cordic.

In Chapter 3, we address the problem of the relatively high cost of computation of the Cordic algorithm. We introduce the concept of fast rotations: related to Cordic arithmetic, these are methods to rotate over (a set of fixed) angles, but at a much lower cost. In this chapter, we follow the original work on fast rotations as presented by Hekstra in [29], and of Hekstra and Götze in [30], and in recent publications such as in [35]. We present the background, different methods, ways to implement, and properties of fast rotations. This chapter serves as the reference in fast rotation methods.

Referring to Figure 1.1, these first two Chapters 2 and 3 can be read independently from each other. The lines in the diagram indicate dependencies between the chapters. The remaining of the thesis focuses on the application of fast rotations, and depends on Chapter 3.

The topic of fast rotations as an alternative to Cordic has become quite popular in a short time, judging from the number of recent papers [29, 36, 37, 30, 38, 39, 40,

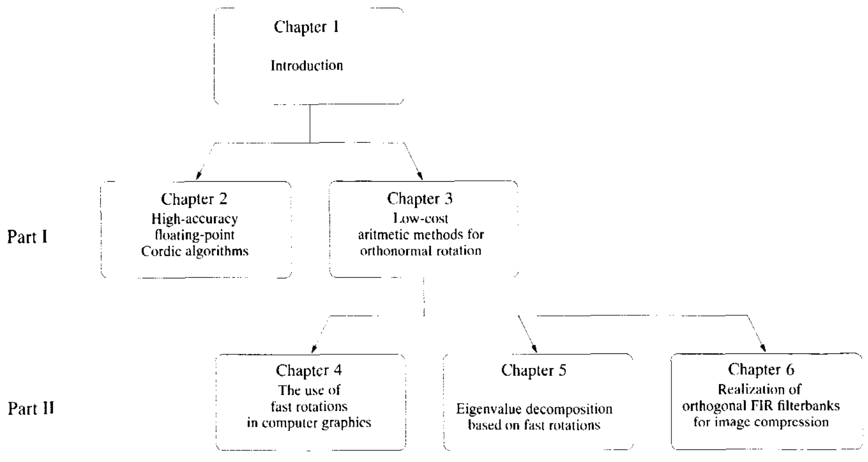

Chapter 1
Introduction
Part I
Chapter 2
High-accuracy floating-point Cordic algorithms
Chapter 3
Low-cost aritmetic methods for orthonormal rotation
Part II
Chapter 4
The use of fast rotations in computer graphics
Chapter 5
Eigenvalue decomposition based on fast rotations
Chapter 6
Realization of orthogonal FIR filterbanks for image compression

Figure 1.1. How to read this thesis.

35, 41, 42, 43] and diversity of applications. Hence, in this thesis, we pay particular attention to a number of applications from diverse fields in Chapters 4 to 6.

Looking at the history of the development of fast rotations, we see two parallel developments, both starting around the same time. One by the author himself [29], and one by Jürgen Götze et al. in [36]. In the former, the basis is laid for what we call the set of maximal fast rotations. The latter employs what they call a double rotation and scaling method, which is a special form of what we call factored fast rotations. Both came to be, purely out of the necessity to greatly reduce the computational complexity of certain operations within a specific application. Hence, we start with these two specific applications in Chapters 4 and 5 that have led to the development of fast rotations.

In Chapter 4, we show how fast rotations are applied in the various related problems of rendering of photo-realistic images using radiosity and ray-tracing. The general field of this application is that of computer graphics. It is also the application that has been the incentive behind the development of fast rotation methods. In this chapter, we follow the work previously presented in many of the internal documents of the Radiosity Engine project [44, 45, 29, 46, 47, 48, 49]. We present the background and details of two computation-intensive problems in radiosity rendering: intersection computation and cell traversal. For both we present an algorithm that relies heavily on the use of fast rotations, and show that at least an order of magnitude reduction of the computational complexity is gained, with respect to a solution with conventional arithmetic.

In Chapter 5, we show how fast rotations are applied to the problem of eigenvalue decomposition of symmetric matrices. This is a typical application in the field of matrix algebra. Again, almost an order of reduction in computational complexity is achieved, with respect to known solutions based on Cordic arithmetic. In this chapter, we mainly follow the work as presented in [36, 37, 30, 38], making a few improvements here and there. We provide the background on the problem of eigenvalue decomposition(EVD), the Jacobi algorithm to solve the EVD, and present our solution, based on fast rotations.

In Chapters 6 we present an important application of fast rotations in the field of digital signal processing: the high-quality compression and reconstruction of medical images. The compression relies on transform coding, using a Lapped Orthogonal Transform (LOT) [50] filterbank, to achieve the required compression ratios, while maintaining the quality. In this chapter, we rely in a large part on the work of Heusdens [51], for the background on transform coding of images, and focus more on the implementation of the FIR filterbank of the LOT using fast rotations. For this we follow the work which was presented separately by Hekstra et al. in [52, 40] and [39] and combine this into a whole. We present a programmable architecture and corresponding VLSI chip: the Parallel TransForm Engine (PTFE), that is capable of implementing the LOT transform. The programmability of the architecture provides for a flexibility to implement any other commonly used transform. We show that through the use of fast rotations combined with other techniques, the computational complexity is greatly reduced, and that the implementation has many desirable properties.

Appendix

1.A Examples of Cordic arithmetic

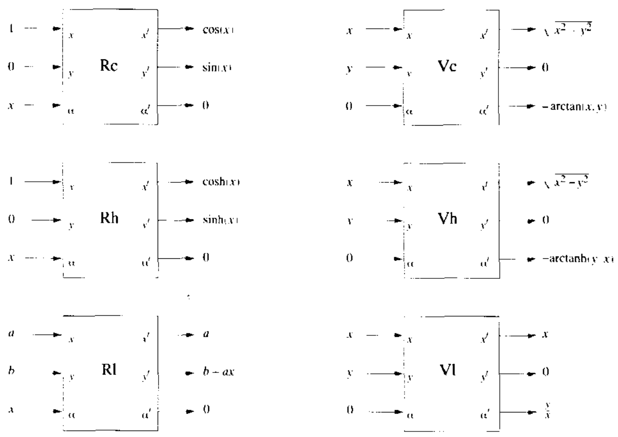


Figure 1.2. Functions evaluated by Cordic for the different modes.

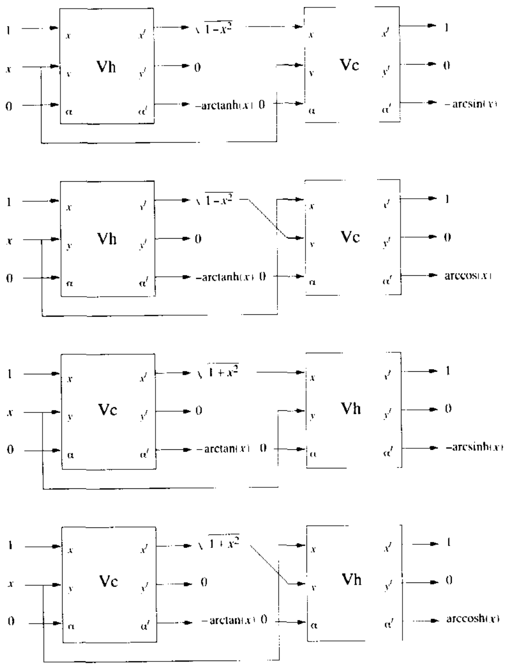


Figure 1.3. Cordic schemes for the evaluation of the inverse trigonometric functions.

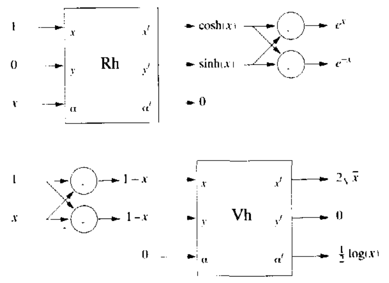


Figure 1.4. Cordic schemes for the evaluation of exponents, logarithms and square roots.

1.B Efficient implementation schemes for the inverse magnification factor

N_{mant}	scaling sequence	length
4...5	$2^{-1}(1+2^{-2})$	2
6...8	$2^{-1}(1+2^{-2})(1-2^{-5})$	3
9...11	$(1-2^{-2})(1-2^{-2})(1+2^{-4})(1+2^{-6})$	4
12...16	$2^{-1}(1+2^{-2})(1-2^{-5})(1+2^{-9})(1+2^{-10})$	5
17...23	$2^{-1}(1+2^{-2})(1-2^{-5})(1+2^{-9})(1+2^{-10})(1+2^{-16})$	6
24...27	$2^{-1}(1+2^{-2})(1-2^{-5})(1+2^{-9})(1+2^{-10})(1+2^{-16})(1-2^{-23})$	7
28	$(1-2^{-2})(1-2^{-2})(1+2^{-4})(1+2^{-6})(1+2^{-11})(1-2^{-14})(1-2^{-19})(1+2^{-22})$	8
29...31	$2^{-1}(1+2^{-2})(1-2^{-5})(1+2^{-9})(1+2^{-10})(1+2^{-16})(1-2^{-23})(1+2^{-28})$	8
32...34	$2^{-1}(1+2^{-2})(1-2^{-5})(1+2^{-9})(1+2^{-10})(1+2^{-16})(1-2^{-23})(1+2^{-28})(1+2^{-31})$	9

Table 1.1. Low-cost factored scaling sequences for a given wordlength.

Bibliography

[1] Jack. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on electronic computers*, pp. 330–334, Sept. 1959.

[2] Jack. E. Volder, "Binary computation algorithms for coordinate rotation and function generation," Tech. Rep. Convair report IAR-1 148 Aeroelectronics group, Convair, June 1956, A copy of the report is in possession by the author.

[3] Jack E. Volder, "About the first Cordic and the IAR-1 148 report," private communication, August 1997.

[4] Advanced Systems Group, "Technical description of fixtaking tie-in equipment," Tech. Rep. Aerosystems Report FZE-052, Advanced Systems Group, General Dynamics / Fort Worth, Aug. 1962, A copy of the report is in possession by the author.

[5] Bernhard Kuchta, "About application and (non-)continuation of Cordic in navigation computers, the fate of Convair, analog resolvers, the size of the CORDIC-1 processor.," private communication, August 1997.

[6] J.S Walther, "A unified algorithm for elementary functions," *proceedings of the AFIPS Spring Joint Computer Conference*, pp. 379–385, 1971.

[7] Jean-Michel Muller, "About the application of Cordic in Intel 'x87 co-processors," private communication, July 1997.

[8] Jean-Michel Muller, *Elementary Functions, Algorithms and Implementation*, Birkhäuser, 1997.

[9] David S. Cochran, "Algorithms and accuracy in the HP-35," *Hewlett-Packard Journal*, pp. 10–11, June 1972.

[10] M. A. Liccardo, "An interconnect processor with emphasis on Cordic mode operation," M.S. thesis, University of California at Berkeley, Sept. 1968.

[11] Gene L. Haviland and Al A. Tuszynski, "A CORDIC arithmetic processor chip," *IEEE journal of solid-state circuits*, vol. SC-15, no. 1, pp. 4–14, Feb. 1980.

[12] A.A.J. de Lange, A.J. van der Hoeven, E.F. Deprettere, and J. Bu, "An optimal floating-point pipeline CMOS Cordic processor," in *IEEE International Symposium on Circuits and Systems.*, June 1988.

[13] TRW, TRW LSI Products Inc., PO Box 2472, La Jolla, CA 92038, *TMC2330 CMOS Coordinate Transformer 16x16-bit, 25MOPS Advance Information*, July 1989.

[14] TRW, TRW LSI Products Inc., PO Box 2472, La Jolla, CA 92038, *Digital Synthesizer, Dual 16-bit, 25Mops*, 1990.

[15] Alvin M. Despain, "Fourier transform computers using CORDIC iterations," *IEEE Transactions on Computers*, vol. c-23, no. 10, pp. 993–1001, Oct. 1974.

[16] Jos Huisken, "About the application of Cordic in the DAB chip," private communication, August 1997.

[17] W.M. Gentleman and H.T. Kung, "Matrix triangularization by systolic arrays," in *Proceedings of the SPIE, Real-Time Signal Processing IV*, 1981, pp. 298–303.

[18] Miloš D. Ercegovac and Tomás Lang, "On-line scheme for computing rotation factors," *Journal of Parallel and distributed computing*, vol. 5, pp. 209–227, 1988, Year and volume unknown.

[19] W.M. Gentleman, "Least-squares computations by Givens transformations without square-roots," *Journal Inst. Math Applics.*, vol. 12, pp. 329–336, 1973.

[20] K.J. Raghunath and K. K. Parhi, "High-speed RLS using scaled tangent rotations (STAR)," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '93)*, July 1993, pp. 1959–1962.

[21] K.J. Raghunath and K.K. Parhi, "A 100MHz pipelined RLS adaptive filter," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 1995, pp. 3187–3190.

[22] M.D. Ercegovac and T. Lang, "Implementation of an svd processor using redundant cordic," in *Advanced Algorithms and Architectures for Signal Processing III*. SPIE, 1988, vol. 975, pp. 300–313.

[23] M.D. Ercegovac and T. Lang, "Implementation of fast angle calculation and rotation using on-line cordic," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1988, pp. 2703–2706.

[24] Hai Xiang Lin and Henk J. Sips, "On-line CORDIC algorithms," *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1039–1052, Aug. 1990.

[25] Gerben J. Hekstra and Ed F. Deprettere, "Floating point Cordic," in *Proceedings ARITH 11*, Windsor, Ontario, June 30 - July 2, 1993 1993.

[26] G.J. Hekstra and E.F.A. Deprettere, "Cordic algorithms and architectures," European Patent Office, Bulletin 95/01, January 1995, Publication number: 0 632 369 A1, date of filing: June 26, 1993.

[27] Miloš D. Ercegovac and Tomás Lang, *Division and Square root, Digit-recurrence algorithms and implementations*, Kluwer Academic Publishers, 1994.

[28] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE transactions on electronic computers*, pp. pp. 14–17, February 1964.

[29] Gerben Hekstra, "Definition and structure of hemisphere and viewing rays," Tech. Rep. ET/NT/Radio-9, TU Delft, November 1993.

[30] J. Götze and G. Hekstra, "An algorithm and architecture based on orthonormal μ-rotations for computing the symmetric EVD," *Integration, the VLSI Journal*, vol. 20, pp. 21–39, 1995.

[31] R.W. Reitwiesner, "Binary Arithmetic," *Advances in Computers*, vol. 1, pp. 231–308, 1966.

[32] Daniel A. Magenheimer, Liz Peters, Karl W. Petis, and Dan Zuras, "Integer multiplication and division on the HP precision architecture," *IEEE transactions on computers*, vol. vol. 37, pp. pp. 980–990, August 1988.

[33] Gerben Hekstra, "Sequential floating-point Cordic," Tech. Rep., Dept. Electrical Engineering, Delft University of Technology, 1992.

[34] Gerben Hekstra, "Parallel floating-point Cordic," Tech. Rep., Dept. Electrical Engineering, Delft University of Technology, 1993.

[35] Gerben J. Hekstra and Ed F.A. Deprettere, "Fast rotations: Low-cost arithmetic methods for orthonormal rotation," in *Proceedings of the 13th Symposium on Computer Arithmetic*, Tomas Lang, Jean-Michel Muller, and Naofumi Takagi, Eds. IEEE, July 1997, pp. 116–125.

[36] J. Götze, S. Paul, and M. Sauer, "An efficient Jacobi-like algorithm for parallel eigenvalue computation," *IEEE Trans. Comput.*, vol. 42, pp. 1058–1065, 1993.

[37] Jürgen Götze and Gerben J. Hekstra, "Adaptive approximate rotations for computing the EVD," in *Proceedings 3rd Int. Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, Aug. 1994.

[38] J. Götze and G. Hekstra, "Adaptive approximate rotations for computing the symmetric EVD," in *Algorithms and Parallel VLSI Architectures III*, Marc Moonen and Francky Catthoor, Eds., pp. 73–84. Elsevier, 1995.

[39] Gerben J. Hekstra, Ed F. Deprettere, Richard Heusdens, and Monica Monari, "Recursive approximate realization of image transforms with orthonormal rotations," in *Proceedings International Workshop on Image and Signal Processing*, Manchester, UK, November 1996.

[40] Gerben J. Hekstra, Ed F. Deprettere, Richard Heusdens, and Zhiqiang Zeng, "Efficient orthogonal realization of image transforms," in *Proceedings SPIE*, Denver, Colorado, US, August 1996.

[41] J. Götze, "CORDIC-based approximate rotations for SVD and QRD," in *European Signal Processing Conference*, Edinburgh, Scotland, 1994, pp. 1867–1871.

[42] J. Götze, P. Rieder, and J.A. Nossek, "Parallel SVD-updating using approximate rotations," in *SPIE Conference on Advanced Signal Processing: Algorithms and Applications*, San Diego, USA, 1995, SPIE, pp. 242–252.

[43] P. Rieder, J. Götze, M. Sauer, and J.A. Nossek, "Orthogonal approximation of the discrete cosine transform," in *Proc. European Conference on Circuit Theory and Design*, Istanbul, Turkey, Aug. 1995, pp. 1003–1006.

[44] Gerben Hekstra, "Intersection computation using bounding plane subdivision," Tech. Rep. ET/NT/Radio-7, TU Delft, November 1993.

[45] Gerben Hekstra, "Spherical Bounding Box computation," Tech. Rep. ET/NT/Radio-8, TU Delft, November 1993.

[46] Shen Li-Sheng, "Grid-Ray Cell Traversal," Tech. Rep. ET/NT/Radio-10, TU Delft, November 1993.

[47] Gerben Hekstra, "Derivation of a suitable 2-dimensional ϕ, θ sampling grid for fast cell traversal," Tech. Rep. ET/NT/Radio-15, TU Delft, November 1993.

[48] Gerben Hekstra, "Image Plane Bounding Box Computation," Tech. Rep. ET/NT/Radio-16, TU Delft, May 1994.

[49] Gerben Hekstra, "Spherical Bounding Box: Computation of the minimum distance to a patch," Tech. Rep. ET/NT/Radio-17, TU Delft, December 1993.

[50] H.S. Malvar and D.H. Staedlin, "The LOT: Transform coding without blocking effects," *IEEE Trans. on ASSP*, vol. 37, no. 4, pp. 553–559, 1989.

[51] Richard Heusdens, *Overlapped Transform Coding of Images: Theory, Application, and Realization*, Ph.D. thesis, Delft University of Technology, 1996.

[52] Ed F. Deprettere, Gerben Hekstra, and Richard Heusdens, "Fast VLSI overlapped transform kernel," in *1995 IEEE Workshop on VLSI Signal Processing*, Osaka, Japan, October 1995, vol. VLSI Signal Processing VIII, pp. 287–302.

Part I

High Performance Rotation Techniques

Chapter 2

FLOATING-POINT CORDIC ALGORITHMS AND ARCHITECTURES

Contents

2.1 Introduction

In this chapter, we present a floating-point Cordic algorithm that uses a floating-point representation for the angle of rotation as well as for the vector data. We call this a *full floating-point* Cordic algorithm. We also present the implementation of this algorithm on both a sequential and a parallel architecture. The algorithm relies on a special floating-point representation of the angle. This representation allows angle computations to be done with a guaranteed *relative* precision, which is necessary for a correct implementation of a floating-point Givens operator. In this chapter, we will mainly follow the work on full floating-point Cordic as presented by Hekstra and Deprettere in [1, 2, 3, 4]. We state that our algorithm is the only known, to our current knowledge, to have an efficient implementation on a parallel, pipelined architecture.

The underlying interest in the full floating-point Cordic algorithm is the accurate and robust realization of a Givens rotation [5] that works on floating-point data. Such a floating-point Givens rotation is a useful computational element in DSP applications that rely on numerical matrix algebra. As an illustration we mention a practical application of radar signal processing [6, 7, 8] that relies extensively on the Minimum Variance Distortionless Response (MVDR) algorithm. For the definition of the Givens rotation we refer the reader to Golub and Van Loan in [5]. We state that a correct realization of a floating-point Givens rotation is able to zero a given element in a matrix of floating-point values to a certain relative precision. We illustrate this in Example 2.1 below.

As early as 1971, Walther [9] has laid the foundation for Cordic arithmetic for when the arguments, both vector *and* angle, are in floating-point number representation. The use of a floating-point number representation of the angle was essential for the application in question, which was the evaluation of the transcendental functions for calculators and floating-point numerical co-processors. Since then, a few researchers [10, 11, 12, 13] have published on advances in floating-point Cordic algorithms and architectures for computation on floating-point numbers. For all the above publications it is the case, however, that the input vector is in a floating-point number representation, while the angle is in a fixed-point representation, conform Volder's classical algorithm [14]. Cavallaro and Luk, in [11], indeed argue that for matrix computations, such as QR decomposition and Singular Value Decomposition (SVD), a fixed-point representation of the angle is sufficient. A fixed-point representation of the angle implies an absolute accuracy, or absolute **angle resolution**, with which any angle can be approximated.

We choose to argue differently, which we illustrate by our example below. To expose the destructive effects of the absolute accuracy in the angle, we consider the following example:

Example 2.1
Let the 2×2 *matrix* $\mathbf{A}$ *be given by*

$$\mathbf{A} = \begin{bmatrix} 1.2000 \cdot 10^{+6} & 1.4000 \cdot 10^{+6} \\ 2.6400 \cdot 10^{-5} & 3.4700 \cdot 10^{-5} \end{bmatrix}, \tag{2.1}$$

and the 2×1 *vector* $\mathbf{y}$ *be given by*

$$\mathbf{y} = \begin{bmatrix} 1.1000 \cdot 10^{+6} \\ 2.3700 \cdot 10^{-5} \end{bmatrix}. \tag{2.2}$$

Let us define $\mathbf{x}$ *as the* 2×1 *vector*

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{2.3}$$

with unknowns x_1 *and* x_2*, such that the equation*

$$\mathbf{Ax} = \mathbf{y} \tag{2.4}$$

is satisfied.

Let us consider the problem of solving $\mathbf{x}$ *using QR decomposition and back-substitution. For a treatise on QR decomposition, we refer the reader to Golub and Van Loan [5, sec. 6.2]. The QR decomposition step transforms Equation (2.4) to*

$$\mathbf{Rx} = \mathbf{Qy}. \tag{2.5}$$

where $\mathbf{Q}$ *is an orthogonal matrix, and* $\mathbf{R} = \mathbf{QA}$ *is by definition an upper triangular matrix. For this simple example,* $\mathbf{Q}$ *is a* 2×2 *rotation matrix, and hence is characterized by a single parameter (angle)* θ *in*

$$\mathbf{Q} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}. \tag{2.6}$$

The exact solution for θ *in our example is* $\theta = -2.2000 \cdot 10^{-11}$*. Applying this to Equation (2.5) results in the set of equations:*

$$\begin{bmatrix} 1.2000 \cdot 10^{+6} & 1.4000 \cdot 10^{+6} \\ 0.0 & 3.9000 \cdot 10^{-6} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.1000 \cdot 10^{+6} \\ -5.0000 \cdot 10^{-7} \end{bmatrix}. \tag{2.7}$$

Using standard back-solving techniques, we first solve for $x_2 = -1.2821 \cdot 10^{-1}$*, and use this to solve for* $x_1 = 1.0662 \cdot 10^{+0}$*.*

Of course, any arithmetic technique that uses finite precision to determine θ, and to perform the rotation $\mathbf{Q}$ *is bound to introduce errors. We look at the error in θ only, and define $\tilde{\theta} = \theta + \varepsilon$ as the approximation to θ with error ε. Let us define correspondingly the orthogonal matrix* $\tilde{\mathbf{Q}}$ *as the approximation to* $\mathbf{Q}$, *and which is given by:*

$$\tilde{\mathbf{Q}} = \begin{bmatrix} \cos(\tilde{\theta}) & -\sin(\tilde{\theta}) \\ \sin(\tilde{\theta}) & \cos(\tilde{\theta}) \end{bmatrix}. \tag{2.8}$$

Let us now define δ as the absolute angle resolution of the classical Cordic algorithm. This is the bound of the error in approximating the angle. For our example, we assume a precision of 12 *bits, which results in a practical value for* $\delta = \arctan(2^{-12}) \approx 2.4414 \cdot 10^{-4}$. *Twelve bit precision may not sound like much, but looking at the entire range of $-\pi \dots \pi$ we have a resolution of around* $2.4414 \cdot 10^{-4}/(2\pi) = 0.0039\%$, *which at first sight would appear adequate. The resolution limits the error of approximation ε to the domain $-\delta \leq \varepsilon \leq \delta$. Even then, we observe that this resolution is still several orders of magnitude larger than the θ of our example. To see what the destructive effect of this absolute angle resolution is on our solution, let us consider the worst-case situations, when $\varepsilon = \pm\delta$, and disregarding any other sources of error.*

For the case $\varepsilon = \delta = 2.4414 \cdot 10^{-4}$, *we have* $\tilde{\theta} = 2.2000 \cdot 10^{-10} + 2.4414 \cdot 10^{-4} \approx 2.4414 \cdot 10^{-4}$, *and see clearly that this error completely masks the value of θ. Applying the approximate* $\tilde{\mathbf{Q}}$ *to Equation (2.5) results in the set of equations:*

$$\begin{bmatrix} 1.2000 \cdot 10^{+6} & 1.4000 \cdot 10^{+6} \\ 2.9297 \cdot 10^{+2} & 3.4180 \cdot 10^{+2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.1000 \cdot 10^{+6} \\ 2.6855 \cdot 10^{+2} \end{bmatrix}. \tag{2.9}$$

Note that the $(2,1)$ entry of the matrix $\tilde{\mathbf{R}} = \tilde{\mathbf{Q}}\mathbf{A}$ *is non-zero, due to the approximation. For the back-solving, we however implicitly* assume *it to be (close enough to) zero. Again, we first solve for* $x_2 = 7.8571 \cdot 10^{-1}$, *and use this to solve for* $x_1 = 1.2166 \cdot 10^{-8}$. *Note the large discrepancy between these and the exact solutions for x_1, x_2, in this case about 8 respectively 1 orders of magnitude.*

In the same way we solve the system of equations x_1, x_2 for the case of $\varepsilon = -\delta = -2.4414 \cdot 10^{-4}$. *The results are shown below for comparison.*

	exact, $\varepsilon = 0$	$\varepsilon = +\delta$	$\varepsilon = -\delta$
x_1	$1.0662 \cdot 10^{+0}$	$1.2166 \cdot 10^{-8}$	$-1.2166 \cdot 10^{-8}$
x_2	$-1.2821 \cdot 10^{-1}$	$7.8571 \cdot 10^{-1}$	$7.8571 \cdot 10^{-1}$

Clearly, both x_1 and x_2 are strongly affected by the error in approximation due to the fixed, absolute resolution.

One may argue that this example is an extreme case, but similar large differences in the magnitude of the input do occur *commonly in practice. They are due to the poor balancing of the equations, a situation that occurs in problems, such as both the QR-based and the inverse QR-based Recursive Least Squares (QRD-RLS, iQR-RLS) and the Minimum Variance Distortionless Response (iQR-MVDR) algorithms [8, 15, 7].*

□

Walther's full floating-point Cordic algorithm would be able to overcome this problem, as it makes use of a floating-point representation of the angle. However, it is not entirely suitable for our purposes due to the following drawbacks:

- It explicitly assumes that the numerical value of the angle is computed (the z - iteration of the Cordic algorithm). This is a valid assumption for its initial application, that of a floating-point numerical co-processor capable of calculation of elementary functions. This, however, poses problems how to accurately represent angles of vectors which are almost parallel to the x- or y-axes. These angles would be very close to $k \times \frac{\pi}{2}, k \in \{1,2,3\}$, and their finite precision representation would give rise to similar problems as exposed in Example 2.1.
- It is particularly unsuitable for an efficient parallel implementation. In a parallel, pipelined implementation, the first stage implements the first step of the algorithm, the second stage implements the second step, and so on. For Walther's algorithm, the starting value of the iteration index, the shifts for the x and y iterations, and the scaling factor correction all depend upon the exponents of the inputs. This implies that the functionality of each stage in the pipeline is such that it should cover all possible situations. In practice this means that each stage needs a variable shifter for each of the x and y datapaths. For VLSI this is a very unattractive and area-consuming solution.

Both of the above problems are solved with the floating-point Cordic algorithm and the introduction of a proper floating-point angle format as presented in [4]. The results of this floating-point Cordic have been presented separately for a sequential architecture in [2] and for a parallel architecture in [3]. In this chapter, we unify our approach for both architectures.

2.1.1 Outline of this chapter

In Section 2.2, we start off by treating the floating-point representation of numbers. Analogous to this, in Section 2.3, we present a floating-point representation of angles. We exploit certain properties of the angles which are crucial to achieve a higher relative accuracy than would be possible in a regular floating-point number representation. In Section 2.4, we present our full floating-point Cordic algorithm. Based on the algorithms reported on separately by the author in [2, 3] we have unified this algorithm to be, at this stage, independent of the target architecture. We treat the implementation of this algorithm on a sequential architecture in Section 2.5. This work is based on that which has been presented earlier by the author in [2]. Similarly, we treat the implementation of this algorithm on a parallel architecture in the adjoining Section 2.6. For this section, we follow the work as presented earlier by the author in [3]. In Section 2.7, we validate the algorithm and its implementations and prove its worthiness. We examine the behaviour of the error in accuracy from the results of extensive simulations, and show that it is conform to the designed limits. We discuss the advantages and disadvantages of our floating-point Cordic algorithms and implementations in Section 2.8, seen against the background of the available alternatives. Finally, in Section 2.9, we give our conclusions. In Appendix 2.A we enclose the proofs and derivations relevant to this chapter. In Appendix 2.B, we present angle-bases for the floating-point Cordic algorithms and architectures for various wordwidths.

2.2 Floating-point representation of numbers

Let us take a look at how floating-point numbers are represented. We assume that on the interface to the algorithm, the in- and outputs are represented in an IEEE 754 standard compliant format. For the finer details of this representation, we refer the reader to the text of the IEEE 754 standard [16]. Internally to the algorithm, we use a representation with a signed mantissa and exponent, which we explain in more detail below.

For the sake of clarity, let us first define a notation to represent unsigned numbers in base 2 (binary numbers). Let p, q be positive integers. Let $d_i \in \{0, 1\}$, and $-q \leq i \leq p-1$. The notation:

$$(d_{p-1}d_{p-2}\ldots d_1 d_0.d_{-1}d_{-2}d_{-3}\ldots d_{-q})_2, \tag{2.10}$$

represents a number with the numerical value:

$$\sum_{i=-q}^{p-1} d_i 2^i. \tag{2.11}$$

The d_i are the **digits** of the number, the terms 2^i are the **weights**. The term p signifies the number of digits in the integer part, while q signifies the number of digits in the fractional part. The point in the representation is the **binary point** (compare: decimal point for base 10). If there is no fractional part, we may leave out the binary point. For example: the number $(110.101)_2$ is the base 2 notation for the number 6.625 in decimal notation.

The floating-point representation of a number x is a tuple (s, m, e) with:

sign A sign bit s, indicating the sign of the number and representing values from $\{-1, 1\}$.

mantissa An N_{mant}-bit, unsigned, mantissa m. The number of bits N_{mant} determine the *precision* of the representation. Assuming correct rounding, the mantissa m has an inherent accuracy of $\pm\frac{1}{2}$lsb, where lsb $= 2^{-N_{\text{mant}}}$ is the weight for the least significant bit.

exponent An N_{exp}-bit exponent e. The number of bits N_{exp} determine the dynamic range of the representation. The exponent may have a **bias** b, which indicates at what value the range of exponents start. This bias does not affect computations such as additions and rotations, so we will not see it later in the floating-point Cordic algorithm.

The corresponding value of x is given by:

$$x = sm2^e. \tag{2.12}$$

Later on in this chapter, we will use a floating point representation as a tuple (m, e) that makes use of a *signed* mantissa. This is to keep certain equations simple, and does not affect anything significantly. The value of x is then given by:

$$x = m2^e. \tag{2.13}$$

The mantissa is said to be **normalized** if it is restricted to

$$(0.1000\ldots00)_2 \leq |m| \leq (0.1111\ldots11)_2. \tag{2.14}$$

Note that the most significant digit $m_{-1} = 1$ for normalized numbers. If the above restriction is lifted, then the mantissa is said to be **denormalized**, and takes on values in the domain:

$$(0.0000\ldots00)_2 \leq |m| \leq (0.1111\ldots11)_2. \tag{2.15}$$

Example 2.2
Let us consider the case when $N_{\text{mant}} = 4$, $N_{\text{exp}} = 2$. Let the bias for the exponent be set at $b = -1$. The value of the exponent is limited to:

$$b = -1 \leq e \leq 2 = (2^{N_{\text{exp}}} - 1) + b. \tag{2.16}$$

For normalized numbers, the value of the mantissa is limited to:

$$(0.1000)_2 \leq |m| \leq (0.1111)_2. \tag{2.17}$$

However, for denormalized numbers, the domain of possible values is extended to include zero:

$$(0.0000)_2 \leq |m| \leq (0.1111)_2. \tag{2.18}$$

We construct the 2-D floating-point space, by considering all possible vectors (x,y), where both x and y are numbers in our floating-point representation. The result is shown in Figure 2.1.

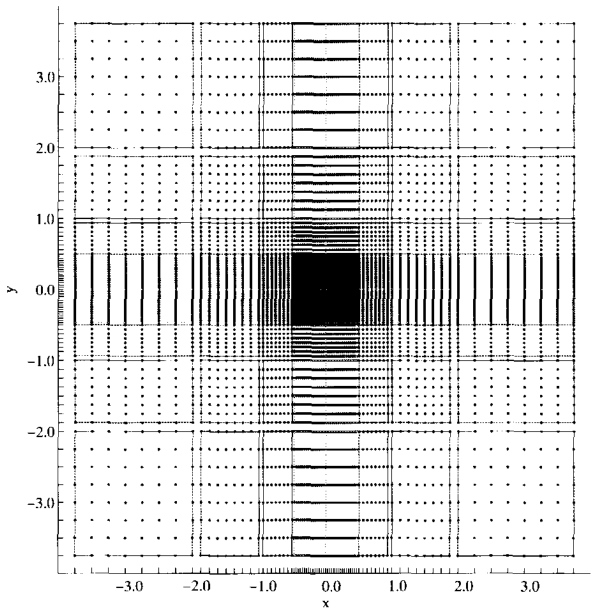


Figure 2.1. *The 2-D Floating-point space*

Let x,y be numbers whose floating-point representation given by (s_x, m_x, e_x) and (s_y, m_y, e_y) respectively. Let us define the normalized floating-point domain $\mathcal{D}_{e_x,e_y}$ as the set of 2-D points (x,y), where the mantissas m_x, m_y are both normalized:

$$\mathcal{D}_{e_x,e_y} = \{(x,y)|x = \pm m_x 2^{e_x}, y = \pm m_y 2^{e_y}, 1 \leq m_x, m_y < 2 - \text{lsb}\}. \tag{2.19}$$

Let us likewise define the partial normalized floating point domain $\mathcal{D}^{(p)}_{e_x,e_y}$ as the set of 2-D points (x,y), where only the mantissa m_x is normalized:

$$\mathcal{D}^{(p)}_{e_x,e_y} = \{(x,y)|x = \pm m_x 2^{e_x}, y = \pm m_y 2^{e_y}, 1 \le m_x < 2 - \text{lsb}\}, \qquad (2.20)$$

and let us define the denormalized floating-point domain $\mathcal{D}^{(d)}_{e_x,e_y}$ as the set of 2-D points (x,y), where the normalization restriction is lifted for both m_x and m_y.

$$\mathcal{D}^{(d)}_{e_x,e_y} = \{(x,y)|x = \pm m_x 2^{e_x}, y = \pm m_y 2^{e_y}\}. \qquad (2.21)$$

We state the following for these domains. Any vector $\mathbf{v} = (x,y) \in \mathcal{D}_{e_x,e_y}$ has a counterpart $\mathbf{v}' = (x',y') \in \mathcal{D}_{e_x-e_y,0}$ such that they have the same angle of inclination and have the same relative accuracy, as determined by the precision of the coordinates of the vectors.

The floating point domains are visualized in Figure 2.1. The boxed areas represent the domains $\mathcal{D}_{e_x,e_y}$ in which the exponents of the x and y coordinates of the vector are constant, and their mantissas are normalized. The points within such a region represent all possible values for the floating-point numbers within the domain.

2.2.1 Accuracy requirements

Looking at the floating-point space, as shown in Figure 2.1, we can identify the problem of the floating-point representation of the input vector, and its repercussions on the representation of the angle. Let $\mathbf{v} = (x,y)$ be a vector from the region $\mathcal{D}_{e_x,e_y}$. The coefficients x,y are represented in a floating-point format. Let us assume that $e_x > e_y$, and $x > 0$. This means that the vector $\mathbf{v}$ is lying close to the positive x-axis. Let us also concentrate on the angle α between the vector $\mathbf{v}$ and the positive x-axis. The larger the exponent e_x is compared to e_y, the smaller the angle α, which is given by:

$$\alpha = \arctan\left(\frac{m_y \cdot 2^{e_y}}{m_x \cdot 2^{e_x}}\right) = \arctan\left(\frac{m_y}{m_x} \cdot 2^{e_y - e_x}\right), \qquad (2.22)$$

in which this effect is also clearly visible.

The floating-point representation of numbers carries with it an inherent error in the representation. At the best, this error is bounded by $\varepsilon_{\text{round}} = \text{lsb}/2$ for perfect rounding. Taking this into account in Equation (2.22) results in:

$$\alpha + \varepsilon_\alpha = \arctan\left(\frac{(m_y \pm \varepsilon_{\text{round}}) \cdot 2^{e_y}}{(m_x \pm \varepsilon_{\text{round}}) \cdot 2^{e_x}}\right) = \arctan\left(\frac{m_y \pm \varepsilon_{\text{round}}}{m_x \pm \varepsilon_{\text{round}}} \cdot 2^{e_y - e_x}\right), \qquad (2.23)$$

where ε_α is the error induced by the representation of the vector $\mathbf{v}$, on the angle α. What we deduce from Equation (2.23) is that the error ε_α, remains *relative* to the angle α. A smaller angle α results in a smaller error ε_α, but they remain proportional to each other.

So, if we want to accurately represent an angle α, derived from a floating-point vector $\mathbf{v}$, we need to have an error of the representation of the angle, or **angle resolution**, which is smaller than, and comparable in size to, the angle ε_α induced by the error in representation of the vector $\mathbf{v}$. See also Appendix 2.A.3 for the derivation of such an angle resolution.

2.3 Floating-point representation of angles

A proper floating-point representation of angles has been crucial in the development of the floating-point Cordic algorithm. We will go into detail to why we need a floating point representation, and how it is built up.

2.3.1 The Angle base

Hekstra , in [1, 2, 3], introduced the notion of an **angle base** to represent angles.

We define the angle base $\mathcal{A}$ as the ordered sequence of N angles $\bar{\alpha}_i$, as given by $\mathcal{A} = \{\bar{\alpha}_0, \bar{\alpha}_1, \ldots, \bar{\alpha}_{N-1}\}$. We call the angles $\bar{\alpha}_i$ the **base angles** of the representation. They are similar in function to the weights in the floating-point representation of numbers. The base angles are ordered in size, satisfying $\bar{\alpha}_i \geq \bar{\alpha}_{i+1}$. The number of base angles is given by N. The largest base angle is given by $\bar{\alpha}_0$, and the smallest is given by $\bar{\alpha}_{N-1}$.

Representation of angles

The angle base is a means to bind the **base angles** for the representation together. We can represent an angle α using these base angles as:

$$\alpha = \sum_{i=0}^{N-1} \sigma_i \bar{\alpha}_i + \varepsilon_\alpha , \tag{2.24}$$

where the σ_i form the digits of the angle representation. Normally speaking, for the Cordic algorithm, $\sigma_i \in \{-1, +1\}$. The error in representation is given here as ε_α. From this equation we can clearly see that the sequence of $\bar{\alpha}_i$ forms a number base for a number representation, where the $\bar{\alpha}_i$ are the *weights*, where σ_i are the *digits* of the representation, and where ε_α is the *representation error*.

Let us define the digit sequence ς as the sequence of digits σ_i, given by:

$$\varsigma = (\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{N-1}) . \tag{2.25}$$

We state that ς is a finite accuracy **representation** of the angle α.

We define the **angle resolution** δ as the bound of the error in the representation of the angle, using a given angle base $\mathcal{A}$. The error of representation ε_α is then bound by:

$$|\varepsilon_\alpha| \leq \delta . \tag{2.26}$$

We state that the bound δ is given by the value of the smallest angle $\bar{\alpha}_{N-1}$ of the angle base:

$$\delta = \bar{\alpha}_{N-1} . \tag{2.27}$$

We define the angle **domain of convergence** γ as the largest positive representable angle for a given angle base $\mathcal{A}$. Any angle α that satisfies:

$$|\alpha| \leq \gamma , \tag{2.28}$$

can be represented using the angle base $\mathcal{A}$. The value of γ follows for when all the σ_i are equal to $+1$, and is given by:

$$\gamma = \sum_{i=0}^{N-1} \bar{\alpha}_i . \tag{2.29}$$

Similarly, the largest negative angle is given by $-\gamma$, for when all σ_i are equal to -1.

For use in a Cordic algorithm, the angle base $\mathcal{A}$ must satisfy the condition:

$$\bar{\alpha}_k \leq \sum_{i=k+1}^{N-1} \bar{\alpha}_i + \delta . \tag{2.30}$$

Muller, in [17], refers to this as a *discrete basis of the order 1*.

Relation to micro-rotations

Let $\mathbf{F}_i$ be a generalized micro-rotation, as defined in Equation (1.5), with approximation pair c_i, s_i, angle of rotation α_i, and magnification factor m_i.

The angle base $\mathcal{A}$ is coupled to a micro-rotation base $\mathcal{F}$ which is an ordered sequence of the micro-rotations, N in number, given by $\mathcal{F} = \{\mathbf{F}_0, \mathbf{F}_1, \ldots, \mathbf{F}_{N-1}\}$. The micro-rotation $\mathbf{F}_i$ is capable of rotating over the angle $\alpha_i = \sigma_i \cdot \bar{\alpha}_i$. In this way the link is made to the base angles $\bar{\alpha}_i$, and is it possible to rotate over the angle α with this sequence of micro-rotations. See also Section 1.5.3.

Likewise, we define the overall magnification factor K, as the product of the magnification factors m_i, as given by:

$$K = \prod_{i=1}^{N-1} m_i . \tag{2.31}$$

Indexing of floating-point angle bases

The classical Cordic algorithm, which works on fixed-point data, uses only a single angle base $\mathcal{A}$ and micro-rotation base $\mathcal{F}$, with which it performs the rotations. This approach would fail for floating-point Cordic since the dynamic range of the angles, coupled to the required accuracy, would implicate an angle base of enormous size.

Instead, we use a collection of angle bases and *match* these to regions in the 2-D floating-point space. For a given region, it is possible to determine the maximum angle γ that needs to be represented, and also the required angle resolution δ. By proper choice of these regions, we can keep the size N of the angle base small. If the union of the regions covers the entire 2-D floating-point space, then this collection of angle bases is capable of representing any angle α, corresponding to a vector $\mathbf{v}$ from this space. This is the key concept behind the floating-point Cordic algorithm.

It is useful if we can index an arbitrary angle base $\mathcal{A}$. Let us define the **angle exponent** κ as an integer parameter, by which we index the angle base $\mathcal{A}_\kappa$, and all its related properties. We shall see the precise function of the angle exponent later on.

The base angles of $\mathcal{A}_\kappa$ are denoted by $\bar{\alpha}_{\kappa,i}$. The total number of base angles is given by N_κ, which may be different per angle base. Likewise, the smallest and largest base angles are given by $\bar{\alpha}_{\kappa,N_\kappa-1}$ and $\bar{\alpha}_{\kappa,0}$. The angle resolution is denoted by δ_κ, and is given by:

$$\delta_\kappa = \bar{\alpha}_{\kappa,N_\kappa-1}\,. \tag{2.32}$$

The domain of convergence is denoted by γ_κ and is given by:

$$\gamma_\kappa = \sum_{i=0}^{N_\kappa-1} \bar{\alpha}_{\kappa,i}\,. \tag{2.33}$$

The angle exponent κ also indexes the micro-rotation base, as in $\mathcal{F}_\kappa$, and its related properties, analogous to the angle base. The micro-rotations are denoted by $\mathbf{F}_{\kappa,i}$, with approximation pair $c_{\kappa,i}, s_{\kappa,i}$, angle of rotation $\alpha_{\kappa,i}$, and magnification factor $m_{\kappa,i}$. The overall magnification factor is denoted by K_κ, and is given by:

$$K_\kappa = \prod_{i=1}^{N_\kappa-1} m_{\kappa,i}\,. \tag{2.34}$$

Similarly, we re-define the digit sequence, ς as the sequence of digits σ_i, given by:

$$\varsigma = (\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{N_\kappa-1})\,. \tag{2.35}$$

Note that ς has a variable length N_κ, dependent on the angle exponent κ.

We state that the tuple (κ, ς) is the floating-point representation of a floating point angle α. This is analogous to the floating-point representation of numbers, κ plays the role of the exponent e, while ς plays the role of the mantissa m.

2.4 The floating-point Cordic algorithm

In this section we present the floating-point Cordic algorithm, which at this stage is still independent on the final choice of the architecture. We do, however, present a global architecture, from which we later derive both a sequential and a parallel, pipelined architecture. We present the iteration scheme for the floating-point Cordic which is the foundation to both architectures. This iteration scheme makes use of the angle exponent κ and the angle bases $\mathcal{A}_\kappa$ to attain the required floating-point accuracy. We finish of with an observation on what the constraints are for an efficient implementation.

2.4.1 Global behaviour of the floating-point Cordic

The floating-point Cordic algorithm consists of operations at three distinctive levels. They are:

1. **floating-point interface** The input vector is translated from the outside floating-point format (IEEE 754 compliant) to the internal floating-point format. Likewise, the result is translated back from the internal format back to the outside format. This may require renormalisation, range checking etc.

2. **pre- and post-rotations** These perform exact rotations over π and $\pi/2$ radians, at a very low cost. The pre-rotations always bring the input vector to a given region, so as to simplify the arithmetic within the core, and to guarantee convergence.

3. **floating-point core** This performs the actual floating-point Cordic iteration to rotate over a floating-point angle. The implementation of this core is a further point of discussion, as this changes for the sequential or parallel architecture.

A global prototype architecture for the floating-point Cordic is shown in Figure 2.2, which displays these levels. The input to the floating-point Cordic is a vector $\mathbf{v}_{\text{in}}$ with components $x_{\text{in}}, y_{\text{in}}$, and a floating point angle, with components β_{in} and α_{in}. The output of the floating-point Cordic is analogous to the input, however with subscript "out". See also Figure 2.2.

We do not go into detail on how the floating-point format conversions are done, but refer only to [16], and state that this is similar to that of floating-point addition. Instead, we assume that the input is available in a (signed mantissa, exponent) format such as $(m_{x,\text{in}}, e_{x,\text{in}})$, with the relationship:

$$x_{\text{in}} = m_{x,\text{in}} 2^{e_{x,\text{in}}} \tag{2.36}$$

to the input x_{in}, and similarly for y_{in}. We also assume that it is enough that the output is delivered in the same internal format, with $(m_{x,\text{out}}, e_{x,\text{out}})$ and $(m_{y,\text{out}}, e_{y,\text{out}})$.

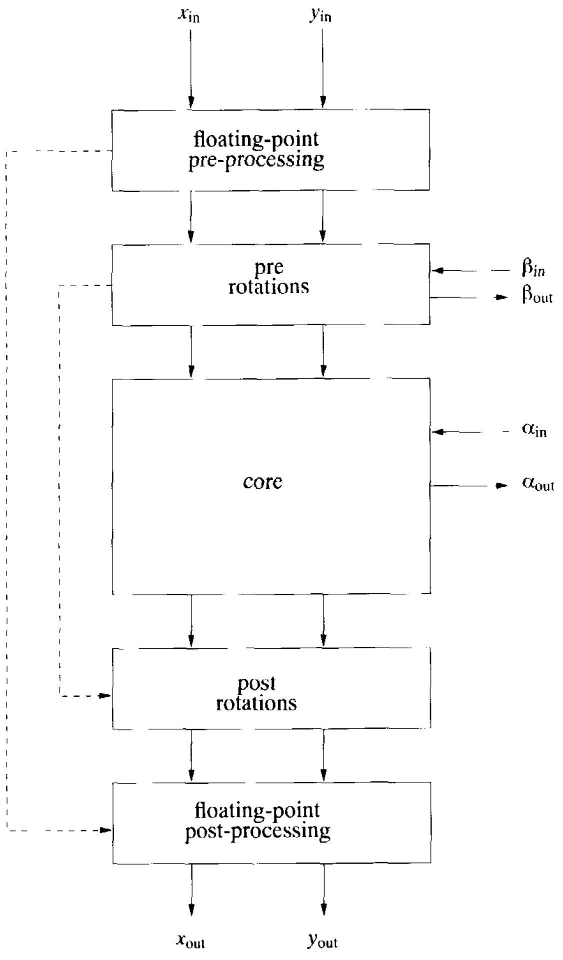


Figure 2.2. A global, prototype architecture for the floating-point Cordic.

Modes of operation

The Cordic algorithm, both the classical and the floating-point one, have two modes of operation, namely **rotation** and **vectoring**. In both modes a rotation is performed, the only difference is how the angle of rotation is obtained. Let β and α be the internally available angles of rotation for the pre- and post-rotation stage, and for the core respectively. Let us define θ as the overall floating-point angle of rotation, which is given by the sum of the individual angles:

$$\theta = \beta + \alpha. \tag{2.37}$$

The angle β is a so-called **exact**, or accuracy preserving angle. For circular rotation, it is a multiple of $\pi/2$ and very cheap to implement. It is used to take advantage of the available symmetry of the floating-point domain, as visible in Figure 2.1. The angle α is a floating-point angle, as treated in the previous section, and restricted to $|\alpha| \leq \pi/4$, thanks to the rotation over β.

In both modes of operation, the rotation of the input vector over these angles is performed, resulting in the output vector:

$$\begin{bmatrix} x_{\text{out}} \\ y_{\text{out}} \end{bmatrix} = \mathbf{R}(\alpha)\mathbf{R}(\beta) \begin{bmatrix} x_{\text{in}} \\ y_{\text{in}} \end{bmatrix}. \tag{2.38}$$

For the 'rotation' mode, these angles of rotation are taken from the input, and used to perform the rotation, as given by:

$$\begin{aligned} \beta &= \beta_{\text{in}} \\ \alpha &= \alpha_{\text{in}} . \end{aligned} \tag{2.39}$$

For the 'vectoring' mode, these angles are determined from the input vector $(x_{\text{in}}, y_{\text{in}})$, in such a way that, when rotated according to Equation (2.38), the output vector $(x_{\text{out}}, y_{\text{out}})$ lies along the positive x axis, and satisfies:

$$\begin{aligned} x_{\text{out}} &\geq 0 \\ y_{\text{out}} &= 0 . \end{aligned} \tag{2.40}$$

Since the rotation is norm-preserving, the component x_{out} is equal to the length of the input vector. The input angles are ignored. In both cases, the actual angle rotated over is sent to the output, as given by:

$$\begin{aligned} \beta_{\text{out}} &= \beta \\ \alpha_{\text{out}} &= \alpha . \end{aligned} \tag{2.41}$$

Notation

Since we are dealing with a multitude of blocks and signals in the floating-point Cordic algorithm, we define an implicit notation for the variables used within the algorithm, to avoid a multitude of definitions. Let 'block' be the name of one of the blocks in the scheme of Figure 2.2: 'pre', 'post', 'core' and otherwise 'in' or 'out'. We consider the notation only for the x component. The same notation holds for the y component, and possibly the angle. Let x_{block} be defined as the x-coordinate output of that block. The exponent and mantissa of the floating-point representation of x_{block} are implicitly given as $e_{x,\text{block}}$ and $m_{x,\text{block}}$, with:

$$x_{\text{block}} = m_{x,\text{block}} \cdot 2^{e_{x,\text{block}}}. \tag{2.42}$$

Inside the block, we identify as well iteration variables, which we denote by $x_{\text{block}}[i]$, $e_{x,\text{block}}[i]$ and $m_{x,\text{block}}[i]$.

2.4.2 Pre- and post-rotations

The combination of both the pre- and post rotations perform a rotation over β. Let β_{pre} be the angle belonging to the pre-rotation, and let β_{post} be the angle for the post-rotation. Both are exact angles, and multiples of $\pi/2$. The angle β is the combination of the above, and given by:

$$\beta = \beta_{\text{pre}} + \beta_{\text{post}}. \tag{2.43}$$

The pre-rotations by itself are used to condition the input vector, such that it is brought to a pre-defined region of convergence. This is done to simplify the operations within the core.

The input vector $(x_{\text{in}}, y_{\text{in}})$, is rotated to the region of convergence using β_{pre}, such that the following properties hold for the resulting vector $(x_{\text{pre}}, y_{\text{pre}})$:

$$e_{x,\text{pre}} \geq e_{y,\text{pre}} \tag{2.44}$$

$$m_{x,\text{pre}} \geq 0 \tag{2.45}$$

This is done with rotations over $\pi/2$, and over π. These rotations take effect in changing the signs and exchanging the exponents and mantissas between x and y and therefore do not affect the precision that is present in the data. The first pre-rotation over $\pi/2$ is performed if $e_{y,\text{in}} > e_{x,\text{in}}$, otherwise no rotation is performed. The implementation of this rotation, working on floating-point data is shown in Algorithm 1. The result of the rotation is an intermediate vector $(x_{\text{pre}}[1], y_{\text{pre}}[1])$.

ALGORITHM 1
```
if ($e_{y,\text{in}} > e_{x,\text{in}}$) then
    /* rotate over $\pi/2$ */
    $\sigma_{0,\text{pre}} := 1$
    $(e_{x,\text{pre}}[1], m_{x,\text{pre}}[1]) := (e_{y,\text{in}}, -m_{y,\text{in}})$
    $(e_{y,\text{pre}}[1], m_{y,\text{pre}}[1]) := (e_{x,\text{in}}, m_{x,\text{in}})$
**else**
    /* no rotation */
    $\sigma_{0,\text{pre}} := 0$
    $(e_{x,\text{pre}}[1], m_{x,\text{pre}}[1]) := (e_{x,\text{in}}, m_{x,\text{in}})$
    $(e_{y,\text{pre}}[1], m_{y,\text{pre}}[1]) := (e_{y,\text{in}}, m_{y,\text{in}})$
**end if**
```

The second pre-rotation, over π radians this time, is performed if the resulting $m_{x,\text{pre}}[1]$ is negative, see Algorithm 2.

```
ALGORITHM 2
**if** $(m_{x,\text{pre}}[1] < 0)$ **then**
    /* rotate over $\pi$ */
    $\sigma_{1,\text{pre}} := 1$
    $(e_{x,\text{pre}}, m_{x,\text{pre}}) := (e_{x,\text{pre}}[1], -m_{x,\text{pre}}[1])$
    $(e_{y,\text{pre}}, m_{y,\text{pre}}) := (e_{y,\text{pre}}[1], -m_{y,\text{pre}}[1])$
**else**
    /* no rotation */
    $\sigma_{1,\text{pre}} := 0$
    $(e_{x,\text{pre}}, m_{x,\text{pre}}) := (e_{x,\text{pre}}[1], m_{x,\text{pre}}[1])$
    $(e_{y,\text{pre}}, m_{y,\text{pre}}) := (e_{y,\text{pre}}[1], m_{y,\text{pre}}[1])$
**end if**
```

The combination of these rotations bring the input data to the halfplane with $m_{x,\text{pre}} \geq 0$. The pre-rotations are visualized in Figure 2.3. The shaded areas indicate the region in in which the vector $(x_{\text{pre}}, y_{\text{pre}})$ resides after this pre-rotation.

The angle β_{pre} over which has been rotated, is given by:

$$\beta_{\text{pre}} = \sigma_{0,\text{pre}} \cdot \frac{\pi}{2} + \sigma_{1,\text{pre}} \cdot \pi. \tag{2.46}$$

The post-rotations compensate for the angle of rotation, if necessary. For the 'vectoring' mode, the input needs to be brought to this region of convergence anyway, and hence:

$$\begin{aligned} \beta &= \beta_{\text{pre}} \\ \beta_{\text{post}} &= 0 \end{aligned} \tag{2.47}$$

For the 'rotation' mode, however, β is enforced, and we need to compensate as follows:

$$\beta_{\text{post}} = \beta - \beta_{\text{pre}} \tag{2.48}$$

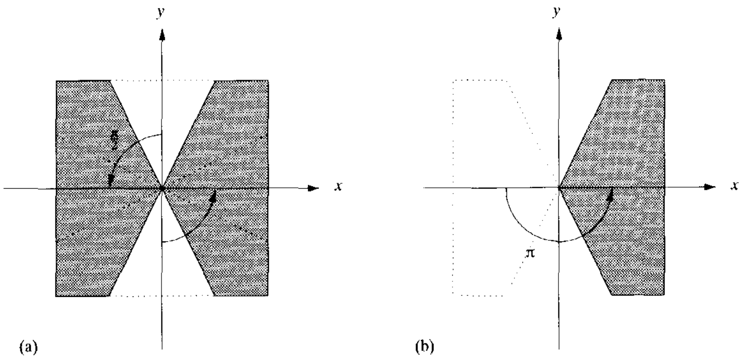


Figure 2.3. Pre-rotations of the data over (a) $\pi/2$ and (b) π radians.

In both cases, Equation 2.43 is satisfied. The post-rotations are performed with a similar technique as shown for the pre-rotations, and we will not discuss this further.

2.4.3 The floating-point core

The floating-point core of the Cordic algorithm performs a rotation over the floating-point angle α. The input to the core is the vector $(x_{\text{pre}}, y_{\text{pre}})$, coming from the pre-rotation unit. This vector is rotated over the floating-point angle α, to form the result $(x_{\text{core}}, y_{\text{core}})$. Depending upon the mode of operation, the angle α is either enforced from outside ('rotation' mode), or determined by the input vector ('vectoring' mode).

We assume that the angle α is represented by the tuple (κ, ς), as treated in Section 2.3. We also assume that the necessary angle bases $\mathcal{A}_\kappa$, indexed by the angle exponent κ are pre-calculated and available.

We will show how the angle bases are employed to perform the rotation over a floating-point angle.

The angle exponent and selection of the angle base

When we look at the floating-point domain in Figure 2.1 we see that when two regions $\mathcal{D}$ have the same difference in exponents $e_y - e_x$ they share the same domain of convergence and accuracy in angles. This is due to the equivalence:

$$\arctan\left(\frac{m_y \cdot 2^{e_y}}{m_x \cdot 2^{e_x}}\right) = \arctan\left(\frac{m_y}{m_x} \cdot 2^{e_y - e_x}\right) \tag{2.49}$$

We can match a single angle base $\mathcal{A}_d$ to these domains, with the index $d = e_y - e_x$ being the difference in exponents. This angle base is then suited to represent any angle that corresponds to a vector that lies in one of those regions, and with the correct relative accuracy. In such a way, we can cover the entire floating-point domain.

Let us consider the two modes of operation. For the 'vectoring' mode, we have to compute the angle α, such that the vector $(x_{\text{pre}}, y_{\text{pre}})$ is rotated to the positive x-axis. In this case, we compute the angle exponent, as given by:

$$\kappa = e_{y,\text{pre}} - e_{x,\text{pre}}, \tag{2.50}$$

and use this to select the angle base $\mathcal{A}_\kappa$ to perform the rotations in the core. Combining Equation (2.50) with (2.44), we obtain the bounds for the angle exponent, as given by:

$$\kappa_{\min} \leq \kappa \leq 0, \tag{2.51}$$

where $\kappa_{\min} = -(2^{N_{\exp}} - 1)$ is the smallest value that the angle exponent can attain in Equation (2.50).

We state that we can use the angle base $\mathcal{A}_\kappa$ to represent the angle α with the proper relative accuracy, and such that the vector $(x_{\text{pre}}, y_{\text{pre}})$ is indeed rotated to align to the x-axis .

For the 'rotation' mode, the angle α is given by the input α_{in}. The angle exponent κ is part of this representation, and this is used for selection the angle base $\mathcal{A}_\kappa$ to perform the rotations in the core. We *have* to use the angle base $\mathcal{A}_\kappa$, since the floating-point angle α is represented in *this* angle base.

For the moment, we will assume that the angle bases $\mathcal{A}_\kappa$ are given, and matched to the regions in the floating-point domain. The construction of these angle bases is deferred to Section 2.5 and the Appendices.

The floating-point Cordic iteration

For the rotation over the angle α, we use the micro-rotations, as given by the micro-rotation base $\mathcal{F}_\kappa$, also indexed by the angle exponent κ.

These micro-rotations are steered by the digits σ_i of the digit sequence ς. The number of digits, and hence the number of micro-rotations is given by N_κ.

The floating-point Cordic iteration that implements the sequence of micro-rotations is given, in broadest terms, by:

$$\begin{aligned} x[i+1] &= c_{\kappa,i} \cdot x[i] - \sigma_i s_{\kappa,i} \cdot y[i] \\ y[i+1] &= c_{\kappa,i} \cdot y[i] + \sigma_i s_{\kappa,i} \cdot x[i] \ . \end{aligned} \tag{2.52}$$

for $i = 0, 1, \ldots, N_k - 1$. The initial input to the iteration is given by:

$$\begin{aligned} x[0] &= \dot{K}_\kappa x_{\text{pre}} \\ y[0] &= \dot{K}_\kappa y_{\text{pre}} \end{aligned} \tag{2.53}$$

Note that the input is prescaled with $\check{K}_K$ to compensate for the overall magnification factor K_K. The result of the iteration is given by:

$$\begin{aligned} x_{\text{core}} &= x[N_K] \\ y_{\text{core}} &= y[N_K] \ , \end{aligned} \tag{2.54}$$

which is then passed on to the post-rotation unit.

We have dropped the subscripts for the core variables in order to keep the equations readable. Each iteration performs a micro-rotation $\mathbf{F}_{K,i}$ over the angle $\sigma_i \cdot \bar{\alpha}_{K,i}$.

For the 'vectoring' mode, the digits σ_i are determined by:

$$\sigma_i = \begin{cases} +1 & \text{if} \quad y[i] \geq 0 \\ -1 & \text{if} \quad y[i] < 0 \end{cases} , \tag{2.55}$$

as in classical Cordic, and such that $y_{\text{core}} \to 0$.

For the 'rotation' mode, the digits σ_i are provided in ς, which is part of the representation of α.

The modified floating-point Cordic iteration

The floating-point iteration, as given by Equations (2.52) to (2.55) not of practical use, as it hides the operations on the floating-point representation of $x[i], y[i]$. Writing the iteration of Equation (2.52) out in terms of mantissas and exponents results in:

$$\begin{aligned} m_x[i+1] \cdot 2^{e_x[i+1]} &= c_{K,i} \cdot m_x[i] \cdot 2^{e_x[i]} - \sigma_i s_{K,i} \cdot m_y[i] \cdot 2^{e_y[i]} \\ m_y[i+1] \cdot 2^{e_y[i+1]} &= c_{K,i} \cdot m_y[i] \cdot 2^{e_y[i]} + \sigma_i s_{K,i} \cdot m_x[i] \cdot 2^{e_x[i]} \ , \end{aligned} \tag{2.56}$$

which is a step in the right direction, but still too cluttered. We propose the use of **block floating-point** arithmetic inside the core, for as much as possible. This implies that the exponents remain fixed during a sequence of computations. This simplifies and even speeds up arithmetic, when compared to individual floating-point operations. Let $e_{x,\text{core}}, e_{y,\text{core}}$ be the exponents, which are determined before the iteration starts, and remain fixed throughout the computation. The iteration can then be greatly simplified to:

$$\begin{aligned} m_x[i+1] &= c_{K,i} \cdot m_x[i] - \sigma_i s_{K,i} \cdot m_y[i] \cdot 2^{e_{y.\text{core}} - e_{x.\text{core}}} \\ m_y[i+1] &= c_{K,i} \cdot m_y[i] + \sigma_i s_{K,i} \cdot m_x[i] \cdot 2^{e_{x.\text{core}} - e_{y.\text{core}}} \ . \end{aligned} \tag{2.57}$$

Note that the iteration is now completely written out in terms of operations on mantissas only. The only difference with a classical Cordic iteration is the presence of the terms $2^{e_{y.\text{core}} - e_{x.\text{core}}}$.

We would like to fix the exponents $e_{x,\text{core}}, e_{y,\text{core}}$, such that no overflow occurs for the mantissas. This is guaranteed [1] for:

$$\begin{aligned} e_{x,\text{core}} &= \max(e_{x,\text{pre}}, \kappa + e_{y,\text{pre}}) = e_{x,\text{pre}} \\ e_{y,\text{core}} &= \max(e_{y,\text{pre}}, \kappa + e_{x,\text{pre}}) \quad . \end{aligned} \tag{2.58}$$

This result is based on the analysis of the largest possible value that may occur in the iterations. The simplification for $e_{x,\text{core}}$ holds by grace of Equations (2.44) and (2.51).

The input to the iteration, as given before in Equation (2.53), can also be expressed in terms of mantissas only, as given by:

$$\begin{aligned} m_x[0] &= \check{K}_\kappa m_{x,\text{pre}} \\ m_y[0] &= \check{K}_\kappa m_{y,\text{pre}} \cdot 2^{e_{y,\text{pre}} - e_{y,\text{core}}} \end{aligned} \tag{2.59}$$

Note that only the y mantissa needs to be aligned.

The result of the iteration, and output of the core, is simply given by:

$$\begin{aligned} m_{x,\text{core}} &= m_x[N_\kappa] \\ m_{y,\text{core}} &= m_y[N_\kappa] \quad . \end{aligned} \tag{2.60}$$

Note that the exponents $e_{x,\text{core}}, e_{y,\text{core}}$ were already fixed.

Micro-rotation model

We take a short look at the types of micro-rotations that we consider for the implementation. We are in principle free to choose any type. Adhering to the convention started in [18, 2], we consider the **simple** and **complex** micro-rotation models.

The simple model is given by:

$$\begin{aligned} c_{\kappa,i} &= 1 \\ s_{\kappa,i} &= 2^{S_{i,\kappa}} \\ \bar{\alpha}_{\kappa,i} &= \arctan(2^{S_{i,\kappa}}) \\ m_{\kappa,i} &= \sqrt{1 + 2^{2S_{i,\kappa}}} \quad . \end{aligned} \tag{2.61}$$

where $S_{i,\kappa}$ is an integer, and $S_{i,\kappa} \leq \kappa$.

The complex model is similar, and given by:

$$\begin{aligned} c_{\kappa,i} &= 1 \\ s_{\kappa,i} &= 2^{S_{i,\kappa}} + \eta_{i,\kappa} 2^{S'_{i,\kappa}} \quad . \end{aligned} \tag{2.62}$$

where $\eta_{i,\kappa} \in \{-1, 0, +1\}$, $S_{i,\kappa}, S'_{i,\kappa}$ are integer, and $S'_{i,\kappa} \leq S_{i,\kappa} \leq \kappa$. The formulas for $\bar{\alpha}_{\kappa,i}$ and $m_{\kappa,i}$ are analogous to the simple method, but left out for brevity.

We gain an idea of the realization of the floating-point iteration, when we substitute the simple model of Equation (2.61) in the iteration of Equation (2.57), and obtain:

$$\begin{aligned} m_x[i+1] &= m_x[i] - \sigma_i \cdot m_y[i] \cdot 2^{S_{i,\kappa}+e_{y,\text{core}}-e_{x,\text{core}}} \\ m_y[i+1] &= m_y[i] + \sigma_i \cdot m_x[i] \cdot 2^{S_{i,\kappa}+e_{x,\text{core}}-e_{y,\text{core}}} \quad , \end{aligned} \tag{2.63}$$

which we re-write to:

$$\begin{aligned} m_x[i+1] &= m_x[i] - \sigma_i 2^{S_x[i]} m_y[i] \\ m_y[i+1] &= m_y[i] + \sigma_i 2^{S_y[i]} m_x[i] \quad , \end{aligned} \tag{2.64}$$

where we introduce of the **local shifts** for the x, y datapaths, $S_x[i], S_y[i]$, as given by:

$$\begin{aligned} S_x[i] &= S_{i,\kappa} + (e_{y,\text{core}} - e_{x,\text{core}}) \\ S_y[i] &= S_{i,\kappa} - (e_{y,\text{core}} - e_{x,\text{core}}) \end{aligned} \tag{2.65}$$

It is clear from Equation (2.64) that the cost of a floating-point Cordic iteration, for the simple model, is also two shift-add pairs, the same as for a fixed point implementation.

2.5 A Sequential Architecture for the Floating-point Core

In this section, we present a sequential architecture for the floating-point core of the Cordic algorithm. For the implementation of this architecture we have chosen a word-serial approach, similar to [19].

The set of angle bases $\mathcal{A}_\kappa$ —necessary to implement the floating-point Cordic— have been calculated for 12- and 24-bit mantissa sizes. For this purpose, a heuristic search program, `Bangles` has been written. A detailed analysis of the implementation parameters is made, which serve as input to the search program.

A simulation model was built to validate the correct working of the architecture and algorithm, the results of which are presented in Section 2.7.

2.5.1 Practical Implementation

The sequential architecture is shown in Figure 2.4. The mantissa datapath, which forms part of the core unit, is implemented to operate in word-serial fashion. This means that in one cycle, only one micro-rotation is computed. An entire vectoring or rotation operation will therefore take at most $\max\{N_\kappa\}$ cycles plus additional cycles for the scaling. In the construction of the angle bases $\mathcal{A}_\kappa$, we have taken

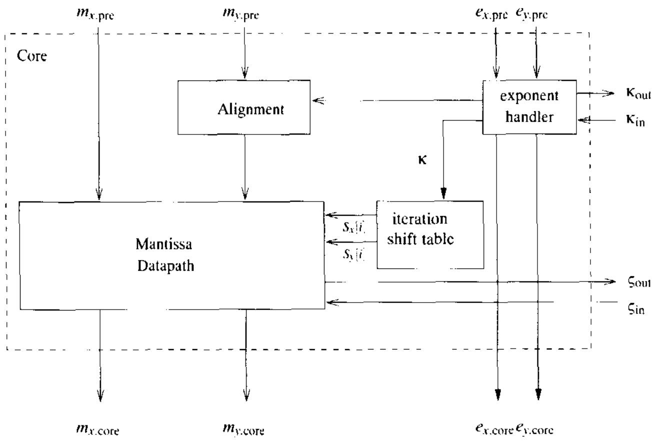


Figure 2.4. Diagram of the sequential floating-point core.

care to keep the N_κ roughly the same and to force the magnification factor to such a value that scaling is performed in one cycle.

The reason for developing a word-serial architecture is two-fold. The obvious reason is of course to save chip area. Since our goal is to produce a floating-point Cordic that works on IEEE 754 single-precision data, requiring a mantissa datapath of 24 bits, a fully parallel approach would yield an impractically large chip area. The second reason is that the floating-point Cordic *algorithm* presented in this chapter lends itself more to a word-serial implementation.

Requirements on the domain of convergence

The largest angle within a region $\mathcal{D}_{e_x,e_y}$ with fixed exponents $(e_{x.\text{pre}}, e_{y.\text{pre}})$ occurs when:

$$\begin{cases} x = (1.00\ldots00)_2 \cdot 2^{e_{x.\text{pre}}} \\ y = (1.11\ldots11)_2 \cdot 2^{e_{y.\text{pre}}} \end{cases} \tag{2.66}$$

Since this angle determines the domain of convergence, a suitable bound for γ_κ, of the angle base $\mathcal{A}_\kappa$ with $\kappa = e_{y.\text{pre}} - e_{x.\text{pre}}$, is given in Equation (2.67).

$$\gamma_\kappa \geq \arctan(2^{\kappa+1}) \tag{2.67}$$

Angle accuracy requirements

The angle resolution δ is determined by the smallest angle in the angle base. There are a number of criteria to determine what the bound is for the necessary accuracy in the angle. We will adopt one which takes into account the given accuracy of the input data, making sure that the error induced by the Cordic computation is less than the error which can be accounted to the N_{mant}-bit accuracy of the mantissa representation. A bound for the minimum angle δ_κ is derived in Appendix 2.A.3. From this bound, we can state the following for the angle resolution , keeping in mind the requirement that $s_{\kappa,i}$ is easy to implement, say, a power of two:

$$\left\{\begin{array}{lcl} s_{\kappa,N-1} & = & 2^{\kappa-N_{\mathrm{mant}}-1} \\ \bar{\alpha}_{\kappa,N-1} & = & \arctan(2^{\kappa-N_{\mathrm{mant}}-1}) \\ S_{N-1,\kappa} & = & \kappa-N_{mant}-1 \end{array}\right. . \qquad (2.68)$$

Magnification factor

We would like to force the magnification factor K_κ to be such that the multiplication with the approximative inverse $\acute{K}_\kappa$ is cheap in hardware. In [18, 20] the magnification factor is forced to 2, using complex or additional micro-rotations. However, this approach will not work for the Floating-point Cordic. If we force $K_\kappa \to 2$ for *every* angle base $\mathcal{A}_\kappa$, then the resulting domain of convergence γ_κ and the number of base angles N_κ will be unnecessary large. This is especially so for angle bases with a small angle exponent κ.

In Appendix 2.A.2 we derive a suitable value for the magnification factor, that is optimal in the sense of combining an easy multiplication of the inverse magnification factor and a minimal number of micro-rotations. For an angle base $\mathcal{A}_\kappa$, with domain of convergence satisfying condition (2.67), the ideal magnification factor turns out to be:

$$K_\kappa \approx (1-2^{2\kappa-1})^{-1} \qquad (2.69)$$

The multiplication with the inverse magnification factor then degenerates to a shift and subtract operation, similar to a Cordic micro-rotation.

The generic angle base

As we can see from Equation (2.69), the magnification factor quickly approaches 1 as the angle exponent κ gets smaller. Hence, after a certain limit, we revert to a *generic* angle base which has a magnification factor of $K_\kappa = 1$ to the required precision and its angle base defined by the template:

$$\left\{\begin{array}{lcl} s_{\kappa,i} & = & 2^{\kappa-i} \\ \bar{\alpha}_{\kappa,i} & = & \arctan(2^{\kappa-i}) \\ S_{i,\kappa} & = & \kappa-i \end{array}\right. . \qquad (2.70)$$

It can be proven that this set of angles spans the required domain of convergence γ_κ as set in Equation (2.67). The proof of the required angle accuracy is given in Appendix 2.A.3.

What remains to be found is the limiting value of κ at which the magnification factor of the angle base, which is given by Equation (2.70), becomes equal to 1 to enough precision. We call this the generic angle base limit L_{gen}. In Appendix 2.A.1 we derive this limit to be:

$$L_{\text{gen}} = \left\lfloor \frac{-N_{\text{mant}} - 1}{2} \right\rfloor \tag{2.71}$$

In practice this means that, for say a 12-bit floating-point Cordic with $L_{\text{gen}} = -7$, the 7 angle bases $\mathcal{A}_0 \ldots \mathcal{A}_{-6}$ must be implemented.

Datapath Accuracy

The accuracy for the core datapath of the Floating-point Cordic follows the guidelines presented in [18] for block floating-point Cordic. The datapath is extended with a guard bit on the left (msb) to prevent overflow and extra bits on the right (lsb) to defy the effect of accumulative round-off errors in the micro-rotations. Simulations have proven these measures to be effective.

Computing the angle bases

Now that the domain of convergence γ_κ and accuracy $\alpha_{\text{min},\kappa}$ are known, the set of angles can be computed that make up the angle base $\{\alpha_i\}_\kappa$.

For this purpose, we have written a heuristic search program `Bangles` that finds an optimal set of angles under the given constraints of domain of convergence, accuracy and magnification factor. Additional parameters to the program are to make use of complex micro rotations and hence have the ability to force the magnification factor to what Equation (2.69) dictates.

The result for a 12-bit floating-point Cordic is presented in Table 2.1.

Similar tables have been calculated for a 24-bit mantissa, as required by the IEEE 754 standard for single precision floating-point numbers.

2.6 A Parallel Architecture for the Floating-point Core

In this section, we present a parallel architecture for the floating-point core of the Cordic algorithm. For the implementation of this architecture we have chosen a pipelined, word-parallel approach, similar to [18].

The set of angle bases $\mathcal{A}_\kappa$ —necessary to implement the floating-point Cordic— have been calculated for 12- and 24-bit mantissa sizes, similar to the sequential architecture.

A simulation model was built to validate the correct working of the architecture and algorithm, the results of which are presented in Section 2.7.

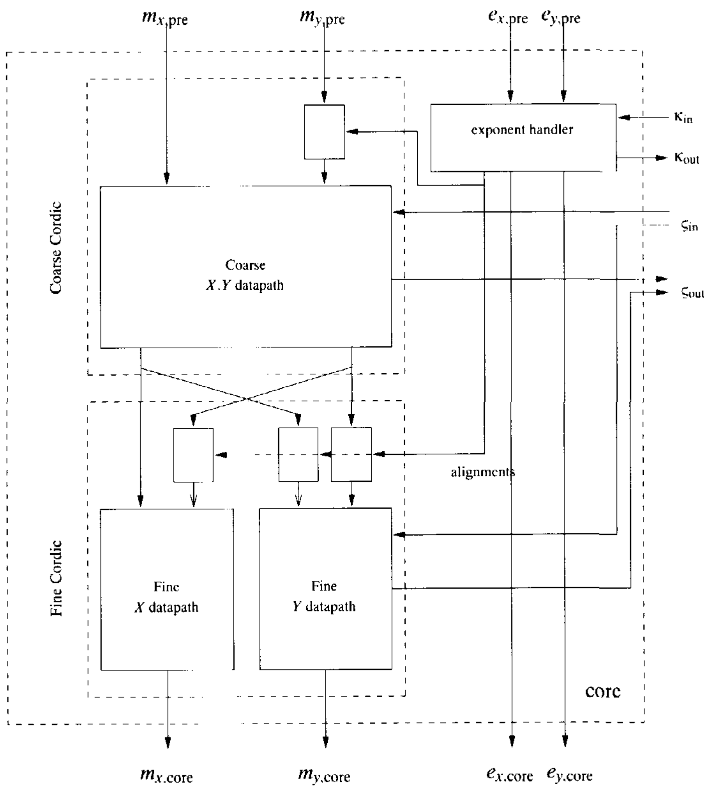


Figure 2.5. Diagram of the parallel floating-point core.

The parallel architecture is shown in Figure 2.5. We place the following constraints on the architecture. These constraints are the result of fine-tuning between algorithm and architecture in order to optimize both for an efficient parallel, pipelineable implementation.

- The micro-rotations take place in block floating-point datapaths that compute on mantissas only, and are devoid of floating-point adders. These datapaths can have different exponents, and thus mantissas have to be re-aligned when transferred from one datapath to another.

- The programmability of the datapath is low. The shifts that are necessary to implement multiplications with a_i in the coarse micro-rotations, to perform division and multiplication in the fine micro-rotations, or for scaling with K^{-1} are *fixed* and *hardwired* in the implementation of the datapath.

- The datapath is minimal in the sense of number of micro-rotations and width of the internal data, under the above constraint of fixed shifts and while still achieving the required relative precision.

This implies that the approach as used in the sequential architecture will fail. Looking at the angle bases $\mathcal{A}_\kappa$ derived for this architecture would require too much flexibility from the core in order to implement everything. Instead, we propose to merge a number of angle bases together, so that one large one that encompasses all is formed, with the domain of convergence of the largest, and the angle resolution of the smallest. This has certain implications on the architecture concerning the size of the datapaths. The proposed angle bases are shown in Table 2.2.

We distinguish four block-floating-point datapaths:

- Two for the coarse Cordic, the x and y datapath, with (fixed) exponents $e_{x.\mathrm{coarse}}$ and $e_{y.\mathrm{coarse}}$.
- Two for the fine Cordic, the x and y datapath, with (fixed) exponents $e_{x.\mathrm{fine}}$ and $e_{y.\mathrm{fine}}$.

Figure 2.5 shows the architecture, showing the block floating-point coarse and fine datapath, and how they fit into their surroundings.

The exponents $e_{x.\mathrm{coarse}}, \ldots, e_{y.\mathrm{fine}}$ are computed such that overflow is controlled and *maximum* relative precision is maintained while having a *minimum* allowable datapath width.

Knowing that $e_{x.\mathrm{pre}} \geq e_{y.\mathrm{pre}}$ from the pre-rotations, we choose:

$$e_{x.\mathrm{fine}} = e_{x.\mathrm{coarse}} = e_{x.\mathrm{pre}} \tag{2.72}$$

and extend the fine and coarse x datapath with 1 msb guard digit for overflow [3].

Similarly, for the fine y datapath, we control overflow by taking:

$$e_{y.\mathrm{fine}} = \max(e_{y.\mathrm{pre}}, e_{x.\mathrm{pre}} + \kappa) \tag{2.73}$$

and extend the fine y datapath with 2 msb guard digits [3].

As for the remaining coarse y datapath, when rotating over a coarse angle for which $\kappa > L$, the exponents of the coarse datapath must be equal to each other. Otherwise, for fine angles with $\kappa \leq L$, the exponent is taken equal to that of the fine y-datapath that follows it.

$$e_{y.\mathrm{coarse}} = \begin{cases} e_{x.\mathrm{coarse}} & \text{if} \quad \kappa > L \\ e_{y.\mathrm{fine}} & \text{if} \quad \kappa \leq L \end{cases} \tag{2.74}$$

As the exponents $e_{x.\mathrm{coarse}}, \ldots, e_{y.\mathrm{fine}}$ of the individual datapaths take on different values, *alignment* of the mantissas is necessary when they transfer from one datapath to the other. We distinguish four alignments.

1. $a_{y,\text{coarse}}$, from the pre-rotations to the y, coarse datapath.
2. $a_{y,\text{fine}}$, from the y, coarse to the y, fine datapath.
3. $a_{x,\text{fixed}}$, from the x, coarse to the y, fine datapath.
4. $a_{y,\text{fixed}}$, from the y, coarse to the x, fine datapath.

The places where alignment occurs are shown in Figure 2.5. The first two alignments are given by:

$$\begin{aligned} a_{y,\text{coarse}} &= e_{y,\text{pre}} - e_{y,\text{coarse}} \\ a_{y,\text{fine}} &= e_{y,\text{coarse}} - e_{y,\text{fine}} \end{aligned} \tag{2.75}$$

The last two are dependent on other factors, their value is given in Equation (2.90). There is no alignment necessary in the x datapath, since $e_{x,\text{fine}} = e_{x,\text{coarse}} = e_{x,\text{pre}}$.

Definition of the Coarse Cordic

The "coarse" Cordic is characterized by:

1. the base angles numbered $\alpha_0, \ldots, \alpha_{N-1}$
2. a magnification factor K_{coarse} The magnification factor K_{coarse} is chosen such that the multiplication with K_{coarse}^{-1} (division by K_{coarse}) is cheap to implement in hardware.
3. an angle domain of convergence, γ_{coarse}, given by:

$$\gamma_{\text{coarse}} = \sum_{i=0}^{N-1} \alpha_i + \delta \tag{2.76}$$

   and satisfying $\gamma_{\text{coarse}} \geq \pi/4$.

4. an angle resolution δ as determined by the minimum angle δcoarse $= \alpha_{n-1}$. This angle resolution δ should satisfy $\delta \leq \gamma_{\text{fine}}$, where γ_{fine} is the angle domain of convergence of the "fine" Cordic.
5. a sufficient datapath width, such that the relative precision in the coordinates of the input vector v_{in} is not affected for $|\alpha| > \gamma_{\text{fine}}$.

The function of the coarse Cordic micro-rotations is to rotate the vector $v_{\text{pre}} = (x,y)_{\text{pre}}$ over an angle α_{coarse} to $v_{\text{coarse}} = (x,y)_{\text{coarse}}$.

In the case that the angle exponent $\kappa \leq L$, indicating that α is already a "fine" angle and $\alpha_{\text{coarse}} = 0$, then the micro-rotations are skipped, irrespective of whether the mode is vectoring or rotation. nevertheless, the y-mantissa is already aligned to the datapath of the fine Cordic that follows this. In this case, the recursion for the mantissas is given by:

$$\begin{aligned} m_{x,\text{coarse}}[i+1] &= m_{x,\text{coarse}}[i] \\ m_{y,\text{coarse}}[i+1] &= m_{y,\text{coarse}}[i] \end{aligned} \tag{2.77}$$

for $i = 0 \ldots N-1$ with initialization

$$\begin{aligned} m_{x,\text{coarse}}[0] &= m_{x,\text{pre}} \\ m_{y,\text{coarse}}[0] &= m_{y,\text{pre}} \cdot 2^{a_{y,\text{coarse}}} \end{aligned} \tag{2.78}$$

Otherwise, for $L < \kappa \leq 0$, the recursion for the coarse micro-rotations is given by:

$$\begin{aligned} m_{x,\text{coarse}}[i+1] &= m_{x,\text{coarse}}[i] - \sigma_i \cdot a_i \cdot m_{y,\text{coarse}}[i] \\ m_{y,\text{coarse}}[i+1] &= m_{y,\text{coarse}}[i] + \sigma_i \cdot a_i \cdot m_{y,\text{coarse}}[i] \end{aligned} \tag{2.79}$$

for $i = 0 \ldots N-1$ with initialization

$$\begin{aligned} m_{x,\text{coarse}}[0] &= K^{-1} \cdot m_{x,\text{pre}} \\ m_{y,\text{coarse}}[0] &= K^{-1} \cdot m_{y,\text{pre}} \cdot 2^{a_{y,\text{coarse}}} \end{aligned} \tag{2.80}$$

In the vectoring mode, the σ_i are determined from

$$\sigma_i = \begin{cases} 1 & \text{if} \quad m_y[i] < 0 \\ -1 & \text{if} \quad m_y[i] \geq 0 \end{cases} \tag{2.81}$$

In the rotation mode, the σ_i are determined from the input angle:

$$\sigma_i = \sigma_{i,\text{in}} \tag{2.82}$$

In either case, the alignment $a_{y,\text{coarse}}$ for the y datapath is given by

$$a_{y,\text{coarse}} = e_{y,\text{pre}} - e_{y,\text{coarse}} \tag{2.83}$$

and the resulting vector v_{coarse} is given by:

$$\begin{aligned} m_{x,\text{coarse}} &= m_{x,\text{coarse}}[N] \\ m_{y,\text{coarse}} &= m_{y,\text{coarse}}[N] \end{aligned} \tag{2.84}$$

Definition of the Fine Cordic

The "fine" Cordic is characterized by:

1. a maximum angle domain of convergence γ_{fine}
2. an angle resolution $\varepsilon_{\text{fine}}$ that varies with the angle exponent, such that the relative precision of the angle is constant and that the precision in the input vector (from which the angle is computed by vectoring) is preserved.
3. no need for scaling, as $K_{\text{fine}} = 1$ to enough precision by definition.
4. a sufficient datapath width, such that the precision of the mantissas is not affected by roundoff errors, and also such that no overflow occurs in computations.

The fine Cordic is not a Cordic as such, but more like a fused multiply-add. It relies on a linearisation of the rotation over α_{fine}, such that $\sin\alpha_{\text{fine}} \rightarrow S$ and $\cos\alpha_{\text{fine}} \rightarrow 1$.

$$R(\alpha_{\text{fine}}) \approx \begin{bmatrix} 1 & -S \\ S & 1 \end{bmatrix} \tag{2.85}$$

The approximation $S = \sin\alpha_{\text{fine}}$ is then written as

$$S = \sum_{i=0}^{M-1} s_i \cdot 2^{-i+h} + \varepsilon_{\text{fine}} \tag{2.86}$$

where the fine micro-rotation quotient bits $s_i \in \{-1, 1\}$, the angle resolution $\varepsilon_{\text{fine}} = \mathcal{O}(2^{h-M})$, and the exponent h defined as

$$\begin{aligned} h &= \begin{cases} L & \text{if} \quad \kappa > L \\ \kappa & \text{if} \quad \kappa \leq L \end{cases} \\ &= \min(L, \kappa) \end{aligned} \tag{2.87}$$

The s_i play a similar role to the σ_i in the coarse micro-rotations.

The recursion for the fine micro-rotations is given by:

$$\begin{aligned} m_{x.\text{fine}}[i+1] &= m_{x.\text{fine}}[i] - s_i \cdot 2^{-i} \cdot m_{y.\text{fixed}} \\ m_{y.\text{fine}}[i+1] &= m_{y.\text{fine}}[i] + s_i \cdot 2^{-i} \cdot m_{y.\text{fixed}} \end{aligned} \tag{2.88}$$

for $i = 0 \ldots N-1$ with initialization

$$\begin{aligned} m_{x.\text{fine}}[0] &= m_{x.\text{coarse}} \\ m_{y.\text{fine}}[0] &= m_{y.\text{coarse}} \cdot 2^{a_{y.\text{fine}}} \\ m_{x.\text{fixed}} &= m_{y.\text{coarse}} \cdot 2^{a_{x.\text{fixed}}} \\ m_{y.\text{fixed}} &= m_{y.\text{coarse}} \cdot 2^{a_{y.\text{fixed}}} \end{aligned} \tag{2.89}$$

The alignments are computed from the differences in the exponents of the individual datapaths, and given by:

$$\begin{aligned} a_{y.\text{fine}} &= e_{y.\text{coarse}} - e_{y.\text{fine}} \\ a_{x.\text{fixed}} &= e_{x.\text{coarse}} - e_{y.\text{fine}} + h \\ a_{y.\text{fixed}} &= e_{y.\text{coarse}} - e_{x.\text{fine}} + h \end{aligned} \tag{2.90}$$

Like for the coarse micro-rotations, the s_i are determined in the vectoring mode from

$$s_i = \begin{cases} 1 & \text{if} \quad m_y[i] < 0 \\ -1 & \text{if} \quad m_y[i] \ddagger 0 \end{cases} \tag{2.91}$$

In the rotation mode, the s_i are determined from the input angle:

$$s_i = s_{i,\text{in}} \tag{2.92}$$

In either case, the resulting vector v_{fine} is given by:

$$\begin{aligned} m_{x.\text{fine}} &= m_{x.\text{fine}}[N] \\ m_{y.\text{fine}} &= m_{y.\text{fine}}[N] \end{aligned} \tag{2.93}$$

2.7 Validation of the algorithm

In an actual implementation of the floating-point Cordic algorithm, there are many possible sources of errors which may influence the overall numerical accuracy of the operations. We identify the following possible sources:

- **finite wordlength computation.**
  All the computational elements in the implementation, such as shifters and adders, have to compute to a given fixed wordlength. This condition must be met in order for the implementation to be realizable in for instance a VLSI circuit. The finite-wordlength introduces round-off errors in the computations.

- **finite parameters.**
  Similar to the problem of finite wordlength representation, the parameters that define the floating-point Cordic have to be finite. For the angle bases this results in a given angle resolution. Also the magnification factor correction has to be done in a fixed number of steps resulting in magnification errors.

- **input data.**
  The input data contributes in two ways to the error. Due to the fixed wordlength representation, there is an inherent representation error in the input data. Secondly, the algorithm is most likely to exhibit a data-dependent behaviour for the error, in which the magnitude of the error depends on the actual value of the data.

It is practically impossible to predict in advance how these errors accumulate through the actual implementation, and hence we must resort to simulations to validate the algorithm. For this purpose, we have built simulation models of the floating-point Cordic algorithm for the implementation on both sequential and parallel architectures.

We have performed extensive simulations with these models, in both modes of the Cordic (vectoring and rotation) and for a large range of possible input data.

We show the absolute and relative error behaviour in the angle-, the x- and the y-components as a result of vectoring a large set of random input vectors. We do this for both a simulation model of a parallel architecture, and that of a sequential architecture. The designed precision for both is 12-bits. Internally, the computations are done to a higher precision, so to avoid accumulation of round-off errors in the operations. For the precision in the angle, we compare the result to that when computed to a much higher precision. In all cases, we assume that the input is exact, that is, we ignore any representation error in the input data. For the y-component, we test how close the final result is to zero, which is the ultimate precision result for vectoring. For the x-component, we compare the result to the length of the input vector, computed to a much higher precision.This also gives us insight into the error introduced in the correction of the magnification.

Let us first discuss the results for the parallel architecture. In Figure 2.6 we present both the absolute and the relative error in the calculation of the angle. We see in Figure 2.6(b) that the relative error stays constant for the smaller angles, which is what was desired. For the classical fixed-point Cordic algorithm, this relative error would increase all the time for smaller and smaller angles, and would eventually exceed the actual value of the angle, at the point when it becomes close to the angle resolution. The relative error becomes very much smaller for the larger angles ($|a|$ ‡ p/4), which is due to the angle representation.

In Figure 2.7, we present the absolute and the relative error in how close the result of the y-component is to zero. We can see that the residue is well below the representation of the numbers in 12-bit accuracy.

In Figure 2.8, we present the absolute and relative errors in the x-component. We see the influence of the error in the correction of the magnification factor. For very small angles, with $|x| >> |y|$ this error decreases in the way shown, since x is assumed to be precise, and is very close to the length for these small angles.

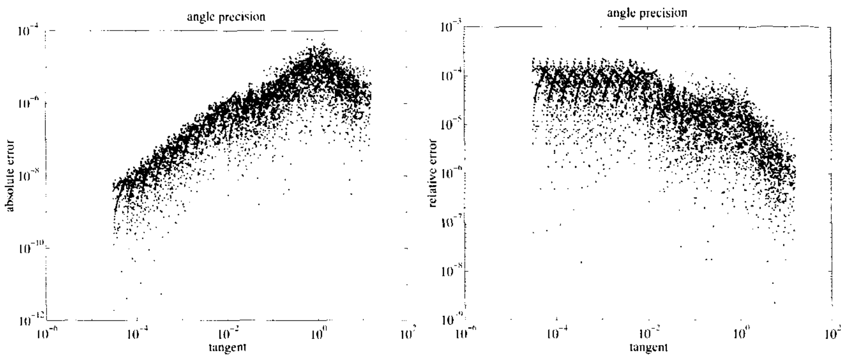


Figure 2.6. absolute and relative error in the approximation of the angle for a 12-bit floating-point Cordic parallel architecture

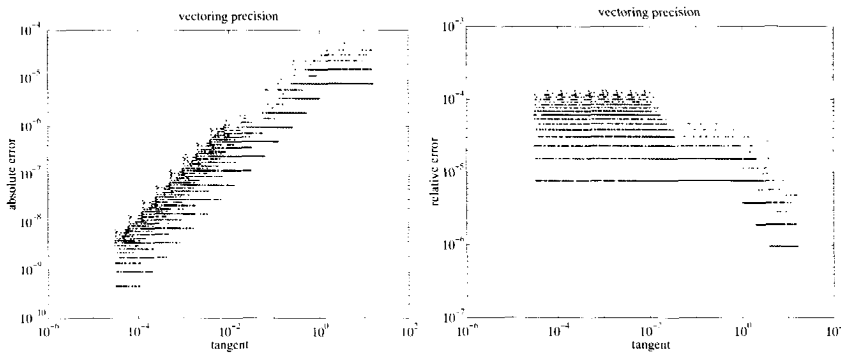


Figure 2.7. absolute and relative error in the y-component for a 12-bit floating-point Cordic parallel architecture

Similarly, In Figures 2.9, 2.10, and 2.11 we show the same results, but then for a sequential architecture. We note that the allover performance is comparable to that of the parallel architecture. Note the difference in the error in the x-component between Figures 2.11 and 2.8. The behaviour for that of the sequential architecture can be explained that we are working with a number of different angle bases, each having their own distinct error in the correction of the magnification factor.

2.8 Discussion

Ercegovac and Lang present a realization of a floating-point Givens rotation in [21], which is not based on Cordic arithmetic. Instead they propose the use of (on-line) scheme to compute the rotation including floating-point square-root computation, division, multiplication and addition. This approach uses more "standard" arith-

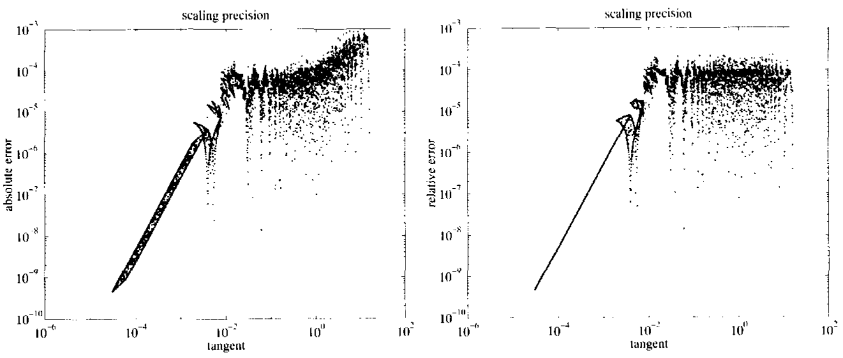


Figure 2.8. absolute and relative error in the magnification for a 12-bit floating-point Cordic parallel architecture

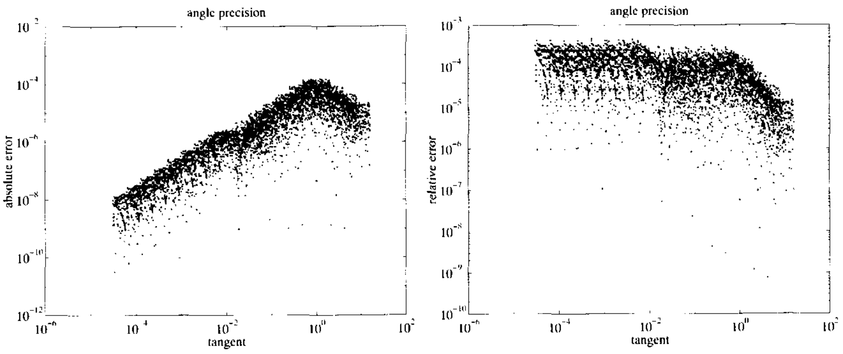


Figure 2.9. absolute and relative error in the approximation of the angle for a 12-bit floating-point Cordic sequential architecture

metic, for which the floating-point arithmetic issues are well known, as opposed to the floating-point Cordic, and computes the result to a similar relative precision. However, the approach is not inherently robust, and knows exceptions, for example when both inputs are zero.

2.9 Conclusions

We have presented a floating-point Cordic Algorithm that calculates angles to full floating-point precision. We have introduced a new floating-point angle representation that preserves the accuracy present in the input data. This representation is inherently more accurate than a fixed-precision floating-point representation. An additional benefit of staying in the proposed angle representation is that the latency between consecutive vectoring and rotation operations is reduced to a single stage.

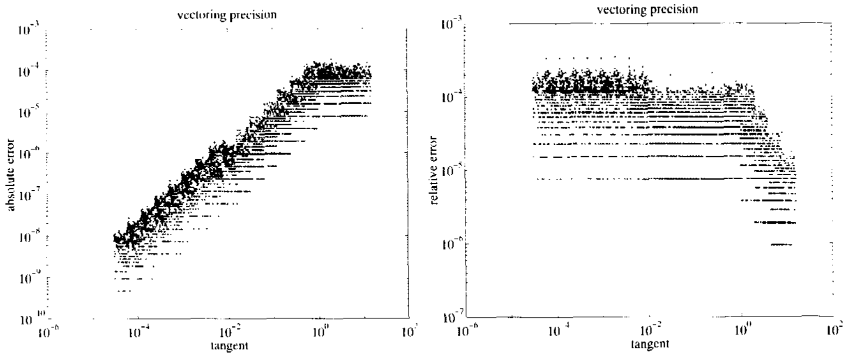


Figure 2.10. absolute and relative error in the y-component for a 12-bit floating-point Cordic sequential architecture

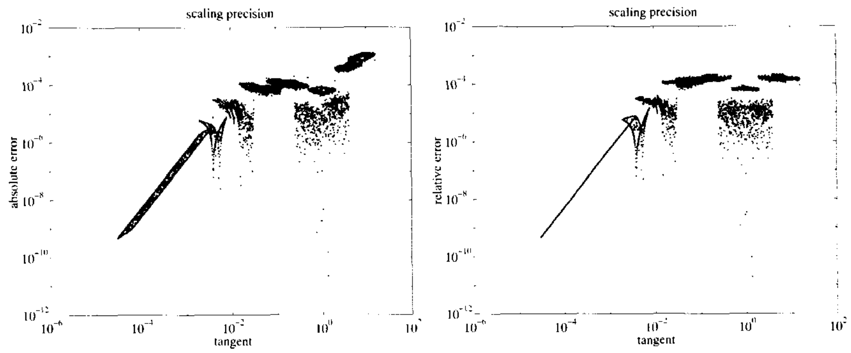


Figure 2.11. absolute and relative error in the magnification for a 12-bit floating-point Cordic sequential architecture

This is analogous to the technique proposed in [22].

We have presented the implementation of this floating-point Cordic algorithm on both a sequential and a parallel architecture. The advantage of the sequential architecture is that it is has a relatively low cost in (VLSI) area, which is traded off for a reduced throughput. When high throughput becomes an issue, we propose a pipelined parallel architecture. Throughput is traded off here against a higher cost of implementation (VLSI area). An added advantage is the use of our proposed floating-point angle representation. By grace of this, the latency between consecutive vectoring and (multiple) rotation operations over the same angle is reduced to a single stage. This causes that they can follow each other with one stage difference in the pipe.

Our experience has been that for less "regular" binary arithmetic algorithms, such as the floating point Cordic, the interplay between algorithm and architecture is quite strong. This means that the choice of architecture can have great consequences on the algorithm. Disregarding this, we have managed to unify the algorithm so that it is still independent on the final target architecture.

We have validated the implementation of the algorithm for both architectures. For this purpose we have built appropriate simulation models. We have examined the behaviour of the error, from the results of extensive simulations, and have shown that it is conform to the designed limits. The floating-point Cordic algorithm has been employed in a MVDR application by Fantini in [6, 7, 8]. Experiments have shown the superiority over Cordic algorithms with a fixed-point angle representation. Concerning other Cordic algorithms that *do* employ a floating-point angle representation, such as Walther's [9], our algorithm has a clear advantage in that it is the only known one for which an efficient implementation on a parallel architecture exists. This becomes a serious issue for when high throughput, high accuracy, and robust numerical computations are required by the application. Such a parallel, pipelined floating-point Cordic architecture has been designed and verified by Looye [23]. It is used as a high throughput robust numerical coprocessor in a larger system.

Appendix

2.A Proofs for this chapter

2.A.1 Derivation of the generic Cordic limit, L_{gen}

The generic angle base $\mathcal{A}_k$, with $k \geq L_{\text{gen}}$, is defined as the angle base with magnification factor $K_k = 1$ to sufficient precision, and where the base angles $\{\alpha_i\}$ given by:

$$\left\{\begin{array}{rcl} S_i & = & -k+i \\ a_i & = & 2^{k-i} \\ \alpha_i & = & \arctan(2^{k-i}) \\ N_k & = & N_{\text{mant}}+1 \end{array}\right. , i \in \{0, 1, \ldots, N_{\text{mant}}\} \tag{2.94}$$

The limit L_{gen} is equal to the transition value of k where the magnification factor K_k becomes equal to $1+\varepsilon$, with the error $\varepsilon < 2^{-N_{\text{mant}}-1}$. This bound comes from the multiplicative nature of scaling with K_k. The error ε must not be larger then the smallest *relative* error, which is $\frac{1}{2}$lsb divided by the maximum value for the mantissa. This works out to be:

$$\begin{aligned} \varepsilon &< \frac{2^{-N_{\text{mant}}}}{2(1-2^{-N_{\text{mant}}})} \\ &< 2^{-N_{\text{mant}}-1} \end{aligned} \tag{2.95}$$

We will now derive an approximation and an upper bound for K_k, and therefore for the error ε, from which we derive the limit L_{gen}.

If we substitute Equation (2.94) in (2.34), we obtain:

$$K_k = \prod_{i=k}^{k-N_{\text{mant}}} (1+2^{2i})^{\frac{1}{2}} \tag{2.96}$$

Using the inequality:

$$(1+2^{k})(1+2^{k-1}) \text{ £ } 1+2^{k+1}. \quad \text{for } k \text{ £ } 0 \tag{2.97}$$

and complete induction we can prove that

$$\begin{aligned} K_k^2 &= \overset{k-N_{\text{mant}}}{\underset{i=k}{}} (1+2^{2i}) \\ &= (1+2^{2k})\dots \\ &\quad (1+2^{2(k-N_{\text{mant}}+1)})(1+2^{2(k-N_{\text{mant}})}) && (2.98) \\ &< 1+2^{2k+1} && (2.99) \end{aligned}$$

Again, continuing with this bound for K_k^2 we derive a less tight bound for K_k following

$$\begin{aligned} K_k^2 &< 1+2^{2k+1} \\ &< 1+2^{(2k)+1}+2^{2(2k)} = (1+2^{2k})^2 && (2.100) \\ K_k &< 1+2^{2k} && (2.101) \end{aligned}$$

Combining this bound and Equation (2.95), we come to:

$$\begin{aligned} 1+2^{2k} &\text{ £ } 1+2^{-N_{\text{mant}}-1} \\ 2k &\text{ £ } -N_{\text{mant}}-1 && (2.102) \end{aligned}$$

and hence:

$$L_{\text{gen}} = \left\lfloor \frac{-N_{\text{mant}}-1}{2} \right\rfloor \tag{2.103}$$

2.A.2 Relation between Domain Of Convergence r and Magnification factor K

We have found that there is a definite relation between the domain of convergence r of an angle base, its magnification factor K, and the number of base angles n.

For a given domain of convergence r there is an angle base with a maximum magnification factor K_{max}, and likewise, to attain a magnification factor of K, there is a lower bound on the domain of convergence q_{min}. Without proof, we state that the maximum value for K occurs when the base angles are chosen according to $a_i = r \cdot 2^{-(i+1)}$. A suitable bound for the magnification factor is then given by:

$$K_{\text{max}} = \overset{¥}{\underset{i=0}{}} \left(\cos \frac{r}{2^{i+1}}\right)^{-1} \tag{2.104}$$

The lower bound to the magnification factor is equal to 1. This bound is attained by taking n base angles of $\alpha_i = r/n$, and then forcing n to infinity.

$$K_{\min} = \lim_{n\to\infty} \prod_{i=0}^{n} \left(\cos\frac{r}{n}\right)^{-1} = \lim_{n\to\infty} \left(\cos\frac{r}{n}\right)^{-n} = 1 \tag{2.105}$$

Likewise, from the above we can prove that for a given K there is no upper bound $\gamma_{\max}$ for the domain of convergence.

Clearly, we do not want an infinite number of base angles. In general we can say that decreasing the magnification factor for a fixed domain of convergence will result in an increase of the number of base angles which is also equal to the number of micro-rotations.

The objectives in our search for an ideal magnification factor K_κ for an angle base with domain of convergence γ_κ satisfying (2.67) are then the following:

1. To come as close as possible to the maximum magnification factor. This will guarantee a minimal number of base angles.
2. To express this magnification factor in such a way that its inverse is a sum or difference of two powers of two. This guarantees a simple implementation of the prescaling function.

To satisfy both objectives we propose that the scaling multiplication takes place as $1-2^{k_{\text{scale}}}$, where the k_{scale} is chosen in such a way that the resulting magnification factor, given by

$$K_\kappa = \left(1-2^{k_{\text{scale}}}\right)^{-1} \tag{2.106}$$

is the closest to $K_{\max}$.

Practical calculations have shown that $k_{\text{scale}} = 2\kappa - 1$ satisfies the above conditions. Hence we will force the magnification factor to be equal to

$$K_\kappa \approx \left(1-2^{2\kappa-1}\right)^{-1} \tag{2.107}$$

with the same relative precision ε as given by condition (2.95).

2.A.3 Derivation of a bound for the angle resolution δ_K

There are a number of criteria to determine the required accuracy of the Cordic. Since most of these amount to the same bound, we adopt one that takes into account the inherent accuracy in the representation of the floating-point input. Let the inherent accuracy ε in the mantissa be given by:

$$\varepsilon = \frac{1}{2}\text{lsb} = 2^{-N_{\text{mant}}}. \tag{2.108}$$

Our reasoning is as follows. If we were to take a Cordic that produces angles which have an error which is larger than can be accounted to the error in the input data, we are obviously not computing to the full extent of the available input data, i.e. we are ignoring least-significant bits. If, on the other hand, we would make a Cordic that is arbitrarily accurate by adding extra micro-rotations beyond the necessary limit, this accuracy would be masked by the error in the representation of the input data. Our goal is, of course, only to perform necessary computations. For this purpose we will derive a bound for the minimum angle δ_K.

The bound for the minimum angle δ_K is given by the difference between the angle β, computed from the full precision input, and the angle $\tilde{\beta}$, computed from the perturbed input.

$$\delta_K \leq \beta - \tilde{\beta} \tag{2.109}$$

The full precision input data will amount to an angle β given by

$$\tan\beta = \frac{m_y \cdot 2^{e_y}}{m_x \cdot 2^{e_x}} \tag{2.110}$$

while the worst-case deviation from this is

$$\tan\tilde{\beta} = \frac{(m_y - \text{lsb}/2) \cdot 2^{e_y}}{(m_x + \text{lsb}/2) \cdot 2^{e_x}} \tag{2.111}$$

The smallest, worst-case difference between these angles occurs when both mantissas are at their maximum: $m_x, m_y = 2 - \text{lsb}$. However, in order to simplify the proof, we will slacken the bound by assuming $m_x, m_y = 2$.

Substituting the above, together with Equations (2.109), (2.110) and (2.111) in the trigonometric equivalence

$$\tan(\beta - \tilde{\beta}) = \frac{\tan\beta - \tan\tilde{\beta}}{1 + \tan\beta\tan\tilde{\beta}} \tag{2.112}$$

results in the following equation for the bound:

$$\tan\delta_\kappa \leq \frac{2^{\kappa-N_{\text{mant}}}}{1+2^{2\kappa}-2^{2\kappa-N_{\text{mant}}-1}+2^{-N_{\text{mant}}-1}} \tag{2.113}$$

When we analyze the denominator, we see that it is less or equal to 2 for $\kappa \leq 0$. Therefore, we can safely assume the following bound for $\tan\delta_\kappa$, keeping in mind that it has to be a power of two for easy implementation.

$$\tan\delta_\kappa = 2^{\kappa-N_{\text{mant}}-1} \tag{2.114}$$

Further analysis shows us that the denominator becomes smaller than 1 at $\kappa \leq L_{\text{gen}}$. This means that, for the generic angle base, we can suffice with:

$$\tan\delta_\kappa = 2^{\kappa-N_{\text{mant}}} \tag{2.115}$$

2.B Examples of angle bases for common sizes

Angle base	$\mathcal{A}_0$	$\mathcal{A}_{-1}$	$\mathcal{A}_{-2}$	$\mathcal{A}_{-3}$	$\mathcal{A}_{-4}$	$\mathcal{A}_{-5}$	$\mathcal{A}_{-6}$	$\mathcal{A}_\kappa$
κ	0	−1	−2	−3	−4	−5	−6	≤ -7
k_{scale}	−1	−3	−5	−7	−9	−11	−13	—
K	2	$\frac{8}{7}$	$\frac{32}{31}$	$\frac{128}{127}$	$\frac{512}{511}$	$\frac{2048}{2047}$	$\frac{8192}{8191}$	1
Cordic iteration	$(S_i\ S'_i\ \eta_i)_\kappa$							$S_{i,\kappa}$
1	(0 4 1)	(1 —)	(3 4 1)	(4 5 1)	(5 7 1)	(6 —)	(6 —)	$-\kappa+0$
2	(1 3 1)	(3 —)	(3 7 1)	(4 —)	(5 —)	(6 —)	(7 —)	$-\kappa+1$
3	(1 —)	(3 —)	(4 6 1)	(5 7 1)	(5 —)	(6 —)	(8 —)	$-\kappa+2$
4	(2 7 -1)	(4 —)	(4 —)	(5 —)	(6 —)	(7 —)	(9 —)	$-\kappa+3$
5	(3 —)	(4 —)	(5 —)	(6 —)	(7 —)	(8 —)	(10 —)	$-\kappa+4$
6	(4 —)	(4 —)	(6 —)	(7 —)	(8 —)	(9 —)	(11 —)	$-\kappa+5$
7	(5 —)	(5 —)	(7 —)	(8 —)	(9 —)	(10 —)	(12 —)	$-\kappa+6$
8	(6 —)	(6 —)	(8 —)	(9 —)	(10 —)	(11 —)	(13 —)	$-\kappa+7$
9	(7 —)	(7 —)	(9 —)	(10 —)	(11 —)	(12 —)	(14 —)	$-\kappa+8$
10	(8 —)	(8 —)	(10 —)	(11 —)	(12 —)	(13 —)	(15 —)	$-\kappa+9$
11	(9 —)	(9 —)	(11 —)	(12 —)	(13 —)	(14 —)	(16 —)	$-\kappa+10$
12	(10 —)	(10 —)	(12 —)	(13 —)	(14 —)	(15 —)	(17 —)	$-\kappa+11$
13	(11 —)	(11 —)	(13 —)	(14 —)	(15 —)	(16 —)	(18 —)	$-\kappa+12$
14	(12 —)	(12 —)	(14 —)	(15 —)	(16 —)	(17 —)	(19 —)	—
15	(13 —)	(13 —)	(15 —)	(16 —)	(17 —)	(18 —)	—	—
16	—	(14 —)	—	—	—	—	—	—

Table 2.1. The series of angle bases $\mathcal{A}_\kappa$ for a 12-bit sequential floating-point Cordic.

Angle base	$\mathcal{A}_\kappa$	$\mathcal{A}_\kappa$
κ	$-6 \le \kappa \le 0$	$\kappa \le -7$
k_{scale}	-1	—
K	2	1
Cordic iteration	$(S_i\ S'_i\ \eta_i)_\kappa$	$S_{i,\kappa}$
1	(0 4 1)	—
2	(1 3 1)	—
3	(1 —)	—
4	(2 7 -1)	—
5	(3 —)	—
6	(4 —)	—
7	(5 —)	—
8	(6 —)	—
9	(7 —)	$-\kappa+0$
10	(8 —)	$-\kappa+1$
11	(9 —)	$-\kappa+2$
12	(10 —)	$-\kappa+3$
13	(11 —)	$-\kappa+4$
14	(12 —)	$-\kappa+5$
15	(13 —)	$-\kappa+6$
16	(14 —)	$-\kappa+7$
17	(15 —)	$-\kappa+8$
18	(16 —)	$-\kappa+9$
19	(17 —)	$-\kappa+10$
20	(18 —)	$-\kappa+11$
21	(19 —)	$-\kappa+12$

Table 2.2. The series of angle bases $\mathcal{A}_\kappa$ for a 12-bit parallel floating-point Cordic.

Bibliography

[1] Gerben J. Hekstra and Ed F. Deprettere, "Floating point Cordic," in *Proceedings ARITH 11*, Windsor, Ontario, June 30 - July 2, 1993 1993.

[2] Gerben Hekstra, "Sequential floating-point Cordic," Tech. Rep., Dept. Electrical Engineering, Delft University of Technology, 1992.

[3] Gerben Hekstra, "Parallel floating-point Cordic," Tech. Rep., Dept. Electrical Engineering, Delft University of Technology, 1993.

[4] G.J. Hekstra and E.F.A. Deprettere, "Cordic algorithms and architectures," European Patent Office, Bulletin 95/01, January 1995, Publication number: 0 632 369 A1, date of filing: June 26, 1993.

[5] Gene H. Golub and Charles F. van Loan, *Matrix Computations*, The John Hopkins University Press, 1983.

[6] Simone P. Fantini, "Matlab program for the MVDR-iQR algorithm," June 1997.

[7] Edwin Rijpkema, Gerben Hekstra, Ed F. Deprettere, and Jun Ma, "A strategy for determining a Jacobi Specific Dataflow Processor," Tech. Rep. ET/CAS/Jacobium-27, TU Delft, May 1997.

[8] Simone P. Fantini, "Application of VLSI CORDIC technology to adaptive processing of radar signals," Tech. Rep. ET/CAS/Jacobium-32, TU Delft, July 1997.

[9] J.S Walther, "A unified algorithm for elementary functions," *proceedings of the AFIPS Spring Joint Computer Conference*, pp. 379–385, 1971.

[10] A.A.J. de Lange, A.J. van der Hoeven, E.F. Deprettere, and J. Bu, "An optimal floating-point pipeline CMOS Cordic processor," in *IEEE International Symposium on Circuits and Systems.*, June 1988.

[11] Joseph R. Cavallaro and Franklin T. Luk, "Floating-point Cordic for matrix computations," in *IEEE International Conference on Computer Design: VLSI in Computers & Processors*, Oct. 1988, pp. 40–42.

[12] Alfons. A. J. de Lange and Ed F. Deprettere, "Design and implementation of a floating-point quasi-systolic general purpose CORDIC rotator for high-rate parallel data and signal processing," in *Proceedings of the 10th Symposium on Computer Arithmetic*, Peter Kornerup and David W. Matula, Eds., June 1991, pp. 272–281.

[13] S. C. Bass, G. M. Butler, R. L. Williams, F. Barlos, and D. R. Miller, "A bit-serial, floating point CORDIC processor in VLSI," 1991, pp. 1165–1168.

[14] Jack. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on electronic computers*, pp. 330–334, Sept. 1959.

[15] Simone P. Fantini, "Simulations on MVDR-iQR algorithm," July 1997.

[16] The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, New York 10017, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985*, July 1985.

[17] Jean-Michel Muller, "Discrete basis and computation of elementary functions," *IEEE Transactions on Computers*, vol. C-34, no. 9, pp. 857–862, September 1985.

[18] Jichun Bu, Ed F. A. Deprettere, and Fons de Lange, "On the optimization of a pipelined silicon Cordic algorithm," in *Signal Processing III: Theories and Applications*, I.T. Young et al., Ed., 1986, pp. 1227–1230.

[19] Gene L. Haviland and Al A. Tuszynski, "A CORDIC arithmetic processor chip," *IEEE journal of solid-state circuits*, vol. SC-15, no. 1, pp. 4–14, Feb. 1980.

[20] Jeong-A Lee, *Redundant CORDIC: Theory and its Applications to Matrix Computations*, Ph.D. thesis, University of California, Los Angeles, 1990.

[21] Miloš D. Ercegovac and Tomás Lang, "On-line scheme for computing rotation factors," *Journal of Parallel and distributed computing*, vol. 5, pp. 209–227, 1988, Year and volume unknown.

[22] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and pipelined VLSI implementations of signal processing algorithms," in *VLSI and Modern Signal Processing*, S.Y. Kung, H.J. Whitehouse, and T. Kailath, Eds., pp. 257–276. Prentice-Hall, 1985.

[23] Alco O. Looye, "Multiport memory and floating-point Cordic pipeline on Jacobium processing elements," 1997, to be published.

Chapter 3

LOW-COST ARITHMETIC METHODS FOR ORTHONORMAL ROTATION

Contents

3.1 Introduction

Fast rotations [1] are arithmetic methods for performing orthonormal rotation at a very low cost in implementation. Related to Cordic[2, 3], they form a viable, low-cost alternative to the more expensive Cordic arithmetic for certain applications. Like Cordic, they maintain the desired properties of robustness and numerical stability. One can say that what fast rotations are for Cordic is the analog of what Canonic Signed Digit (CSD) optimization [4, 5, 6, 7], and the work of Magenheimer et al. [8], are for multiplication.

Fast rotations were introduced at the same time independently by the author [1] and by Götze (as orthonormal μ-rotations) [9] in 1993. Though the above published methods differ, the ideas behind them are the same, and they both belong to the class of fast rotations, as we will show later in this chapter in more detail.

In this chapter, we intend to provide the reader with a comprehensive inventory of the different classes and methods of fast rotations. Partially this includes previous work by the author, published before in [10], partially joint work with Götze [11, 12, 13], as well as previously unpublished work.

Fast rotations have been successfully employed as an alternative to Cordic in a number of applications in signal and image processing and in computer graphics [1, 9, 11, 12, 13, 14, 15, 10, 16, 17, 18]. A selection of these applications are presented in the adjoining Chapters 4 to 6, where it is shown how they can have a significant advantage over other arithmetic techniques. In most cases, their use has led to a reduction of the computational complexity by around one order, in some cases by even more.

3.1.1 Definition of a Fast rotation

A fast rotation over an angle α is given by the matrix $\mathbf{F}$ as:

$$\mathbf{F} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}, \tag{3.1}$$

where (c, s) is a pairwise approximation of a (cosine, sine) pair, with magnification factor m given by

$$m = \sqrt{c^2 + s^2}, \tag{3.2}$$

and rotation angle α given by

$$\alpha = \arctan(\frac{s}{c}), \tag{3.3}$$

satisfying the following fast rotation conditions:

1. **close-to-orthonormal:** The magnification factor m, which is the norm of $\mathbf{F}$, is close to unity, $m = 1 + \varepsilon$, where the error ε falls well below the precision of the number representation. Hence, this error is negligible compared to that of rounding-off the data, making the operation in practice *orthonormal.* We also call ε the *magnification error*, as we desire the magnification to be (close to) unity.

2. **low-cost implementation:** The fast rotation $\mathbf{F}$ is cheap to implement in terms of hardware cost, in the order of a few shift and add operations. This cost is a function of the magnitude of the angle of rotation and of the required precision.

We illustrate this in the following example.

Example 3.1
Let us consider the approximation pair (c, s) *as given by:*

$$\begin{aligned} c &= 1-2^{-9} \\ s &= 2^{-4}-2^{-15} \ . \end{aligned} \tag{3.4}$$

If we calculate the magnification factor according to Equation (3.2), we obtain:

$$\begin{aligned} m &= \sqrt{(1-2^{-9})^2+(2^{-4}-2^{-15})^2} \\ &= \sqrt{1-2^{-8}+2^{-18}+2^{-8}-2^{-18}+2^{-30}} \\ &= \sqrt{1+2^{-30}} \\ &\approx 1+2^{-31} \qquad . \end{aligned} \tag{3.5}$$

Notice how the intermediate products all cancel out, except for 1 *and for* 2^{-30}*, resulting in a magnification error of* $\varepsilon \approx 2^{-31} \approx 0.0000000005$*. This is equivalent to ten decimal places of precision, and hence we can consider this error to be quite small already. We can observe now some aspects of the applicability of this approximation pair for a fast rotation. If we are to work with 24-bit precision numbers, then a magnification error of this size is certainly negligible with respect to the representation error, and the approximation pair satisfies the first condition for fast rotations. If the working precision is for example 64-bit, then the magnification error is no longer negligible, and the first condition no longer satisfied. If, on the other hand, we decrease the working precision to say 12-bits, then it does not make sense to use this approximation pair for a fast rotation —even though the error is negligible—, as the last term* 2^{-15} *in* s *is rendered ineffective, and no longer contributes to forming the precision.*

We can compute the angle of rotation in a similar way, by substituting the approximation pair into Equation (3.3) and obtain:

$$\begin{aligned} \alpha &= \arctan\left(\frac{2^{-4}-2^{-15}}{1-2^{-9}}\right) \\ &\approx \arctan(0.062591) \\ &\approx 0.062510(3.5816\,\mathit{degrees}) \ . \end{aligned} \tag{3.6}$$

Let us now consider the actual implementation. Let the input vector $\mathbf{v}$ *be given as* $\mathbf{v} = [v_x\, v_y]^T$. *Let us define the output vector* $\mathbf{v}' = [v'_x\, v'_y]^T$ *as the result of applying the fast rotation* $\mathbf{F}$ *on the input vector* $\mathbf{v}$.

$$\mathbf{v}' = \mathbf{F}\mathbf{v}\,. \tag{3.7}$$

Writing out Equation (3.7) into its scalar equations, substituting Equations (3.1) and (3.4) results in:

$$\begin{aligned} v'_x &= v_x - v_y \cdot 2^{-4} - v_x \cdot 2^{-9} + v_y \cdot 2^{-15} \\ v'_y &= v_y + v_x \cdot 2^{-4} - v_y \cdot 2^{-9} - v_x \cdot 2^{-15} \ . \end{aligned} \tag{3.8}$$

The multiplications with a power of two translate to shift operations in the implementation. The calculation of the result as given by Equation (3.8) requires 6 shift and 6 add/subtract operations. We prefer to speak in terms of shift-add pairs comprising of two shifts and two adds, and so the cost is 3 shift-add pair operations. A rotation over the same angle and at the same precision, but then using Cordic would require around 36 shift-add pairs. Hence we can conclude that also the second property for fast rotations is satisfied by this approximation pair.

□

There are many ways to implement this particular fast rotation. We show a few important ones in Figure 3.1(a) to (d). We distinguish between a **direct** and a **factored** implementation.

The schemes in Figure 3.1(a), (b) and (c) are examples of direct implementations. Every individual term in Equation (3.8) is apparent in the scheme. The number of shift-add pairs for a direct implementation is hence the same, irrespective of the structure of the scheme. This is the most clear in schemes (a) and (b), where the shift factors have a one to one correspondence to those in Equation (3.8). Of these two, scheme (b) is an implementation with an optimized flattened adder tree, and is most likely to have the least latency of all. However, we are more interested in the schemes with a **cascade** structure, such as schemes (a), (c) and (d). Due to their regularity and repeated structure, they lend themselves better to efficient sequential or pipelined implementations. Scheme (c) uses a cascade shift chain to form

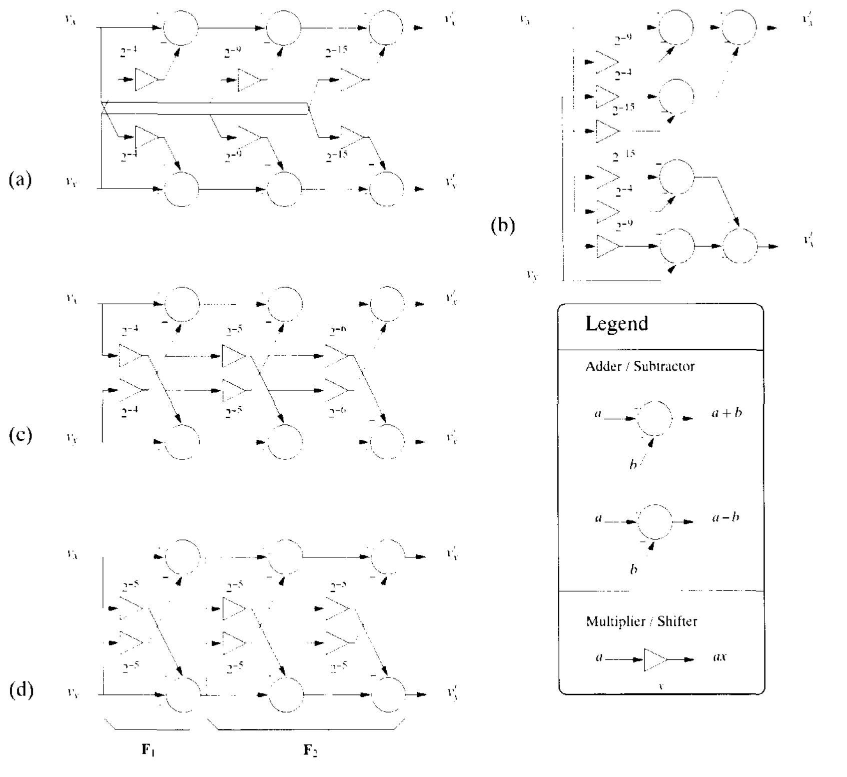


Figure 3.1. Different schemes to implement the fast rotation of Example 3.1

the shifted terms, reusing already shifted results, and hence reducing the overall complexity (area) of the shifters.

The scheme in Figure 3.1(d) is an example of a factored implementation. It is possible to split the fast rotation **F** into a product $\mathbf{F}_2 \cdot \mathbf{F}_1$, where the $\mathbf{F}_i$ are 2×2 matrices, having the same form as in Equation (3.1), though not necessarily fast rotations themselves. For this example, $\mathbf{F}_1$ is realized by the first stage, while $\mathbf{F}_2$ is realized by the latter two stages.

In the case of this particular example, all schemes result in the same cost: three shift-add pairs. In subsequent chapters we will show the superiority of factored implementations for higher order fast rotations.

3.1.2 Taxonomy

We can classify the fast rotations according to the taxonomy shown in Figure 3.2.

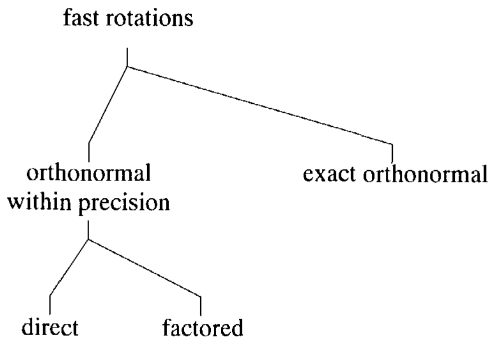


Figure 3.2. Taxonomy of fast rotations

The first dichotomy we see is that of into the class **orthonormal within precision** and the class **exact orthonormal** . The former class is for when there is a magnification error but not significant; $|\varepsilon| > 0$. The latter class is for when there is no magnification error at all; $\varepsilon = 0$.

We pay particular attention to the class **orthonormal within precision** in this chapter. From the taxonomy, we see that this class is split up into two others: those with a **direct** implementation, and those with a **factored** implementation.

- **direct** The direct form implementation is when each term in the approximation pair c, s is explicitly realized as a shift-add operation. The cost of implementation is hence directly linked to the complexity of the approximation pair.

- **factored** Factored fast rotations allow an implementation which is built up out of a sequence of several smaller operations. The total cost of implementation, which is the sum of the cost of the smaller operations, is significantly lower than that of the direct form implementation. In Section 3.4 we present two variations of factored fast rotations: Götze's method of double rotations with a fast scaling sequence [9, 19], and Hekstra's method of extended rotations [10].

Note that these classes are not mutually exclusive, as we have seen from the possible implementations of Example 3.1. A fast rotation with a factored implementation always has a direct form implementation, though the reverse is not always true.

On the other main branch of the taxonomy, there is the class of **exact orthonormal** fast rotations. For circular rotation, these are are trivial rotations over $\frac{k\pi}{2}$. However, for hyperbolic rotation, there are non-trivial solutions, such as Walther's [3] method to extend the range of hyperbolic Cordic. We treat both in more detail in Section 3.5.

We furthermore define a **fast rotation method** to be a systematic method by which to generate the approximation pairs for a number of different fast rotations. These fast rotations are all derived from a common formula —usually parameterized— for the method.

3.1.3 Outline of this chapter

In Section 3.2, we present the first few fast rotation Methods I through to V, and a general method for so-called maximal fast rotations. We discuss the direct form implementation of these methods, in terms of shift-add operations. In order get a better grip on the underlying theory, we introduce a polynomial representation for fast rotations in Section 3.3 and show the relationship between circular and hyperbolic fast rotations. In Section 3.4, we present the class of factored fast rotations. We show how a fast rotation can have a factored form, when it is built up as a sequence of other operations. The total cost of implementation, as the sum of the cost of these many smaller operations, can be significantly lower than that of the direct form implementation. This leads to higher order fast rotations with a low cost. A special class of fast rotations, the exact fast rotations, are treated in Section 3.5. These are fast rotations over relatively large angles, with absolutely no error in orthonormality whatsoever. They exist for both circular and hyperbolic coordinate systems. We show their use in extending the domain of the angle of rotation. Finally, in Section 3.6, we give our conclusions.

3.2 Direct Form Fast Rotation Methods

We present the first few fast rotation methods, along with their properties, and a generalization for a subset of these fast rotations.

For the sake of simplicity in the formulas, we will assume positive rotations in the first quadrant only, with both $c.s > 0$.

3.2.1 Method I

The simplest of the fast rotation methods is the same as the standard Cordic [2] micro-rotation, with the approximation pair given by:

$$\begin{aligned} c &= 1 \\ s &= 2^{\kappa} \ . \end{aligned} \tag{3.9}$$

where the parameter κ is an integer and, due to its nature, is also called the **angle exponent** . Both the magnification factor m and angle of rotation α are a function of this angle exponent and, after substitution of Equation (3.9) into (3.2) and (3.3), are given by:

$$m = \sqrt{1+2^{2\kappa}} \tag{3.10}$$

and

$$\alpha = \arctan(2^{\kappa}) \, . \tag{3.11}$$

Obviously, given a required precision, not all values of the angle exponent κ are suitable. For small enough values of κ, the error in magnification becomes negligible, and hence any additional scaling is unnecessary. We aim to use these rotations only in that domain for which this indeed occurs.

In most cases, we let the finite word length in the computations determine the required precision of the fast rotations. We assume a mantissa size of N_{mant} bits, with the weight of the least significant bit given by $\text{lsb} = 2^{-N_{\text{mant}}}$. When employing rounding-to-nearest, the error due to rounding is given by $\varepsilon_{\text{round}} = \text{lsb}/2$. For rounding down, the error is given by $\varepsilon_{\text{chop}} = \text{lsb}$. For simplicity's sake, we will assume the rounding-down model, and base our results on it. Equivalent results can be found for the round-to-nearest model.

For a fast rotation to be considered orthonormal, the magnification factor m must satisfy:

$$1 - \varepsilon_{\text{chop}} < m < 1 + \varepsilon_{\text{chop}} \, . \tag{3.12}$$

By inspection of Equation (3.10), it follows that $m > 1$, so we need only consider the right side inequality. Substitution of m, and squaring of both sides results in:

$$1 + 2^{2\kappa} < 1 + 2\varepsilon_{\text{chop}} + \varepsilon_{\text{chop}}^{2}. \tag{3.13}$$

Subtracting 1 from both sides, and tightening the condition by removing the $\varepsilon_{\text{chop}}^{2}$ term on the right-hand side, we arrive at:

$$2^{2\kappa} \leq 2\varepsilon_{\text{chop}} < 2\varepsilon_{\text{chop}} + \varepsilon_{\text{chop}}^{2}. \tag{3.14}$$

Substituting $\varepsilon_{\text{chop}}$ by $2^{-N_{\text{mant}}}$, and writing out the left hand inequality of (3.14) in terms of the exponents only, we arrive at the upper bound κ_u of the angle exponent as:

$$\kappa \leq \kappa_u = \frac{-N_{\text{mant}} + 1}{2}. \tag{3.15}$$

This result is also well known from Cordic literature, where the "tail" of micro-rotations with $\kappa \leq \kappa_u$ do not influence the magnitude of the scaling factor any more.

Additionally, we define the lower bound for the angle exponent. Looking at the smallest shift factor [1] which has to be accounted for, we find the term 2^{κ} in c. If this exceeds the width of the mantissa, *i.e.* when $\kappa < -N_{\text{mant}}$, then due to the finite wordlength arithmetic, the argument is shifted beyond the precision, and effectively becomes zero, regardless of its value. This implies that the shift is ineffective for those values of the angle exponent, and may have been skipped. Hence, this leads to the lower bound κ_l for the angle exponent in:

$$\kappa > \kappa_l = -N_{\text{mant}}. \tag{3.16}$$

For values of the angle exponent κ beyond this bound, the fast rotation degrades to the identity operator.

The implementation of the fast rotation is illustrated by the rotation of the vector $\mathbf{v} = [v_x \, v_y]^T$ over the angle α to the vector $\mathbf{v}' = [v'_x \, v'_y]^T$ in:

$$\begin{bmatrix} v'_x \\ v'_y \end{bmatrix} = \mathbf{F} \cdot \begin{bmatrix} v_x \\ v_y \end{bmatrix}. \tag{3.17}$$

Writing out the separate equations for the resulting vector $\mathbf{v}'$, and substituting (3.9) leads to the classical Cordic equation.

$$\begin{cases} v'_x &= v_x - 2^{\kappa} v_y \\ v'_y &= v_y + 2^{\kappa} v_x \end{cases}. \tag{3.18}$$

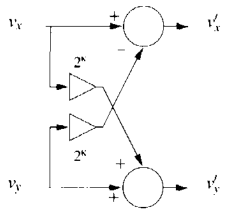


Figure 3.3. Implementation of a Method I fast rotation.

It is plain to see that this operation can be done in direct form with two shift- and two add operations. The implementation is shown in Figure 3.3. We prefer to express the cost of implementation in terms of shift-add pairs , as the computation in (3.18) can be performed pairwise for v'_x, v'_y. This particular fast rotation method hence requires 1 shift-add pair.

3.2.2 Method II

The next fast rotation method is based on the more accurate approximation pair:

$$\begin{aligned} c &= 1-2^{2\kappa-1} \\ s &= 2^{\kappa} \end{aligned} \quad . \tag{3.19}$$

The magnification factor m and rotation angle α follow from substitution of the c, s pair of Equation (3.19) into (3.2) and (3.3), and are given by:

$$m = \sqrt{1+2^{4\kappa-2}} \tag{3.20}$$

and

$$\alpha = \arctan(\frac{2^{\kappa}}{1-2^{2\kappa-1}}) . \tag{3.21}$$

Note that, for the same angle exponent κ, this method produces rotations over angles of similar magnitude, but with a higher precision, in terms of how close m is to unity, as compared to Method I with the approximation pair of Equation (3.9).

Note also that, in writing out m^2 as the sum of the squares of c and s, all terms cancel out except for the lowest power 1 and the highest power $2^{4\kappa-2}$. We call such fast rotations **maximal**, as the maximum number of terms cancel out against each other. The term with the highest power of 2^κ, is the only term remaining that is a function

[1]Based on the assumption that $\kappa \leq 0$, the shift factors in question are all negative. To avoid confusion, we say that a shift or shift factor is the smallest if it is the most negative. The amount of places shifted over is of course the largest, but in an absolute sense.

of the angle exponent κ, and hence the smallest possible error ε is attained. The Method I fast rotations, although a trivial example, are also maximal.

We can use this rotation method, given the same required precision, for larger values of the angle exponent κ than the previous method. The upper bound κ_u for the angle exponent follows from a similar derivation as in Equations (3.12) to (3.16) as:

$$\kappa \leq \kappa_u = \frac{-N_{\text{mant}} + 3}{4} . \tag{3.22}$$

The lower bound κ_l for this method follows from the limit value of the angle exponent κ for which the term $2^{2\kappa-1}$ in c vanishes, given a mantissa size of N_{mant}. This happens for

$$\kappa > \kappa_l = \frac{-N_{\text{mant}} + 1}{2} . \tag{3.23}$$

Note that this lower bound is the same as the upper bound of Method I. At this boundary, the term $2^{2\kappa-1}$ in c vanishes, and the method automatically degrades to the Method I fast rotation.

Concerning the implementation; writing out Equation (3.17), substituting (3.19), leads to:

$$\left\{ \begin{array}{lcl} v'_x & = & v_x - 2^{\kappa} v_y - 2^{2\kappa-1} v_x \\ v'_y & = & v_y + 2^{\kappa} v_x - 2^{2\kappa-1} v_y \end{array} \right. . \tag{3.24}$$

The cost of implementing this fast rotation is 4 shift- and 4 add operations, or two shift-add pairs.

The implementation of the Method II fast rotation is shown in Figure 3.4. It is a cascade realization, which makes use of a shift chain to keep the overall cost of shifters low. This realization is based on the same principles as the one shown in Figure 3.1(c).

3.2.3 Method III

The next fast rotation method, Method III, is yet more accurate, and is given by the approximation pair:

$$\begin{array}{lcl} c & = & 1 - 2^{2\kappa-1} \\ s & = & 2^{\kappa} - 2^{3\kappa-3} \end{array} . \tag{3.25}$$

The magnification factor m and rotation angle α for this method are given by:

$$m = \sqrt{1 + 2^{6\kappa-6}} \tag{3.26}$$

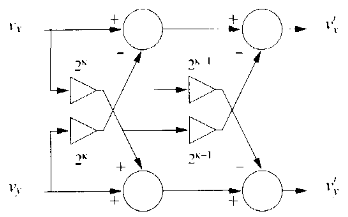


Figure 3.4. Implementation of a Method II fast rotation.

and

$$\alpha = \arctan\left(\frac{2^{\kappa} - 2^{3\kappa-3}}{1 - 2^{2\kappa-1}}\right). \tag{3.27}$$

Note that this is, due to the form of m in Equation (3.26), also a **maximal** fast rotation.

The upper bound κ_u for the angle exponent for this method follows from an analogous derivation as of Equations (3.12) to (3.16) as:

$$\kappa_u = \frac{-N_{\text{mant}} + 7}{6}. \tag{3.28}$$

The lower bound κ_l follows in a similar way as for the previous methods as:

$$\kappa_l = \frac{-N_{\text{mant}} + 3}{3}. \tag{3.29}$$

which shows a slight overlap with the upper bound of the Method II fast rotations.

Concerning the implementation; writing out Equation (3.17), substituting (3.25), leads to:

$$\begin{cases} v'_x &= v_x - 2^{\kappa}v_y - 2^{2\kappa-1}v_x + 2^{3\kappa-3}v_y \\ v'_y &= v_y + 2^{\kappa}v_x - 2^{2\kappa-1}v_y - 2^{3\kappa-3}v_x \end{cases}. \tag{3.30}$$

The implementation of the Method III fast rotation, as a cascade realization with a shift chain, is shown in Figure 3.5. Other schemes to implement this particular fast rotation method are shown in Figure 3.1(a) to (d), including a factored realization. The fast rotation we used in Example 3.1 is also a Method III fast rotation, generated for the angle exponent $\kappa = -4$.

The cost of implementing this fast rotation is three shift-add pairs. Note that the cost in shift-add pairs for the direct form implementation of all the fast rotations is equal to the sum of the number of terms in c and s minus one, to account for the term 1 in c. From the trend of the fast rotation methods presented so far, we can see that a higher accuracy at a fixed order of magnitude of the angle rotation, or a larger angle of rotation at a fixed accuracy both naturally imply a higher cost of implementation.

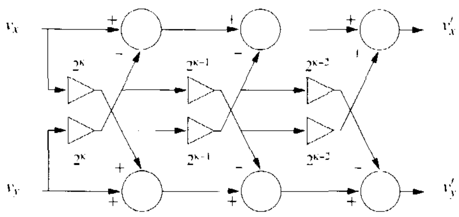


Figure 3.5. Implementation of a Method III fast rotation.

3.2.4 Higher order fast rotations

These first three simple forms of fast rotation have been found by solving for c and s having a fixed number of power-of-two terms, and constraining m so that maximal fast rotations are found. While this is possible for a small number of terms, it becomes unmanageable for higher order fast rotations. Moreover, they do not always exist for certain combinations of the number of terms, as in the case when c and s have three and two terms respectively.

Instead, we propose the Method IV rotation as given by the approximation pair:

$$\begin{aligned} c &= 1 - 2^{2\kappa-1} - 2^{4\kappa-3} \\ s &= 2^{\kappa} - 2^{5\kappa-4} \end{aligned} \quad . \tag{3.31}$$

The magnification factor m and rotation angle α are given by:

$$m = \sqrt{1 + 2^{8\kappa-6} + 2^{10\kappa-8}} \tag{3.32}$$

and

$$\alpha = \arctan\left(\frac{2^{\kappa} - 2^{5\kappa-4}}{1 - 2^{2\kappa-1} - 2^{4\kappa-3}}\right). \tag{3.33}$$

Note that this is *not* a maximal fast rotation as we can see by the form of m in Equation (3.32).

Likewise, we propose the Method V rotation, which again is maximal, as given by the approximation pair:

$$\begin{aligned} c &= 1 - 2^{2\kappa-1} + 2^{4\kappa-3} \\ s &= 2^{\kappa} - 2^{3\kappa-2} + 2^{5\kappa-5} \end{aligned} \quad . \tag{3.34}$$

The magnification factor m and rotation angle α are given by:

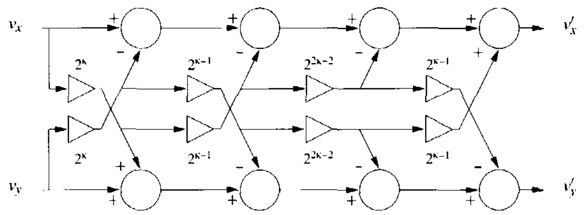
$$m = \sqrt{1 + 2^{10\kappa-10}} \tag{3.35}$$

and

$$\alpha = \arctan\left(\frac{2^{\kappa} - 2^{3\kappa-2} + 2^{5\kappa-5}}{1 - 2^{2\kappa-1} + 2^{4\kappa-3}}\right). \tag{3.36}$$

The implementations of the Method IV and V fast rotations are shown in Figure 3.6(a) and (b).

(a)



(b)

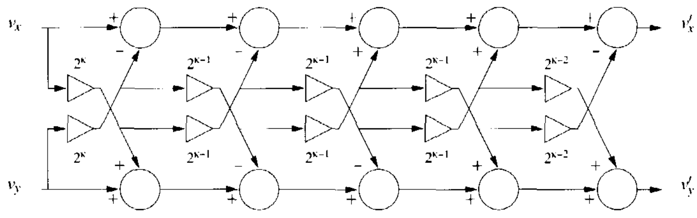


Figure 3.6. Implementations of Method IV and V fast rotations.

A general expression exists for a subset of maximal fast rotations, parameterized in N, with the approximation pair given by:

$$\begin{aligned} c &= 2\sum_{i=0}^{N}(-x^2)^i - 1 \\ s &= x\left(2\sum_{i=0}^{N}(-x^2)^i - (-x^2)^N\right), \end{aligned} \tag{3.37}$$

where the variable x is introduced both for sake of simplicity, and to illustrate the polynomial representation. Substituting $x = 2^{\kappa}$ for $N = 0$, and $x = 2^{\kappa-1}$ for $N \geq 1$ leads to fast rotation methods, expressed as a function of the angle exponent κ.

The previously presented Methods I, III and V are the first three members of this series for $N = 0, 1, 2$ respectively. Note well that the terms in c and s do *not* follow the Taylor series expansion of $\cos(x)$ and $\sin(x)$. The fast rotation methods generated by Equation (3.37) only form a subset of the possible maximal fast rotations. In Section 3.4 we present methods of construction that give a much larger coverage, coupled with a more efficient implementation.

The magnification factor m follows from substitution of c, s of Equation (3.37) into (3.2) and by successive elimination, and is given by

$$\begin{aligned} m &= \sqrt{1-(-x^2)^{2N+1}} \\ &= \sqrt{1+x^{4N+2}} \quad , \end{aligned} \tag{3.38}$$

which shows that the method is indeed maximal.

When implemented in a direct form, the cost of this general method is equal to $(2N+1)$ shift-add pairs. In practice, it does not pay to go beyond Method III $(N=1)$ with direct form implementation, as more efficient *factored* implementation schemes exist for certain higher order fast rotations, as we shall see in Section 3.4.

3.2.5 Evaluation and trade-offs

Let us now look at the trade-offs between the three main parameters of a fast rotation, namely: cost, accuracy, and angle of rotation.

We define the cost L, of a fast rotation as the number of shift-add pair operations required for its implementation. For example, the cost for a Method III fast rotation is $L=3$.

We define the accuracy q, of a fast rotation as how close the magnification m is to unity, or as how small the magnification error ε is. We measure the accuracy as the base 2 logarithm of the magnification error, as given by:

$$q = -\log_2|m-1| = -\log_2|\varepsilon| . \tag{3.39}$$

The accuracy q is an indication for at which bit position, relative to the most significant bit position, the magnification error starts having its influence. The larger the value of q, the higher the accuracy. For the angle of rotation α we make the assumption that the different methods produce almost the same angle of rotation for the same angle exponent κ.

Example 3.2
Let us consider the five methods presented so far, keeping the angle exponent fixed, at for example $\kappa=-4$, which should keep the angle of rotation α close to the same value, and examining the effect the accuracy q and cost L. The cost is of course directly related to the chosen method. The result of this experiment is shown in the table below.

Method	κ	α	ε	q	L
I	-4	$6.2419\cdot10^{-2}$	$1.9512\cdot10^{-3}$	9.001	1
II	-4	$6.2541\cdot10^{-2}$	$1.9073\cdot10^{-6}$	19.000	2
III	-4	$6.2510\cdot10^{-2}$	$4.6566\cdot10^{-10}$	31.000	3
IV	-4	$6.2541\cdot10^{-2}$	$1.8208\cdot10^{-12}$	38.999	4
V	-4	$6.2480\cdot10^{-2}$	$4.4409\cdot10^{-16}$	51.000	5

Note that the angle of rotation for the different methods stays indeed more or less the same.

Similarly, we consider the five methods, and try to tune the angle exponent κ for each, such that the resulting fast rotations have at least the same fixed accuracy, say $q \geq 30$, and then examine the effect on the angle of rotation. The result of this experiment is shown in the table below.

Method	κ	α	ε	q	L
I	-15	$3.0518\cdot10^{-5}$	$4.6566\cdot10^{-10}$	31.0000	1
II	-7	$7.8126\cdot10^{-3}$	$4.6566\cdot10^{-10}$	31.0000	2
III	-4	$6.2510\cdot10^{-2}$	$4.6566\cdot10^{-10}$	31.0000	3
IV	-3	$1.2533\cdot10^{-1}$	$4.6748\cdot10^{-10}$	30.9944	4
V	-2	$2.4868\cdot10^{-1}$	$4.6566\cdot10^{-10}$	31.0000	5

Note that this example is construed in such a way, that the resulting accuracies q of the methods are very close together, $q \approx 31$, for integer values of the angle exponent κ. In this case, this has been done on purpose to accurately show the relation between the angle of rotation α and the cost L. Normally speaking, it would be difficult to find such values for q and κ.[2]

□

Looking at the results of Methods I through to V so far, it is obvious that the following relationships hold:

- For a given minimum required accuracy q, the cost L grows with an increase of the angle of rotation α.
- For a given fixed angle of rotation α, which we achieve by fixing the angle exponent κ, the cost L grows with an increase of the accuracy q.
- For a fixed cost L, which amounts to a fixed method, the accuracy q diminishes with an increase of the angle of rotation α.

[2]The next values for which this happens is when the angle exponent κ takes on the values $(-75, -37, -24, -18, -14)$ for Methods I to V, which results in $q \approx 151$, so an accuracy of 151 bits!

Let us consider the subset of maximal fast rotations, parameterized in N and x (which is a function of the angle exponent κ), as given by Equations (3.37) and (3.38). We will use this to arrive at more precise relationships for the trade-offs. The cost of such a fast rotation, when implemented in the direct form, is given by:

$$L = (2N+1), \tag{3.40}$$

the accuracy follows from substitution of Equation (3.38) in (3.39) as:

$$q = -\log_2 |\sqrt{1+x^{4N+2}} - 1|, \tag{3.41}$$

and the angle of rotation is given by:

$$\alpha = \frac{s}{c} = \frac{x(2\sum_{i=0}^{N}(-x^2)^i - (-x^2)^N)}{2\sum_{i=0}^{N}(-x^2)^i - 1}. \tag{3.42}$$

We apply the replacement rule $\sum_{i=0}^{N} t^i = \frac{1-t^{N+1}}{1-t}$ to Equation (3.42), which results in:

$$\begin{aligned}\alpha &= \arctan\left(x\frac{2\frac{1-(-x^2)^{N+1}}{1+x^2} - (-x^2)^N)}{2\frac{1-(-x^2)^{N+1}}{1+x^2} - 1}\right) \\ &= \arctan\left(x\frac{2-(-x^2)^N - (-x^2)^{N+1}}{1-(-x^2) - 2(-x^2)^{N+1}}\right) \quad .\end{aligned} \tag{3.43}$$

We propose to use approximations for q and α, to simplify the formulas. For small t, the approximation for square-root $\sqrt{1+t} \approx \frac{1}{2}t$ is valid. Applying this to Equation (3.41), we approximate q with $\tilde{q}$, resulting in:

$$\begin{aligned}\tilde{q} &= -\log_2 |\tfrac{1}{2}x^{4N+2}| \\ &= 1-(4N+2)\cdot\log_2(x) \quad .\end{aligned} \tag{3.44}$$

We state that α is a weak function of N, and propose to use the approximation $\tilde{\alpha}$, defined as the limit for α when N goes to infinity, with $x < 1$, as given by:

$$\begin{aligned}\tilde{\alpha} &= \lim_{N\to\infty} \alpha \\ &= \arctan\left(\frac{2x}{1+x^2}\right) \\ &= 2\arctan(x) \quad .\end{aligned} \tag{3.45}$$

This approximation is valid for large N,and $x < 1$ as is the case for higher-order methods. Solving for x in Equation (3.45) results in $x = \tan(\tilde{\alpha}/2)$. Similarly, solving for N in Equation (3.40) results in $N = (L-1)/2$. Substituting then x and N into Equation (3.44), gives us the relationship between the three parameters, which we sought after:

$$\tilde{q} = -2L\log_2(\tan(\frac{\tilde{\alpha}}{2})) + 1\,. \tag{3.46}$$

Using a first order approximation $\tan(t) = t$ for the tangent, and forgetting about the multiplicative or additive constants, we derive the proportionality relationship between cost, accuracy and angle to be:

$$q \propto -L\log(\alpha)\,. \tag{3.47}$$

Note well that this holds for a direct form implementation of the fast rotation. Equation (3.46) is particularly useful as a quick indicator for the cost, accuracy, or angle of rotation for a fast rotation, given that the other parameters are known.

As an illustration, Figure 3.7 shows for which parameters of angle of rotation α and accuracy q the different methods can be applied. The regions in this figure show the domain of application, as determined by the lower and upper bounds κ_l, κ_u on the angle exponent κ. The overlapping of the methods is shown by a dotted line, the cheaper methods overlap the more expensive ones.

In the lower left corner of the plot, there is a region for which performing any rotation is ineffective, the angle is too small to result in any change of the output vector from the input, given the precision. Adversely, in the upper right corner, there is a region where even higher order fast rotations must be used, and where the aforementioned methods no longer suffice.

3.3 Polynomial representation

We have already seen an example of a polynomial representation for the maximal fast rotations of Equation (3.37). We will use a polynomial representation as it facilitates the representation of fast rotations, simplifies the proofs, and allows us to gain more insight into the underlying theory.

Let us define the polynomial approximation pair to be given by two polynomials in x: $c(x)$ and $s(x)$, having similar roles as c and s.

Similarly we define the (polynomial) magnification $m(x)$ and the angle of rotation $\alpha(x)$ to be given by:

$$m(x) = \sqrt{c^2(x) + s^2(x)} \tag{3.48}$$

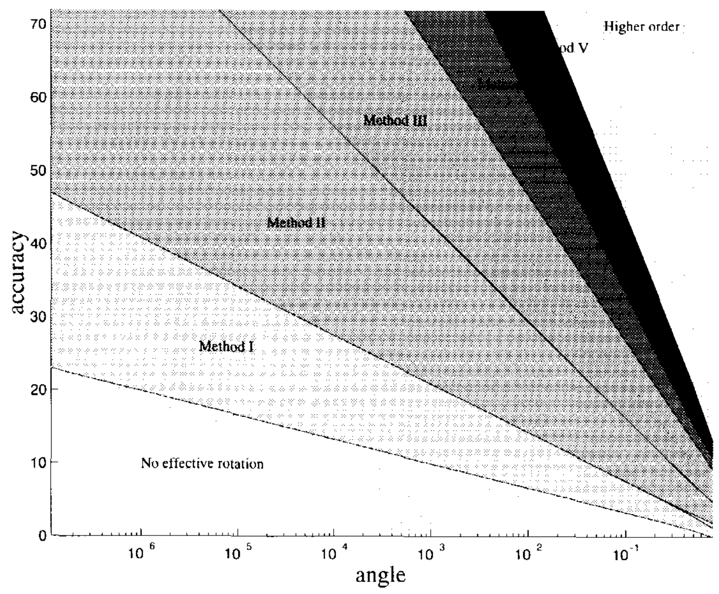


Figure 3.7. Domains of application of the fast rotation Methods I to V.

and

$$\alpha(x) = \arctan(\frac{s(x)}{c(x)}) . \tag{3.49}$$

In the polynomial representation of a fast rotation method, the powers of the angle exponent in 2^{κ} are replaced by powers of the parameter x.

Example 3.3
The polynomial approximation pair for the fast rotation Method I is given by:

$$\begin{aligned} c(x) &= 1 \\ s(x) &= x . \end{aligned} \tag{3.50}$$

Likewise, the polynomial representation of the magnification $m(x)$ and angle of rotation $\alpha(x)$ are given by

$$m(x) = \sqrt{1+x^2} \tag{3.51}$$

and

$$\alpha(x) = \arctan(x) . \tag{3.52}$$

□

The relation between the approximation pair c, s and the polynomials $c(x), s(x)$ for the Method I fast rotations from the above example is given by:

$$\begin{aligned} c &= c(2^{\kappa}) \\ s &= s(2^{\kappa}) . \end{aligned} \tag{3.53}$$

We observe that the polynomials $c(x)$ and $s(x)$ *generate* approximation pairs for the rational points $x = 2^{\kappa}$.

The polynomial approximation pair $c(x), s(x)$ of Equation (3.50) is **canonic**. This means that the term with the lowest power of x in $s(x)$ is x itself. The polynomial approximation pairs that we will work with are not necessary canonic, but can be made to be with a simple transformation. However, we *do* adhere to the convention that the lowest power of x in $s(x)$ must be 1.

Example 3.4
The polynomial representation of the fast rotation Method II is given by:

$$\begin{aligned} c(x) &= 1-2x^2 \\ s(x) &= 2x \\ m(x) &= \sqrt{1+4x^4} \\ \alpha(x) &= \arctan\left(\frac{2x}{1-2x^2}\right) . \end{aligned} \tag{3.54}$$

Note that this representation is on purpose not canonic, so as to avoid fractional coefficients in the polynomials. Equations (3.19) to (3.21) are obtained by substitution of x by $2^{\kappa-1}$.

□

We present the polynomial representation of the fast rotation Methods I through to V in Table 3.1. The realization of these methods is displayed in Figure 3.8.

method	$c(x)$	$s(x)$	$m(x)$	canonic	maximal	substitution
I	1	x	$\sqrt{1+x^2}$	yes	yes	$x = 2^{\kappa}$
II	$1-2x^2$	$2x$	$\sqrt{1+4x^4}$	no	yes	$x = 2^{\kappa-1}$
III	$1-2x^2$	$2x-x^3$	$\sqrt{1+x^6}$	no	yes	$x = 2^{\kappa-1}$
IV	$1-2x^2-2x^4$	$2x-2x^5$	$\sqrt{1+4x^8+4x^{10}}$	no	no	$x = 2^{\kappa-1}$
V	$1-2x^2+2x^4$	$2x-2x^3+x^5$	$\sqrt{1+x^{10}}$	no	yes	$x = 2^{\kappa-1}$

Table 3.1. Polynomial representation of fast rotation Methods I through to V.

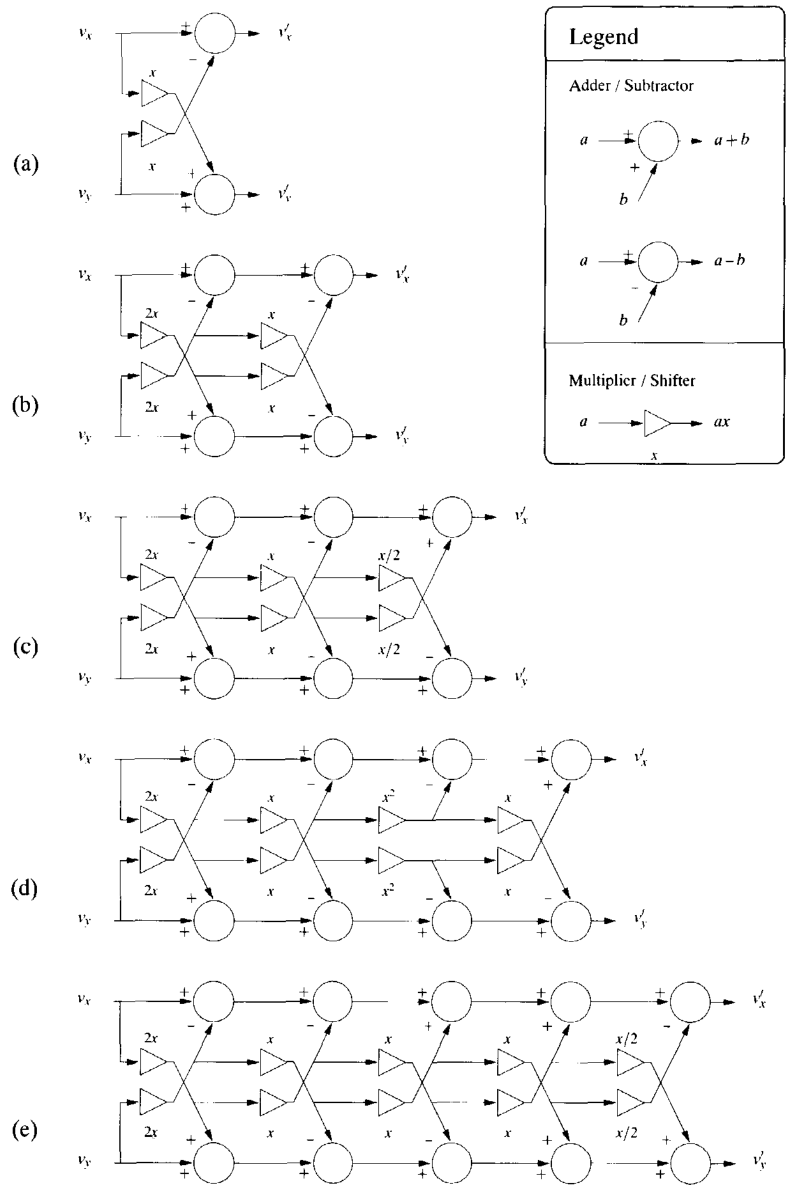


Figure 3.8. Polynomial realization of fast rotation Methods I through to V.

To aid further discussion, we introduce the auxiliary polynomial $U_N(x)$ as being:

$$U_N(x) = \sum_{i=0}^{N} x^i . \tag{3.55}$$

We will use this polynomial extensively in the next sections, both to aid a compact representation, and to simplify proofs, using the set of rules below.

$$\begin{aligned}
(1-x)U_N(x) &= 1-x^{N+1} && (3.56)\\
U_N(x) &= \frac{1-x^{N+1}}{1-x} && (3.57)\\
(1+x^N)U_{N-1}(x) &= U_{2N-1}(x) && (3.58)\\
(1+x^{2^L})U_{2^L-1}(x) &= U_{2^{L+1}-1}(x) && (3.59)\\
U_{2^L-1}(x) &= \prod_{l=0}^{L-1} 1+x^{(2^l)} && (3.60)\\
U_{N-1}(x)U_{M-1}(x^N) &= U_{MN-1}(x) && (3.61)
\end{aligned}$$

Observing Equation (3.37), we replace the summations with instances of the polynomial $U(-x^2)$, and are able to compactly represent the maximal fast rotations as:

$$\begin{aligned}
c(x) &= 2U_N(-x^2)-1\\
s(x) &= x\left(2U_N(-x^2)-(-x^2)^N\right) . && (3.62)
\end{aligned}$$

The formulas for $m(x)$ and $\alpha(x)$ follow by application of the rules as:

$$m(x) = \sqrt{1-(-x^2)^{2N+1}} \tag{3.63}$$

and

$$\alpha(x) = \arctan\left(x\frac{2-(-x^2)^N-(-x^2)^{N+1}}{1-(-x^2)^N-2(-x^2)^{N+1}}\right) . \tag{3.64}$$

Note that Equation (3.62), for the values of the parameter $N = 0, 1, 2$ respectively, gives the polynomial representation of the fast rotation Methods I, III, and V as presented in Table 3.1.

3.3.1 Hyperbolic fast rotations

We define $c_{\mathrm{h}}(x), s_{\mathrm{h}}(x)$ as the **hyperbolic approximation pair**, and the hyperbolic fast rotation $\mathbf{F}_{\mathrm{h}}$ as given by:

$$\mathbf{F}_{\mathrm{h}}(x) = \begin{bmatrix} c_{\mathrm{h}}(x) & s_{\mathrm{h}}(x) \\ s_{\mathrm{h}}(x) & c_{\mathrm{h}}(x) \end{bmatrix}, \tag{3.65}$$

with the hyperbolic magnification factor $m_{\mathrm{h}}(x)$, and the hyperbolic angle of rotation $\alpha_{\mathrm{h}}(x)$, given by:

$$m_{\mathrm{h}}(x) = \sqrt{c_{\mathrm{h}}{}^2(x) - s_{\mathrm{h}}{}^2(x)} \tag{3.66}$$

and

$$\alpha_{\mathrm{h}}(x) = \operatorname{arctanh}(\frac{s_{\mathrm{h}}(x)}{c_{\mathrm{h}}(x)}). \tag{3.67}$$

The relationships between the circular and hyperbolic functions are well known. From [20] they are given for the hyperbolic sine and cosine as:

$$\begin{aligned} \cosh(x) &= \cos(ix) \\ \sinh(x) &= -i\sin(ix) \ , \end{aligned} \tag{3.68}$$

where $i^2 = -1$. A similar relationship holds between circular and hyperbolic fast rotations in:

$$\begin{aligned} c_{\mathrm{h}}(x) &= c(ix) \\ s_{\mathrm{h}}(x) &= -is(ix) \ . \end{aligned} \tag{3.69}$$

Due to the nature of the circular polynomial approximation pairs that we have seen so far, with $c(x)$ and $s(x)$ being even and odd functions respectively[3], the relationship (3.69) results in $c_{\mathrm{h}}(x)$ and $s_{\mathrm{h}}(x)$ having *real* coefficients. Hence the hyperbolic fast rotation $\mathbf{F}_{\mathrm{h}}$ is realizable.

Substitution of the relationships of (3.69) in Equations (3.66) and (3.67) leads to the relationships between the hyperbolic and circular counterparts:

$$\begin{aligned} m_{\mathrm{h}}(x) &= \sqrt{c^2(ix) + s^2(ix)} &= m(ix) \\ \alpha_{\mathrm{h}}(x) &= \operatorname{arctanh}(-i\frac{s(ix)}{c(ix)}) &= -i\alpha(ix) \ . \end{aligned} \tag{3.70}$$

[3]Note well that $c(x)$ and $s(x)$ being even and odd functions is not a natural consequence for fast rotations. There exist polynomial approximation pairs that do not fit this pattern, and hence have no realizable hyperbolic counterpart.

The relationships (3.69) and (3.70) can be applied to obtain the dual, hyperbolic forms of the fast rotation methods presented so far, as well as for most of the fast rotations which are obtained with the techniques described in the next section.

Example 3.5
The hyperbolic fast rotation Method IIIh, given below, follows from application of the above relationships on the circular fast rotation Method III, as given by:

$$\begin{aligned} c_{\mathrm{h}}(x) &= 1+2x^2 \\ s_{\mathrm{h}}(x) &= 2x+x^3 \\ m_{\mathrm{h}}(x) &= \sqrt{1-x^6} \\ \alpha_{\mathrm{h}}(x) &= \operatorname{arctanh}(\tfrac{2x+x^3}{1+2x^2}) \ . \end{aligned} \tag{3.71}$$

□

Similarly, we can apply these relationships to derive the other hyperbolic fast rotations. The polynomial representation of the hyperbolic fast rotation Methods Ih through to Vh are presented in Table 3.2.

method	$c_{\mathrm{h}}(x)$	$s_{\mathrm{h}}(x)$	$m_{\mathrm{h}}(x)$	canonic	maximal	substitution
Ih	1	x	$\sqrt{1-x^2}$	yes	yes	$x=2^{K}$
IIh	$1+2x^2$	$2x$	$\sqrt{1+4x^4}$	no	yes	$x=2^{K-1}$
IIIh	$1+2x^2$	$2x+x^3$	$\sqrt{1-x^6}$	no	yes	$x=2^{K-1}$
IVh	$1+2x^2-2x^4$	$2x-2x^5$	$\sqrt{1+4x^8-4x^{10}}$	no	no	$x=2^{K-1}$
Vh	$1+2x^2+2x^4$	$2x+2x^3+x^5$	$\sqrt{1-x^{10}}$	no	yes	$x=2^{K-1}$

Table 3.2. Polynomial representation of hyperbolic fast rotation Methods Ih through to Vh.

It is evident that a circular fast rotation which is maximal, leads to a hyperbolic counterpart that is also maximal.

Similarly, applying the relationships on Equation (3.62), we arrive at the general expression for the subset of maximal hyperbolic fast rotations:

$$\begin{aligned} c_{\mathrm{h}}(x) &= 2U_N(x^2)-1 \\ s_{\mathrm{h}}(x) &= x\left(2U_N(x^2)-(x^2)^N\right) \ . \end{aligned} \tag{3.72}$$

The corresponding formulas for $m_{\mathrm{h}}(x)$ and $\alpha_{\mathrm{h}}(x)$ follow as:

$$m_{\mathrm{h}}(x) = \sqrt{1-(x^2)^{2N+1}} \, . \tag{3.73}$$

and

$$\alpha_h(x) = \arctan\left(x\frac{2-(x^2)^N-(x^2)^{N+1}}{1-(x^2)^N-2(x^2)^{N+1}}\right). \tag{3.74}$$

3.3.2 Unification of circular and hyperbolic rotations

A unification exists for the circular and hyperbolic fast rotations, quite like the one given for Cordic by Walther [3].

To capture both the circular and hyperbolic modes of rotation in one common formula, we define the circular/hyperbolic parameter t as:

$$t = \begin{cases} +1 & \text{for hyperbolic mode} \\ -1 & \text{for circular mode} \end{cases}. \tag{3.75}$$

and use this parameter in the definition of the unified approximation pair $c_u(t,x), s_u(t,x)$ as:

$$\begin{aligned} c_u(t,x) &= \begin{cases} c_h(x) & \text{for} \quad t=+1 \\ c(x) & \text{for} \quad t=-1 \end{cases} \\ s_u(t,x) &= \begin{cases} s_h(x) & \text{for} \quad t=+1 \\ s(x) & \text{for} \quad t=-1 \end{cases} \end{aligned}. \tag{3.76}$$

The unified fast rotation $\mathbf{F}_u(t,x)$ is in turn defined as:

$$\mathbf{F}_u(t,x) = \begin{bmatrix} c_u(t,x) & t\,s_u(t,x) \\ s_u(t,x) & c_u(t,x) \end{bmatrix}. \tag{3.77}$$

Note that we have purposefully omitted the "linear" mode of rotation, defined in [3] by $t=0$. Whereas the formula of Equation (3.77) would still be correct, it makes no sense to use the typical fast rotation formulas, as the linear case is best handled by other low-cost techniques, such as canonic signed digit (CSD) decomposition [4] or the multiplicative decomposition into a minimum number of CSD terms [8].

If we consider a term in $c(x)$, we can always write it as Cx^{4n+k}, where C is a rational constant, n,k are both integer, and $k \in \{0,1,2,3\}$. Then, applying Equation (3.69) to it, the corresponding term in $c_h(x)$ becomes $C(ix)^{4n+k} = i^k Cx^{4n+k}$. For *even* powers of x, that is when $k \in \{0,2\}$, the term remains real. For $k=0$, it stays the same, while for $k=2$ it changes sign.

Likewise, we can re-write any term in $s(x)$ as Sx^{4n+k}, with S a rational constant. Again, applying Equation (3.69) to it, the corresponding term in $s_h(x)$ becomes $-iS(ix)^{4n+k} = i^{k-1}Sx^{4n+k}$. For *odd* powers of x, when $k \in \{1,3\}$, the term remains real. For $k=1$ it stays the same, while for $k=3$ it changes sign.

Summarizing, for circular approximation pairs with strictly even $c(x)$ and odd $s(x)$, we can obtain the unified approximation pair through the substitution rules presented in Table 3.3 below.

Hence, the methods presented in Tables 3.1 and 3.2 can be combined into their unified counterparts, presented in Table 3.4 below.

k	term in $c(x)$	term in $c_h(x)$	term in $c_u(t,x)$
0	Cx^{4n+0}	Cx^{4n+0}	Cx^{4n+0}
2	Cx^{4n+2}	$-Cx^{4n+2}$	$-tCx^{4n+2}$
k	term in $s(x)$	term in $s_h(x)$	term in $s_u(t,x)$
1	Sx^{4n+1}	Sx^{4n+1}	Sx^{4n+1}
3	Sx^{4n+3}	$-Sx^{4n+3}$	$-tSx^{4n+3}$

Table 3.3. Replacement rules for unified fast rotations.

method	$c_u(t,x)$	$s_u(t,x)$	$m_u(t,x)$
Iu	1	x	$\sqrt{1-tx^2}$
IIu	$1+t2x^2$	$2x$	$\sqrt{1+4x^4}$
IIIu	$1+t2x^2$	$2x+tx^3$	$\sqrt{1-tx^6}$
IVu	$1+t2x^2-2x^4$	$2x-2x^5$	$\sqrt{1+4x^8-4tx^{10}}$
Vu	$1+t2x^2+2x^4$	$2x+t2x^3+x^5$	$\sqrt{1-tx^{10}}$

Table 3.4. Polynomial representation of unified fast rotation Methods Iu through to Vu.

3.4 Factored fast rotation

Referring to Equation (3.47), we have shown that the cost of a direct form implementation grows with the increase of the accuracy and with the increase of the magnitude of the angle of rotation, although very slowly. Inspection of Equations (3.47) and (3.41) shows that the accuracy q is linearly proportional to the cost, and hence the magnification error ε is exponentially proportional to the cost.

In this section we will present factored implementation schemes that perform better in the sense that the cost is even lower to achieve a certain small magnification error for a given angle. These methods aim at factoring a fast rotation into simpler operations, so that the overall cost is kept very low. Much lower in fact, than the direct form implementation. We do not propose to factor any given fast rotation in this way. Rather, we present two variations of parameterized factored fast rotation methods, of which the factorization is known beforehand. We use this to our advantage to construct higher order fast rotations. The two variations of factored fast rotations that we present are Götze's method of double rotations with a fast scaling sequence [9, 19] in Section 3.4.1, and Hekstra's method of extended rotations [10] in Section 3.4.2.

We define a factored realization of a fast rotation $\mathbf{F}$ as the product of N factors $\mathbf{F}_i$, $i = 1.2...N$, as given by:

$$\mathbf{F} = \mathbf{F}_N \cdot \mathbf{F}_{N-1} \cdots \mathbf{F}_2 \cdot \mathbf{F}_1 . \tag{3.78}$$

Whereas we are in principle free to choose the form of the factors $\mathbf{F}_i$, we constrain the $\mathbf{F}_i$ to be 2×2 μ-rotations, with approximation pair c_i, s_i. For the factored fast rotations that we consider in this section, this is sufficient. The individual factors $\mathbf{F}_i$ are given by:

$$\mathbf{F}_i = \begin{bmatrix} c_i & -s_i \\ s_i & c_i \end{bmatrix} . \tag{3.79}$$

The magnification factor m_i and angle of rotation α_i given by:

$$m_i = \sqrt{c_i^2 + s_i^2} , \tag{3.80}$$

and

$$\alpha_i = \arctan \frac{s_i}{c_i} . \tag{3.81}$$

We do not put any constraints on the individual magnification factors, as we did for fast rotations. Only the overall magnification factor m *should* satisfy these constraints.

Using the property (1.9) for the concatenation of μ-rotations, we able to derive the formulas for the overall magnification factor m and the angle of rotation α of $\mathbf{F}$. The magnification factor m is given by the product of the individual magnification factors m_i in:

$$m = \prod_{i=1}^{N} m_i . \tag{3.82}$$

Likewise, the overall angle of rotation α is given by the sum of the individual angles of rotation α_i in:

$$\alpha = \sum_{i=1}^{N} \alpha_i . \tag{3.83}$$

Let us now define $\bar{c}_k, \bar{s}_k$ to be the intermediate approximation pair that corresponds to the intermediate product of the $\mathbf{F}_i$, for $i = 1, \ldots k$, with $k \leq N$. We use property (1.11) to recursively express the intermediate approximation pair as:

$$\begin{aligned} \bar{c}_i &= \bar{c}_{i-1} c_i - \bar{s}_{i-1} s_i \\ \bar{s}_i &= \bar{c}_{i-1} s_i + \bar{s}_{i-1} c_i , \end{aligned} \tag{3.84}$$

for $i \geq 2$, and with initial condition

$$\begin{aligned} \bar{c}_1 &= c_1 \\ \bar{s}_1 &= s_1 . \end{aligned} \tag{3.85}$$

Clearly it follows from the definition that the overall approximation pair c, s of $\mathbf{F}$ is given by the final intermediate approximation pair $\bar{c}_k, \bar{s}_k$, when $k = N$.

$$\begin{aligned} c &= \bar{c}_N \\ s &= \bar{s}_N \ . \end{aligned} \tag{3.86}$$

We will refer to fast rotations that have a factored implementation in a shorthand way as **factored fast rotations** .

3.4.1 Götze's double rotation method

Götze presented, in [9], a factored fast rotation method which was based on a double rotation $\mathbf{F}_1$, followed by $N-1$ scaling stages $\mathbf{F}_i$, with $i = 2, \ldots, N$. The method makes use of the fact that the magnification factor of a double rotation over a simple micro-rotation, such as Method I, no longer has a square-root in the expression, and allows a fast scaling sequence to bring it to any arbitrary precision.

The approximation pair of the first factor $\mathbf{F}_1$ is given by:

$$\begin{aligned} c_1 &= 1 - x^2 \\ s_1 &= 2x \quad . \end{aligned} \tag{3.87}$$

This is a double rotation of Method I, with magnification factor m_1 given by:

$$m_1 = \sqrt{(1+x^2)^2} = 1 + x^2 , \tag{3.88}$$

and angle of rotation α_1 given by:

$$\alpha_1 = \arctan(\frac{2x}{1-x^2}) = 2\arctan(x) . \tag{3.89}$$

Note that there is no longer a square-root in the expression of the magnification factor, and that the angle of rotation is twice that of Method I, which was to be expected.

The approximation pairs of the next factors $\mathbf{F}_i$, with $i \geq 2$, are given in parameterized form as:

$$\begin{aligned} c_i &= 1 + (-x^2)^{(2^{i-2})} \\ s_i &= 0 \quad . \end{aligned} \tag{3.90}$$

Note the absence of the terms s_i, which indicates that this is a pure scaling operation. The magnification factors m_i are given, for $i \geq 2$, by:

$$m_i = c_i = 1 + (-x^2)^{(2^{i-2})} . \tag{3.91}$$

There is no effective rotation, so the angles of rotation α_i are given by:

$$\alpha_i = 0 . \tag{3.92}$$

Looking at the overall behaviour of all N factors $\mathbf{F}_i$, we observe the following. Let us define $m_{2,N}$ as the overall magnification factor of the $\mathbf{F}_i$ with $2 \le i \le N$, so the ones that implement the scaling. As a direct result of the Rule (3.60), it follows that $m_{2,N}$ can be written as:

$$m_{2,N} = U_{2^{N-1}-1}(-x^2) . \tag{3.93}$$

Since we know that the scaling factors do not contribute to the rotation, we can express the overall approximation pair c, s as given by:

$$\begin{array}{rclcl} c & = & (1-x^2)m_{2,N} & = & (1-x^2)U_{2^{N-1}-1}(-x^2) \\ s & = & 2x \cdot m_{2,N} & = & 2x \cdot U_{2^{N-1}-1}(-x^2) \end{array} \tag{3.94}$$

The overall magnification factor m, which is given by the product $m = m_1 m_{2,N}$, follows, as a direct result of Rule (3.56), as given by:

$$\begin{aligned} m &= m_1 m_{2,N} \\ &= (1+x^2)U_{2^{N-1}-1}(-x^2) \\ &= 1-(-x^2)^{(2^{N-1})} \quad . \end{aligned} \tag{3.95}$$

The overall angle of rotation α, is simply given by:

$$\alpha = 2\arctan(x) . \tag{3.96}$$

We can use this factored fast rotation method to construct fast rotations of any arbitrary precision, by supplying enough many scaling stages, *i.e.* by increasing N. The scaling sequence shows quadratic convergence.

The cost of implementation of the first factor $\mathbf{F}_1$ is two shift-add pairs, that of the remaining $\mathbf{F}_i, i \ge 2$ is one shift-add pair each. This results in the overall cost L, parameterized in N, as given by:

$$L = N + 1 . \tag{3.97}$$

Example 3.6
For example, let us consider Götze's factored fast rotation method for length $N = 4$. This is a double rotation stage, followed by three scaling stages.

The fast rotation **F** *is given by the matrix product:*

$$\begin{aligned}\mathbf{F} &= \begin{bmatrix} 1-x^2 & -2x \\ 2x & 1-x^2 \end{bmatrix} \cdot \begin{bmatrix} 1-x^2 & 0 \\ 0 & 1-x^2 \end{bmatrix} \cdot \\ & \begin{bmatrix} 1+x^4 & 0 \\ 0 & 1+x^4 \end{bmatrix} \cdot \begin{bmatrix} 1+x^8 & 0 \\ 0 & 1+x^8 \end{bmatrix} . \end{aligned} \tag{3.98}$$

Written out in full, the approximation pair is given by:

$$\begin{aligned} c &= (1-x^2)U_{2^3-1}(-x^2) \\ s &= 2xU_{2^3-1}(-x^2) \quad , \end{aligned} \tag{3.99}$$

and the magnification factor and angle of rotation are given by:

$$\begin{aligned} m &= 1-(-x^2)^{(2^3)} \\ &= 1-x^{16} \end{aligned} \tag{3.100}$$

and

$$\alpha = 2\arctan(x) . \tag{3.101}$$

The cost of this fast rotation is only 5 shift-add pairs. Compared to the Method V fast rotation, the factored fast rotation has a significantly larger accuracy for a comparable angle of rotation.

□

The cascade realization of the factored fast rotation of Example 3.6 is shown in Figure 3.9.

The accuracy q for this factored fast rotation method, can be derived from Equations (3.39) and (3.95) as given by:

$$\begin{aligned} q &= -\log_2|m-1| \\ &= -\log_2|(-x^2)^{2^{N-1}}| \\ &= -\log_2(x^{(2^N)}) \\ &= -2^N\log_2(x) \quad . \end{aligned} \tag{3.102}$$

Via Equations (3.96), (3.97), and (3.102) we can derive the trade-off relation between angle, accuracy and cost to be given by:

$$q = -2^{L-1}\log_2(\tan(\frac{\alpha}{2})) . \tag{3.103}$$

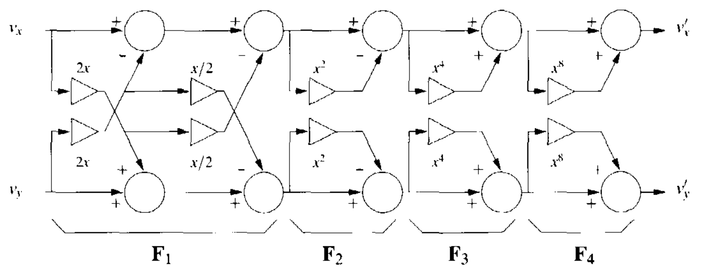


Figure 3.9. Cascade polynomial realization of Götze's double rotation method

We can simplify the above, to reveal the proportionality between the trade-off factors, as given by:

$$q \propto -2^L \log(\alpha) . \tag{3.104}$$

Note that, compared to Equation (3.47) this is clearly a superior method to direct form implementations, but only for L large enough.

3.4.2 Hekstra's extended rotation method

Hekstra presented the extended factored fast rotation method in [10], where each of the factors $\mathbf{F}_i$ contribute to the overall rotation, and their individual magnification factors m_i compensate out, to form a highly accurate fast rotation. This is in contrary to Götze's method, where only the first factor contributes to the rotation. This new method provides extra flexibility in the construction of fast rotations, as the different choices of the direction of rotation for the individual rotation factors result in different overall angles of rotation. Again it holds that, by increasing the number of factors N, an arbitrary high accuracy can be reached.

We will demonstrate one type of such factored fast rotation methods only. Many other types exist, but they all follow the same kind of construction that we will show for this particular case. See [10] for more details. The approximation pair of the first factor $\mathbf{F}_1$ is given by:

$$\begin{aligned} c_1 &= 1 \\ s_1 &= x \ . \end{aligned} \tag{3.105}$$

This is nothing more than a Method I rotation, with magnification factor m_1 and angle of rotation α_1 as given by:

$$m_1 = \sqrt{1+x^2}, \tag{3.106}$$

and

$$\alpha_1 = \arctan(x)\,. \tag{3.107}$$

The remaining factors $\mathbf{F}_i$, with $i = 2,\ldots,N$,are given in parameterized form as:

$$\begin{aligned} c_i &= 1 - x^{(2\cdot 3^{i-2})} \\ s_i &= \eta_i x^{(3^{i-2})} \end{aligned} \quad , \tag{3.108}$$

where the parameter $\eta_i \in \{-1,+1\}$ indicates the direction of rotation of the factor $\mathbf{F}_i$.

The magnification factors m_i, for $i \geq 2$, are given by:

$$m_i = \sqrt{1 - x^{(2\cdot 3^{i-2})} + x^{(4\cdot 3^{i-2})}}\,. \tag{3.109}$$

Note that the parameter η_i is no longer present in the magnification factor. The structure of this magnification factor may look daunting, but it has been constructed so that the overall magnification factor m is accurate and also maximal. Let us define $m_{1,2}$ as the product of the first two magnification factors, $m_{1,2} = m_1 m_2$. Writing this out in full results in:

$$\begin{aligned} m_{1,2} &= m_1 m_2 \\ &= \sqrt{1+x^2}\sqrt{1-x^2+x^4} \\ &= \sqrt{1+x^6} \end{aligned} \quad , \tag{3.110}$$

which shows a three-fold increase in accuracy. By means of induction, we can prove that the overall magnification factor m, for N factors, is given by:

$$m = \sqrt{1 + x^{(2\cdot 3^{N-1})}}\,. \tag{3.111}$$

The angles of rotation α_i of the remaining factors $\mathbf{F}_i$, with $i = 2,\ldots,N$,are given in parameterized form as:

$$\begin{aligned} \alpha_i &= \arctan\left(\frac{\eta_i x^{(3^{i-2})}}{1 - x^{(2\cdot 3^{i-2})}}\right) \\ &= \eta_i \arctan\left(\frac{x^{(3^{i-2})}}{1 - x^{(2\cdot 3^{i-2})}}\right) \\ &= \eta_i \bar{\alpha}_i \end{aligned} \quad . \tag{3.112}$$

where $\bar{\alpha}_i$ is given by:

$$\bar{\alpha}_i = \arctan\left(\frac{x^{(3^{i-2})}}{1-x^{(2\cdot 3^{i-2})}}\right). \tag{3.113}$$

By means of the above simplification, we can write the overall angle of rotation α as given by the summation:

$$\alpha = \alpha_1 + \sum_{i=2}^{N} \eta_i \bar{\alpha}_i . \tag{3.114}$$

Note that the overall angle α depends on the choice of the direction parameters η_i. For practical reasons we have to fix $\eta_2 = +1$, otherwise we have that the first two factors combine to implement a Method I rotation, but at a much greater cost than necessary.

There is no nice, closed expression for the approximation pair c, s, as these too depend on the choice of the η_i. The general formulas for factored fast rotations, as given in Equations (3.84) to (3.86) still hold, of course.

The cost of the first factor $\mathbf{F}_1$ of this particular factored fast rotation is one shift-add pair. The cost for each of the remaining $\mathbf{F}_i$, the so called extensions, is two shift-add pairs. Hence, the overall cost L, parameterized in the number of factors N, is given by:

$$L = 2N - 1 . \tag{3.115}$$

Example 3.7
We illustrate this factored fast rotation method for $N = 3$.

The fast rotation $\mathbf{F}$ is given by the matrix product:

$$\mathbf{F} = \begin{bmatrix} 1 & -x \\ x & 1 \end{bmatrix} \cdot \begin{bmatrix} 1-x^2 & -x \\ x & 1-x^2 \end{bmatrix} \cdot \begin{bmatrix} 1-x^6 & \eta_3 x^3 \\ \eta_3 x^3 & 1-x^6 \end{bmatrix} . \tag{3.116}$$

Note that we have fixed $\eta_2 = +1$. Two different fast rotations result still for $\eta_3 \in \{-1, +1\}$. In general, with $N \geq 2$ factors, we obtain 2^{N-2} different fast rotations.

The overall magnification factor angle of rotation are given by:

$$\begin{aligned} m &= m_1 m_2 m_3 \\ &= \sqrt{1+x^2}\sqrt{1-x^2+x^4}\sqrt{1-x^6+x^{12}} \\ &= \sqrt{1-x^{18}} \end{aligned} \tag{3.117}$$

and

$$\begin{aligned}\alpha &= \alpha_1 + \bar{\alpha}_2 + \eta_3\bar{\alpha}_3 \\ &= \arctan(x) + \arctan(\tfrac{x}{1-x^2}) + \eta_3 \arctan(\tfrac{x^3}{1-x^6}) \ . \qquad (3.118)\end{aligned}$$

The cost of this fast rotation is only 5 shift-add pairs. Again, compared to both Method V and Götze's factored fast rotation, this rotation has a significantly larger accuracy for a comparable angle of rotation.

□

The cascade realization of the factored fast rotation of Example 3.7 is shown in Figure 3.10.

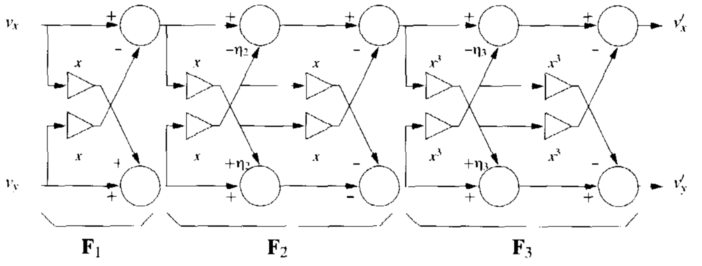


Figure 3.10. Cascade polynomial realization of Hekstra's extended rotation method.

The accuracy q for this factored fast rotation method can be approximated only, and follows by:

$$\begin{aligned}q &= -\log_2|m-1| \\ &= -\log_2|\sqrt{1+x^{(2\cdot 3^{N-1})}}-1| \\ &\approx -\log_2(\frac{1}{2}x^{(2\cdot 3^{N-1})}) \\ &\approx -2\cdot 3^{N-1}\log_2(\frac{1}{2}x) \qquad . \qquad (3.119)\end{aligned}$$

It is difficult to find a closed formula for the trade-offs, since the angle is dependent on the directions η_i. However, for fixed $\eta_i = +1$ which signifies the maximum angle of rotation, we can use a similar approximation for the angle of rotation as in Equation (3.45), given by:

$$\alpha \approx 2\arctan(x)\,. \tag{3.120}$$

Via Equations (3.120), (3.115), and (3.119) we can derive the trade-off relation between angle, accuracy and cost to be given by:

$$q \approx -2 \cdot 3^{\frac{L-1}{2}} \log_2(\frac{1}{2}\tan(\frac{\alpha}{2}))\,. \tag{3.121}$$

The proportionality between angle, accuracy, and cost is derived by simplifying the above, resulting in:

$$q \propto -\sqrt{3}^L \log(\alpha)\,. \tag{3.122}$$

Note that, though Equation (3.104) compared to (3.122) suggests that Götze's method provides more efficient fast rotations, this is only so for very high accuracy. For practical values, Hekstra's method is favourable.

3.5 Exact orthonormal rotation

We define the **exact orthonormal rotations** as the class of fast rotations, for which the special condition

$$\varepsilon = 0 \tag{3.123}$$

holds. They introduce no scaling error. For the circular mode, the members of this class are trivial. For the hyperbolic mode, however, they are non-trivial, as we shall see further on.

3.5.1 Circular mode exact orthonormal rotations

For the circular mode of rotation, the approximation pair must satisfy the condition:

$$c^2 + s^2 = 1\,, \tag{3.124}$$

and simultaneously satisfy the other conditions on fast rotations, such as being cheap to realize. For binary arithmetic, these conditions naturally imply that both c and s must be integer multiples of a power of two. This means that we can write them as:

$$\begin{aligned} c &= X/2^N \\ s &= Y/2^N\,, \end{aligned} \tag{3.125}$$

where $X.Y.N$ are all integer. Substituting (3.125) into Equation (3.124) leads to the Diophantine equation:

$$X^2+Y^2=4^N. \quad X.Y.N \text{ integer}. N \geq 0. \tag{3.126}$$

which only has the trivial solutions:

$$\begin{cases} X &= \pm 2^N \\ Y &= 0 \end{cases} \tag{3.127}$$

and

$$\begin{cases} X &= 0 \\ Y &= \pm 2^N \end{cases} . \tag{3.128}$$

and no others. The proof of this is given in the appendix. A similar proof was also given in [21, 22]. Substituting the solutions (3.127) and (3.128) of the Diophantine equation back into Equation (3.125) results in the approximation pairs for the circular mode exact rotations:

$$\begin{aligned} c &= \pm 1 \\ s &= 0 \end{aligned} \tag{3.129}$$

and

$$\begin{aligned} c &= 0 \\ s &= \pm 1 \end{aligned} . \tag{3.130}$$

Note that the approximation pair of Equation (3.129) corresponds to the well-known, albeit trivial, rotations over $\alpha = 0$ or $\alpha = \pi$, while that of Equation (3.130) corresponds to rotations over $\alpha = \pm\frac{\pi}{2}$.

These rotations are cheap to realize, they require no additions, only negation. Furthermore, they are **accuracy-preserving**, as no error is introduced in the result for finite-wordlength computations.

3.5.2 Hyperbolic mode exact orthonormal rotations

For the hyperbolic mode of rotation, the approximation pair must satisfy the condition:

$$c_h^2 - s_h^2 = 1. \tag{3.131}$$

and simultaneously satisfy the other conditions on fast rotations. Again, we write the approximation pair in terms of integer multiples of powers of two in:

$$\begin{aligned} c_h &= X/2^N \\ s_h &= Y/2^N \end{aligned} . \tag{3.132}$$

where X, Y, N are all integer. Due to the nature of the hyperbolic functions, with $\cosh(x) \geq 0$, and $|\sinh(x)| \leq \cosh(x)$ we impose the following constraints:

$$\begin{aligned} X &\geq 0 \\ X &\geq |Y| \,. \end{aligned} \tag{3.133}$$

Substituting (3.132) into Equation (3.131) leads to the Diophantine equation for the hyperbolic mode:

$$X^2 - Y^2 = 4^N, \quad X, Y, N \text{ integer}, N \geq 0, \tag{3.134}$$

Contrary to the circular mode, this equation has non-trivial solutions, given by:

$$\begin{cases} X = (\frac{2^{N_1}+2^{N_2}}{2}) \\ Y = \pm(\frac{2^{N_1}-2^{N_2}}{2}) \end{cases}, \tag{3.135}$$

where N_1, N_2 are integer, and satisfy $N_1 + N_2 = 2N$. The proof of this is given in the appendix. Substituting the solution (3.135) of the Diophantine equation back into Equation (3.132), and defining $M = N_1 - N$ with M integer, results in the approximation pair for the hyperbolic mode exact rotations:

$$\begin{aligned} c_h &= (\frac{2^M+2^{-M}}{2}) \\ s_h &= \pm(\frac{2^M-2^{-M}}{2}) \,. \end{aligned} \tag{3.136}$$

The angle of rotation, $\alpha_h = \text{arctanh}(s_h/c_h)$, is found by rewriting the terms $2^{\pm M}$ in Equation (3.136) as natural exponents $e^{\pm M \ln 2}$. Substituting this into the equation for the angle of rotation results in:

$$\alpha_h = \text{arctanh}\left(\pm\frac{e^{M\ln 2} - e^{-M\ln 2}}{e^{M\ln 2} + e^{-M\ln 2}}\right), \tag{3.137}$$

which simplifies to:

$$\alpha_h = \pm M \ln 2. \tag{3.138}$$

Note that these exact hyperbolic rotations are the same as proposed by Walther [3] to extend the range of hyperbolic rotation for the Cordic algorithm.

3.6 Conclusions

In this chapter, we have presented the theoretical background and a comprehensive inventory of methods for fast rotations. Fast rotations are arithmetic methods for performing orthonormal rotation at a very low cost in implementation, as an alternative to Cordic. Although fast rotations exist only for certain angles of rotation, they form a sufficient set to efficiently implement any orthogonal operation.

We have presented a taxonomy for the classes of fast rotations, and the significant methods from each class. We have presented a polynomial representation for fast rotation methods. The representation facilitates the understanding of the underlying theory, and provides a link to other disciplines. We have shown the relationship between the circular and the hyperbolic fast rotation methods, and how they can be unified into one model. We have shown the trade-offs that exist between the angle of rotation, accuracy, and cost of fast rotation methods.

In the adjoining chapters, we will present a number of applications that have made use of fast rotations.

Appendix

3.A Exact orthonormal rotation

3.A.1 Circular mode exact orthonormal rotations

THEOREM 3.1
The Diophantine equation:

$$X^2 + Y^2 = 4^N, \quad X, Y, N \text{ integer}, N \geq 0, \tag{3.139}$$

only has the trivial solutions:

$$\begin{cases} X &= \pm 2^N \\ Y &= 0 \end{cases} \tag{3.140}$$

and

$$\begin{cases} X &= 0 \\ Y &= \pm 2^N \end{cases} . \tag{3.141}$$

PROOF

For $N = 0$, it follows that the only solutions are the trivial ones:

$$\begin{cases} X &= \pm 1 \\ Y &= 0 \end{cases} \tag{3.142}$$

and

$$\begin{cases} X &= 0 \\ Y &= \pm 1 \end{cases} . \tag{3.143}$$

Note that $1 = 2^0$, and hence the theorem holds for $N = 0$.

For $N > 0$, the right-hand term of (3.139) is even, and hence we state that if X is odd, then Y is odd, and likewise if X is even, then Y is even too.

For X,Y odd, we make use of the well-known and easily verified fact that an odd square is always a multiple of four plus one. Hence we can write:

$$\begin{aligned} X^2 &= 4L+1 \\ Y^2 &= 4M+1 \ , \end{aligned} \tag{3.144}$$

where L,M integer. Substitution of (3.144) into (3.139) leads to a disagreement, since the left-hand term cannot be a multiple of four, while the right-hand term is, for $N > 0$. From this we can conclude that there are no solutions for X,Y odd.

For the case where both X,Y are even we can perform a problem reduction. We introduce the relationships $X = 2X', Y = 2Y', N+1 = N'$, with X',Y',N' integer and $N' \geq 0$. Substitution of this into (3.139) results in:

$$X'^2 + Y'^2 = 4^{N'}, \quad X',Y',N' \text{ integer}, N' \geq 0, \tag{3.145}$$

which is of the exact same form as the original (3.139). We can repeat this process until we end up with a situation where either $N = 0$, resulting into the known trivial solution, or $N > 0$ and both X,Y odd, with no solutions.

From induction of (3.145) it follows that, for $N > 0$, and X,Y both even, the only solutions are those given by Equations (3.140) and (3.141). Hence the theorem holds.

3.A.2 Hyperbolic mode exact orthonormal rotations

THEOREM 3.2
The Diophantine equation:

$$X^2 - Y^2 = 4^N, \quad X,Y,N \text{ integer}, N \geq 0, \tag{3.146}$$

has the non-trivial solutions:

$$\left\{ \begin{aligned} X &= \eta\left(\frac{2^{N_1}+2^{N_2}}{2}\right) \\ Y &= \nu\left(\frac{2^{N_1}-2^{N_2}}{2}\right) \end{aligned} \right. , \tag{3.147}$$

with N_1,N_2 integer, satisfying $N_1 + N_2 = 2N$, and $\eta,\nu = \pm 1$.

PROOF

The left-hand side of Equation (3.146) can be factored as follows:

$$X^2 - Y^2 = (X+Y)(X-Y)\,. \tag{3.148}$$

These factors must integer, as they are the sum and difference of integers. The only integer factorizations into two parts for Equation (3.146) are given by:

$$4^N = (\eta 2^{N_1})(\eta 2^{N_2})\,, \tag{3.149}$$

where N_1, N_2 are non-negative integers, satisfying $N_1 + N_2 = 2N$, and $\eta = \pm 1$. Assigning the factors on both sides gives us:

$$\left\{ \begin{array}{lcl} (X+Y) & = & (\eta 2^{N_1}) \\ (X-Y) & = & (\eta 2^{N_2}) \end{array} \right. . \tag{3.150}$$

Solving the above system of equations for X and Y results in:

$$\left\{ \begin{array}{lcl} X & = & \eta(\frac{2^{N_1}+2^{N_2}}{2}) \\ Y & = & \eta(\frac{2^{N_1}-2^{N_2}}{2}) \end{array} \right. , \tag{3.151}$$

The extra degree of freedom in the sign ν of the expression for Y in Equation (3.147) comes from interchanging the role of N_1 and N_2, which changes the sign. The sign η of the expression for X then remains unaffected.

Bibliography

[1] Gerben Hekstra, "Definition and structure of hemisphere and viewing rays," Tech. Rep. ET/NT/Radio-9, TU Delft, November 1993.

[2] Jack. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on electronic computers*, pp. 330–334, Sept. 1959.

[3] J.S Walther, "A unified algorithm for elementary functions," *proceedings of the AFIPS Spring Joint Computer Conference*, pp. 379–385, 1971.

[4] R.W. Reitwiesner, "Binary Arithmetic," *Advances in Computers*, vol. 1, pp. 231–308, 1966.

[5] H. Samueli, "An improved search algorithm for the design of multiplierless FIR filters with power-of-two coefficients," *IEEE Trans. on Circuits and Systems*, vol. 36, pp. 1044–1047, July 1989.

[6] D. Li, J. Song, and Y. C. Lim, "A polynomial-time algorithm for designing digital filters with power-of-two coefficients," in *Proceedings of 1993 IEEE ISCAS*, Chicago, Il, May 1993, pp. 84–87.

[7] D. A. Parker and K. K. Parhi, "Area-efficient parallel FIR digital filter implementations," in *International Conference on Applications-Specific Systems, Architectures and Processors*, Chicago, IL, Aug. 1996, pp. 93–111.

[8] Daniel A. Magenheimer, Liz Peters, Karl W. Petis, and Dan Zuras, "Integer multiplication and division on the HP precision architecture," *IEEE transactions on computers*, vol. vol. 37, pp. pp. 980–990, August 1988.

[9] J. Götze, S. Paul, and M. Sauer, "An efficient Jacobi-like algorithm for parallel eigenvalue computation," *IEEE Trans. Comput.*, vol. 42, pp. 1058–1065, 1993.

[10] Gerben J. Hekstra and Ed F.A. Deprettere, "Fast rotations: Low-cost arithmetic methods for orthonormal rotation," in *Proceedings of the 13th Symposium on Computer Arithmetic*, Tomas Lang, Jean-Michel Muller, and Naofumi Takagi, Eds. IEEE, July 1997, pp. 116–125.

[11] Jürgen Götze and Gerben J. Hekstra, "Adaptive approximate rotations for computing the EVD," in *Proceedings 3rd Int. Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, Aug. 1994.

[12] J. Götze and G. Hekstra, "An algorithm and architecture based on orthonormal μ-rotations for computing the symmetric EVD," *Integration, the VLSI Journal*, vol. 20, pp. 21–39, 1995.

[13] J. Götze and G. Hekstra, "Adaptive approximate rotations for computing the symmetric EVD," in *Algorithms and Parallel VLSI Architectures III*, Marc Moonen and Francky Catthoor, Eds., pp. 73–84. Elsevier, 1995.

[14] Gerben J. Hekstra, Ed F. Deprettere, Richard Heusdens, and Monica Monari, "Recursive approximate realization of image transforms with orthonormal rotations," in *Proceedings International Workshop on Image and Signal Processing*, Manchester, UK, November 1996.

[15] Gerben J. Hekstra, Ed F. Deprettere, Richard Heusdens, and Zhiqiang Zeng, "Efficient orthogonal realization of image transforms," in *Proceedings SPIE*, Denver, Colorado, US, August 1996.

[16] J. Götze, "CORDIC-based approximate rotations for SVD and QRD," in *European Signal Processing Conference*, Edinburgh, Scotland, 1994, pp. 1867–1871.

[17] J. Götze, P. Rieder, and J.A. Nossek, "Parallel SVD-updating using approximate rotations," in *SPIE Conference on Advanced Signal Processing: Algorithms and Applications*, San Diego, USA, 1995, SPIE, pp. 242–252.

[18] P. Rieder, J. Götze, M. Sauer, and J.A. Nossek, "Orthogonal approximation of the discrete cosine transform," in *Proc. European Conference on Circuit Theory and Design*, Istanbul, Turkey, Aug. 1995, pp. 1003–1006.

[19] J. Götze and G. Hekstra, "An algorithm and architecture based on orthonormal μ-rotations for computing the symmetric EVD," *Integration, the VLSI Journal*, vol. 20, pp. 21–39, 1995.

[20] D. Zwillinger, Ed., *Standard Mathematical Tables and Formulae*, CRC press, 30th edition, 1989.

[21] Jean Duprat and Jean-Michel Muller, "The Cordic algorithm: new results for a fast VLSI implementation," Tech. Rep. 90-04, LIP-IMAG, Jan. 1990.

[22] Peter Montgomery, "Concerning the mathematical background of maximal fast rotations, factoring of polynomials over the complex numbers," private communication, July 1997.

Part II

Applications of Fast Rotations

Chapter 4

THE USE OF FAST ROTATIONS IN COMPUTER GRAPHICS

Contents

4.1 Introduction

In this chapter, we show how fast rotations have been applied in computer graphics, and what has been gained from their use over that of alternative arithmetic techniques. The computer graphics application which we consider is that of photo-realistic rendering of artificial images, by means of ray-tracing and radiosity shading. The rendering technique that we propose is described in detail in [1]. This technique, however, requires an enormous amount of quite high complexity arithmetic functions to achieve the desired photo-realism. Typical operations that we encounter are: geometric transformations of polygons, computing bounding boxes of a polygon in the spherical coordinate system, and the intersection computation of a line segment with a polygon. The work presented in this chapter concerns the implementation of a VLSI chip-set that forms a high-performance co-processor for the Radiosity Engine[1] [2, 3, 1] that is capable of computing typical ray-tracing operations, at a very high throughput. We will pay particular attention to the process of intersection computation, and its implementation in the Intersection Computation Unit (ICU).

4.1.1 Outline of this chapter

In Section 4.2, we describe the problem of photo-realistic rendering of artificial scenes. We identify that the most computationally intensive problem is the computation of the form-factors by means of sampling with a large ray frustum. In Section 4.3, we go deeper into the details of sampling with a hemisphere ray frustum. We show how hemisphere rays are constructed, and that the sampling requires the computation of the intersection of rays with patches. Section 4.4 extensively covers the problem of intersection between a patch and a bundle of rays. We propose our solution, which uses an incremental computation scheme to lower the amount of operations. The further introduction of fast rotation operations gives rise to a large reduction of the computational complexity, and brings the implementation of a high-throughput VLSI intersection computation unit (ICU) within reach. In Section 4.5 we show that the introduction of fast rotations does not necessarily complicate other operations related to the intersection. We show how the index space bounding box (ISBB) computation is affected. Section 4.6 covers the implementation aspects of the Radiosity Engine system for high performance photo-realistic rendering. We focus on the implementation of the ASIC chip-set which forms the computational heart of the system. Finally, in Section 4.7, we give our conclusions.

[1]This research was funded by the Dutch Technology Foundation STW (Stichting voor de Technische Wetenschappen) under contract DEL99.1982

4.2 Photo-realistic rendering

Rendering, or the production of photo-realistic images, requires the actual simulation of the propagation of light through an environment. Shen, in [1], states that realism can only be achieved by taking the global illumination into account, which includes the effects of all objects in the environment. Two commonly used methods that do this are **ray-tracing** [4], which considers only specular reflections, and **radiosity** [5], which considers only diffuse reflections. The rendering technique we employ is a mixture of both. We refer the interested reader to [1] for further details. Likewise, we do not pretend to give a full treatise on the rendering technique here, but only present that which is relevant for later discussion.

4.2.1 The environment

The use of radiosity requires it to work with preferably a closed environment. For our case, the environment is bounded in 3D by a unit cube with its center in the origin of the coordinate system. Any point V with coordinates $V = (x,y,z)$ in the environment is hence limited to:

$$\begin{array}{rcl} -\frac{1}{2} \leq & x & \leq \frac{1}{2} \\ -\frac{1}{2} \leq & y & \leq \frac{1}{2} \\ -\frac{1}{2} \leq & z & \leq \frac{1}{2} \; . \end{array} \tag{4.1}$$

We call the Cartesian coordinate system in which the environment is defined, the **global coordinate system**. Unless otherwise specified, we assume that all points and vectors are defined within this coordinate system.

We position the objects (i.e. teapot, painting, table, etc.) in this environment. Objects, in turn, are built up out of patches. The patches make up the surface of the object. To the patches are assigned certain surface properties, such as colour, emission energy, reflective and refractive coefficients, etc., see [1] for more details. Within this chapter, we confine our focus to only the geometric properties of patches. Whereas we are in essence free to choose any type of surface primitive for patches, we choose them to be coplanar, convex polygons for reasons of practicality, and to keep computations simple.

Definition of patches

Let $P_0, P_1, \ldots$ be points in the global coordinate system, and let $\mathbf{p}_0, \mathbf{p}_1, \ldots$ be their corresponding vectors. We define a **patch** as a polygon having four vertices P_0, P_1, P_2, P_3, with respective vectors $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$. The polygon is assumed to be both convex and coplanar. A patch has a front face and a back face. The orientation of a patch is determined by the sequence of vertices. This orientation follows the "right-hand" rule that states that, when looking at the back of a patch and the front facing away, the vertices are counted clock-wise. This is illustrated in Figure 4.1.

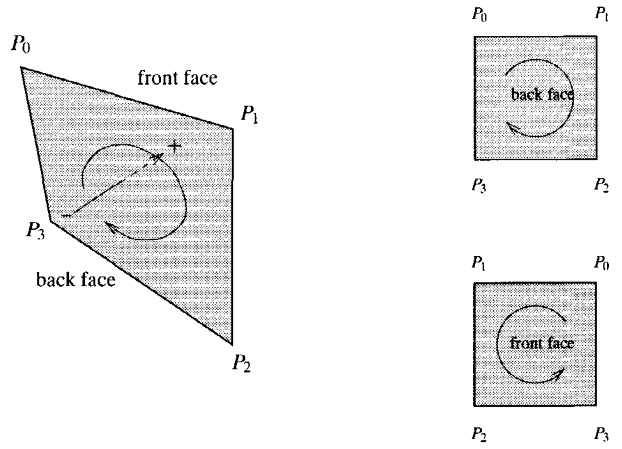


Figure 4.1. The orientation of a patch, showing the front and back faces, and the sequence of the vertices.

The surface properties, mentioned above, are only valid for the front face of the patch. Objects are decomposed into patches in such a way that only the front faces are "seen" in the environment.

4.2.2 Form-factor computation

Essential in the radiosity method is the computation of so-called form-factors. Let us consider two patches in the environment, with indices k and ℓ. We take patch k to be the source patch, that is where the energy originates, and patch ℓ as a destination patch.

The **form-factor** $F_{k,\ell}$ between these two patches, from patch k to patch ℓ, is defined as the fraction of energy that leaves the source patch k and arrives on the destination patch ℓ. This takes into account the "visibility" and possible partial occlusion of the destination patch ℓ, as seen from the source patch k. If we know the energy emitted by the source patch, we can use this form-factor to determine the amount of energy that lands on the destination patch. This energy is then added to the input energy of that patch; what the resulting output energy will be is a function of the patch's properties, such as its reflectance coefficient. When we consider all patches in the environment in turn as source, and the remaining as destination, we arrive at a system of equations in terms of the energy emitted by the patches. Solving the equilibrium state of this system results in the radiosity solution for the environment. See [1] for more details. Let A_i be the area of a given patch with index i. Let dA_i be an infinitesimal differential area on the patch, located at an arbitrary sample point $\mathbf{s}_i$. Let us consider the line segment between the sample points on the source patch, $\mathbf{s}_k$ and on the destination patch, $\mathbf{s}_\ell$. Let $\beta_i, i = \{k, \ell\}$ be the angle between the normal of patch i and the said line segment. See Figure 4.2 for an illustration of the setup of the form-factor computation. The formula for the form-factor is given by the

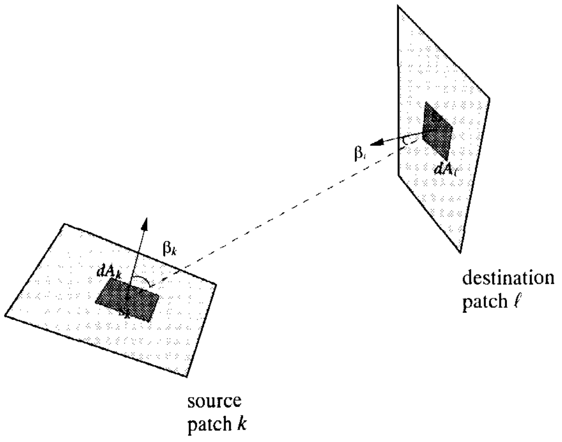


Figure 4.2. Setup for the form-factor computation

double surface integral:

$$F_{k,\ell} = \frac{1}{A_k} \int_{A_k} \int_{A_\ell} \frac{\cos(\beta_k)\cos(\beta_\ell)}{\pi R^2} HID \, dA_\ell \, dA_k \tag{4.2}$$

where *HID* is a parameter for the occlusion, which takes the value 1 if the differential areas can see each other, and 0 if not. The sum of the form-factors is, by definition, equal to unity. Let $\mathcal{P}$ be the set of patches in the environment[2].

$$\sum_{\substack{\ell \in \mathcal{P} \\ \ell \neq k}} F_{k,\ell} = 1 \tag{4.3}$$

The analytical evaluation of the double surface integral of Equation (4.2) is a daunting, if not impossible, task. It is made especially difficult by the presence of the occlusion parameter *HID*, which is a function of all patches in the environment that lie between the two patches in question.

Instead we propose to (approximately) evaluate this integral using a summation, where we sample the space, as seen from the sample point, with a large collection of rays, as we will describe below.

[2]It suffices here to take the subset of patches that can be 'seen' from the source patch. All patches that cannot be 'seen' have a form-factor of zero.

Sampling with rays

We propose to attack the problem of computing form-factors as follows. [3] Instead of using an infinitesimal amount of sample points on the source patch, we use a small, finite number of sample points. Let **s** now be a sample point on the source patch, and let $\mathcal{S}$ be the set of sample points. For the sake of simplicity, we assume them to be spaced in a regular grid on the source patch. Let dA_k be the differential area, belonging to the sample point **s**, but no longer being infinitesimal.

Let us define a **ray**, r, as the line segment, starting from the said sample point **s**, and continuing along a given ray direction **r**. The vector **x** lies on the ray if it satisfies the line equation:

$$\mathbf{x} = \mathbf{s} + \lambda\mathbf{r} \tag{4.4}$$

and where the parameter λ satisfies $\lambda \geq 0$. The parameter λ is a measure of the distance of the point **x** from the ray origin **s**. Rays are only shot into the half-space, as seen from the front face of the source patch.

Let us define a **ray frustum**, $\mathcal{R}$ as a set of rays $r \in \mathcal{R}$, which all share a common ray origin **s**. Let us assume that, for this treatise, the ray directions are all pre-determined and fixed. Each ray then actually represents a small spherical volume in space, and together, they make up the half-space as seen from the source patch.

Through the use of Nusselt's analogue [1], we assign to each ray r, a pre-computed **delta form-factor** $f(r)$, which is the fraction of energy leaving the source patch, that is represented by that ray. The sum of the delta form-factors is equal to unity.

$$\sum_{r \in \mathcal{R}} f(r) = 1 \tag{4.5}$$

Effectively, the delta form-factor $f(r)$ can be seen as being equivalent to the part $\frac{\cos(\beta_k)\cos(\beta_\ell)}{\pi R^2} dA_\ell$ in Equation (4.2), where $\cos(\beta_k)$ is the angle of inclination of the ray, measured from the normal, and where dA_ℓ is the projection of the spherical volume, belonging to the ray, onto the destination patch.

Let us also define the occlusion function $HID(\mathbf{s}, r)$ as being 1 if the ray r, shot from the sample point **s** on the source patch, 'sees' the destination patch ℓ as the first patch in its line of vision, and as being 0 otherwise. We also use the terminology that the ray 'hits' the destination patch.

We can now approximate the double surface integral of Equation (4.2) with the double summation:

$$F_{k,\ell} = \frac{1}{A_k} \sum_{\mathbf{s} \in \mathcal{S}} dA_k \sum_{r \in \mathcal{R}} HID(\mathbf{s}, r) f(r) \tag{4.6}$$

[3] Most of what we will describe here, is the same as proposed by Shen in [1].

The accuracy of this approximation is heavily dependent on the choice of an adequately fine sampling of the source patch, as determined by $\mathcal{S}$, and of the space, as determined by $\mathcal{R}$. Especially a sufficiently fine density of the rays in $\mathcal{R}$ is necessary to catch fine details that may be present in the environment. Note that for the Radiosity Engine, the typical density of the rays in $\mathcal{R}$ ranges from around 10^4 to 10^7.

We pay particular attention to the inner summation of Equation (4.6), given by $\sum_{r\in\mathcal{R}} HID(\mathbf{s},r)f(r)$, as we identify this as the most computationally complex in the entire rendering. We call this process **ray frustum shooting** from a given sample point s on the source patch k.

Note that the term $HID(\mathbf{s},r)$ here is implicitly linked to a given destination patch ℓ. If we take note of which rays $r\in\mathcal{R}$ hit which patches, we have all the necessary information to compute $HID(\mathbf{s},r)$ for *any* given destination patch $\ell\in\mathcal{P}$.

Our problem of computing all necessary form-factors $F_{k,\ell}$, with $\ell\in\mathcal{P},\ell\neq k$, is now reduced to the problem of shooting a ray frustum into the environment, checking which patches are hit by which rays, and summing up the contributions of the delta form-factors to form the form-factors of the corresponding patches. Essential in this is the computation of the intersection of rays r with patches in the environment to find out which rays hit which patch as the first in their line of vision. We will discuss this next

4.3 Hemisphere ray sampling

First, we need to define a number of concepts for rays and ray frustums before we can tackle the problem of ray-patch intersection.

4.3.1 The local coordinate system

Let s be the sample point on the source patch. Let **n** be the normal of the patch, at the sample point, and such that it points in the direction of the front face of the patch. Let $\mathbf{e}'_x,\mathbf{e}'_y,\mathbf{e}'_z$ be vectors of unit length, defined in the global coordinate system, and together forming a basis. Let us define the **local coordinate system** for the sampling to be the Cartesian coordinate system, with its origin located at the sample point s, and with its z-axis aligned with the normal of the patch. We designate the axes of the local coordinates system to be x',y' and z', with corresponding direction vectors $\mathbf{e}'_x,\mathbf{e}'_y$ and $\mathbf{e}'_z$. See Figure 4.3 for an illustration. We introduce the convention that vectors with a prime, like $\mathbf{p}'$, are defined in the local coordinate system, unless otherwise specified. One notable exception is for the direction vectors $\mathbf{e}'_x,\mathbf{e}'_y,\mathbf{e}'_z$ of the axes of the local coordinate system, which are defined in the global coordinate system.

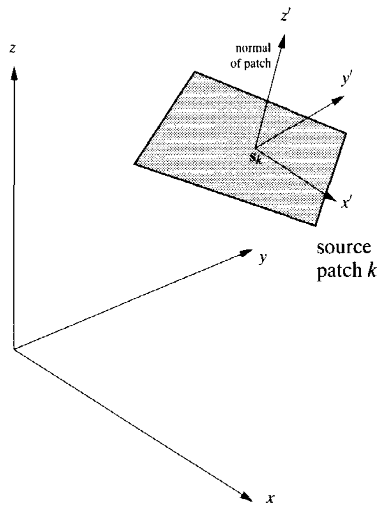


Figure 4.3. The global and local coordinate systems

A point $\mathbf{p}'$ defined in the local coordinate system, has an equivalent point $\mathbf{p}$ in the global coordinate system. They are related to each other by a geometrical transformation, which is a combined rotation and translation in 3-D space. Let us define $T_{L\to G}$ and $T_{G\to L}$ to be geometrical transformations to compute the global coordinates from local ones, and vice versa. The following relations hold:

$$\begin{aligned} \mathbf{p} &= T_{L\to G}(\mathbf{p}') \\ \mathbf{p}' &= T_{G\to L}(\mathbf{p}) \end{aligned} \tag{4.7}$$

Let $\mathbf{R}_{L\to G}$ be the 3×3 rotation matrix, given by:

$$\mathbf{R}_{L\to G} = \left[\ \mathbf{e}'_x \mathbf{e}'_y \mathbf{e}'_z\ \right]. \tag{4.8}$$

Keeping in mind that the origin of the local coordinate system is at the sampling point s, the geometrical transformation $T_{L\to G}$ can be written as:

$$\mathbf{p} = T_{L\to G}(\mathbf{p}') = \mathbf{s} + \mathbf{R}_{L\to G}\cdot\mathbf{p}', \tag{4.9}$$

which is in effect a rotation to the correct coordinate axes, followed by a translation. From the above, combined with Equation (4.7), the inverse transformation $T_{G\to L}$ follows as:

$$\mathbf{p}' = T_{G\to L}(\mathbf{p}) = \mathbf{R}_{G\to L}\cdot(\mathbf{p}-\mathbf{s}), \tag{4.10}$$

where $\mathbf{R}_{G\to L}$ is also a 3×3 rotation matrix, and given by $\mathbf{R}_{G\to L} = \mathbf{R}^{-1}_{L\to G}$.

4.3.2 The hemisphere coordinate system

We define the **hemisphere** as a half sphere with unit radius, positioned on the front face of the source patch, with its origin at the sampling point $\mathbf{s}$, and with its apex in the direction of the patch normal. We use the hemisphere to define the sampling rays. It is closely connected to the local coordinate system of the patch.

Let φ, θ be two angles. We define the hemisphere coordinate system as given by all possible tuples (φ, θ), where φ, θ are constrained by:

$$\begin{array}{lll} 0 \leq & \theta & < \pi/2 \\ 0 \leq & \varphi & < 2\pi \end{array} . \tag{4.11}$$

Let $\mathbf{r}'$ be a vector, defined in the local coordinate system. The point $\mathbf{r}'$ lies on the hemisphere if it satisfies the equation:

$$\mathbf{r}' = \mathbf{R}_{xy}(\varphi)\mathbf{R}_{zx}(\theta) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} , \tag{4.12}$$

where $\mathbf{R}_{xy}(\varphi)$ and $\mathbf{R}_{zx}(\theta)$ are embedded rotation matrices, given by:

$$\mathbf{R}_{xy}(\varphi) = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} , \tag{4.13}$$

and

$$\mathbf{R}_{zx}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} . \tag{4.14}$$

The matrix $\mathbf{R}_{zx}(\theta)$ performs a rotation over the angle θ in the z,x plane, while the matrix $\mathbf{R}_{xy}(\varphi)$ performs a rotation over the angle φ in the x,y plane. The vector $\mathbf{r}'$ has unit length since the rotations do not affect the length of vectors. The relation between the vector $\mathbf{r}'$ on the hemisphere and a point (φ, θ) in the hemisphere coordinate system, as set by Equation (4.12), is also illustrated in Figure 4.4.

Writing out Equation (4.12), substituting (4.13) and (4.14) results in an alternative form for $\mathbf{r}'$, given by:

$$\mathbf{r}' = \begin{bmatrix} \sin(\theta)\cos(\varphi) \\ \sin(\theta)\sin(\varphi) \\ \cos(\theta) \end{bmatrix} . \tag{4.15}$$

though we prefer the form as given by Equation (4.12).

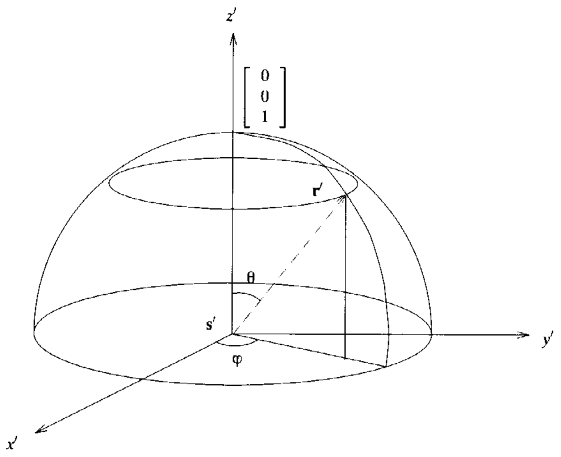


Figure 4.4. Construction of the ray direction $\mathbf{r}'$ in the hemisphere coordinate system, showing the orientation of the angles θ and φ

4.3.3 Hemisphere rays

In Equation (4.4), we have seen the definition of a ray in the global coordinate system. Transformation of the ray equation to the local coordinate system, through application of Equation (4.10), leads to:

$$\mathbf{x}' = \lambda \mathbf{r}', \tag{4.16}$$

where $\mathbf{x}'$ is a point on the ray, and $\mathbf{r}'$ is the direction vector of the ray, both defined in the local coordinate system. The ray parameter λ is the same as in Equation (4.4), and satisfies $\lambda \geq 0$. Note that the origin of the ray is at the origin of the local coordinate system, which is of course the sampling point $\mathbf{s}$.

We define a hemisphere ray, in the local coordinate system, as the line segment starting from the origin of the hemisphere, satisfying Equation (4.16), and where the ray direction vector $\mathbf{r}'$ lies on the hemisphere, and is given by Equation (4.12).

We sample the space, as seen by the sample point on the source patch, with a bundle, or frustum, of rays $\mathcal{R}$. We call this set of rays the **hemisphere rays**. The direction vectors $\mathbf{r}$ of the hemisphere rays are determined by a regular, uniformly spaced grid in the hemisphere coordinate system.

The ray index space

Let us define N_θ as the number of hemisphere rays in the θ dimension, and N_φ as the number of hemisphere rays in the φ dimension. Let us define the **ray index**

(i, j) for the hemisphere rays, with i, j both integer and constrained to:

$$\begin{array}{lcl} 0 \leq & i & \leq N_\varphi - 1 \\ 0 \leq & j & \leq N_\theta - 1 \ , \end{array} \tag{4.17}$$

The pairs (i, j) form a two-dimensional, rectangular, regular grid with sides N_φ and N_φ. The total number of rays N_r in this ray index space, and hence in the hemisphere, is given by

$$N_r = N_\theta N_\varphi . \tag{4.18}$$

We use the ray index (i, j) to index the hemisphere rays. We call the space of ray indices (i, j) the **ray index space**.

Hemisphere ray direction vectors

Let us define the vector $\mathbf{r}'_{i,j}$ as being the ray direction, in the local coordinate system, for the hemisphere ray with ray index (i, j).

Let the angles φ_i and θ_j form two monotonous increasing sequences for the indices $i = 0, 1, \ldots$ and $j = 0, 1, \ldots$, and satisfying:

$$\begin{array}{lcl} 0 \leq & \varphi_i & \leq 2\pi \\ 0 \leq & \theta_j & \leq \frac{\pi}{2} \end{array} \tag{4.19}$$

and where the indices (i, j) satisfy the conditions set in Equation (4.17).

The pair of angles (φ_i, θ_j), forming a point in the hemisphere coordinate system, determine the direction of the hemisphere ray with index (i, j). The direction vector $\mathbf{r}'_{i,j}$, defined in the local coordinate system, of this hemisphere ray is given, analogous to Equation (4.12), by:

$$\mathbf{r}'_{i,j} = \mathbf{R}_{xy}(\varphi_i)\mathbf{R}_{zx}(\theta_j) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} . \tag{4.20}$$

Hemisphere ray resolution

For convenience in computations, as we will show later on, we choose the distribution of the angles of the rays to be uniform. We define a constant angular **ray resolution** Δ_φ and Δ_θ, and use this to define the uniformly distributed angles θ_j and φ_i as:

$$\begin{array}{lcl} \theta_j & = & (j + \frac{1}{2}) \cdot \Delta_\theta \\ \\ \varphi_i & = & (i + \frac{1}{2}) \cdot \Delta_\varphi \ . \end{array} \tag{4.21}$$

Substituting Equation (4.21) into (4.11), and combining this with the conditions posed in Equation (4.17), we derive the relation between the ray resolutions $\Delta_\varphi, \Delta_\theta$ and the number of rays N_φ, N_θ to be:

$$\begin{aligned} N_\varphi &= \left\lfloor \frac{2\pi}{\Delta_\varphi} - \frac{1}{2} \right\rfloor \\ N_\theta &= \left\lfloor \frac{\pi}{2\Delta_\theta} - \frac{1}{2} \right\rfloor . \end{aligned} \tag{4.22}$$

Note that, since we are dealing with a sampling problem, we are free to choose the ray resolutions $\Delta_\varphi, \Delta_\theta$ such that the rotation is easy and cheap to implement, *i.e.* one of the fast rotation methods of Chapter 3.

Practical values for the ray resolution

As mentioned before, the design criteria for the Radiosity Engine, require the density of hemisphere sampling rays N_r to range from around 10^4 to 10^7. From these figures we derive the range of practical values for the number of rays, N_φ and N_θ, and the required angle resolutions Δ_φ and Δ_θ.

The rectangular space spanned by the rays in the hemisphere coordinate space has an aspect ratio of 4 : 1 for φ versus θ. Let us assume that the angle resolution is the same in both dimensions for the extreme cases, so that $\Delta_\varphi = \Delta_\theta$. Then, applying Equation (4.22) to Equation (4.18), ignoring the floor operators, results in the following approximation:

$$\begin{aligned} N_r &= N_\theta N_\varphi \\ &= \left\lfloor \frac{2\pi}{\Delta_\varphi} - \frac{1}{2} \right\rfloor \left\lfloor \frac{\pi}{2\Delta_\theta} - \frac{1}{2} \right\rfloor \\ &\approx \frac{2\pi}{\Delta_\varphi} \frac{\pi}{2\Delta_\theta} \\ &\approx \frac{\pi^2}{\Delta_\varphi^2} \end{aligned} \tag{4.23}$$

We use this approximation to derive target ranges for ray resolutions. For the aforementioned range of the total number of rays, this results in the target ranges presented in Table 4.1.

Note that these values are targets only, and not strict values. The actual ranges are determined by the choice of the resolutions $\Delta_\varphi, \Delta_\theta$ or the number of rays N_φ, N_θ. These in turn depend significantly on the choice of the arithmetical operations at the lowest level, as we shall see in the following sections.

parameter	symbol	min. value	max. value
total number of rays	N_r	$1.0 \cdot 10^4$	$1.0 \cdot 10^7$
number of rays in φ	N_φ	$2.0 \cdot 10^2$	$6.32 \cdot 10^3$
number of rays in θ	N_θ	$5.0 \cdot 10^1$	$1.58 \cdot 10^3$
ray resolution in φ	Δ_φ	$3.14 \cdot 10^{-2}$	$9.93 \cdot 10^{-4}$
ray resolution in θ	Δ_θ	$3.14 \cdot 10^{-2}$	$9.93 \cdot 10^{-4}$

Table 4.1. Target ranges for the ray resolution

4.3.4 Sampling of a patch with hemisphere rays

In the sampling of a destination patch with the hemisphere rays, we are trying to determine *which* of the rays actually hit the patch in question as the first in the line of vision. The delta form-factors of these rays contribute to the form-factor of the patch. This is then used to find out what amount of energy of the source patch is received by the destination patch.

The determination of the rays that hit the patch is done in two phases. First, a subset of the hemisphere rays is selected. It makes no sense to test certain rays if they hit the patch, if you can determine beforehand, on basis of their geometry that they never will. For instance, a ray shot to one corner of the environment will not hit a patch in another corner. Second, the rays are tested if they actually intersect with the patch, and the intersection distances to the patches are sorted, so the closest patch for that ray is found.

In this section, we consider how the selection of the rays is done, on the basis of a bounding box that contains the patch. Any ray not penetrating the bounding box is by definition not going to hit the patch. We choose to use a bounding box in the spherical coordinate system, which simplifies the selection of the rays.

The Spherical Bounding Box

The spherical coordinate system, with coordinates (φ, θ, R), is defined, in relation to a point (x', y', z') in the local coordinate system as:

$$\begin{aligned} x' &= R\cos(\theta) \\ y' &= R\sin(\theta)\cos(\varphi) \\ z' &= R\sin(\theta)\sin(\varphi) \ . \end{aligned} \tag{4.24}$$

In this system, R is the distance of the point (x', y', z') to the origin of the local coordinate system (the sample point s), while the angles θ, φ fix the direction, as in the hemisphere coordinate system. The angle θ is also known as the elevation, and φ as the azimuth.

We define the **spherical bounding box** (SBB) of a patch as a tuple

$$((\varphi_{\min}, \varphi_{\max}), (\theta_{\min}, \theta_{\max}), R_{\min})$$

with:

$$\begin{array}{lll} 0 \leq & \varphi_{\min} \leq \varphi_{\max} & \leq 2\pi \\ 0 \leq & \theta_{\min} \leq \theta_{\max} & \leq \frac{\pi}{2} \\ 0 \leq & R_{\min} & . \end{array} \tag{4.25}$$

The spherical bounding box describes a bounding box in the spherical coordinate system of Equation (4.24), such that the patch in question is completely contained within. The SBB can be used to select rays in the hemisphere coordinate system, to test for intersection with the patch in question. Any ray that lies outside the SBB is, by definition, never going to intersect the patch, and hence need not be tested. In this way, a fraction of the hemisphere rays, only those that matter, are selected, reducing the amount of unnecessary intersection computations. This is the underlying principle of the Shelling Technique [1], using **spatial isolation** of patches, for all three spherical dimensions, to reduce the amount of unnecessary computations. For the full description of the SBB computation, we refer to [6, 7, 8].

The Index Space Bounding Box

The **index space bounding box**, or ISBB for short, is defined as the tuple $((l_1, u_1), (l_2, u_2))$, with l_1, u_1, l_2, u_2 integer, and

$$\begin{array}{lll} 0 \leq & l_1 \leq u_1 & \leq N_\varphi \\ 0 \leq & l_2 \leq u_2 & \leq N_\theta \ . \end{array} \tag{4.26}$$

The tuple (l_1, u_1) represents the lower and upper bounds for the index i, while the tuple (l_2, u_2) represents the lower and upper bounds for the index j, defining a rectangular region in the ray index space.

A ray with ray index (i, j) is contained within the index space bounding box if the conditions:

$$\begin{array}{lll} l_1 & \leq i \leq & u_1 \\ l_2 & \leq j \leq & u_2 \end{array} \tag{4.27}$$

hold.

We use the ISBB to select a subset of the hemisphere rays $\mathcal{R}$ to test with the patch for intersection. Any ray outside of the ISBB is guaranteed, by definition, to miss the patch and needs not to be tested. We need both the concept of an SBB and of an ISBB. The SBB is computed from the (local) coordinates of the patch. It is independent of the actual resolution of the hemisphere rays. The ISBB depends on the resolution of the hemisphere rays. However, since multiple resolutions may be required in the shooting process, it is easier to recompute the ISBB from the SBB of a patch, instead of directly from the patch coordinates.

Computation of the index space bounding box from the SBB

A ray with index (i, j), and therefore with angles φ_i, θ_j, is contained within the spherical bounding box if the conditions

$$\begin{aligned} \varphi_{\min} &\leq \varphi_i \leq \varphi_{\max} \\ \theta_{\min} &\leq \theta_j \leq \theta_{\max} \end{aligned} \tag{4.28}$$

hold.

Our goal is to compute the ISBB from the SBB, such that any ray contained within the SBB, is also contained within the ISBB. Or, to state the dual form: any ray, *not* contained within the ISBB is also *not* contained within the SBB. Figure 4.5 illustrates the relationship between the SBB and the ISBB.

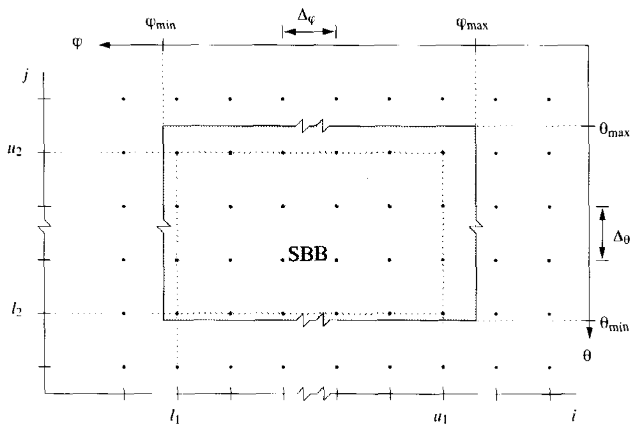


Figure 4.5. Computing the ISBB from the Spherical bounding box

Hence we can state that the necessary conditions for the index bounds are given by:

$$\begin{aligned} \varphi_{(l_1-1)} &< \varphi_{\min} \\ \varphi_{(u_1+1)} &> \varphi_{\max} \\ \theta_{(l_2-1)} &< \theta_{\min} \\ \theta_{(u_2+1)} &> \theta_{\max} \ . \end{aligned} \tag{4.29}$$

Note that the above conditions (4.29) are necessary and adequate. However, in order to find the optimum bounds, we can tighten these conditions. We show this for l_1.

We narrow the condition (4.29) for l_1, so as to find the maximum value that l_1 can attain. In that case, $i = l_1$ is contained within the SBB, while $i = (l_1 - 1)$ lies just outside the SBB. Hence we state:

$$\varphi_{(l_1-1)} < \varphi_{\min} \leq \varphi_{(l_1)} \,. \tag{4.30}$$

Substitution of Equation (4.21) into the above condition leads to:

$$l_1 - 1 < \frac{\varphi_{\min}}{\Delta_\varphi} - \frac{1}{2} \leq l_1 \,, \tag{4.31}$$

from which, through the definition of the ceiling operator, the formula for the optimum bound l_1 follows as:

$$l_1 = \left\lceil \frac{\varphi_{\min}}{\Delta_\varphi} - \frac{1}{2} \right\rceil \,. \tag{4.32}$$

Similarly, we can derive the optimum values for the other bounds, resulting in (no proof given):

$$\begin{aligned} l_1 &= \left\lceil \frac{\varphi_{\min}}{\Delta_\varphi} - \frac{1}{2} \right\rceil & u_1 &= \left\lfloor \frac{\varphi_{\max}}{\Delta_\varphi} - \frac{1}{2} \right\rfloor \\ l_2 &= \left\lceil \frac{\theta_{\min}}{\Delta_\theta} - \frac{1}{2} \right\rceil & u_2 &= \left\lfloor \frac{\theta_{\max}}{\Delta_\theta} - \frac{1}{2} \right\rfloor \,. \end{aligned} \tag{4.33}$$

Note that these are the optimum bounds, and not necessarily the ones we want to use, due to the presence of a division operation in the computation. In later sections we will present sub-optimum bounds which require less complex computations.

4.4 Ray-patch Intersection Computation

In this section we treat the intersection between a patch and a ray frustum. Rather than working on a single ray basis for ray-patch intersection, we can use the knowledge of the structure of the frustum to our advantage to reduce the complexity of the operations. We know that we are dealing with a sampling problem here, and we shall show that we can use this knowledge too to choose certain sampling resolutions that simplify operations.

4.4.1 Requirements for intersection computation

We are interested in finding the intersection points of a patch with a ray frustum. With 'patch' we mean any of the destination patches, as 'seen' from the sample point s. More specifically, we want to know:

1. whether a ray r from the frustum $\mathcal{R}$ does indeed intersect with the patch, or misses it completely. We call this the **hit** or no-hit information. If we can tell beforehand if it misses, then we do not need to compute the intersection point.

2. the distance from the sample point $\mathbf{s}$, which is the origin of the ray, to the intersection point $\mathbf{x}$ on the patch. This is needed to 'sort' the patches which are intersected by the ray, such that the first in line of vision (smallest distance) is selected as being 'seen' from the sample point.

3. the coordinates of the intersection point $\mathbf{x}$. The rendering algorithm that we use requires that, if a ray hits a patch with specular (hard) reflection properties, the ray will continue at a later stage from that intersection point. This in turn requires the knowledge of the exact intersection point on the patch. As we shall see later on, we choose to represent this intersection point in a local patch coordinate system rather than in, say, the global coordinate system.

There are well documented ways to compute the intersection point [9, 10] and the related quantities as required above, but we are looking in particular for a method that is both cheap to implement in VLSI, as well as numerically robust. The latter constraint is necessary for finite wordlength, fixed-point computations.

We base this work on the method proposed by Hekstra in [10]. This method, called the "bounding planes" method, computes the hit/no-hit information, the intersection distance and the coordinates of the intersection point. The latter are given in the patch coordinate system, from which the coordinates of the intersection point in either the local or the global coordinate system can be computed. The method is particularly efficient in the re-use of already computed results, by means of incremental computation schemes. Also it allows the use of fast rotations at the heart of the incremental operation, which greatly reduces the computational complexity. The method is robust, and the corresponding architecture has an efficient VLSI implementation. Our additional contribution here is the use of an alternative patch coordinate system, which reduces the complexity even further.

4.4.2 The patch coordinate systems

We define the patch coordinate system by a tuple $(u_\square . v_\square)$, and state that any point $\mathbf{x}$ on the patch can be represented as a bi-linear combination the patches vertices, in terms of the parameters $u_\square$ and $v_\square$ as given by:

$$\begin{aligned} \mathbf{x} &= (1-v_\square)\left((1-u_\square)\mathbf{p}_0 + (u_\square)\mathbf{p}_1\right) \\ &+ \quad (v_\square)\left((1-u_\square)\mathbf{p}_3 + (u_\square)\mathbf{p}_2\right) \ . \end{aligned} \tag{4.34}$$

where $u_\Box, v_\Box$ are confined to:

$$\begin{array}{rcl} 0 \leq & u_\Box & \leq 1 \\ 0 \leq & v_\Box & \leq 1 \ . \end{array} \tag{4.35}$$

Note that the above only holds if the patch is indeed both convex and coplanar. An illustration of the $(u_\Box, v_\Box)$ patch coordinate system is shown in Figure 4.6.

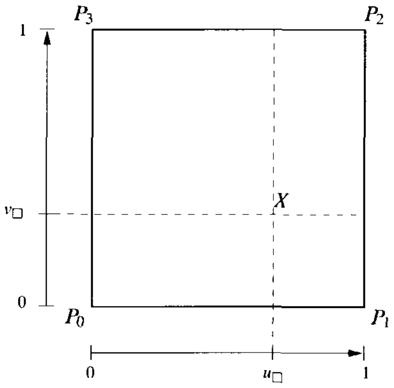


Figure 4.6. The $(u_\Box, v_\Box)$ patch coordinate system.

We propose to use an alternative form of patch coordinate system, which is based on splitting the patch into two triangular polygons, rather than one square polygon. We split the patch along the line P_0P_2 into two triangles T_0 and T_1. Let triangle T_0 be given by the vertices $P_0P_1P_2$, and let T_1 be given by the vertices $P_0P_2P_3$. Let us define the parameter t_Δ as being 0 if a point is inside triangle T_0, and as being 1 if it is inside triangle T_1. We define the alternative patch coordinate system by a tuple $(t_\Delta, u_\Delta, v_\Delta)$, where the parameters $t_\Delta, u_\Delta, v_\Delta$ satisfy:

$$\begin{array}{rcl} & t_\Delta & \in \{0,1\} \\ 0 \leq & u_\Delta & \leq 1 \\ 0 \leq & v_\Delta & \leq 1 \end{array} \tag{4.36}$$

and state that any point **x** on the patch can be represented as a bi-linear combination the patches' vertices, in terms of the parameters t_Δ, u_Δ and v_Δ as given by:

$$\left\{ \begin{array}{rcll} \mathbf{x} & = & (1-v_\Delta)((1-u_\Delta)\mathbf{p}_0 + (u_\Delta)\mathbf{p}_1) & \text{if} \quad t_\Delta = 0 \\ & + & (v_\Delta)((1-u_\Delta)\mathbf{p}_0 + (u_\Delta)\mathbf{p}_2) & \\ & & & \\ \mathbf{x} & = & (1-v_\Delta)((1-u_\Delta)\mathbf{p}_0 + (u_\Delta)\mathbf{p}_2) & \text{if} \quad t_\Delta = 1 \\ & + & (v_\Delta)((1-u_\Delta)\mathbf{p}_3 + (u_\Delta)\mathbf{p}_2) & \end{array} \right. \tag{4.37}$$

which can be simplified to:

$$\begin{cases} \mathbf{x} = (1-u_\Delta)\mathbf{p}_0 + (u_\Delta)\left((1-v_\Delta)\mathbf{p}_1 + (v_\Delta)\mathbf{p}_2\right) & \text{if} \quad t_\Delta = 0 \\ \mathbf{x} = (1-u_\Delta)\left((1-v_\Delta)\mathbf{p}_0 + (v_\Delta)\mathbf{p}_3\right) + (u_\Delta)\mathbf{p}_2 & \text{if} \quad t_\Delta = 1 \end{cases} \tag{4.38}$$

An illustration of the $(t_\Delta, u_\Delta, v_\Delta)$ patch coordinate system is shown in Figure 4.7.

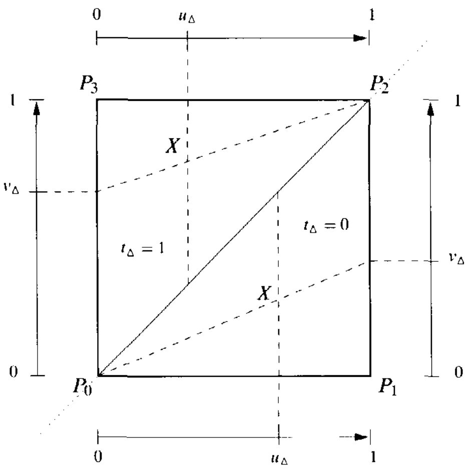


Figure 4.7. The $(t_\Delta, u_\Delta, v_\Delta)$ patch coordinate system.

We use the patch coordinate system for a number of purposes:

1. To formulate the problem of finding the intersection point between a ray and a patch.

2. To test if the ray hits the patch. The patch coordinates of the intersection point satisfy Equation (4.36), *if and only if* the ray intersects the patch.

3. To store the intersection point. From the patch coordinates $(t_\Delta, u_\Delta, v_\Delta)$ of the intersection point, we can compute its coordinates in the global coordinate system using Equation (4.38), or in the local coordinate system, using an analogous formula.

4. To compute the intersection distance. The distance from the sample point to the intersection point can also be written as a bi-linear combination of distances to the vertices of the patch, as we shall see later.

5. To perform radiosity shading of the patch. To compute the light intensity at a given point of the patch, a bi-linear interpolation is done of the radiosity values at the vertices of the patch. Likewise, the patch coordinates can also be used for the inverse problem, that of distributing the ray's energy to the patch vertices.

4.4.3 The bounding planes method

The method we use to compute the intersection point is the "bounding planes" method, as proposed by Hekstra in [10]. In principle, the bounding planes are used to determine whether a ray intersects with a patch or not (hit / no-hit). The effectiveness of the method lies in the fact that we *re-use* many of the results of the bounding plane computations for the computation of the patch coordinates. In turn, the intersection distance computation follows from these patch coordinates.

The essential operation in all the above computations is the computation of an inner product of a given vector (bounding plane normal or patch vertex) with the direction vector **r** of the ray. Since we are dealing with a ray frustum $\mathcal{R}$, as determined by the ISBB of the patch, we have to perform these inner product computations for each ray in the frustum. We show how we can use knowledge of the structure of the ray frustum to incrementally compute these inner products at a very low computational cost using the fast rotations of Chapter 3. These fast rotations exist for certain angles only, and have an inherent magnification factor which is not exactly equal to unity. We conclude by looking at practical values for the ray resolution, as determined by the choice of fast rotations, and the effect on the accuracy of computations.

Hit / no-hit computation

We define a bounding plane as the plane which goes through the sample point S, being the origin of the rays, and two adjacent vertices of the destination patch P_k, P_l, see Figure 4.8.

We define the bounding plane normal $\mathbf{n}_{kl}$, as the normal of the bounding plane, and such that it is on the side of the plane, on which the patch is, when its front face is facing the sample point. The bounding plane normal is given by the cross-product[4]:

$$\mathbf{n}_{kl} = (\mathbf{p}_l - \mathbf{s}) \times (\mathbf{p}_k - \mathbf{s})\,. \tag{4.39}$$

See also Figure 4.8 for the orientation of the bounding plane normal. Let **v** be a vector to a point on the bounding plane. The plane equation of the bounding plane is then given by:

$$\mathbf{n}_{kl} \cdot \mathbf{v} = \mathbf{n}_{kl} \cdot \mathbf{s}\,. \tag{4.40}$$

[4]The definition of the cross-product follows the "right-hand" rule: $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

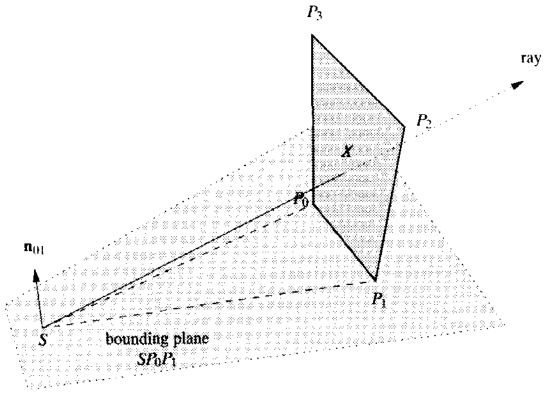


Figure 4.8. Bounding plane setup.

Rewriting Equation (4.39) in the local coordinate system results in:

$$\mathbf{n}'_{kl} = \mathbf{p}'_l \times \mathbf{p}'_k . \tag{4.41}$$

which shows the advantage to compute the bounding plane normals in the local coordinate system. Similarly, the plane equation of Equation (4.40) simplifies to:

$$\mathbf{n}'_{kl} \cdot \mathbf{v}' = 0 . \tag{4.42}$$

We use the bounding planes to find out whether a ray intersects the patch without computing the intersection point explicitly. Let us consider the bounding planes that lie on the circumference of the patch, with bounding plane normals $\mathbf{n}_{01}, \mathbf{n}_{12}, \mathbf{n}_{23}, \mathbf{n}_{30}$. A ray intersects with the patch, if and only if, the ray is inside the pyramidal cone formed by these bounding planes. This means it must lie on the right side of the bounding plane, facing the normal, for all bounding planes simultaneously.

Let us consider the line equation of the ray, given by Equations (4.4) and (4.16) in the global and local coordinate systems. To test whether a point **x** on the ray is on the right side of a bounding plane, we use the following procedure. We can express the point **x** as:

$$\mathbf{x} = \mathbf{v} + d\mathbf{n}_{kl} , \tag{4.43}$$

where **v** is a vector to a point in the bounding plane, and d is a dimensionless parameter. The point **x** is on the right side of the bounding plane or on the plane itself, if and only if $d \geq 0$. Solving for d, substituting Equations (4.43) and (4.4) in the plane Equation (4.40), results in:

$$d = \lambda \frac{\mathbf{n}_{kl} \cdot \mathbf{r}}{\mathbf{n}_{kl} \cdot \mathbf{n}_{kl}}, \tag{4.44}$$

with an analogous formula in the local coordinate system. Regarding the fact that $\lambda \geq 0$ for a valid point on the ray, and $\mathbf{n}_{kl} \cdot \mathbf{n}_{kl} > 0$, we introduce the **pseudo-distance** δ_{kl} as given by:

$$\delta_{kl} = \mathbf{n}_{kl} \cdot \mathbf{r}, \tag{4.45}$$

and state that, for a ray to be on the right side of the bounding plane, it must satisfy:

$$\delta_{kl} \geq 0. \tag{4.46}$$

In the local coordinate system, we can rewrite Equation (4.45) as:

$$\delta_{kl} = \mathbf{n}'_{kl} \cdot \mathbf{r}'. \tag{4.47}$$

The pseudo-distance is a measure of how far the ray is located from the bounding plane. We will see its use later on in the determination of the intersection point.

Concerning the test whether a ray intersects with a patch, we can state the following. A ray with origin $\mathbf{s}$ and direction vector $\mathbf{r}$ intersects the (convex, co-planar) patch if and only if the pseudo-distances $\delta_{01}, \delta_{12}, \delta_{23}, \delta_{30}$ are all simultaneously greater or equal to zero:

$$\begin{array}{rcl} \delta_{01} & \geq & 0 \\ \delta_{12} & \geq & 0 \\ \delta_{23} & \geq & 0 \\ \delta_{30} & \geq & 0 \,. \end{array} \tag{4.48}$$

This is illustrated in Figure 4.9, where the dark shaded region corresponds to all pseudo-distances greater or equal to zero. The lighter shading corresponds to a single bounding plane. In the intersection computation, we perform this hit / no-hit computation first, to see if it is necessary to proceed with calculating the actual intersection.

Due to the nature of the cross-product, the following properties hold for the bounding plane normal $\mathbf{n}_{kl}$ and the pseudo-distance δ_{kl}.

Using the property $\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$ of the cross product, we arrive at

$$\begin{array}{rcl} \mathbf{n}_{kl} & = & -\mathbf{n}_{lk} \\ \delta_{kl} & = & -\delta_{lk} \,. \end{array} \tag{4.49}$$

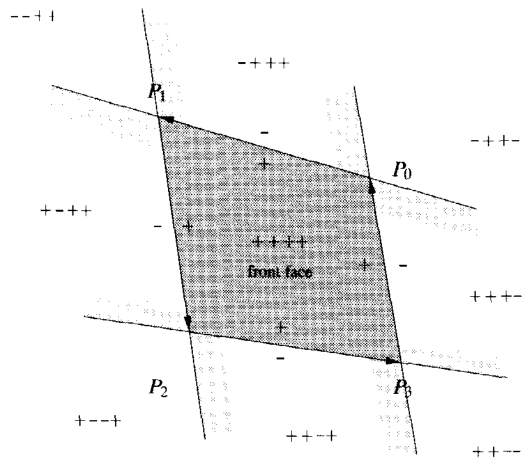


Figure 4.9. The regions defined by the bounding planes, showing the signs of the pseudo-distances.

Let λ, μ be dimensionless parameters. Let the vector $\mathbf{p}_m$ be the linear combination of the vectors $\mathbf{p}_j$ and $\mathbf{p}_k$, as given by:

$$\mathbf{p}_m = \lambda \mathbf{p}_j + \mu \mathbf{p}_k . \tag{4.50}$$

Using the distributive property $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) + (\mathbf{a} \times \mathbf{c})$ and the associative property $\mathbf{a} \times (\lambda \mathbf{b}) = \lambda(\mathbf{a} \times \mathbf{b})$ of the cross product, we arrive at:

$$\begin{aligned} \mathbf{n}_{im} &= \lambda \mathbf{n}_{ij} + \mu \mathbf{n}_{ik} \\ \delta_{im} &= \lambda \delta_{ij} + \mu \delta_{ik} \quad . \end{aligned} \tag{4.51}$$

Patch coordinate computation

Once it is known whether a ray actually intersects the patch, we can continue with finding the intersection point and the intersection distance. We do not explicitly compute the coordinates of the intersection point X in the global or local coordinate system. Instead, we compute the $(t_\Delta, u_\Delta, v_\Delta)$ patch coordinates of the intersection point, as defined by Equations (4.36) and (4.38). From this coordinate system, we can compute the local or global coordinates, following Equation (4.38).

First we must determine in which of triangles T_0, T_1 the intersection point lies. We propose to use an extra bounding plane through S, P_0, and P_2, with triangle T_1 on its right side. We compute the pseudo-distance δ_{02}, which implies that X is in triangle

T_1 if $\delta_{02} \geq 0$ and in triangle T_0 otherwise. From this, we determine t_Δ to be given by:

$$t_\Delta = \begin{cases} 0 & \text{if} \quad \delta_{02} < 0 \\ 1 & \text{if} \quad \delta_{02} \geq 0 \end{cases} . \tag{4.52}$$

Let **x** be the vector that points to the intersection point. The solution of the intersection point follows from setting **x** on the patch in Equation (4.38) equal to **x** on the ray in Equation (4.16). The solution of t_Δ from Equation (4.52) tells us which of the systems of the triangles to choose. Instead of attempting to solve the patch coordinates directly from the system of the individual coordinates, we use a method with parameterized planes, that can be seen as a form of bounding planes.

For this treatise, we assume that $t_\Delta = 0$, so we select only the equations that are valid for this value. We consider only triangle T_0, with vertices P_0, P_1, P_2. Furthermore, we prefer to perform all computations in the local coordinate system. To recapitulate, the ray equation is given by

$$\mathbf{x}' = \lambda \mathbf{r}' ,$$

and the patch equation is given by:

$$\mathbf{x}' = (1-u_\Delta)\mathbf{p}_0' + (u_\Delta)\left((1-v_\Delta)\mathbf{p}_1' + (v_\Delta)\mathbf{p}_2'\right)$$

for $t_\Delta = 0$. Let P_v be the point between vertices P_1 and P_2, with vector $\mathbf{p}_v'$ given by:

$$\mathbf{p}_v' = (1-v_\Delta)\mathbf{p}_1' + (v_\Delta)\mathbf{p}_2' . \tag{4.53}$$

Let us consider the plane that goes through the points S, P_0 and P_v. This plane is parameterized in v_Δ only. The plane normal is given by:

$$\mathbf{n}_v' = \mathbf{p}_v' \times \mathbf{p}_0' , \tag{4.54}$$

and the plane equation for a point **x** on the plane is given by:

$$\mathbf{n}_v' \cdot \mathbf{x}' = 0 . \tag{4.55}$$

See Figure 4.10 for an illustration of the construction of this plane.

Substitution of Equation (4.53) and (4.41) reduces the normal to:

$$\begin{aligned} \mathbf{n}_v' &= (1-v_\Delta)\mathbf{p}_1' \times \mathbf{p}_0' + (v_\Delta)\mathbf{p}_2' \times \mathbf{p}_0' \\ &= (1-v_\Delta)\mathbf{n}_{01}' + (v_\Delta)\mathbf{n}_{02}' \qquad , \end{aligned} \tag{4.56}$$

where we encounter the bounding plane normals again.

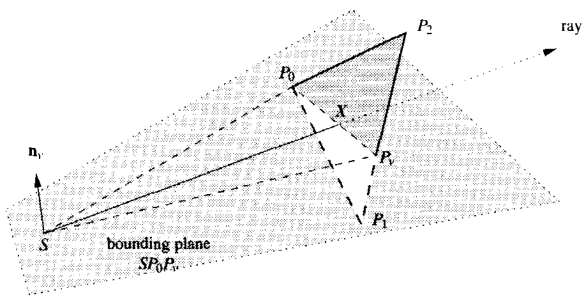


Figure 4.10. Construction of the plane, parameterized in v_Δ.

Setting the ray Equation (4.16) equal to the plane Equation (4.55) and substituting (4.56) results in the partial solution for the intersection point:

$$\lambda\left((1-v_\Delta)\mathbf{n}'_{01}\cdot\mathbf{r}' + (v_\Delta)\mathbf{n}'_{02}\cdot\mathbf{r}'\right) = 0. \tag{4.57}$$

Using our knowledge that $\lambda \geq 0$, and substituting Equation (4.45) results in the equation for the patch coordinate v_Δ, expressed in terms of the pseudo-distances, as given by:

$$(1-v_\Delta)\delta_{01} + (v_\Delta)\delta_{02} = 0. \tag{4.58}$$

Note that the pseudo-distances δ_{01} and δ_{02} have been computed before for the hit / no-hit and the t_Δ calculation. The value of v_Δ follows from the solution of Equation (4.58) as given by:

$$v_\Delta = \frac{\delta_{01}}{\delta_{01}-\delta_{02}}. \tag{4.59}$$

Instead of solving v_Δ by means of Equation (4.59), which requires a division, we prefer to express the coordinates in the form of Equation (4.58), and to solve it with a subdivision process, as we shall show later. This is a more robust operation than division, and has the advantage that other computations, such as the distance computations, can be performed on-line with the subdivision process.

For determining the u_Δ coordinate, we use a similar procedure. Let us define the points P_{u1} and P_{u2}, parameterized in u_Δ only, with vectors $\mathbf{p}'_{u1}, \mathbf{p}'_{u2}$ given by:

$$\begin{aligned}\mathbf{p}'_{u1} &= (1-u_\Delta)\mathbf{p}'_0 + (u_\Delta)\mathbf{p}'_1 \\ \mathbf{p}'_{u2} &= (1-u_\Delta)\mathbf{p}'_0 + (u_\Delta)\mathbf{p}'_2 .\end{aligned} \tag{4.60}$$

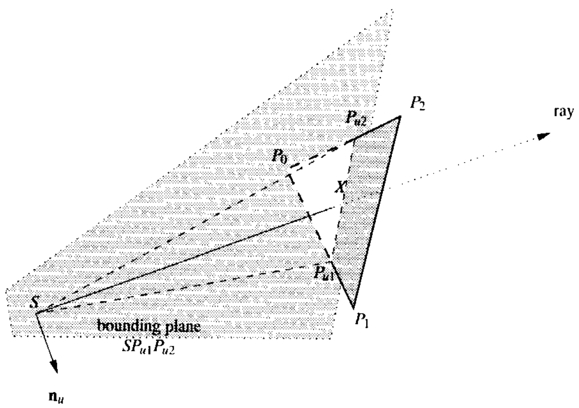


Figure 4.11. Construction of the plane, parameterized in u_Δ.

We use these points to build a bounding plane, going through the points S, P_{u1}, P_{u2}, which is parameterized in u_Δ only. Let $\mathbf{n}'_u$ be the normal of this bounding plane. See Figure 4.11 for an illustration of the construction of this plane. Using a similar derivation as in Equations (4.55) to (4.56), we arrive at the value for $\mathbf{n}'_u$ as given by:

$$\mathbf{n}'_u = (1-u_\Delta)(\mathbf{n}'_{02}-\mathbf{n}'_{01}) + (u_\Delta)\mathbf{n}'_{12}, \tag{4.61}$$

which has a similar structure to Equation (4.56). Note that all the bounding plane normals, $\mathbf{n}'_{01}, \mathbf{n}'_{02}, \mathbf{n}'_{12}$, have been computed before for other calculations. The solution of u_Δ then follows, with a similar derivation as in Equations (4.57) to (4.59), as given by:

$$(1-u_\Delta)(\delta_{20}-\delta_{10}) + (u_\Delta)\delta_{21} = 0. \tag{4.62}$$

We can also derive the solutions for the case $t_\Delta = 1$, that result in the complete set of equations for u_Δ, v_Δ as given by:

$$\begin{cases} (1-u_\Delta)(\delta_{20}-\delta_{10}) + (u_\Delta)\delta_{21} = 0 & \text{if} \quad t_\Delta = 0 \\ (1-u_\Delta)\delta_{30} + (u_\Delta)(\delta_{32}+\delta_{20}) = 0 & \text{if} \quad t_\Delta = 1 \end{cases} \tag{4.63}$$

and

$$\begin{cases} (1-v_\Delta)\delta_{01} + (v_\Delta)\delta_{02} = 0 & \text{if} \quad t_\Delta = 0 \\ (1-v_\Delta)\delta_{02} + (v_\Delta)\delta_{32} = 0 & \text{if} \quad t_\Delta = 1 \end{cases}. \tag{4.64}$$

Together with Equation (4.52), they form the solution of the intersection point, as expressed in the $(t_\Delta, u_\Delta, v_\Delta)$ patch coordinate system.

Intersection distance computation

We define the intersection distance t, as the Euclidean distance between the origin of the ray, S, to the intersection point X. The intersection distance is given by:

$$\begin{aligned} t &= \|\mathbf{x}-\mathbf{s}\| \\ &= \lambda\|\mathbf{r}\| \quad . \end{aligned} \tag{4.65}$$

Normally speaking, computing the distance would require three subtractions, three multiplications or squaring operations, two additions and a square-root operation. If the length of the direction vector $\|\mathbf{r}\|$ is known, then one could also compute λ, which requires at least one division, and then to multiply this with $\|\mathbf{r}\|$ to form the distance.

For hemisphere rays, the ray direction vector $\mathbf{r}$ is normalized, i.e. $\|\mathbf{r}\| = 1$. We use this to formulate a technique to compute the intersection distance, without having to perform expensive square-root or division operations. We choose to concentrate on the inner product $\mathbf{x}\cdot\mathbf{r}$, which we re-write according to:

$$\begin{aligned} \mathbf{x}\cdot\mathbf{r} &= \lambda\mathbf{r}\cdot\mathbf{r} \\ &= \lambda\|\mathbf{r}\|^2 \\ &= t\|\mathbf{r}\| \quad . \end{aligned} \tag{4.66}$$

Under the assumption that the ray direction is normalized, we can state:

$$t = \mathbf{x}\cdot\mathbf{r}. \tag{4.67}$$

From Equation (4.38), we know that we can express the intersection point $\mathbf{x}$ as a bi-linear combination of the patches' vertices, in terms of the $(t_\Delta, u_\Delta, v_\Delta)$ patch coordinates. Substituting this Equation (4.38) in (4.67), results in:

$$\begin{cases} t = (1-u_\Delta)\mathbf{p}_0\cdot\mathbf{r} + (u_\Delta)\left((1-v_\Delta)\mathbf{p}_1\cdot\mathbf{r} + (v_\Delta)\mathbf{p}_2\cdot\mathbf{r}\right) & \text{if} \quad t_\Delta = 0 \\ t = (1-u_\Delta)\left((1-v_\Delta)\mathbf{p}_0\cdot\mathbf{r} + (v_\Delta)\mathbf{p}_3\cdot\mathbf{r}\right) + (u_\Delta)\mathbf{p}_2\cdot\mathbf{r} & \text{if} \quad t_\Delta = 1 \end{cases} . \tag{4.68}$$

Defining the distances t_i as given by:

$$t_i = \mathbf{p}_i\cdot\mathbf{r}, \tag{4.69}$$

we can simplify Equation (4.68) to:

$$\begin{cases} t = (1-u_\Delta)t_0 + (u_\Delta)\left((1-v_\Delta)t_1 + (v_\Delta)t_2\right) & \text{if} \quad t_\Delta = 0 \\ t = (1-u_\Delta)\left((1-v_\Delta)t_0 + (v_\Delta)t_3\right) + (u_\Delta)t_2 & \text{if} \quad t_\Delta = 1 \end{cases} . \tag{4.70}$$

Note that the distance t_i is the distance from the vertex P_i to the plane, going through the sample point S and with normal given by $\mathbf{r}$. In the local coordinate system, the same holds, and the t_i can also be computed accordingly, as given by:

$$t_i = \mathbf{p}'_i \cdot \mathbf{r}' . \tag{4.71}$$

So, if we have the patch coordinates, we can compute the intersection distance using the above formulas. If the t_i are given, this requires at most three multiplications and two additions. It is also possible to construct t on-line, during the evaluation of the patch coordinates in the sub-division process, as we shall see later. This results in an even more efficient implementation.

Normally speaking, the evaluation of the t_i, requiring four 3-D inner product computations, is considered costly. However, we will present a technique of computing these inner products at an extremely low cost, so this is no longer prohibitive.

4.4.4 Fast generation of inner products

As we have seen previously, much of the intersection computation revolve around the calculation of the inner product between a given vector and the ray direction.

In Table 4.2 we recapitulate the different quantities for which we have to compute such an inner product, and where they are used. These quantities have to be re-

type of computation	quantities	formula	equations
hit / no-hit	$\delta_{01}, \delta_{12}, \delta_{23}, \delta_{30}$	$\delta_{kl} = \mathbf{n}'_{kl} \cdot \mathbf{r}'$	(4.48)
patch coordinates	$\delta_{01}, \delta_{12}, \delta_{23}, \delta_{30}, \delta_{02}$	$\delta_{kl} = \mathbf{n}'_{kl} \cdot \mathbf{r}'$	(4.52), (4.63), (4.64)
intersection distance	t_0, t_1, t_2, t_3	$t_k = \mathbf{p}'_k \cdot \mathbf{r}'$	(4.70)

Table 4.2. Inner product computations for intersection

computed for every ray that is tested for intersection with the patch. These rays are, as mentioned before, selected by the ISBB. The bounding plane normals for the patch, five in total, are computed once per patch only, and re-used for every ray in the ISBB.

Let $\mathbf{r}'_{ij}$ be a hemisphere ray with ray index (i, j). The ray is tested against the patch if the ray index satisfies:

$$\begin{aligned} l_1 \leq \; & i \; \leq u_1 \\ l_2 \leq \; & j \; \leq u_2 \; . \end{aligned} \tag{4.72}$$

We are, in principle, free to generate these rays any way we like. Most likely, we will use two nested 'for' loops to generate them.

A naive way of computing the quantities of Table 4.2 would be to first compute the ray direction $\mathbf{r}_{ij}$, and then to compute the inner products. A quick calculation reveals the cost to be: two sine/cosine computations, $2+9\times3=29$ multiplications and $9\times2=18$ additions.

Instead we propose the following scheme. Let $\mathbf{v}'$ be a vector with which to compute the inner product with the ray directions $\mathbf{r}'_{ij}$. The vector $\mathbf{v}'$ can be any of the bounding plane normals $\mathbf{n}'_{kl}$ or the vertices $\mathbf{p}'_k$, as was shown in Table 4.2.

Let v_{ij} be the desired quantity, corresponding to the ray indexed by (i,j), formed by the inner product as given by:

$$v_{ij} = \mathbf{v}' \cdot \mathbf{r}'_{ij}. \tag{4.73}$$

We prefer to write this inner product as $(\mathbf{v}')^T\mathbf{r}'_{ij}$, since we will use it in equations involving matrices.

If we substitute Equation (4.20) for the ray direction vector in the above, we obtain:

$$v_{ij} = \mathbf{v}'^T\mathbf{R}_{xy}(\varphi_i)\mathbf{R}_{zx}(\theta_j)\begin{bmatrix}0\\0\\1\end{bmatrix}. \tag{4.74}$$

Let us define the alternative, indexed rotation matrices $\bar{\mathbf{R}}_{xy}(i)$ and $\bar{\mathbf{R}}_{zx}(j)$, as given by:

$$\bar{\mathbf{R}}_{xy}(i) = \begin{bmatrix}\cos(\varphi_i) & -\sin(\varphi_i) & 0\\ \sin(\varphi_i) & \cos(\varphi_i) & 0\\ 0 & 0 & 1\end{bmatrix} \tag{4.75}$$

and

$$\bar{\mathbf{R}}_{zx}(j) = \begin{bmatrix}\cos(\theta_j) & -\sin(\theta_j)\\ \sin(\theta_j) & \cos(\theta_j)\end{bmatrix}. \tag{4.76}$$

With the above, we can re-write Equation (4.74) to:

$$v_{ij} = \mathbf{v}'^T\bar{\mathbf{R}}_{xy}(i)\begin{bmatrix}0 & 1\\0 & 0\\1 & 0\end{bmatrix}\bar{\mathbf{R}}_{zx}(j)\begin{bmatrix}1\\0\end{bmatrix}. \tag{4.77}$$

We will now explore the possibilities of incremental computation of the v_{ij}. Let us define the vector $\mathbf{v}'_{ij}$ as given by:

$$\mathbf{v}'^T_{ij} = \mathbf{v}'^T\bar{\mathbf{R}}_{xy}(i)\begin{bmatrix}0 & 1\\0 & 0\\1 & 0\end{bmatrix}\bar{\mathbf{R}}_{zx}(j). \tag{4.78}$$

which allows us to re-write Equation (4.74) once more to:

$$v_{ij} = \mathbf{v}_{ij}^{\prime T} \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{4.79}$$

which shows that the computation of v_{ij} is merely the selection of the x component of the vector $\mathbf{v}'_{ij}$.

Let $\mathbf{R}(\alpha)$ be the 2×2 rotation matrix, given by:

$$\mathbf{R}(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}. \tag{4.80}$$

It is easy to see that the incremental relationship:

$$\bar{\mathbf{R}}_{zx}(j+1) = \bar{\mathbf{R}}_{zx}(j)\mathbf{R}(\Delta_\theta) \tag{4.81}$$

holds for the indexed rotation matrix $\bar{\mathbf{R}}_{zx}(j)$. Applying this to Equation (4.79), results in the incremental relationship for the vectors $\mathbf{v}_{ij}$ which is given by:

$$\mathbf{v}_{i,j+1}^{\prime T} = \mathbf{v}_{ij}^{\prime T}\mathbf{R}(\Delta_\theta) \tag{4.82}$$

This shows that $\mathbf{v}_{i,j+1}$ can be computed from $\mathbf{v}_{ij}$ with just a single rotation over the angle resolution Δ_θ. The desired quantity $v_{i,j+1}$ then follows by selection of the x component of $\mathbf{v}_{i,j+1}$, at no extra cost.

The computation scheme that we propose consists of first computing the vector $\mathbf{v}_{i,l_2}$ for index i on a row-by row basis for $l_1 \leq i \leq u_1$. This is typically an expensive computation, but computed only once per i. From here on, we can compute the vectors $\mathbf{v}_{i,j}$ a single rotation using with the incremental scheme of Equation (4.82) for $l_2 \leq j \leq u_2$.

One further, essential improvement is the replacement of the rotation $\mathbf{R}(\Delta_\theta)$ with one of the fast rotation methods of Chapter 3. This does, however, restrict our choice of the angle resolution Δ_θ. Since we are dealing with a *sampling* problem, we are free to choose or adjust our sampling resolution to what we desire, so this restriction is not a problem.

We take $\mathbf{F}_\theta$, conform Equation (3.1) of Chapter 3, to be given by:

$$\mathbf{F}_\theta = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}, \tag{4.83}$$

with magnification factor m and angle of rotation α as given in Equations (3.2) and (3.3).

We propose to use Method II fast rotations, as they are the most suitable over the entire range of angle resolutions, and with the desired accuracy. Let us define the angle exponents κ_θ and κ_φ which determine the angle resolutions Δ_θ and Δ_φ. We will only use the angle exponents κ_θ in actual fast rotations. Even though, we use the angle exponent κ_φ in a similar way to fix the angle resolution Δ_φ. The approximation pair (c, s) is then given, conform Equation (3.19), by:

$$\begin{aligned} c &= 1 - 2^{2\kappa_\theta - 1} \\ s &= 2^{\kappa_\theta} \quad . \end{aligned} \tag{4.84}$$

This, in turn, fixes the angle resolution Δ_θ, as given by:

$$\Delta_\theta = \arctan\left(\frac{2^{\kappa_\theta}}{1 - 2^{2\kappa_\theta - 1}}\right), \tag{4.85}$$

and the magnification factor m, as given by:

$$m = \sqrt{1 + 2^{4\kappa_\theta - 2}}. \tag{4.86}$$

Similarly, the angle resolution Δ_θ is given by:

$$\Delta_\varphi = \arctan\left(\frac{2^{\kappa_\varphi}}{1 - 2^{2\kappa_\varphi - 1}}\right), \tag{4.87}$$

even though it is not used for fast rotations.

To compensate for the magnification factor for low resolutions, we redefine the indexed rotation matrix $\tilde{\mathbf{R}}_{zx}(j)$, as given by:

$$\tilde{\mathbf{R}}_{zx}(j) = \begin{bmatrix} m^j \cos(\theta_j) & -m^j \sin(\theta_j) \\ m^j \sin(\theta_j) & m^j \cos(\theta_j) \end{bmatrix}, \tag{4.88}$$

such that the incremental relationship

$$\tilde{\mathbf{R}}_{zx}(j+1) = \tilde{\mathbf{R}}_{zx}(j)\mathbf{F}_\theta \tag{4.89}$$

holds. With this knowledge, we can re-write Equation (4.82) to derive the incremental scheme with fast rotations as:

$$\begin{aligned} v_{i,j+1} &= \mathbf{v}'^T_{i,j+1} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \mathbf{v}'^T_{ij} \mathbf{F}_\theta \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad . \end{aligned} \tag{4.90}$$

Incorporating fast rotations in the incremental scheme results in a large reduction of the computational complexity. The quantities in Table 4.2 can be computed incrementally with 9 fast rotations per ray, which amounts to 36 shift-add operations. Compared to the naive scheme, with 29 multiplications, 18 additions, and sine/cosine computations, this is more that an order of magnitude less[5]. Moreover, a fast rotation can be executed in a single clock cycle, resulting in a throughput per ray of only 9 cycles.

Practical values for the ray resolution

We have seen that the ray resolution is determined by an integer angle exponent $\kappa_\varphi, \kappa_\theta$. Regarding the target minimum and maximum values for the angle resolution, as presented in Table 4.1, we can determine the range of the angle exponents. The range of angle exponents $\kappa_\varphi, \kappa_\theta$, the corresponding angle resolutions $\Delta_\varphi, \Delta_\theta$, and the number of rays N_φ, N_θ, N_r are shown on Table 4.3. The total number of rays N_r, as presented in this table, is based on the assumption that the angle resolutions $\Delta_\varphi, \Delta_\theta$ are equal. In practical cases, this assumption can be dropped for extra flexibility in possible configurations.

$\kappa_\varphi, \kappa_\theta$	$\Delta_\varphi, \Delta_\theta$	N_φ	N_θ	N_r
-4	$6.25 \cdot 10^{-2}$	99	24	2376
-5	$3.13 \cdot 10^{-2}$	200	49	9800
-6	$1.56 \cdot 10^{-2}$	401	100	$4.01 \cdot 10^4$
-7	$7.81 \cdot 10^{-3}$	803	200	$1.61 \cdot 10^5$
-8	$3.91 \cdot 10^{-3}$	1607	401	$6.44 \cdot 10^5$
-9	$1.95 \cdot 10^{-3}$	3216	803	$2.58 \cdot 10^6$
-10	$9.77 \cdot 10^{-4}$	6433	1607	$1.03 \cdot 10^7$
-11	$4.88 \cdot 10^{-4}$	12867	3216	$4.14 \cdot 10^7$
-12	$2.44 \cdot 10^{-4}$	25735	6433	$1.66 \cdot 10^8$

Table 4.3. Practical values for the ray resolution

We propose to use the range for $\kappa_\varphi, \kappa_\theta$ from -4 (low resolution) up to and including -12 (very high resolution). Judging from the target limits, as proposed in Table 4.3, the range of $-5, \ldots, -10$ would suffice. We have chosen to extend it in both directions with additional resolutions (shown in boldface in the table). The reason behind this is that there are different modes of frustum shooting, aside from shooting *all* hemisphere rays, we also allow:

[5]Assuming a word-size of 24 bit, the multiplications amount in $29 \times 24 = 696$ adder equivalents. Together with the remaining additions, not even taking the sine/cosine into account, this is 714 additions. This is almost a factor of 20.

- shooting of a part of the hemisphere at a locally much higher resolution. An example of this is the shooting of shadow rays towards light sources in the environment only.
- shooting of a narrow frustum at the apex of the hemisphere. An example of this is the shooting of a reflection frustum of an almost specular surface. The construction of the rays, using a regular grid in the hemisphere coordinate system, causes the rays to be very dense at the apex in the φ dimension. Here we can choose a much lower resolution for Δ_φ only, resulting in a lower number of rays.

Accuracy considerations

The use of fast rotations and the incremental computation scheme introduces errors in the results. A single fast rotation introduces a magnification error ε which is inherent in the magnification factor m. This error ε, for the used Method II as given by Equation (3.20), is a function of the angle exponent κ_θ. Fast rotations are only used for rotations in the θ dimension, so we only consider this dimension, and the angle exponent κ_θ.

The incremental computation scheme causes a buildup of these errors. In a single thread of computation, at most N_θ consecutive fast rotations occur. Let us consider the error buildup in n consecutive fast rotations. We define the overall magnification error $\varepsilon_{(n)}$ as the error in magnification of n consecutive fast rotations. The overall magnification factor is given as the product of n individual magnification factors, yielding m^n. This results in the overall magnification error $\varepsilon_{(n)}$ as given by:

$$\varepsilon_{(n)} = m^n - 1\,. \tag{4.91}$$

For ε small enough, we can use the first order approximation:

$$\begin{aligned} \varepsilon_{(n)} &= m^n - 1 \\ &= (1+\varepsilon)^n - 1 \\ &\approx n\varepsilon \qquad . \end{aligned} \tag{4.92}$$

Similarly, we define the overall accuracy $q_{(n)}$, analogous to Equation (3.39), as a measure of how close the overall magnification factor is to unity. It gives an indication at which bit position the error has an influence, or conversely, in how many bits accurate the result is. The overall accuracy $q_{(n)}$ is given by:

$$q_{(n)} = -\log_2 |m^n - 1| = -\log_2 |\varepsilon_{(n)}|\,. \tag{4.93}$$

Applying Equation (4.92), we arrive at the approximation, for small ε, as given by:

$$\begin{aligned} q_{(n)} &= -\log_2 |\varepsilon_{(n)}| \\ &\approx -\log_2 |n\varepsilon| \\ &\approx -\log_2(n) - \log_2 |\varepsilon| \\ &\approx -\log_2(n) + q \qquad . \end{aligned} \tag{4.94}$$

The above approximation shows that $\log_2(n)$ bits of accuracy are lost, compared to the accuracy q of a single fast rotation. This is what we would expect from the error buildup.

For the range of angle exponents that we consider, we have calculated the overall magnification error $\varepsilon_{(n)}$ and overall accuracy $q_{(n)}$, as presented in Table 4.4. The worst case number of consecutive fast rotations that can occur is given by $n = N_\theta$. Note that both N_θ and ε are a function of the angle exponent κ_θ.

κ_θ	N_θ	ε	$\varepsilon_{(N_\theta)}$	$q_{(N_\theta)}$
-4	24	$1.907 \cdot 10^{-6}$	$4.578 \cdot 10^{-5}$	14.42
-5	49	$1.192 \cdot 10^{-7}$	$5.841 \cdot 10^{-6}$	17.39
-6	100	$7.451 \cdot 10^{-9}$	$7.451 \cdot 10^{-7}$	20.36
-7	200	$4.657 \cdot 10^{-10}$	$9.313 \cdot 10^{-8}$	23.36
-8	401	$2.910 \cdot 10^{-11}$	$1.167 \cdot 10^{-8}$	26.35
-9	803	$1.819 \cdot 10^{-12}$	$1.461 \cdot 10^{-9}$	29.35
-10	1607	$1.137 \cdot 10^{-13}$	$1.827 \cdot 10^{-10}$	32.35
-11	3216	$7.105 \cdot 10^{-15}$	$2.285 \cdot 10^{-11}$	35.35
-12	6433	$4.441 \cdot 10^{-16}$	$2.857 \cdot 10^{-12}$	38.35

Table 4.4. Overall accuracy per ray resolution

We see that the overall accuracy is the lowest for the lower resolutions, as we could expect since the angle of rotation is the largest. This error influences only the accuracy of the intersection distance computation. It is not prohibitive, since we compensate for the growth in the magnification in the incremental scheme. For the same ray, the overall magnification is always the same. *Relatively*, in the comparison of the intersection distances of two patches, hit by the same ray, there is no error.

Another source of error buildup, this time possible destructive for the accuracy, is that of round-off errors in the operations. For the Method II fast rotations, this is the error induced by two addition operations. Regarding the maximum number of consecutive fast rotations N_θ for the highest resolution $\kappa_\theta = -12$, as given in Table 4.4, this results in a loss of accuracy of around $\log_2(2N_\theta) = \log_2(12866) \approx 13.6$ bits. The datapath that performs the fast rotations has been extended with an adequate amount of bits to counteract this error buildup.

4.5 Implications on other operations

We have seen in Section 4.4 that the "law of nice numbers" applies to the problem of computing the intersection points between a patch and a frustum of rays. By choosing the angle resolutions $\Delta_\varphi, \Delta_\theta$ such that they correspond to a fast rotation angle (our "nice number"), the computational cost is lowered dramatically.

Such choices, however, influence a whole number of operations which surround the intersection computation. There is no guarantee that the law of nice numbers automatically applies to these operations. Usually we see a trade-off. One example of this is the relation between the angle resolution for one dimension, say Δ_φ, and the number of rays in that dimension, in this case N_φ. Our trade-off here is:

- *either* to have N_φ as a power of two, which is favourable for memory usage. The angle resolution Δ_φ is then such that it requires expensive rotation techniques.
- *or* to choose Δ_φ to be a fast rotation angle, which is favourable to reduce the computational cost. The number of rays N_φ, is then no longer near a power of two, which implies that the memory is not fully used.

Our main objective is to reduce the computational complexity, which in this case outweighs the efficient use of memory, and would make the first option the preferred one.

We will use the computation of the ISBB from the SBB as an example, and see how the choice of the angle resolution influences the computational complexity.

4.5.1 ISBB computations

The computation of the optimal index bounds requires division (or multiplication), subtraction and floor and ceiling operations. Regarding the implementation, we would like to especially avoid the division and ceiling operations in the computation of the bounds, as given in Equation (4.33).

For $\kappa_\varphi, \kappa_\theta$ small enough, we can make use of the following approximation for the value of the angle resolution:

$$\begin{aligned} \Delta_\varphi &\approx 2^{\kappa_\varphi} \\ \Delta_\theta &\approx 2^{\kappa_\theta} \ . \end{aligned} \tag{4.95}$$

Substituting the above approximation in Equation (4.22) gives us the following estimate for the number of rays:

$$\begin{aligned} N_\varphi &\approx \pi 2^{-\kappa_\varphi+1} &< 2^{-\kappa_\varphi+3} \\ N_\theta &\approx \pi 2^{-\kappa_\theta-1} &< 2^{-\kappa_\theta+1} \ . \end{aligned} \tag{4.96}$$

Note that the number of hemisphere rays are not close to a power of two, which would be desired for efficient memory usage.

To simplify the arithmetic operations involved in the computation of the bounds, making use of the fact that Δ_φ is close to 2^{κ_φ}, we propose to use the following formula to obtain the sub-optimal index bounds:

$$\begin{aligned} l_1 &= \left\lfloor \frac{\varphi_{\min}}{2^{\kappa_\varphi}} \right\rfloor & u_1 &= \left\lfloor \frac{\varphi_{\max}}{2^{\kappa_\varphi}} \right\rfloor \\ l_2 &= \left\lfloor \frac{\theta_{\min}}{2^{\kappa_\theta}} \right\rfloor & u_2 &= \left\lfloor \frac{\theta_{\max}}{2^{\kappa_\theta}} \right\rfloor . \end{aligned} \tag{4.97}$$

In this approximation, the division by, for example, Δ_φ is replaced by a multiplication by $2^{-\kappa_\varphi}$, which is basically a shift left operation over $-\kappa_\varphi$ bit positions. Furthermore, we introduce some more slack and use only floor operators. This renders the address translation as no more than the shifting and selecting of the bits of the representations of φ and θ, when these are represented in radians. The suboptimal bounds, as given by Equation (4.97) must still satisfy the conditions as posed in Equation (4.29).

We will examine under what circumstances these conditions are satisfied. For the discussion below, we consider only the bound $\varphi_{\max}$. Substituting the sub-optimal bound $\varphi_{\max}$ of Equation (4.97) into the condition of Equation (4.29) gives:

$$\begin{aligned} \varphi_{\max} &< \varphi_{(u_1+1)} \\ \varphi_{\max} &< (u_1+1+1/2)\Delta_\varphi \\ \varphi_{\max} &< (\left\lfloor \tfrac{\varphi_{\max}}{2^{\kappa_\varphi}} \right\rfloor + 1 + 1/2)\Delta_\varphi . \end{aligned} \tag{4.98}$$

Using the property for the floor operator:

$$x - 1 < \lfloor x \rfloor \leq x . \tag{4.99}$$

and using this to tighten the condition, we obtain:

$$\frac{\varphi_{\max}}{\Delta_\varphi} < \frac{\varphi_{\max}}{2^{\kappa_\varphi}} + 1/2 . \tag{4.100}$$

We can re-write this to form the condition for the value of Δ_φ:

$$\Delta_\varphi > \frac{\varphi_{\max} 2^{\kappa_\varphi}}{\varphi_{\max} + 2^{\kappa_\varphi - 1}} . \tag{4.101}$$

The bound $\varphi_{\max}$ can take on values between 0 and 2π. Analysis of the function in the right term of (4.101), shows that first derivative in φ is always positive. The maximum takes place at the maximum value of $\varphi_{\max} = 2\pi$, resulting in:

$$\Delta_\varphi > \frac{2\pi 2^{\kappa_\varphi}}{2\pi + 2^{\kappa_\varphi - 1}} . \tag{4.102}$$

A similar condition can be established for the other bounds, which combined with the above condition (4.102), results in:

$$\frac{2\pi 2^{\kappa_\varphi}}{2\pi + 2^{\kappa_\varphi - 1}} < \Delta_\varphi < \frac{2\pi 2^{\kappa_\varphi}}{2\pi - 2^{\kappa_\varphi - 1}} \,. \tag{4.103}$$

To evaluate this condition, we must substitute the value for Δ_φ, as given by Equation (4.87), into condition (4.103) for all values in the range $\kappa_\varphi = -12, \ldots, -4$. We consider only the φ component since this is more restrictive due to its larger range. For the Method II fast rotation that we have proposed, the above formulas hold for the whole range of resolutions $\kappa_\varphi \in \{-12, \ldots, -4\}$ that we are interested in.

Compensating for the error in SBB or IPBB computation

The results presented above are valid only if the $\varphi_{\max}$ and $\varphi_{\min}$ values are computed without any residue error. In a practical situation, this is not the case. The current implementation computes the SBB to within an absolute error of $\varepsilon = 2^{-10}$. This error was found to be fine enough for practical application. Let $\varphi_{\max}$ be the non-exact result of the SBB computation, with an inherent error, and let $\tilde{\varphi}_{\max}$ be the exact result. The value of $\tilde{\varphi}_{\max}$ is then limited to the range given by:

$$\varphi_{\max} - \varepsilon < \tilde{\varphi}_{\max} < \varphi_{\max} + \varepsilon \,. \tag{4.104}$$

Unfortunately, the error is not small enough, so that (the modified) condition (4.103) is no longer satisfied for $k \leq -8$. Taking into account the inexact value for $\varphi_{\max}$ and $\varphi_{\min}$, we modify condition (4.29) using (4.104) to:

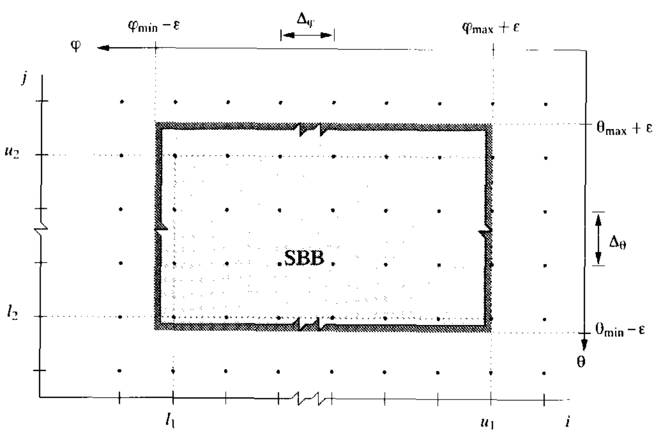
$$\begin{array}{rcl} \varphi_{(l_1-1)} & < & \varphi_{\min} - \varepsilon \\ \varphi_{(u_1+1)} & > & \varphi_{\max} + \varepsilon \\ \theta_{(l_2-1)} & < & \theta_{\min} - \varepsilon \\ \theta_{(u_2+1)} & > & \theta_{\max} + \varepsilon \;. \end{array} \tag{4.105}$$

This relation between the inexact SBB and the ISBB, according to the conditions of Equation (4.105) is illustrated in Figure 4.12.

Without further proof we present the simplified upper and lower bounds of the ISBB, taking into account the inexact computation of the SBB, as given by:

$$\begin{array}{rclcrcl} l_1 & = & \left\lfloor \dfrac{\varphi_{\min} - \varepsilon}{2^{\kappa_\varphi}} \right\rfloor & & u_1 & = & \left\lceil \dfrac{\varphi_{\max} + \varepsilon}{2^{\kappa_\varphi}} \right\rceil \\ l_2 & = & \left\lfloor \dfrac{\theta_{\min} - \varepsilon}{2^{\kappa_\theta}} \right\rfloor & & u_2 & = & \left\lceil \dfrac{\theta_{\max} + \varepsilon}{2^{\kappa_\theta}} \right\rceil \;. \end{array} \tag{4.106}$$

The computation of these bounds requires very little hardware in terms of datapath and control. Only one add-subtract operation plus a shift operation is necessary.



$\varphi_{\min} - \varepsilon$
Δ_φ
$\varphi_{\max} + \varepsilon$
φ
j
u_2
l_2
SBB
$\theta_{\max} + \varepsilon$
Δ_θ
$\theta_{\min} - \varepsilon$
θ
l_1
u_1
i

Figure 4.12. Computing the ISBB from an inexact SBB

The floor operation is nothing more than selecting a certain (fixed) number of bits from the result. The term ε is a power of two, and in this case the least significant bit of the representation. In case the hardware functionality is provided, the adder can be replaced by an incrementer/decrementer.

4.6 Implementation issues

It is a challenging problem to do the mixed software / hardware co-design of a parallel, high performance system, capable of photo-realistic rendering of artificial scenes. Moreover, when the requirement to come up with an efficient implementation has to be met. The Radiosity Engine [3] is such a system.

Smart techniques exist, such as the Shelling Technique [1], to reduce the amount of wasteful computations in the rendering algorithm. Without such preliminary, and in-depth work to reduce the computational complexity of the algorithm first, it would not make sense to attempt to map the algorithm onto any kind of system at all. While these techniques work well on 'software' which runs on a general-purpose uniprocessor, they pose new and challenging problems in VLSI system design when the algorithm is to be mapped on weakly programmable, mixed software-hardware systems. Classical methods for mapping algorithms onto onto parallel architectures, such as presented by Kung in [11] are quite quickly rendered useless.

A typical example of this is when an algorithm contains nested loops of which the loop bounds are non-manifest and data-dependent. This means that the dimensions of the loop space are not known at the moment of mapping. It is not possible to make up the dependence graph of the operations at "compile-time" to map this onto an architecture.

This implies that the system, must have a **self-scheduling** strategy to distribute workload over parallel processors. We see this happen at several levels in the system. At the highest level, workload balancing evens out the computational load over the parallel Engine boards in the system. At the lowest level, tasks are distributed on-line over three identical subdivision units to increase the throughput within the ICU ASIC.

The existence of data-dependent loop bounds also implies that the rates, or computation times of certain operations, within the system are no longer fixed. We are dealing with variable rate systems. With multi-dimensional data-dependent loops, we are dealing with a variable multi-rate system. Systolic architecture solutions are out of the question here.

We have found it necessary to develop models and methodologies to design and implement VLSI systems which are capable of tackling all of the above problems. This has lead to the concept of a multi-rate **peristaltic** pipeline (rather than systolic). which has been applied in the ICU.

In this section, we pay particular attention to the ICU as the most important member of the chip-set, and hope to illustrate some of the methodology and models with it. The ICU is also important as it incorporates, as the first, a fast rotation unit. The application of fast rotations has brought the feasibility of the ICU, at its current performance, well within reach.

4.6.1 The Radiosity Engine

The Radiosity Engine [3] is a high-performance system for photo-realistic image generation of artificial environments. Figure 4.13 shows the architecture of the Radiosity Engine. The host, for which we use a Hewlett-Packard HP9000/747i, is connected to a VME backplane through which it communicates with a cluster of Engine Boards. The Engine boards themselves have a local network with which they exchange information. They are designed specifically for high-speed radiosity and ray-tracing computation, such as cell traversal and intersection computation.

The Engine board itself consists of a Texas Instruments TMS320C40 DSP chip and a chip-set of three dedicated ASICs (CTU, GTU, ICU) to speed up computation. A schematic of the Engine board is shown in Figure 4.14. In this schematic, only the most important details, relevant for this treatise, are depicted. In Figure 4.20 in Appendix 4.A we show a photograph of the Engine board itself, as built by TNO-TPD. The principal components of the Engine board are:

- **DSP** This is a Texas Instruments TMS320C40 DSP chip, which is dedicated mainly to coordination, communication, and data retrieval tasks. It coordinates the operation of the ASICs, providing them with data and instructions.

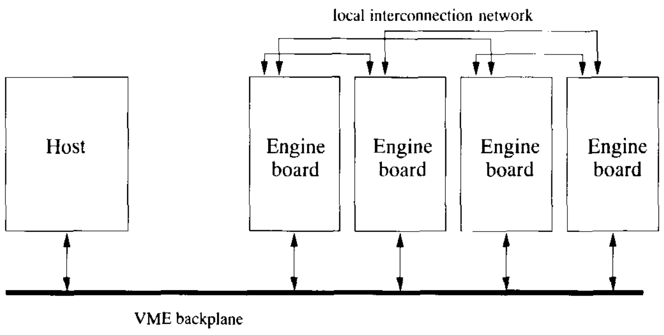


Figure 4.13. Schematic of the Radiosity Engine

It communicates with other Engine boards via the local interconnection network, and with the host over the high bandwidth VME bus. It maintains a database with cell, patch and ray information which is required for the rendering algorithm.

- **Local Memory** This is used for local storage of part of the environment database (the patch and cell databases). The host predicts in advance which part is most likely to be used the most intensive in the computation. This prediction strategy, which is a key component of the Shelling Technique, is meant to reduce the amount of communication over the VME bus. The memory is also used as message-passing buffer for communication between the host and the Engine board.

- **CTU** The Cell Traversal Unit (CTU) is a dedicated ASIC for high-speed cell traversal in the environment. The input for this unit is a description of a spherical region in space —a so-called subshell— which must be traversed. The result of its operation is a set of cells that coincide with that subshell. These cells are necessary to address the patches which are contained in the region.

- **GTU** The Geometry Transformation Unit (GTU) is a dedicated ASIC for geometry transformations. It transforms the coordinates of a patch to a local coordinate system by rotations and translation. Aside this, it also computes a spherical bounding box (SBB) of a patch, which is a description of a spherical region in space within which the patch lies. The SBB is used in the intersection computation to select those rays in the hemisphere with which to intersect the patch.

- **ICU** The Intersection Computation Unit (ICU) is by far the most complicated and diverse of the ASICs. It is dedicated to compute the intersection points between patches and rays. With the found intersection distance it sorts the patches to find, per ray, the first patch in the line of vision. This information about the intersection points is passed on, in compressed form, to the DSP which uses it to compute the form-factors between the patches. The whole of the intersection computation is by far the most computationally demanding operation in the rendering.

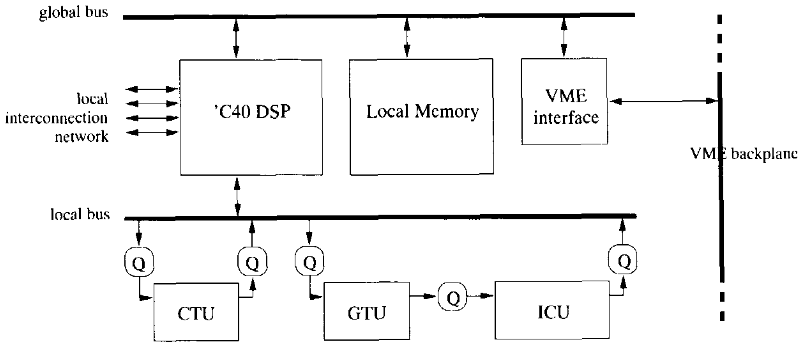


Figure 4.14. Schematic of the Engine Board, showing the principal components

There are a number of FIFO queues between the DSP and ASICs, marked with a "Q" in the schematic. These are added to take up the irregular shocks in the computation. The arrival time of data is non-deterministic, so all ASICs employ handshake mechanisms to control and schedule themselves when valid data arrives.

4.6.2 ASIC co-processor design methodology

The mapping procedure that we employ maps those parts that form a group of operations running at the same rate, *i.e.*, computations within a loop or conditional statement, to a functional unit (FU). The FU is primarily designed to be part of a pipeline, data and instructions flow in one main direction only. The operation of the FU can introduce a change of rate from input to output. The whole algorithm for computing the intersections, which is a nested loop program, is mapped to a chain of functional units. The communication down the FUs is the state information which is sent to the inner loops. The mapping strategy leads to a chain, or pipeline, of parallel working FUs, each of which is dedicated to a specific task. This dedication to a single task greatly reduces the control overhead and increases the fraction of silicon area dedicated to computation. The input / output behaviour of such a pipeline is not that of the classical single token in / single token out model. Instead, a FU produces a data-dependent number of tokens out as a result of a single token in. We dub this kind of pipeline a *peristaltic* pipeline, due to the nature in which it operates. It employs a *self-scheduling* strategy, executing instructions and data in the order at which they arrive at the FUs. The FUs communicate by means of two mechanisms: *message passing* and *memory bank passing.*

4.6.3 The Functional Unit model

The underlying hardware model of the FUs is best explained with actual examples from the implementation of the ICU. Figure 4.15 shows the icon of a typical FU, in this case the Intersect FU, which is part of the ICU.

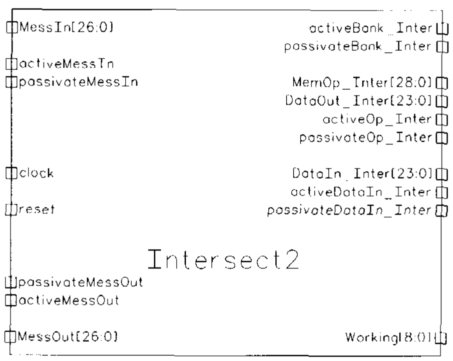


Figure 4.15. The Functional Unit interface of the Intersect unit.

The communication ports of the FU are clearly visible as message in and out ports (MessIn, MessOut) on the left side and as a memory operation port (MemOp) on the right side. All ports have accompanying handshake lines, denoted by 'active' and 'passivate', to indicate the arrival and acceptance of valid data at these ports. The handshake mechanism used is a clock synchronous 2-cycle scheme, allowing data transfer at every clock cycle.

The controller of the FU is designed to be timing-independent by grace of this handshake asynchronous communication scheme. For example, a read request on the memory port can take several cycles to complete, dependent on resource availability, without influencing the operation of the FU or of any other part of the system.

Hierarchical construction of Functional Units

The FUs allow several methods of construction. The most basic is the chaining of FUs, forming a *pipeline*. Such a chain can itself be abstracted to a single FU by hierarchical composition. The Intersect FU —its internal schematic is shown in Figure 4.16— is a fine example of this. Since it contains several FUs which require simultaneous access to the same memory resource, a memory arbiter is added.

Parallel operating FUs are also permitted, as we can see with the Subdivide FU. Internally, as shown in the schematic in Figure 4.17, it consists of three SubUnit

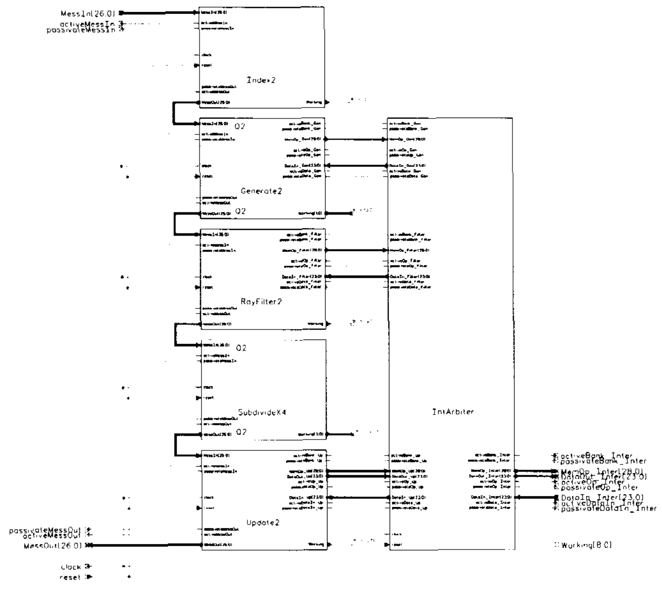


Figure 4.16. Schematic of the Intersect Functional Unit

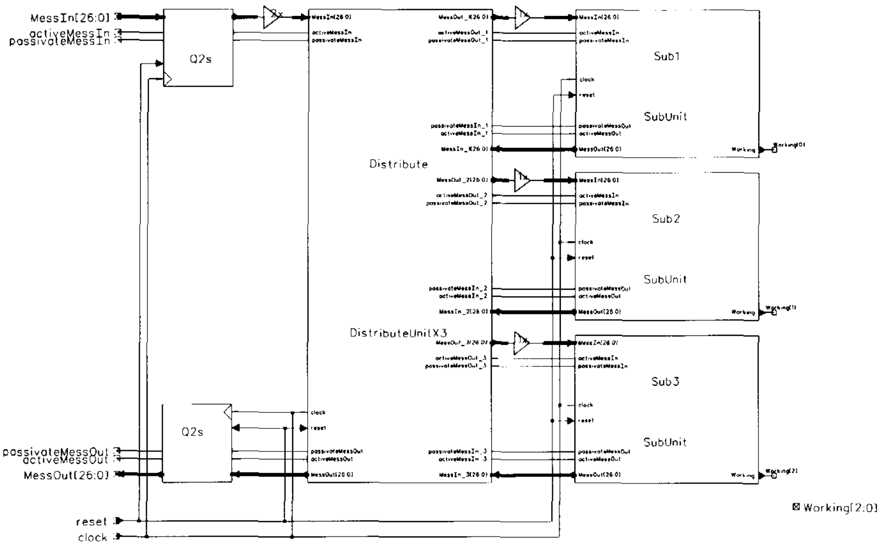


Figure 4.17. Schematic of the Subdivide Functional Unit

FUs working in parallel, and governed by a distribution unit. Any message which is meant for a SubUnit FU is recognized by the distributer, and scheduled accordingly on the next available unit. Any message not meant for these units has to wait until all FUs are idle, and is then passed on to the output port. The sequence ordering of the messages is maintained.

Data transfer mechanisms

There are two mechanisms by which data is transferred between functional units. These are *message passing* and *memory bank passing.*

FU's communicate down the line by means of messages, which is the equivalent of an instruction with accompanying data. A message consists of a single word message header and a data field consisting of a (prescribed) number of data words. Together, they form an instruction for the FU. The controller of the FU is hard-coded to receive, decode and execute such instructions.

Loop variables, variables that change at every iteration of the loop, and basically other variables that do not take up a lot of space are passed down to the FU by means of this message passing. The FU has two ports for this purpose, the MessIn and the MessOut port. Both work with a handshake mechanism to indicate the arrival of valid data.

Any message that is not meant for the FU in question —this is detected by a special field in the header word— is passed on, through the message output port, to subsequent FUs. The FU can, in response to a message, generate by itself new messages for subsequent FUs. This is an example of self-scheduling.

High-volume variables or constants, not bound to loop iterations, are passed down to the FU by means of memory bank passing. A bank of memory, reserved for certain variables, is handed down to the FU once *all* the data in that bank is valid for that operation.

The FU typically performs multiple operations on such a bank of memory, and once finished with the bank, passes it on to the next FU in line. The memory operations on this block of memory are given through the MemOp port of the FU. If the FU in question will never access this block, the port is non-existent.

It is possible that two or more FUs need access to this "pool" of variables in the same block of computations. Hence, we need arbitration over the memory to decide who can access what. The bank is passed on to other FUs once this set of co-operating FUs have *all* given the signal that they are finished with their operations.

The time it takes for the actual transfer of the bank is very fast, basically a few clock cycles. The data stays in the same place in the memory, only the access of the FU to this data is enabled. The actual "switching" of the banks is transparent for the FUs, and is performed by a special memory bank manager unit in the ICU.

4.6.4 Peristaltic pipelined operation

The FUs communicate through asynchronous transfer of messages. They are triggered by the arrival of valid messages, which contain instructions and data for their operation.

The arrival times of these messages are data- and situation dependent. For example, the retrieval time of a patch depends on where it is stored (local memory is short, host memory is long) and on availability of communication channels (bus arbitration, network routing). The execution time of certain operations of FUs is also data-dependent. For instance, in the Subdivide FU of the ICU, a subdivision operation can take more or less time, depending on whether the required accuracy is met (early completion).

The most important property of the peristaltic pipeline operation, however, is that the number of output messages (tokens) for a FU is also data-dependent, due to the non-manifest loop bounds. As an example of this, the Generate FU of the ICU, generates all the rays in a certain region of space, which is determined by the patch geometry. The actual number of the rays generated —and therefore operations at a later FU, such as the Subdivide FU— is hence dependent on the data itself, and is not known at compile- or schedule-time. The FUs are in this sense self-scheduling, *i.e.*, the necessary control and data messages for a certain FU are generated and sent by a previous FU.

Performance

Ideally, a pipelined system such as this operates at maximum performance when balance is achieved in the throughput of the individual stages is matched. For every linked pair of stages in the pipeline, the output rate of the producing stage must match the input rate of the receiving stage. In the case of the peristaltic pipeline, this varies strongly, and so does the location of the bottleneck over time.

The objective that we set instead is to keep those FUs that correspond to the critical inner loops, as busy as possible, as these influence the overall throughput. Even if this is at the expense of other FUs possibly running idle. In the case of the ICU, the critical FU is the Subdivide FU, which comprises of no less than three SubUnit FUs working in parallel to attain the required high performance.

Actual simulation of the circuit of the ICU has given the results for the activity of the FUs, showing the typical peristaltic behaviour in Figure 4.18. The signals, indicating the activity of the FUs, belong to respectively the Index, Generate(2 ×), RayFilter, Subdivide(3 ×) and the Update FUs in the Intersect FU of Figure 4.16.

The same activity signals are shown in Figure 4.19, but then over a shorter time frame, revealing more details. One can clearly see the variation in the activity, and that the position of the bottleneck changes over time.

To measure the performance of the FUs *after* fabrication, a small performance measuring unit (itself an FU) is added as the last in the pipeline. It integrates the

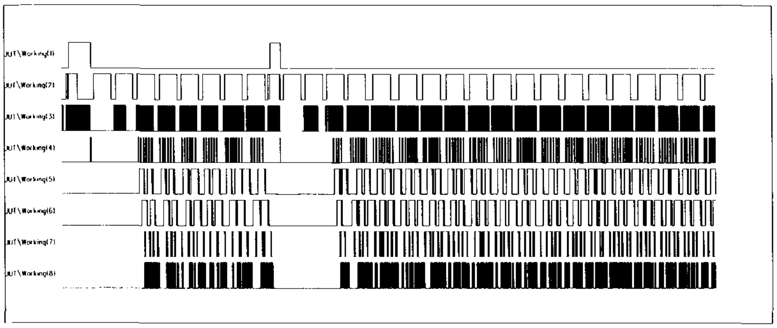


Figure 4.18. The activity signals of the ICU pipeline, showing the peristaltic behaviour

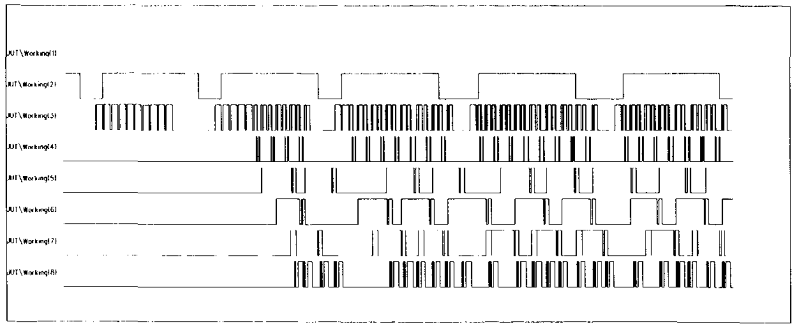


Figure 4.19. The activity signals of the ICU pipeline, detail.

activity signals over a time period, and sends the performance data in a lightweight message to the DSP for evaluation. In this way we can trace the time-varying location of the bottleneck, and how it depends on certain data.

4.6.5 Physical Design

Both the ICU and GTU have been designed in an 1.0μ CMOS process, with Compass tools, using their usc370/vdp300 libraries. They have been fabricated at European Silicon Structures (ES2) in France.

The operating frequency of both ASICs is 25MHz. This gives rise to an equivalent 2 million intersection points per second, at peak performance.

The die size of the GTU measures $9 \times 8\text{mm}^2$. Figure 4.21 in Appendix 4.A shows a photograph of the layout of the GTU. Clearly visible in this photograph are a word-serial Cordic which is used in the computation of the SBB, and the multiplier-accumulator which is used for the geometry transformation. The GTU has been designed by André van der Avoird [12, 13, 14].

The die size of the ICU measures $11.5 \times 11.5\text{mm}^2$, with an active core size of $10 \times 10\text{mm}^2$. The transistor count is around a quarter of a million transistors. Figures 4.22 and 4.23 in Appendix 4.A show a photographs of the layout of the ICU.

4.7 Conclusions

In this chapter, we have looked at a part of the problem of the design of the Radiosity Engine, which is a high performance, parallel architecture, dedicated to photo-realistic rendering of artificial scenes. Algorithms for photo-realistic rendering normally require an enormous amount of high complexity arithmetic functions. We have identified the computation of the intersection between a ray and a patch as potentially the most demanding in terms of computational load and complexity. The Shelling Technique [1], which is the result of an intensive system study of the Radiosity Engine, helps to reduce part of the amount of operations by selecting only those intersection computations that contribute, and eliminating those for which it is known that they don't.

We have presented an integrated scheme that computes whether ray intersects with a patch (hit) and, if so, computes the patch coordinates of the intersection point and the distance to the intersection point. The scheme, which is an improved and extended version of the bounding-plane method [10], relies heavily on the computation of the inner product between the ray direction **r** and a given set of vectors. These inner products can be generated efficiently by means of incremental computation, where every new result requires only one rotation. At this stage we have employed the fast rotation techniques of Chapter 3, by choosing the ray resolution such that a fast rotation can be used in the incremental computation. We are allowed to adapt the ray resolution in this way, because we are dealing with a sampling problem.

We have examined the effect of the choice of the ray resolution on other operations, and found that there were no adverse consequences. The overall reduction of the computational complexity, and hence the workload, has been reduced with more than an order of magnitude. Compared to a naive implementation of computing the inner products, which is the most critical operation, a gain of almost a factor of 20 is made.

The research has led to the feasibility and the implementation of the Intersection Computation Unit (ICU) ASIC. Together with the Geometry Transformation Unit (GTU) and the Cell Traversal Unit (CTU), they form a high performance graphics co-processor subsystem of the Radiosity Engine. The use of the Shelling Technique has resulted in a rendering algorithm with such properties, that its mapping onto a parallel, high performance architecture posed new and challenging problems in VLSI system design. This has led to the development of a new methodology and the concept of a peristaltic —rather than systolic— variable multi-rate pipeline which is used in the design of the ICU. A performance measuring unit has been incorporated in the ICU, to measure the behaviour of such a system.

Appendix

4.A Images of the Radiosity Engine and ASICs

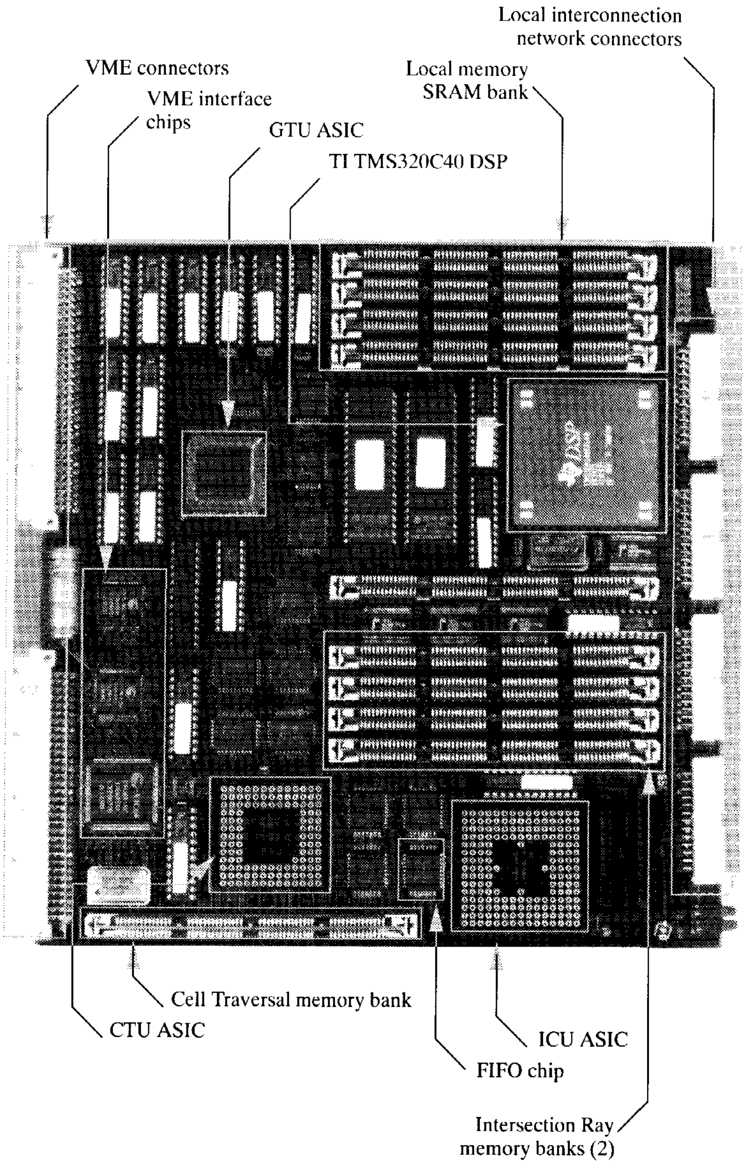


Figure 4.20. Photograph of the Radiosity Engine board (front side).

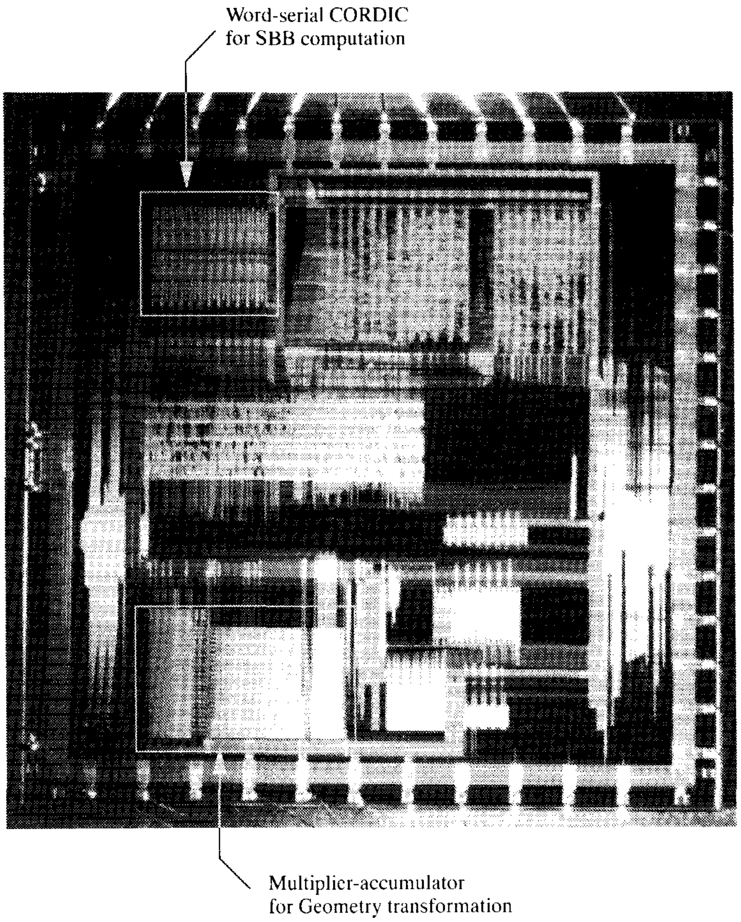


Figure 4.21. Photograph of the layout of the GTU chip.

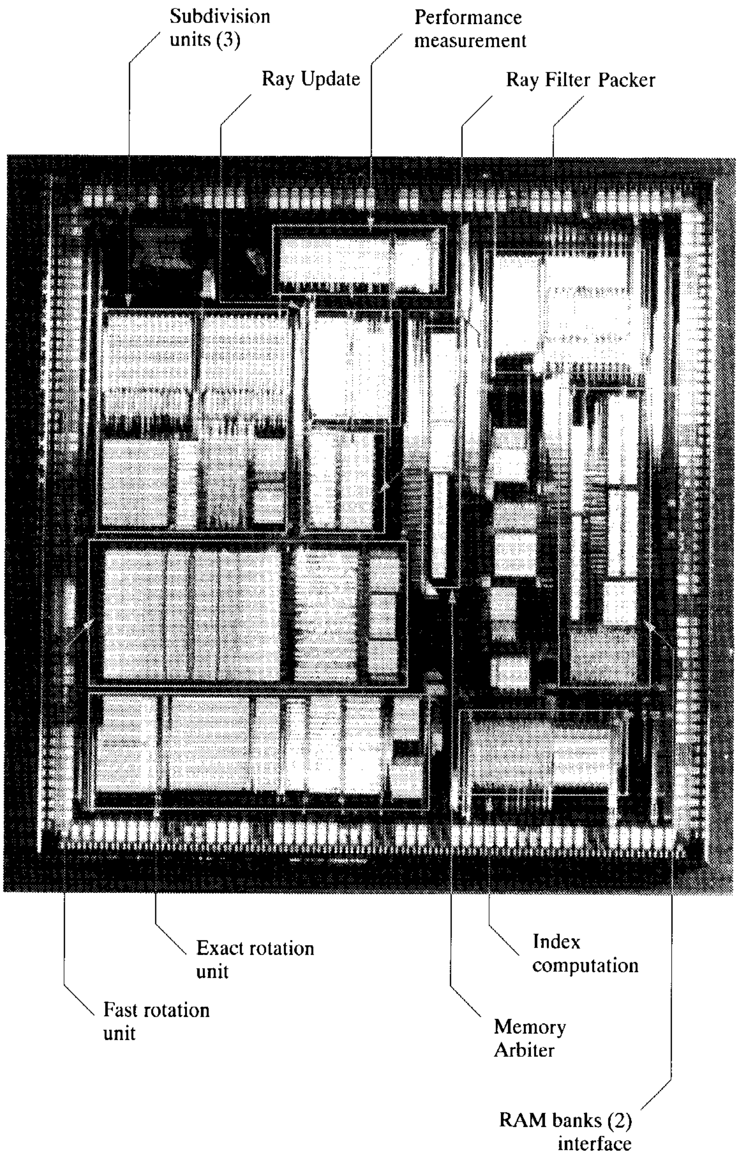


Figure 4.22. Photograph of the layout of the ICU chip, showing the principal units

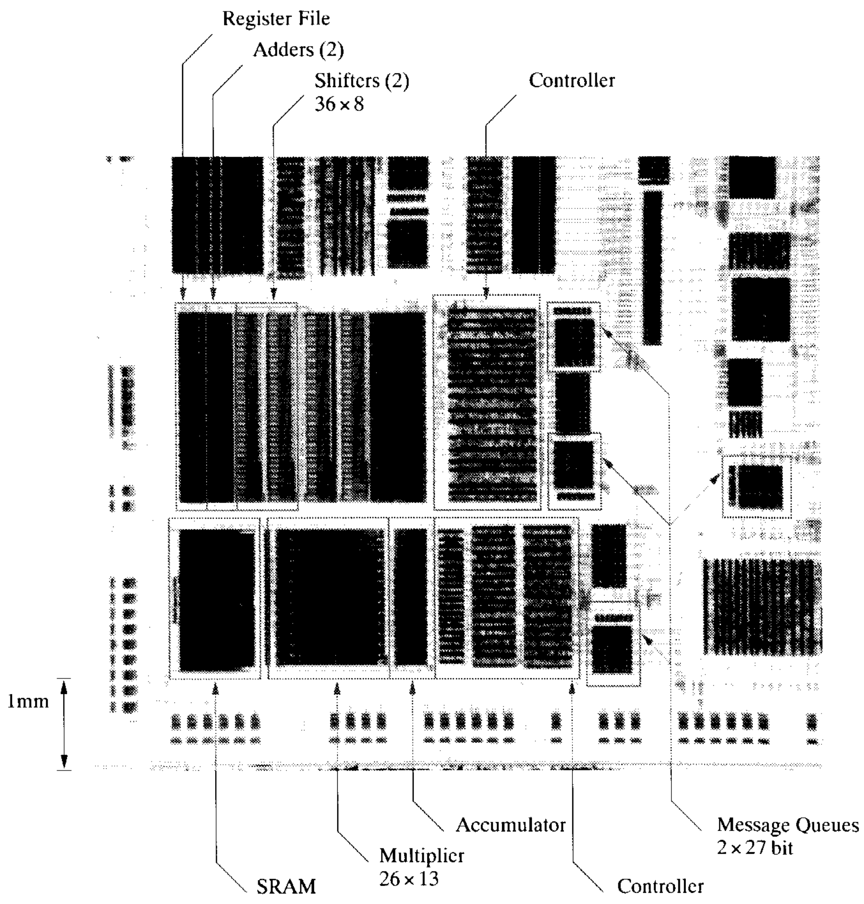


Figure 4.23. Detail photograph of the layout of the ICU chip, showing the exact and fast rotation units.

Bibliography

[1] Li-Sheng Shen, *A Parallel Image Rendering Algorithm and Architecture based on Ray Tracing and Radiosity Shading*, Ph.D. thesis, Delft University of Technology, September 1993, ISBN 90-5326-012-9/CIP.

[2] J.C Bu, G. Boersma, and M.Nelisse, "System Specification of the Radiosity Engine," Tech. Rep. TPD-FSP-RPT-940031, TNO-TPD, April 1993.

[3] Ed F. Deprettere, Gerben J. Hekstra, Li-Sheng Shen, Jichun Bu, and Gerrit Boersma, "A parallel system for photo realistic artificial scene rendering," in *1996 IEEE Workshop on VLSI Signal Processing*. October 1994, vol. VLSI Signal Processing VII, pp. 425–438, IEEE.

[4] A. Appel, "Some techniques for shading machine rendering of solids," in *AFIPS*, 1968, vol. 23, pp. 37–45.

[5] C. Goral, K. Torrance, D. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," in *SIGGRAPH '84*, 1984, vol. 18, pp. 212–222.

[6] Gerben Hekstra, "Spherical Bounding Box computation," Tech. Rep. ET/NT/Radio-8, TU Delft, November 1993.

[7] Gerben Hekstra, "Definition and structure of hemisphere and viewing rays," Tech. Rep. ET/NT/Radio-9, TU Delft, November 1993.

[8] Gerben Hekstra, "Spherical Bounding Box: Computation of the minimum distance to a patch," Tech. Rep. ET/NT/Radio-17, TU Delft, December 1993.

[9] J.D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.

[10] Gerben Hekstra, "Intersection computation using bounding plane subdivision," Tech. Rep. ET/NT/Radio-7, TU Delft, November 1993.

[11] S.Y. Kung, *VLSI Array Processors*, Prentice-Hall International Editions, 1988.

[12] A. van der Avoird, "Geometry Transformation Unit," Tech. Rep. ET/NT/Radio-33, TU Delft, November 1994.

[13] A. van der Avoird, "Geometry Transformation Unit Schematics," Tech. Rep. ET/NT/Radio-37, TU Delft, November 1994.

[14] A. van der Avoird, "Performance estimations radiosity ASICs," Tech. Rep. ET/NT/Radio-38, TU Delft, November 1994.

Chapter 5

EIGENVALUE DECOMPOSITION BASED ON FAST ROTATIONS

Contents

5.1 Introduction

In this chapter we consider the use of fast rotations in the realization of matrix algebra algorithms, in this case the eigenvalue decomposition (EVD) of a symmetric matrix.[1] We present an EVD algorithm that is based on that of the cyclic-by-rows Jacobi algorithm, where our modification is that the angle of the rotations is approximated such that it corresponds to that of a fast rotation, as presented in Chapter 3.

The computation to determine which fast rotation to use in the approximation of the optimal angle, itself consists of only fast rotations, keeping the overall cost low, and facilitating on-line computation.

[1]The work presented in this chapter was performed in part in cooperation with Jürgen Götze, TU München, during his stay at TU Delft.

A significant reduction of the number of required shift–add operations is achieved, with respect to the algorithm based on exact rotation with, for example, Cordic arithmetic.

All types of fast rotations (and the computation to determine the optimal angle) can be implemented as a cascade of only four basic types of shift–add stages. These stages can be executed on a modified sequential floating-point Cordic architecture, making the presented algorithm highly suitable for VLSI–implementation.

5.1.1 Outline of this chapter

In Section 5.2, we provide the background on Eigenvalue decomposition, and present the cyclic-by-rows Jacobi EVD algorithm, on which our algorithm is based. In Section 5.3, we present the modified EVD algorithm that employs fast rotations. We show that all operations, both the approximate rotation and the determination of the optimal angle, are based on fast rotations, keeping the overall cost low. In Section 5.4, we touch on the implementation and arithmetic issues involved in building a high-throughput floating-point EVD architecture. The analysis of the performance of the modified EVD algorithm is given in Section 5.5. Here we show that a great reduction in the computational cost is achieved, when compared to the EVD algorithm based on exact rotations with Cordic arithmetic. Finally, in Section 5.6, we give our conclusions.

5.2 Eigenvalue decomposition

The EVD of a $n \times n$ symmetric matrix $\mathbf{A}$ is defined as

$$\mathbf{A} = \mathbf{Q}^T \Lambda \mathbf{Q},$$

where $\mathbf{Q}$ is an $n \times n$ orthogonal matrix, $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$, and $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$ is an $n \times n$ diagonal matrix containing the eigenvalues λ_i of $\mathbf{A}$.

Computing the EVD of a symmetric matrix is a frequently encountered problem in a great number of scientific applications, e.g. signal processing [1, 2, 3]. In order to meet the real time requirements present in many of these applications, it is often necessary to compute the EVD on a parallel architecture.

The method of choice for the fast parallel computation of the EVD is the Jacobi method, since it offers a significantly higher degree of parallelism than the respective QR–method [4, 5]. One sweep of a cyclic Jacobi method (consisting of $n(n-1)/2$ rotation evaluations and their application to the matrix) can be implemented on an upper triangular array of processors with nearest neighbour interconnections in $O(n)$ time [4, 6]. Usually, $O(\log_2 n)$ sweeps are required [4]. Besides this suitability for a parallel implementation the Jacobi method offers a higher numerical accuracy than the QR–method [7].

The computational complexity of the Jacobi method depends largely on the evaluation and application of the rotations that make up $\mathbf{Q}$. In [8], the different strategies that have been tried to bring down the cost, are categorized. In this chapter, we will follow in large lines the algorithm as given in [8].

5.2.1 The iterative Jacobi algorithm for EVD

The Jacobi method to compute the EVD of an $n \times n$ symmetric matrix $\mathbf{A}$ is given by:

$$\begin{aligned} \mathbf{M}_{(0)} &= \mathbf{A} \\ \mathbf{M}_{(h+1)} &= \mathbf{Q}_{(h)}\mathbf{M}_{(h)}\mathbf{Q}^T_{(h)} \ , \end{aligned} \tag{5.1}$$

where h is the iteration, $\mathbf{Q}_{(h)}$ is an embedded 2×2 rotation at iteration h. After enough many iterations, $\mathbf{M}_{(h)}$ converges to the diagonal matrix of eigenvalues, Λ.

The cyclic–by–row Jacobi method computes the EVD of a $n \times n$ symmetric matrix by applying a sequence of orthonormal rotations to the left and right of $\mathbf{A}$, as shown in the following algorithm:

```
$h := 1$ ; $\mathbf{M}_{(1)} := \mathbf{A}$
for $s := 1$ to number of sweeps
    for $i := 1$ to $n-1$
        for $j := i+1$ to $n$
            $\mathbf{M}_{(h+1)} := \mathbf{Q}_{i,j}(\theta_h)\mathbf{M}_{(h)}\mathbf{Q}^T_{i,j}(\theta_h)$
            $h := h+1$
        end
    end
end
```

where $\mathbf{Q}_{i,j}(\theta)$ is an orthonormal plane rotation over the angle θ in the (i,j) plane, and defined by $(\cos\theta, -\sin\theta, \sin\theta, \cos\theta)$ in the (ii, ij, ji, jj) positions of the $n \times n$ identity matrix.

The index pairs are chosen in the cyclic–by–row manner

$$(i,j) = (1,2)(1,3)\ldots(1,n)(2,3)\ldots(2,n)\ldots(n-1,n) \,. \tag{5.2}$$

The execution of all $N = n(n-1)/2$ index pairs (i,j) according to (5.2) is called a sweep. Since the matrices $\mathbf{M}_{(h)}$ remain symmetric for all h, it is sufficient to work with the upper triangular part of the matrix throughout the algorithm.

Let us define the off–diagonal quantity $E_{(h)}$ for the symmetric matrix $\mathbf{M}$ as:

$$E_{(h)} = \sqrt{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(m_{ij}^{(h)}\right)^2} \,. \tag{5.3}$$

Note that this is the same as the Frobenius norm of the upper triangular part of $\mathbf{M}_{(h)}$ above the main diagonal.

A similarity transformation $\mathbf{M}_{(h+1)} = \mathbf{Q}_{ij}(\theta_h)\mathbf{M}_{(h)}\mathbf{Q}_{ij}^T(\theta_h)$ yields [9]:

$$\left(E_{(h+1)}\right)^2 = \left(E_{(h)}\right)^2 - \left(\left(m_{ij}^{(h)}\right)^2 - \left(m_{ij}^{(h+1)}\right)^2\right). \tag{5.4}$$

Obviously, a reduction of $E_{(h)}$ is obtained if $|m_{i,j}^{(h+1)}| < |m_{i,j}^{(h)}|$. This reduction is maximal if $m_{i,j}^{(h+1)} = 0$. Therefore,

$$\lim_{h\to\infty} E_{(h)} \to 0 \quad \Leftrightarrow \quad \lim_{h\to\infty} \mathbf{M}_{(h)} \to \operatorname{diag}[\lambda_1, \ldots, \lambda_n]. \tag{5.5}$$

Without loss of generality we will drop the index h and only consider the 2×2 symmetric EVD subproblem

$$\begin{bmatrix} m'_{i,i} & m'_{i,j} \\ m'_{i,j} & m'_{j,j} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} m_{i,i} & m_{i,j} \\ m_{i,j} & m_{j,j} \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}^T . \tag{5.6}$$

in the sequel. Thus, $m'_{i,j}$ is given by:

$$m'_{i,j} = \frac{1}{2}\left[(2m_{i,j})\cos 2\theta - (m_{j,j} - m_{i,i})\sin 2\theta\right]. \tag{5.7}$$

Solving $m'_{i,j} = 0$ one obtains the optimal angle of rotation θ_{opt}, for which maximal reduction of E (i.e. $m'_{i,j} = 0$) is achieved:

$$\theta_{\text{opt}} = \frac{1}{2}\arctan(\tau), \tag{5.8}$$

where $\tau = \frac{2m_{i,j}}{m_{j,j} - m_{i,i}}$, and where the range of θ_{opt} is limited to $|\theta_{\text{opt}}| \leq \frac{\pi}{4}$.

5.3 EVD with approximate rotations

Using approximate rotations enables the reduction of the complexity of the vectorization and the rotation mode [6, 10, 11]. For a reduction of the off–diagonal quantity E it is not necessary to meet $m'_{i,j} = 0$, but it is sufficient, when using an approximate angle θ_{app}, that

$$|m'_{i,j}| = |d| \cdot |m_{i,j}| \text{ with } 0 \leq |d| < 1, \tag{5.9}$$

where the reduction d is a function of the approximate angle θ_{app} and the matrix data, given by:

$$d(\theta_{\text{app}}, \tau) = \cos 2\theta_{\text{app}} - \frac{1}{\tau} \sin 2\theta_{\text{app}} . \tag{5.10}$$

This is the basis for the design of approximate rotations, i.e. rotations that meet (5.9).

The approach used is to construct a set $\mathcal{F} = \{\mathbf{F}_0, \mathbf{F}_{-1}, \ldots\}$ of fast rotations. We define the fast rotation $\mathbf{F}_i(\sigma)$ as given by:

$$\mathbf{F}_i(\sigma) = \begin{bmatrix} c & -\sigma s \\ \sigma s & c \end{bmatrix}, \tag{5.11}$$

where $c, s \geq 0$, and where the direction parameter $\sigma \in \{-1, +1\}$ indicates the direction of the rotation. The angle of rotation is given as $\sigma \cdot \alpha_\kappa$, where

$$\alpha_\kappa = \arctan(\frac{s}{c}) . \tag{5.12}$$

The approximation pair c, s is chosen such that $\mathbf{F}_\kappa$ implements the rotation over an angle $\pm\alpha_\kappa$ in a minimal number of shift–add operations, while satisfying the accuracy requirements, as set by the application. The angles α_κ form the ordered set of approximation angles $\mathcal{A} = \{\alpha_0, \alpha_{-1}, \ldots\}$. Both angle α_κ and fast rotation $\mathbf{F}_\kappa$ are selected by the integer angle index κ, satisfying $\kappa \leq 0$. In Section 5.3.1, we will present methods to construct such fast fast rotations and how to determine the optimal angle index κ.

Using approximate rotations must be paid by an increase in the number of required sweeps. It has been shown in [10, 11, 12], however, that the overall cost, in terms of shift–add operations, obtained for the Jacobi method using approximate rotations is significantly lower than for the Jacobi method using exact rotations.

5.3.1 Construction of the set of fast rotations

The ordered set of approximation angles $\mathcal{A}$, for a given accuracy N_{mant}, is constructed using the aforementioned methods. For $N_{\text{mant}} = 32$ Table 5.1 shows when to select which method, depending on the value of the angle index κ, as well as the angle, and the cost for rotation and possible scaling. The fast rotations are chosen such that they satisfy the accuracy condition using the cheapest possible method.

5.3.2 Determining the optimal approximate rotation

The crucial point for approximate rotations is to find the approximate angle $\theta_{\text{app}} = \sigma \cdot \alpha_\kappa$, where $\sigma \in \{-1, +1\}$ is the direction of the rotation and the angle $\alpha_\kappa \in \mathcal{A}$, with the angle index κ, is chosen such that $|d(\theta_{\text{app}}, \tau)|$ is minimal.

angle index	method	(factored) approximation pair	angle	cost
κ		(c,s)	α_κ	
0	*Ext*	$\{(2^{-1},1-2^{-3}),(1-2^{-6},2^{-3}),$ $(1-2^{-18},2^{-6})\}$	1.179911	6
-1	*Ext*	$\{(1-2^{-3},2^{-1}-2^{-6}),(1-2^{-12},2^{-6})\}$	0.521209	5
-2	*Ext*	$\{(1-2^{-5},2^{-2}-2^{-9}),(1-2^{-18},2^{-9})\}$	0.252616	5
-3	*Ext*	$\{(1-2^{-7},2^{-3}),(1-2^{-14},2^{-7})\}$	0.133137	4
-4	*Ext*	$\{(1-2^{-9},2^{-4}),(1-2^{-18},2^{-9})\}$	$6.449377\,10^{-2}$	4
-5	*III*	$(1-2^{-11},2^{-5}-2^{-18})$	$3.12513\,10^{-2}$	3
-6	*III*	$(1-2^{-13},2^{-6}-2^{-21})$	$1.56252\,10^{-2}$	3
-7	*III*	$(1-2^{-15},2^{-7}-2^{-24})$	$7.81252\,10^{-3}$	3
-8	*II*	$(1-2^{-17},2^{-8})$	$3.90626\,10^{-3}$	2
-9	*II*	$(1-2^{-19},2^{-9})$	$1.95313\,10^{-3}$	2
-10	*II*	$(1-2^{-21},2^{-10})$	$9.76563\,10^{-4}$	2
-11	*II*	$(1-2^{-23},2^{-11})$	$4.88281\,10^{-4}$	2
-12	*II*	$(1-2^{-25},2^{-12})$	$2.44141\,10^{-4}$	2
-13	*II*	$(1-2^{-27},2^{-13})$	$1.22070\,10^{-4}$	2
-14	*II*	$(1-2^{-29},2^{-14})$	$6.10352\,10^{-5}$	2
-15	*II*	$(1-2^{-31},2^{-15})$	$3.05176\,10^{-5}$	2
-16	*I*	$(1,2^{-16})$	$1.52588\,10^{-5}$	1
-17	*I*	$(1,2^{-17})$	$7.62939\,10^{-6}$	1
-18	*I*	$(1,2^{-18})$	$3.81470\,10^{-6}$	1
-19	*I*	$(1,2^{-19})$	$1.90735\,10^{-6}$	1
-20	*I*	$(1,2^{-20})$	$9.53674\,10^{-7}$	1
-21	*I*	$(1,2^{-21})$	$4.76837\,10^{-7}$	1
-22	*I*	$(1,2^{-22})$	$2.38419\,10^{-7}$	1
-23	*I*	$(1,2^{-23})$	$1.19209\,10^{-7}$	1
-24	*I*	$(1,2^{-24})$	$5.96046\,10^{-8}$	1
-25	*I*	$(1,2^{-25})$	$2.98023\,10^{-8}$	1
-26	*I*	$(1,2^{-26})$	$1.49012\,10^{-8}$	1
-27	*I*	$(1,2^{-27})$	$7.45058\,10^{-9}$	1
-28	*I*	$(1,2^{-28})$	$3.72529\,10^{-9}$	1
-29	*I*	$(1,2^{-29})$	$1.86265\,10^{-9}$	1
-30	*I*	$(1,2^{-30})$	$9.31323\,10^{-10}$	1
-31	*I*	$(1,2^{-31})$	$4.65661\,10^{-10}$	1
-32	*I*	$(1,2^{-32})$	$2.32831\,10^{-10}$	1

Table 5.1. The set $\mathcal{A}$ of fast rotations for 32-bit accuracy, showing the method used, the angle and the cost in shift-add pairs.

The direction of rotation σ follows from the sign of the optimal angle θ_{opt}, and is given by:

$$\sigma = \text{sign}(\tau) = \text{sign}(m_{i,j}) \cdot \text{sign}(m_{j,j} - m_{i,i}) . \tag{5.13}$$

In order to determine the angle index κ, we introduce the working domain limits g_κ, and define the condition for choosing the angle α_κ as being:

$$g_{\kappa-1} < |\tau| \leq g_\kappa , \tag{5.14}$$

where g_κ is determined such that, when τ satisfies (5.14), then $|d(\sigma \cdot \alpha_\kappa, \tau)|$ is minimal over the set of angles $\mathcal{A}$. Note, that minimizing $|d(\sigma \cdot \alpha_\kappa, \tau)|$ is equivalent to choosing the angle $\alpha_\kappa (= \theta_{\text{app}})$ which is closest to the optimal angle θ_{opt}. The limit g_κ follows from the solution of:

$$d(\alpha_\kappa, g_\kappa) = -d(\alpha_{\kappa+1}, g_\kappa) , \tag{5.15}$$

i.e. the point in the domain $|\tau|$ where α_κ and $\alpha_{\kappa+1}$ lead to the same reduction of the off–diagonal element. The solution of (5.15) yields:

$$g_\kappa = \tan(\alpha_\kappa + \alpha_{\kappa+1}) = \tan \gamma_\kappa , \tag{5.16}$$

where $\gamma_\kappa = \alpha_\kappa + \alpha_{\kappa+1}$ is the working limit in the angle domain.

In [12], for this particular set of approximative angles, and through the use of floating-point arithmetic the search is narrowed down to check which of three consecutive angles gives the maximal reduction (minimal $|d|$). Here we show that this selection can also be executed by using fast rotations.

Construct the vector $\mathbf{v}$ given by:

$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} m_{j,j} - m_{i,i} \\ 2m_{i,j} \end{bmatrix} . \tag{5.17}$$

The angle between $\mathbf{v}$ and the x-axis is equal to the required $\arctan(\tau) = 2\theta_{\text{opt}}$. To test whether $|\tau|$ is greater or smaller than the limit g_κ is equivalent to checking whether this angle is resp. greater or smaller in absolute value to the limit angle γ_κ.

Hence we rotate $\mathbf{v}$ over the angle $-\sigma \cdot \gamma_\kappa = -\sigma \cdot (\alpha_\kappa + \alpha_{\kappa+1})$, to obtain $\mathbf{v}' = [v'_x \, v'_y]^T$ (see Figure 5.1). When the y-component v'_y is of the *same* sign as v_y, then the angle between $\mathbf{v}$ and the x-axis is larger than γ_κ, and we will select $\alpha_{\kappa+1}$ as the approximate angle. If, however, v'_y is of *opposite* sign to v_y, then the angle is smaller, and we select α_κ. The rotation over $-\sigma \cdot \gamma_\kappa$ is performed as two consecutive rotations over α_κ and $\alpha_{\kappa+1}$, implemented as fast rotations:

$$\mathbf{v}' = \mathbf{Q}(-\sigma\gamma_\kappa) \cdot \mathbf{v} = \mathbf{F}_{\kappa+1}(-\sigma) \cdot \mathbf{F}_\kappa(-\sigma) \cdot \mathbf{v} . \tag{5.18}$$

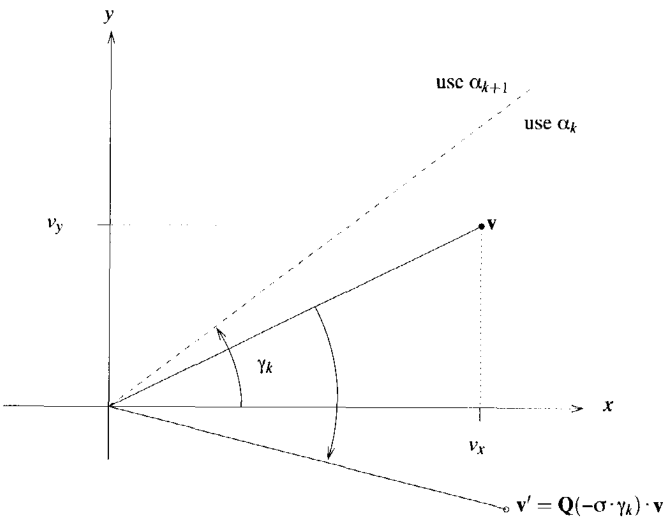


Figure 5.1. Separation of consecutive angles α_κ and $\alpha_{\kappa+1}$.

Obviously, for our example in Figure 5.1 we have to use α_κ since $v_y > 0$ and $v'_y < 0$ holds.

The search for the optimal angle $\alpha_\kappa \in \mathcal{A}$, which effects in the greatest reduction of the off–diagonal element $m_{i,j}$, can be narrowed to a check of three consecutive angles. This can simply be achieved by looking at the exponents of $[v_x, v_y]$. We obtain an estimate for the optimal angle index κ as follows.

Let the components v_x, v_y be given in a floating-point representation, where m_x, m_y are the mantissas, and e_x, e_y are the exponents of this representation. We compute an indication k for the angle index κ, based on the exponents of v_x, v_y only, and which is given by $k = e_y - e_x$. Based on this indication, we state that the optimal angle index κ can be found in the range given by $\kappa \in \{\kappa - 1, \kappa, \kappa + 1\}$. This is illustrated in Figure 5.2. The domains of possible values for v_x and v_y are indicated by the black bars on the x and y axis, respectively. This domain is determined by the range of the mantissas of the floating-point representation of v_x, v_y. These domains of the mantissa describe a rectangle in the (x,y)-space. It is easy to show that this rectangle covers at most three possible rotation angles, i.e., $\theta_{\text{app}} \in \{\alpha_{k-1}, \alpha_k, \alpha_{k+1}\}$.

Now the method described above (Figure 5.1) can be adapted to test which one of the three consecutive angles to use. For this we compute $\mathbf{v}'$ and $\mathbf{v}''$ according to

$$\begin{aligned}\mathbf{v}' &= \mathbf{Q}(-\sigma\gamma_k)\cdot\mathbf{v} = \mathbf{F}_{k+1}(-\sigma)\cdot\mathbf{F}_k(-\sigma)\cdot\mathbf{v}\\ \mathbf{v}'' &= \mathbf{Q}(-\sigma\gamma_{k-1})\cdot\mathbf{v} = \mathbf{F}_{k-1}(-\sigma)\cdot\mathbf{F}_k(-\sigma)\cdot\mathbf{v}\ ,\end{aligned} \tag{5.19}$$

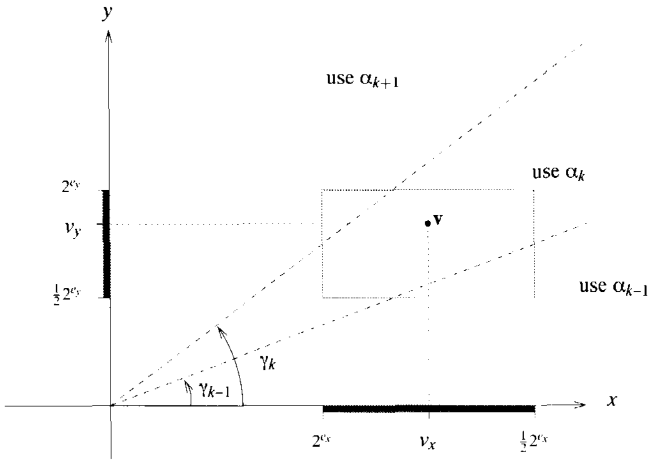


Figure 5.2. Determination of optimal approximate angle.

and choose the correct angle of rotation, using the selection tree, based on the resulting signs of v'_y, v''_y

$$\kappa = \begin{cases} k+1 & \text{if } \mathrm{sign}(v_y) = \mathrm{sign}(v'_y) \\ k-1 & \text{if } \mathrm{sign}(v_y) \neq \mathrm{sign}(v''_y) \\ k & \text{otherwise} \end{cases} \tag{5.20}$$

If the intermediate results of the rotations are re-used, this selection mechanism costs at most three fast rotations (note that this selection can also be executed by unscaled rotations). For our example in Figure 5.2 we obtain $v_y > 0$, $v'_y < 0$, $v''_y > 0$, such that α_κ is the optimal approximate angle θ_{app}.

The resulting total reduction $|d(\tau, \theta_{\text{app}})|$, using the above method for determining the approximate angle is shown in Figure 5.3 for the set of angles $\mathcal{A}$ with 32-bit accuracy.

5.4 Implementation issues for a fast rotation architecture

We now take a closer look at the possible architecture of the fast rotation unit. We state that the four basic types of shift–add stages, as shown in Figure 5.4(a) are sufficient to implement any kind of fast rotation.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{F}_\kappa(\sigma) \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{5.21}$$

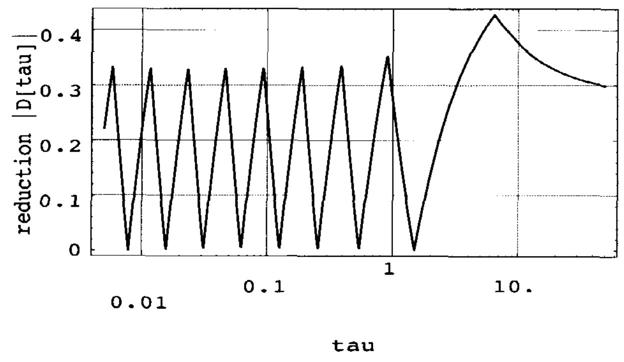


Figure 5.3. The total reduction as a function of $|\tau|$, for the set of angles with 32-bit accuracy

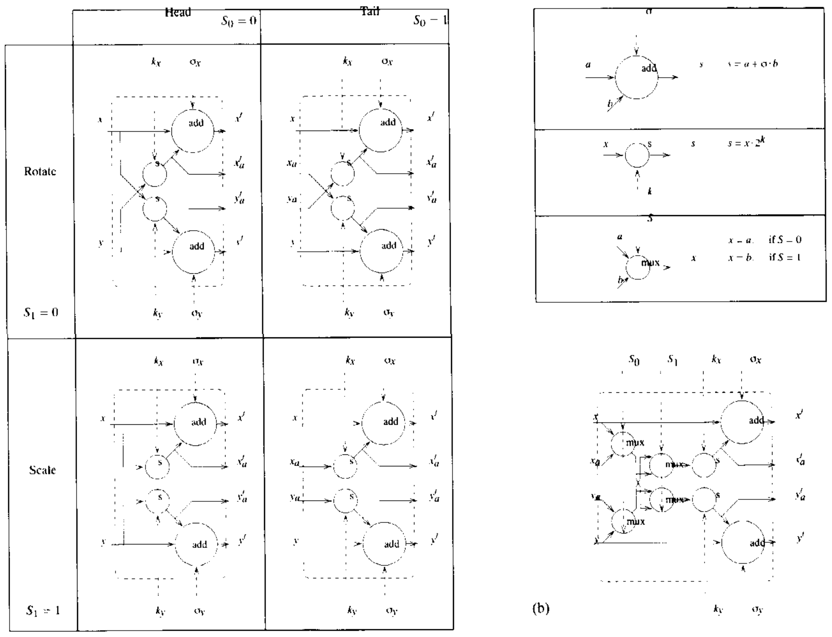


Figure 5.4. The four basic shift–add stages (a) and their unification (b).

Each of the stages implements a pair of shift-add operations. In turn, these four types of stages are combined in the unified stage, as shown in Figure 5.4(b), which forms the basis of the fast fast rotation architecture.

Method III is used for illustrating the architecture design. Writing out (5.21) for the fast rotation of method III yields:

$$\begin{aligned} x' &= x - \sigma(y \cdot 2^{\kappa}) - (x \cdot 2^{2\kappa-1}) + \sigma(y \cdot 2^{3\kappa-3}) \\ y' &= y + \sigma(x \cdot 2^{\kappa}) - (y \cdot 2^{2\kappa-1}) - \sigma(x \cdot 2^{3\kappa-3}) \ . \end{aligned} \tag{5.22}$$

The rotation of (5.22) is realized as a cascade of simple stages, as shown in Figure 5.5(c). The sequence of shifted values of x, being $\{x \cdot 2^{\kappa}, x \cdot 2^{2\kappa-1}, x \cdot 2^{3\kappa-3}\}$, are computed through consecutive shifts over $\kappa, \kappa-1$ and $\kappa-2$ positions. The same is done for the shifted values of y. Intermediate shifted results are stored in auxiliary variables x_a, y_a. Similar sequences exist for the other fast rotation methods (I, II) as is shown in Figures 5.5(a) and 5.5(b), respectively. The extended rotation method is a factored rotation method of Chapter 3, and is shown in Figure 5.5(d), for three factors.

5.4.1 Floating-point realization

The realization of an fast rotation of floating-point x,y data over the angle α_{κ} is performed in three stages.

1. floating point preprocessing; alignment of mantissas
2. block floating-point execution of the rotation
3. floating-point post processing; renormalisation

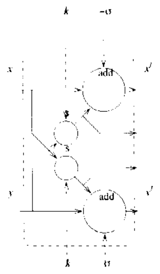
We assume a block floating-point datapath to realize the rotation. This means that the exponents of the x and y datapaths remain constant throughout a sequence of operations, in this case: the shift and add operations in the cascade realization. The x and y summation path *each* has their own fixed exponent $e_x^{(\mathrm{dp})}$ resp. $e_y^{(\mathrm{dp})}$.

The first stage is the floating-point pre-processing. From the exponents e_x, e_y and the angle index (or angle *exponent*) κ, we compute the exponents $e_x^{(\mathrm{dp})}, e_y^{(\mathrm{dp})}$ which are used inside the block floating-point datapath.

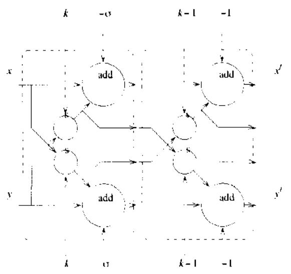
$$\begin{aligned} e_x^{(\mathrm{dp})} &= \max(e_x, e_y + \kappa) \\ e_y^{(\mathrm{dp})} &= \max(e_y, e_x + \kappa) \ . \end{aligned} \tag{5.23}$$

These datapath exponents are chosen such that the final and intermediate results do not overflow the individual datapaths, while still maintaining full accuracy. Only

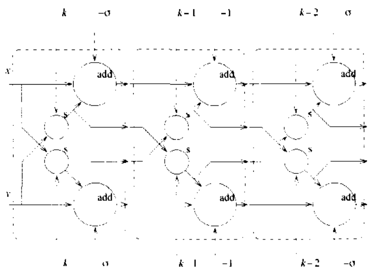
(a)



(b)



(c)



(d)

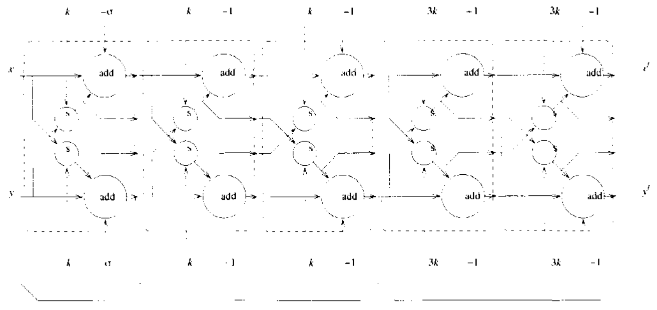


Figure 5.5. Cascade realization of the fast rotations (method I (a), method II (b), method III (c), a factored method with $N = 3$ (d)).

one extra MSB bit must be added as a precaution. The mantissas are aligned accordingly before they enter the block floating-point datapath.

$$\begin{aligned} m_x^{(\mathrm{dp})} &= m_x \cdot 2^{e_x - e_x^{(\mathrm{dp})}} \\ m_y^{(\mathrm{dp})} &= m_y \cdot 2^{e_y - e_y^{(\mathrm{dp})}} \ . \end{aligned} \tag{5.24}$$

The second stage is the block floating-point execution of the rotation. Operations are performed on the mantissas only, greatly speeding up operations since expensive floating-point additions are eliminated. Writing out the rotation in terms of mantissas only, we arrive at the following equation for block floating-point fast rotation:

$$\begin{aligned} m'_x &= c \cdot m_x^{(\mathrm{dp})} - \sigma s \cdot 2^{\Delta} \cdot m_y^{(\mathrm{dp})} \\ m'_y &= c \cdot m_y^{(\mathrm{dp})} + \sigma s \cdot 2^{-\Delta} \cdot m_x^{(\mathrm{dp})} \ , \end{aligned} \tag{5.25}$$

where $\Delta = e_y^{(\mathrm{dp})} - e_x^{(\mathrm{dp})}$ is the difference in exponents between the x and y datapath. The above formula is the basis for the same cascade realizations as in Figure 5.5. The only difference in the execution of the cascaded stages is that, for the "Rotation" stages only, the k_x and k_y shifts are offset by Δ resp. $-\Delta$. This is necessary to align data transferred from the x to the y datapath resp. from the y to the x datapath. From Equations (5.23) we can prove that Δ remains bounded by $\kappa \leq \Delta \leq -\kappa$. This guarantees that only negative shifts (shift right operations) occur during the computation.

The third and last step is the floating-point post-processing. The results of the fixed-point computation is paired with the corresponding exponents as $(m'_x, e_x^{(\mathrm{dp})})$ and $(m'_y, e_y^{(\mathrm{dp})})$ and renormalized to proper floating-point representation.

5.4.2 Architecture considerations

Due to the variable length of the different fast rotations, a sequential architecture for the computational core of the processor is highly favourable. The communication between processors in a parallel system, however, must be capable to handle the variations in computation time.

As for the computational core itself, the floating-point sequential Cordic architecture presented in [13] only needs slight modifications to accommodate the functionality of the unified stage of Figure 5.4(b), making it suitable to perform fast rotations, as well as the normal Cordic operations.

5.5 Analysis of the performance

A random symmetric matrix A of dimension 20×20 is used to illustrate the performance of the algorithm. We assume a wordlength of 32 bit for all our imple-

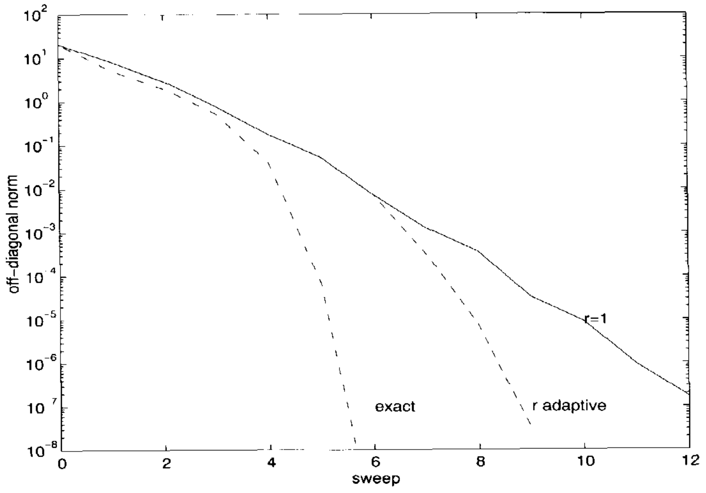


Figure 5.6. Off-diagonal norm vs. sweeps for the Jacobi method.

mentations. The Jacobi method is terminated if the off–diagonal norm is smaller than $10^{-8} \cdot \|\mathbf{M}\|_F$.

In Figure 5.6 the reduction of the off–diagonal norm vs. the sweeps is shown. Obviously, using only one fast rotation for approximating the rotation shows no ultimate quadratic convergence (solid line) as does the Jacobi method with exact rotations (dash–dotted line).

As shown in [12] the quadratic convergence of the Jacobi method can be regained by adapting (increasing) the number of fast rotations executed per plane rotation $\mathbf{Q}_{i,j}$. Executing $r \geq 1$ fast rotations per plane rotation, increases the accuracy of the approximate rotation. Therefore, the conditions for quadratic convergence [11, 12] are met as the diagonalization advances. Here, the Jacobi method is executed with an adaptive number r of fast rotations per plane rotation (dashed line), where $r = \lfloor |\kappa_{mean}|/10 \rfloor$ and κ_{mean} is the mean value of the angle indices used in the previous sweep ($r = 1$ in the first sweep).

In Figure 5.7 the required number of shift–add operations per sweep is shown.

The Jacobi method using the original (exact) Cordic requires a constant number of shift–add operations per sweep, since each rotation requires a constant number of shift–add operations. For a 32 bit Cordic, we need 80 shift–add operations per rotation: 64 for the micro-rotations plus 16 for the scaling of both vector components. The use of approximate rotations, i.e., one fast rotation per plane rotation reduces the number of shift–add operations per sweep significantly. Furthermore,

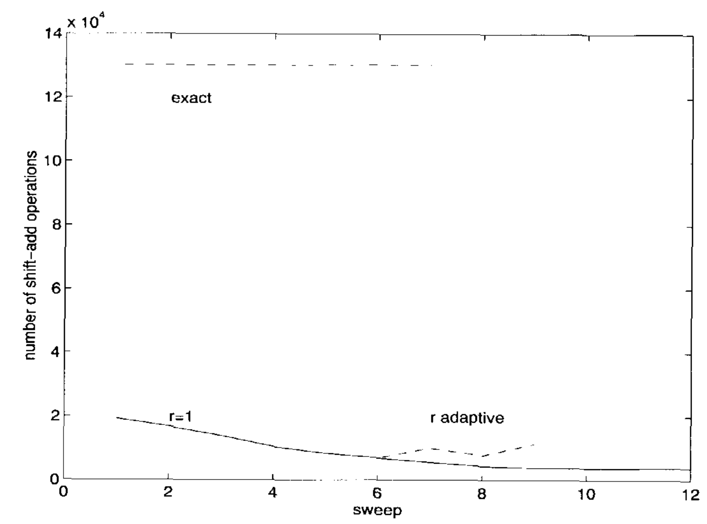


Figure 5.7. Shift-add operations per sweep for the Jacobi method.

Method	exact	$r = 1$	r adaptive
# of sweeps	7	12	9
total # of shift–add	912000	101280	105120

Table 5.2. Total number of sweeps and total number of shift-add operations.

the nature of the Jacobi method guarantees that the less complex fast rotation methods are used as the matrix becomes more and more diagonally dominant during the course of the algorithm , i.e. ultimately only method *I* (the least complex method) is required. Therefore, the number of required shift–add operations per sweep reduces during the course of the Jacobi method (solid line). Since the quadratic convergence is lost this must be paid by a greater number of sweeps.

However, this trade–off between number of sweeps and computational cost (number of shift–add operations) works out to be in favour of the approximate rotation scheme. Table 5.2 shows the total number of sweeps and the total number of shift–add operations required for computing the EVD of the matrix.

Finally, in Figure 5.8 we show the reduction of the off–diagonal norm vs. the shift–add operations. Obviously, the Jacobi methods using approximate rotations obtain the reduction of the off–diagonal norm with significantly less computational cost, i.e., the overall reduction per shift–add operation is significantly better for the ap-

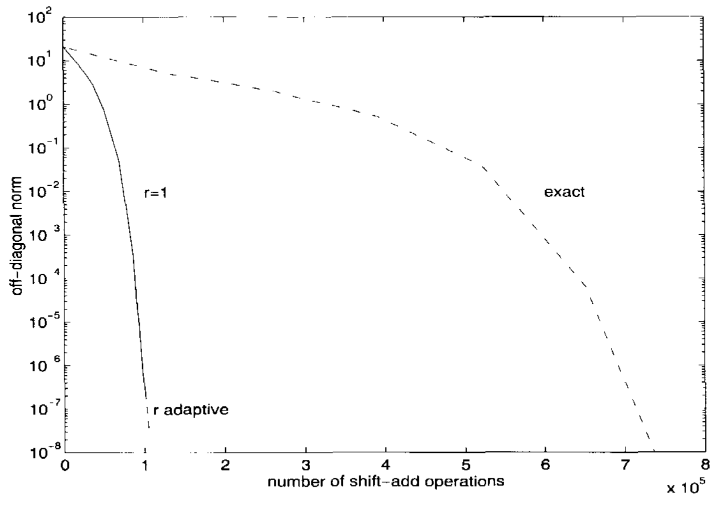


Figure 5.8. Off–diagonal norm vs. shift–add operations for the Jacobi method.

proximate rotations than for the exact rotations. Note, that the exact method actually reduces the off–diagonal norm to $O(10^{-15})$, although the stopping criteria is $10^{-8} \cdot \|\mathbf{M}\|_F$. This is due to the fast reduction of the off–diagonal norm when quadratic convergence of the algorithm is reached. Therefore, the Jacobi method with exact rotations actually requires 7 sweeps and 912000 shift–add operations (Figures 5.6 and 5.8 are only shown for the actual stopping criteria).

The adaptive scheme works as well as the simple $r = 1$ scheme. When working on a parallel array of processors, the off-chip communication becomes a function of the number of sweeps. In this case it is advantageous to use the adaptive scheme, with less sweeps, as it reduces the off–chip communication.

5.6 Conclusions

In this chapter a Jacobi–type algorithm for computing the EVD of a symmetric matrix was presented. It uses different types of fast rotations as approximate rotations. These fast rotations are characterized that the cost decreases with the size of the angle of rotation. We construct a set of fast rotations, which are indexed by an angle index κ, which may be compared to the angle exponent. The evaluation of the angle index κ, which determines the approximate rotation angle and the used type of fast rotation, can also be performed, using at most three fast rotations. Therefore, the entire Jacobi–type method can be performed by the execution of fast rotations. It is shown how these fast rotations are executed on a floating point Cordic-like architecture, making it is highly suitable for VLSI implementation.

The nature of the Jacobi method causes the size of the angles to decrease as the algorithm progresses. This has a positive effect on the use of fast rotations, which become even cheaper for such smaller angles, eventually ending up at only one shift-add pair per fast rotation, for Method I.

The modification that we have shown for EVD, that of replacing exact rotations with approximate, fast rotations, can also be applied with success on other matrix algebra algorithms. It works for the SVD of a rectangular matrix if the SVD problem is mapped to a symmetric EVD problem with increased dimension [9, 14]. The modification can also be applied to the SVD without this mapping procedure [15] by applying the approximate rotation scheme to the Cordic based SVD methods presented in [16, 17]. It has quite recently also been reported to work for QR and QR-RLS algorithms.

Bibliography

[1] R.O. Schmidt, "Multiple emitter location and signal parameter estimation," *IEEE Trans. on Antennas and Propagation*, vol. AP-34, pp. 276–280, 1986.

[2] A. Paulraj, R. Roy, and T. Kailath, "A subspace approach to signal parameter estimation," *Proceedings of the IEEE*, vol. 74, pp. 1044–1045, 1986.

[3] M. Verhaegen and P. Dewilde, "Subspace model identification. part I: The output–error state–space model identification class of algorithms," *Int. J. Control*, vol. 56, pp. 1187–1210, 1992.

[4] R.P. Brent and F.T. Luk, "The solution of singular value and symmetric eigenvalue problems on multiprocessor arrays," *SIAM J. Sci. Stat. Comput.*, vol. 6, pp. 69–84, 1985.

[5] J.-M. Delosme, "Bit-level systolic algorithm for the symmetric eigenvalue problem," in *Proc. Int. Conf. on Application Specific Array Processors*, Princeton (USA), 1990, pp. 771–781.

[6] J.J. Modi and J.D. Pryce, "Efficient implementation of Jacobi's diagonalization method on the DAP," *Numer. Math.*, vol. 46, pp. 443–454, 1985.

[7] J. Demmel and K. Veselic, "Jacobi's method is more accurate than QR," *SIAM J. Matrix Anal. Appl.*, vol. 13, pp. 1204–1245, 1992.

[8] J. Götze and G. Hekstra, "An algorithm and architecture based on orthonormal μ-rotations for computing the symmetric EVD," *Integration, the VLSI Journal*, vol. 20, pp. 21–39, 1995.

[9] G.H. Golub and C.F. van Loan, *Matrix Computations*, The John Hopkins University Press, second edition, 1989.

[10] J.P. Charlier, M. Vanbegin, and P. van Dooren, "On efficient implementations of Kogbetliantz's algorithm for computing the singular value decomposition," *Numer. Math.*, vol. 52, pp. 279–300, 1988.

[11] J. Götze, "On the parallel implementation of Jacobi and Kogbetliantz algorithms," *SIAM J. Sci. Comput.*, vol. 15, pp. 1331–1348, 1994.

[12] J. Götze, S. Paul, and M. Sauer, "An efficient Jacobi–like algorithm for parallel eigenvalue computation," *IEEE Trans. on Computers*, vol. 42, pp. 1058–1065, 1993.

[13] G.J. Hekstra and E.F. Deprettere, "Floating point CORDIC," in *11th Symp. on Computer Arithmetic*, Windsor (Canada), 1993.

[14] J. Götze, "CORDIC–based approximate rotations for SVD and QRD," in *To appear Proc. European Signal Processing Conference*, Edinburgh (Scotland), 1994.

[15] J. Götze, P. Rieder, and J.A. Nossek, "Parallel SVD–updating using approximate rotations," in *SPIE Advanced Signal Processing: Algorithms, Architectures and Implementations VI*, San Diego (USA), 1995.

[16] B. Yang and J.F. Böhme, "Reducing the computations of the singular value decomposition array given by Brent and Luk," *SIAM J. Matrix Anal. Appl.*, vol. 12, pp. 713–725, 1991.

[17] J.R. Cavallaro and F.T. Luk, "CORDIC arithmetic for an SVD processor," *J. Parallel & Distributed Computing*, vol. 5, pp. 271–290, 1988.

Chapter 6

REALIZATION OF ORTHOGONAL FIR FILTERBANKS FOR IMAGE COMPRESSION

Contents

6.1 Introduction

In this chapter, we consider the use of fast rotations in the efficient realization of orthogonal filterbanks. The application in which these filterbanks are used is that of transform coding for high quality compression of medical images without blocking artifacts. We propose to realize the orthogonal filterbank as a network of fast rotations. Since the filter coefficients are fixed, we can perform the analysis of the realisation off-line. This implies that we have extra degrees of freedom, and can spend a large amount of computational effort to come up with a realisation with the lowest achievable cost.

Image transforms, such as the Lapped Orthogonal Transform (LOT), various modifications of the LOT, the Discrete Cosine Transform (DCT), and Wavelet transforms which are all commonly used in transform coding for data compression, can be recursively decomposed as a network of orthogonal matrices of decreasing size. The basis functions on which the transform is built can approximated to any order of accuracy by realizing the set of orthogonal matrices in its decomposition by means of so-called fast rotations which are orthonormal within the range of the required accuracy. For the approximation to be optimal, all orthogonal matrices in the decomposition must be simultaneously expressed in terms of fast rotations. We present a procedure to compute the optimal solutions being either the solution of minimum cost for a given lower bound of the accuracy or the solution with the highest accuracy for a given upper bound of the cost.

Compression of X-ray image series with the transforms implemented as proposed in the chapter has shown that high accuracy at low bit rates can be achieved at a small implementation cost. Real time compression and coding of image sequences giving rise to Gbit/sec data-rates is achievable using a single chip and transforms of any size between 8×8 and 32×64.

6.1.1 Outline of this chapter

In Section 6.2, we present the background on data compression of images with orthogonal transforms. We show how the LOT transfer can be recursively decomposed into a network of smaller orthogonal matrix operations and butterfly operations. In Section 6.3, we propose a method to realize orthogonal matrices by decomposing them into a network of fast rotations. This realization can be continued until the required precision is met, making it a successive approximation. This decomposition allows a low-cost realization, but one that is orthogonal, at every step of the approximation. In Section 6.4, we apply the approximation technique on the network of orthogonal matrix operations of the LOT of Section 6.2. We pose the problem to find an optimal approximation for the whole LOT, *i.e.*, one that meets the accuracy requirements, and spends a minimal amount of computations. We show that this leads to a multi-parameter optimization problem, with an enormous search space. We present a heuristic search algorithm to tackle this problem, and present the solutions for the LOT. In Section 6.5, we present the architecture for a Parallel Transform Engine (PTFE) which is capable of implementing the complete network of the LOT, by sequential execution of the low-level operations in this network. These operations are of the type of rotation, scaling, and butterfly operations.

All of these can be expressed in terms of basic shift-and-add operations. We also present the implementation of a prototype PTFE ASIC that has been fabricated. In Section 6.6, we give our conclusions. In Appendix 6.A, we show photographs of the layout of the prototype PTFE ASIC.

6.2 Orthogonal image transforms

Data compression of images —such as X-ray image sequences— for storage purposes is heavily constrained by the requirement that the reconstructed images should not reveal coding artifacts. Compression techniques using discrete cosine transforms (DCT) [1] or conventional lapped orthogonal transforms (LOT) [2] fail to meet these requirements at high compression ratios. The modified lapped transform (MLOT) overcomes some of these problems, but it is not orthogonal which is a disadvantage from the point of view of implementation. In [3], Heusdens has presented a new LOT which is orthogonal and does not introduce any blocking artifacts when applied to medical image compression. The new LOT, however, does *not* allow a realization in terms of DCT or DST (Discrete Sine Transform) operations. Even, then, it still allows a recursive decomposition into orthogonal operations of decreasing size, a fact which we can use to our advantage.

6.2.1 The structure of lapped orthogonal transforms

The lapped orthogonal transform is a lower triangular banded Toeplitz matrix operator $\mathcal{A}(\mathbf{A})$ with upper triangular banded Toeplitz inverse $\mathcal{S}(\mathbf{S})$, $\mathcal{SA} = \mathcal{I}$, where $\mathcal{A}$ is the analysis matrix and $\mathcal{S}$ is the synthesis matrix, which are given by:

$$
\begin{aligned}
\mathcal{A} &= \begin{bmatrix} \ddots & \ddots & & & & \\ & \mathbf{A}_2 & \mathbf{A}_1 & & & \\ & 0 & \mathbf{A}_2 & \mathbf{A}_1 & & \\ & 0 & 0 & \mathbf{A}_2 & \mathbf{A}_1 & \\ & 0 & 0 & 0 & \mathbf{A}_2 & \mathbf{A}_1 \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \\
\mathcal{S} &= \begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & \mathbf{S}_1 & \mathbf{S}_2 & 0 & 0 & 0 \\ & & \mathbf{S}_1 & \mathbf{S}_2 & 0 & 0 \\ & & & \mathbf{S}_1 & \mathbf{S}_2 & 0 \\ & & & & \mathbf{S}_1 & \mathbf{S}_2 \\ & & & & & \ddots & \ddots \end{bmatrix}
\end{aligned}
\qquad (6.1)
$$

For reason of convention, we write $\mathbf{A} = [\mathbf{A}_1\mathbf{A}_2]$ and $\mathbf{S} = [\mathbf{S}_1\mathbf{S}_2]$ Both matrices have dimension $N \times 2N$ and have the following structure:

$$\mathbf{A} = \mathbf{P}_A \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{11}J \\ \mathbf{A}_{21} & -\mathbf{A}_{21}J \end{bmatrix} \quad \mathbf{S} = \mathbf{P}_S \begin{bmatrix} \mathbf{S}_{11} & \mathbf{S}_{11}J \\ \mathbf{S}_{21} & -\mathbf{S}_{21}J \end{bmatrix}, \tag{6.2}$$

where $\mathbf{P}_A$ and $\mathbf{P}_S$ are permutation matrices, $\mathbf{A}_{21} = \mathbf{B}_A\mathbf{A}_{11}$ and $\mathbf{S}_{21} = \mathbf{B}_A{}^{-T}\mathbf{S}_{11}$ for some invertible matrix $\mathbf{B}_A$, and $\mathbf{J}$ the mirror identity.

We will concentrate on the analysis side only, and hence only look at the realization of $\mathbf{A}$. A similar derivation, as we are about to present, holds for $\mathbf{S}$ too. Putting $\mathbf{A}_e = \frac{1}{2}(\mathbf{A}_{11} + \mathbf{A}_{11}J)$, $\mathbf{A}_o = \frac{1}{2}(\mathbf{A}_{11} - \mathbf{A}_{11}J)$and $\mathbf{U}_0 = [\mathbf{A}_e{}^T\mathbf{A}_o{}^T]^T$, and similarly for $\mathbf{S}$, the matrix $\mathbf{A}$ is decomposed as:

$$\mathbf{A} = \mathbf{P}_A \begin{bmatrix} \mathbf{I} & \\ & \mathbf{B}_A \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{I} \\ \mathbf{I} & -\mathbf{I} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{I} & \mathbf{I} & & \\ & & \mathbf{I} & -\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{U}_0 & \\ & \mathbf{U}_0 \end{bmatrix}. \tag{6.3}$$

Figure 6.1 shows the network corresponding to this decomposition of $\mathbf{A}$, when applied to a part of the image as:

$$\mathbf{y}_i = \mathbf{A} \begin{bmatrix} \mathbf{x}_i \\ \mathbf{x}_{i+1} \end{bmatrix}, \tag{6.4}$$

where $\mathbf{x}_i, \mathbf{x}_{i+1}$ are two $N \times 1$ vectors of consecutive samples of the image, and the $N \times 1$ vector $\mathbf{y}_i$ is the result of the filtering operation. The blocks $\mathbf{BF}_1, \mathbf{BF}_2$ in the diagram reprcsent the two butterfly operators that appear in Equation (6.2).

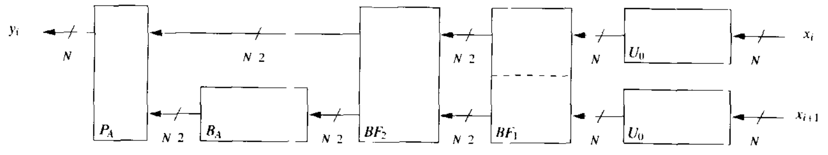


Figure 6.1. The first step in the decomposition of the LOT analysis matrix

Due to the overlap in the computation of consecutive $\mathbf{y}_i$, one of the $\mathbf{U}_0$ operators can be eliminated from the network [3], by introducing state between consecutive transforms, and leads to a more efficient computation. The Equation (6.3) is split up into two parts, introducing the $\frac{N}{2} \times 1$ vectors $\mathbf{s}_i, \mathbf{s}_i'$ as the external and internal state variables, and is written out as:

$$\begin{aligned} \begin{bmatrix} \mathbf{s}_i \\ \mathbf{s}_i' \end{bmatrix} &= \begin{bmatrix} \mathbf{I} & -\mathbf{I} \\ \mathbf{I} & \mathbf{I} \end{bmatrix} \cdot \mathbf{U}_0 \cdot \mathbf{x}_i \\ \mathbf{y}_i &= \mathbf{P}_A \begin{bmatrix} \mathbf{I} & \\ & \mathbf{B}_A \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{I} \\ \mathbf{I} & -\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{s}_i' \\ \mathbf{s}_{i+1} \end{bmatrix}. \end{aligned} \tag{6.5}$$

The more efficient network corresponding to Equation (6.5) is shown in Figure 6.2 and will be used later on in the realization of the LOT.

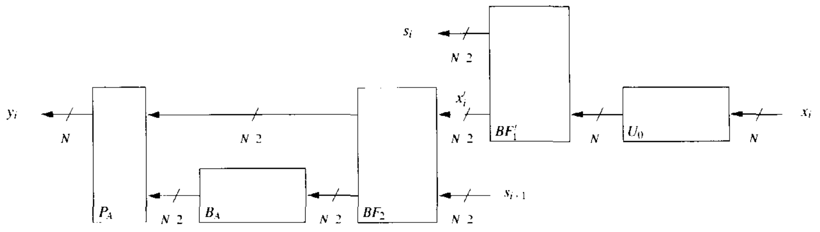


Figure 6.2. The reduced network of the LOT analysis matrix

Now since $\mathbf{U}_0 = [\mathbf{A}_e{}^T \mathbf{A}_o{}^T]^T$ with $\mathbf{A}_e$ and $\mathbf{A}_o$ even and odd, respectively, it holds that this matrix has a similar decomposition to $\mathbf{A}$ as given by:

$$\mathbf{U}_0 = \mathbf{P}_{U_0} \begin{bmatrix} \mathbf{I} & \\ & \mathbf{B}_{U_0} \end{bmatrix} \begin{bmatrix} \mathbf{U}_1 & \\ & \mathbf{U}_1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{J} \\ \mathbf{I} & \textbf{-J} \end{bmatrix}. \tag{6.6}$$

It turns out that the same type of decomposition holds for $\mathbf{U}_1$, and in fact, the entire decomposition of $\mathbf{U}_0$ can be written out recursively as:

$$\mathbf{U}_k = \mathbf{P}_{U_k} \begin{bmatrix} \mathbf{I} & \\ & \mathbf{B}_{U_k} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{k+1} & \\ & \mathbf{U}_{k+1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{J} \\ \mathbf{I} & \textbf{-J} \end{bmatrix}. \tag{6.7}$$

where the $\mathbf{B}_{U_k}$ are decreasing in size by $N \times N, \frac{N}{2} \times \frac{N}{2}, \ldots, 1 \times 1$. The recursion is a remarkable property and will in general not exist, that is will stop after the decomposition of $\mathbf{A}$ [4]. The recursively defined network that belongs to this decomposition of $\mathbf{U}_k$ is shown in Figure 6.3, the block **BF** again representing the butterfly operator in the expression.

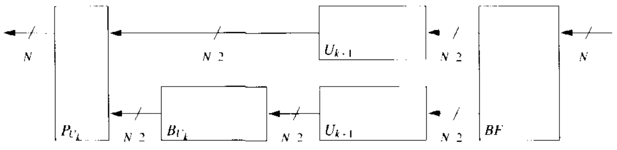


Figure 6.3. The recursive decomposition of the $\mathbf{U}_k$ submatrices

In [4], it is shown that the synthesis matrix **S** has the property that the network parameters are exactly the same as those for the decomposition of the analysis matrix **A**, though only reversed in sequence.

6.3 Efficient orthogonal realization

We consider the matrix-vector multiplication of an arbitrary matrix **A** with a vector **x**, as given by:

$$\mathbf{y} = \mathbf{A}\mathbf{x}. \tag{6.8}$$

For sake of brevity, we consider only the case of **A** having the dimensions $m \times n_a$, with $m \leq n_a$. Other cases follow from transposition. As well as this we impose that **A** has all singular values smaller than or equal to one, $\sigma_i \leq 1$, or more specific: $\mathbf{A}\mathbf{A}^T \leq \mathbf{I}$. This constraint can always be satisfied with proper scaling of the matrix.

We now define the **orthogonal embedding G** of **A** as:

$$\mathbf{G} = [\mathbf{A}_c | \mathbf{A}], \tag{6.9}$$

with the matrix **G** having orthonormal rows and of size $m \times n$, $n = n_a + n_c$, and with the matrix $\mathbf{A}_c$ of size $m \times n_c$, $n_c \leq m$, chosen such that the orthogonality condition holds for the matrix **G** in:

$$\mathbf{G}\mathbf{G}^T = \mathbf{A}_c\mathbf{A}_c^T + \mathbf{A}\mathbf{A}^T = \mathbf{I}. \tag{6.10}$$

We briefly state that such $\mathbf{A}_c$ exists for an arbitrary matrix **A**, and can be constructed by the Cholesky factorization of $\mathbf{I} - \mathbf{A}\mathbf{A}^T$.

It is easily shown that the product of Equation (6.8) can be written as $\mathbf{y} = \mathbf{G}\mathbf{P}_1 x$, with the **extension matrix** $\mathbf{P}_1$ given by:

$$\mathbf{P}_1 = \begin{bmatrix} \mathbf{O}_{n_c \times n_a} \\ \mathbf{I}_{n_a \times n_a} \end{bmatrix}. \tag{6.11}$$

For the case that **A** is already orthogonal, the matrix $\mathbf{A}_c$ is of dimension zero, $n = n_a$, and $\mathbf{P}_1$ becomes the identity matrix.

Without proof, we state that there exists an orthogonal matrix **Q**, with $\mathbf{Q}\mathbf{Q}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I}$, of size $n \times n$, that factorizes **G** such that

$$\mathbf{Q}\mathbf{G}^T = \mathbf{P}_2 = \begin{bmatrix} \mathbf{I}_{m \times m} \\ \mathbf{O}_{(n-m) \times m} \end{bmatrix}. \tag{6.12}$$

We call the matrix $\mathbf{P}_2$, of size $n \times m$, a **selection matrix** for reasons that will become clear later on. For the case that **A** is a square matrix, with $m = n$, $\mathbf{P}_2$ becomes the identity matrix.

Substituting the result of Equation (6.12) into the product of Equation (6.11) we obtain the desired expression for the matrix-vector product in terms of an orthogonal rotation matrix.

$$\mathbf{y} = \mathbf{P}_2^T\mathbf{Q}\mathbf{P}_1\mathbf{x} \tag{6.13}$$

6.3.1 Factorization of Q by means of fast rotations

Any orthogonal rotation matrix **Q** can be implemented as a sequence of elementary rotations. A common example of this is QR decomposition. We opt to factorize the matrix **G**, where **Q** is composed of a sequence of 2×2 fast rotations q_t, as presented in Chapter 3.

The factorization of **Q**, by means of a so-called **greedy** algorithm, proceeds as follows. Setting the initial matrices $\mathbf{Q}_0 = \mathbf{I}, \mathbf{R}_0 = \mathbf{G}^T$, we factorize **G** with the recursion:

$$\begin{aligned} \mathbf{Q}_t &= q_t \cdot \mathbf{Q}_{t-1} \\ \mathbf{R}_t &= q_t \cdot \mathbf{R}_{t-1}, \end{aligned} \tag{6.14}$$

where the embedded 2×2 fast rotation matrix q_t, determined by the index pair (i, j) and rotation angle α, is given by the $n \times n$ identity matrix with embedded entries:

$$\begin{bmatrix} q_{ii} & q_{ij} \\ q_{ji} & q_{jj} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}. \tag{6.15}$$

The angle of rotation α satisfies:

$$\alpha = \arctan(\frac{s}{c}). \tag{6.16}$$

At every step t of the factorization, the index pair (i, j) and the approximation pair c, s that make up q_t are chosen according to the **greedy** rule that the quantity:

$$\Delta = \mathrm{Off}(\mathbf{R}_{t-1}) - \mathrm{Off}(\mathbf{R}_t), \tag{6.17}$$

is maximized. The function $\mathrm{Off}(\mathbf{R}) = \sum_{i\neq j} r_{ij}^2$ is a metric for the amount of off-diagonal energy in the matrix **R**, and measures the progress of the annihilation of the off-diagonal entries. The quantity Δ, being the difference in off-diagonal energy gained in one step, is a measure of how fast the algorithm converges. Note that this algorithm belongs to the class of iterative Jacobi algorithms, and ultimately converges to the required solution of:

$$\begin{aligned} \lim_{T\to\infty} \mathbf{Q}_T &= \mathbf{Q} \\ \lim_{T\to\infty} \mathbf{R}_T &= \mathbf{P}_2. \end{aligned} \tag{6.18}$$

6.3.2 Finding the optimal rotation parameters

The optimal q_t that maximizes Δ is found by going over all combinations of the index pair (i, j), and for each pair determining the optimal rotation angle α_{opt} and evaluating Δ. For determining the optimal rotation angle α_{opt} we need only to look at small submatrices of the i^{th} and j^{th} columns and rows of $\mathbf{R}' = \mathbf{R}_t$, q_t, and $\mathbf{R} = \mathbf{R}_{t-1}$, as other entries in the matrices have no influence on the annihilation of off-diagonal energy.

$$\begin{bmatrix} r'_{ii} & r'_{ij} \\ r'_{ji} & r'_{jj} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \cdot \begin{bmatrix} r_{ii} & r_{ij} \\ r_{ji} & r_{jj} \end{bmatrix}. \tag{6.19}$$

For maximal annihilation of the off-diagonal energy, we have to minimize the quantity $(r'_{ij})^2 + (r'_{ji})^2$. It is easy to prove that, by differentiating the said quantity and setting this to zero, the solution for the optimal angle then follows as being:

$$\alpha_{\text{opt}} = \arctan\left(\frac{-2(r_{ii}r_{ji} - r_{jj}r_{ij})}{r_{ii}^2 + r_{jj}^2 - r_{ij}^2 - r_{ji}^2}\right)/2. \tag{6.20}$$

The optimal angle α_{opt} itself is not used in the rotation. Instead we choose the closest possible approximation pair c, s with angle of rotation α, so that the cost of rotation is minimized. The quantity Δ is then evaluated for this combination of index pair (i, j) and angle α by application of Equation (6.17) on Equation (6.19) as:

$$\Delta = (r_{ij}^2 + r_{ji}^2) - ((r'_{ij})^2 + (r'_{ji})^2). \tag{6.21}$$

6.3.3 Practical realization

For practical cases, the factorization need not be continued to infinity. For a preset accuracy, usually expressed in terms of remaining off-diagonal energy in $\mathbf{R}_t$, the factorization can be terminated after, say, T steps. The approximation $\hat{\mathbf{Q}} = \mathbf{Q}_T$ of the matrix $\mathbf{Q}$ is then used in the approximate realization of $\hat{\mathbf{y}}$ in:

$$\hat{\mathbf{y}} = \mathbf{P}_2 T \hat{\mathbf{Q}} \mathbf{P}_1 \mathbf{x}. \tag{6.22}$$

Here, the matrix $\mathbf{A}$ is approximated by $\hat{\mathbf{A}} = \mathbf{P}_2^T \hat{\mathbf{Q}} \mathbf{P}_1$.

The matrix vector product, as expressed in Equation (6.22), is realized as a network of the T 2×2 fast rotation operations in the factorization of $\hat{\mathbf{Q}}$. The extension matrix $\mathbf{P}_1$ and the selection matrix $\mathbf{P}_2$ have no other task than to add zeros at certain inputs of the network and to select certain outputs of the network as the result. They do not contribute in any additional cost.

6.3.4 Realization of the recursive network for the LOT transform

The recursive network of the LOT transform, as shown in Figures 6.1 and 6.3, contains the orthogonal rotation matrices $\{\mathbf{B}_A, \mathbf{B}_{U_0}, \mathbf{B}_{U_1}, \mathbf{B}_{U_2}, \ldots\}$, as well as some trivial butterfly operations. For each of these rotation matrices we apply the approximate realization of Equation (6.22). The optimal number of steps required for the realization of each matrix is calculated by a heuristic optimization program, working with the constraints that a certain accuracy criteria is met for the total realization, while maintaining a minimal cost.

6.4 Recursive approximate realization of the LOT

A property of the matrices $\mathbf{B}_A, \mathbf{B}_{U_0}, \mathbf{B}_{U_1}, \ldots$ in the recursive network of the LOT is that all are orthogonal and already mainly diagonal. This property is essential for an even faster convergence of the approximations of these matrices. For the full recursion of the proposed 16×32 LOT, we approximate each of the matrices with the matrices $\hat{\mathbf{B}}_A, \hat{\mathbf{B}}_{U_0}, \hat{\mathbf{B}}_{U_1}, \hat{\mathbf{B}}_{U_2}$, using the technique described in the previous section, with respectively $T_A, T_{U_0}, T_{U_1}, T_{U_2}$ steps in the approximations. The approximation $\hat{\mathbf{A}}$ follows from the reconstruction using these approximates. For completeness, we have to mention that the recursive decomposition ends with the matrices $\mathbf{B}_{U_3}, \mathbf{U}_4$, and that these too are used in the reconstruction. Since both are trivial 1×1 matrices, $\mathbf{B}_{U_3} = [1], \mathbf{U}_4 = [\frac{1}{8}]$, with an exact realization of no real cost, they need not be approximated.

The number of steps used in the respective approximations of the matrices form an *index* $\mathbf{i} = (T_A, T_{U_0}, T_{U_1}, T_{U_2})$ to a P-parameter approximation (in this case, $P = 4$). We call the corresponding approximation $\hat{A} = f_{\mathrm{mat}}(\mathbf{i})$ the *solution* belonging to this index, where $f_{\mathrm{mat}}(\mathbf{i})$ is the reconstruction function, in terms of the index $\mathbf{i}$. Clearly this function depends on a given factorization of the matrices.

We also define the function $f_{\mathrm{cost}}(\mathbf{i})$ and the function $f_{\mathrm{acc}}(\mathbf{i})$, both in terms of the index $\mathbf{i}$, as the overall cost function of the solution, and the overall accuracy of the solution. We measure the overall cost as total number of shift-add operations in the resulting network, which is a weighted sum of the cost of the realization of submatrices and of the introduced butterfly operations. We measure the accuracy of the solution as the norm of the difference between original and approximation, for which we take $f_{\mathrm{acc}}(\mathbf{i}) = -\log_2(\|\mathbf{A} - \hat{\mathbf{A}}\|)$, where the approximation $\hat{\mathbf{A}}$ is given by $\hat{\mathbf{A}} = f_{\mathrm{mat}}(\mathbf{i})$. Analysis of the cost and accuracy functions reveals that the cost function $f_{\mathrm{cost}}(\mathbf{i})$ is monotonous throughout the solution space. Hence we can write, for any index $\mathbf{i}$, and any positive increment $\delta \geq 0$:

$$f_{\mathrm{cost}}(\mathbf{i} + \delta) \geq f_{\mathrm{cost}}(\mathbf{i}), \qquad \forall \mathbf{i}, \forall \delta \geq 0. \tag{6.23}$$

The accuracy function $f_{\mathrm{acc}}(\mathbf{i})$ is close-to-monotonous, that is, for *most* indices $\mathbf{i}$ and increment $\delta \geq 0$, but not for *all* the monotonicity holds. The disturbance of the monotonicity is small and very local in nature, so that we can set an —empirically determined— error bound $\varepsilon > 0$, such that we can write:

$$f_{\mathrm{acc}}(\mathbf{i} + \delta) + \varepsilon \geq f_{\mathrm{acc}}(\mathbf{i}), \qquad \forall \mathbf{i}, \forall \delta \geq 0. \tag{6.24}$$

Further analysis reveals that, when choosing an increment δ in only one dimension, the accuracy function exhibits *saturation*. This means that an increase in accuracy in only *one* of the matrices is only cost-effective up to a certain level, until the combined accuracies of the other matrices start to play a role, and saturation sets in. This is a clear indication that, in order to find cost-effective solutions, a *simultaneous* approximation of the matrices must be made.

6.4.1 The search for optimal solutions

For the search of a cost-effective solution to the approximation problem, let us first define the target accuracy a_{target} and target cost c_{target} for the search, and state that any solution i must satisfy both $f_{\text{acc}}(\mathbf{i}) \geq a_{\text{target}}$ and $f_{\text{cost}}(\mathbf{i}) \leq c_{\text{target}}$. Furthermore, we define a *cell* $C(\mathbf{p},\mathbf{q})$ in the solution space as the collection of points lying between the bounding indices $\mathbf{p},\mathbf{q}$ with $\mathbf{p} \leq \mathbf{q}$, and write $\mathbf{i} \in C(\mathbf{p},\mathbf{q}) \iff \mathbf{p} \leq \mathbf{i} \leq \mathbf{q}$. Hence, we formulate the discriminative property of a cell $C(\mathbf{p},\mathbf{q})$, that it contains *no* cost-effective solutions if $f_{\text{acc}}(\mathbf{q}) + \varepsilon < a_{\text{target}}$ or $f_{\text{cost}}(\mathbf{p}) > c_{\text{target}}$.

We have implemented a heuristic 2^P-tree branch and bound search algorithm [5], that is capable of finding cost-effective solutions either by finding the best solution $\mathbf{s}$ for a given target accuracy a_{target}, such that $f_{\text{acc}}(\mathbf{s}) \geq a_{\text{target}}$ and $f_{\text{cost}}(\mathbf{s})$ is minimal, or by finding the best solution $\mathbf{s}$ for a given target cost c_{target}, such that $f_{\text{cost}}(\mathbf{s}) \leq c_{\text{target}}$ and $f_{\text{acc}}(\mathbf{s})$ is maximal.

To explain the operation we take the first case, with a given target accuracy a_{target}, and initial target cost of $c_{\text{target}} = \infty$, and search the entire solution space as follows. First, we factorize each the matrices $\mathbf{B}_A, \mathbf{B}_{U_0}, \ldots$ in the recursive decomposition of the LOT independently, until they reach a sufficient level of (maximum) accuracy, thus setting the bounds of the index to the solution as the number of steps required to reach this maximum accuracy. For the 16×32 LOT, the upper bounds are $(128, 128, 27, 5)$, leading to a solution space of size 2.2×10^6. For the 32×64 LOT, this results in upper bounds of $(511, 511, 128, 27, 5)$ and a solution space of size 4.3×10^9. Next, a given cell (the root of the search is the entire solution space) is split into at most 2^P subcells, and each of these is tested whether they could contain any solutions. If a cell may contain solutions, it is split and checked recursively. If not, the corresponding branch of the search tree is cut off. If a solution is found during the search, it is used to set the new target cost c_{target} dynamically, so that less cells need to be examined. The result of the search is a solution that satisfies the constraints and has guaranteed minimum cost.

We have made the interface between the search program and the objective functions $f_{\text{acc}}(\mathbf{i}), f_{\text{cost}}(\mathbf{i})$ such that it can be used for other transforms. We have used it for approximated networks for MLOT, DCT, and wavelet transforms with success.

6.4.2 Results

In Table 6.1, we show the results for approximate realizations of the 16×32 LOT, of increasing accuracy. The accuracy is shown here as the norm of the differences, $||\mathbf{A} - \hat{\mathbf{A}}||$.

Our method shows a rapid convergence, so that solution number 16 in the table, with cost only 776 shift-add-pair operations, is already (visually) indistinguishable form the original, both in smoothness of the basis-functions and in the frequency responses. As a comparison, a direct implementation would require 512 high-accuracy multiply-add operations ($\approx$ 10.000 additions), without having the

solution	index	target acc.	actual acc.	cost
4	(13 8 3 1)	0.4210	0.4163	314
8	(25 18 5 2)	0.1606	0.1578	494
12	(36 32 7 2)	0.0613	0.0602	644
16	(45 44 11 3)	0.0234	0.0231	776
20	(64 56 11 3)	0.0089	0.0088	900
24	(82 77 13 3)	0.0034	0.0033	1046
28	(90 85 18 4)	0.0013	0.0013	1122
32	(103 96 22 5)	0.0005	0.0005	1194
36	(116 115 24 5)	0.0002	0.0002	1266
40	(127 124 26 5)	0.0001	0.0001	1314

Table 6.1. Optimal solutions for the 16×32 LOT with full recursion depth and varying target accuracy.

desirable properties that orthogonal implementations like this one has. Of course, a multiplier implementation following the DCT decomposition is cheaper, but fails for many transforms, such as our new LOT.

We have also tested our search program on partial depth recursive decompositions of the LOT. In Table 6.2, we show the results for full depth (level = 5) until direct (level = 0) implementations.

recursion depth	approximated matrices	index	actual acc.	cost
5	$\mathbf{B}_A, \mathbf{B}_{U_0}, \mathbf{B}_{U_1}, \mathbf{B}_{U_2}, (\mathbf{B}_{U_3}, \mathbf{U}_4)$	(45 44 11 3)	0.0231	776
4	$\mathbf{B}_A, \mathbf{B}_{U_0}, \mathbf{B}_{U_1}, \mathbf{B}_{U_2}, \mathbf{U}_3$	(45 45 11 3 3)	0.0230	780
3	$\mathbf{B}_A, \mathbf{B}_{U_0}, \mathbf{B}_{U_1}. \mathbf{U}_2$	(49 49 9 17)	0.0231	948
2	$\mathbf{B}_A, \mathbf{B}_{U_0}, \mathbf{U}_1$	(49 45 66)	0.0211	1428
1	$\mathbf{B}_A, \mathbf{U}_0$	(56 260)	0.0230	1988
0	$\mathbf{A}$	(691)	0.0226	3808

Table 6.2. Optimal solutions for the 16×32 LOT with different levels of partial recursion and a fixed target accuracy of 0.0233.

The solutions are targeted at the accuracy of solution number 16 of Table 6.1. The results clearly show that the full depth recursive decomposition of the LOT, with simultaneous approximation of the submatrices leads to the best results.

6.5 Architecture of the transform engine

The TransForm Engine (TFE) [6, 7] implements the complete recursive flowgraph of Figure 6.1 and 6.3 by sequential execution of the low-level operations in this flowgraph. These operations are of the type of rotation, scaling, and butterfly operations. All of these can be expressed in terms of basic shift-and-add operations.

We have deliberately chosen for a sequential architecture approach for the TFE for two important reasons:

- **routing / communication.** As the flowgraph of a single transform operation is of a highly irregular nature at the lowest level, a parallel version would have high routing overhead due to the communication demand. In a sequential version, this irregular communication translates itself into a randomly accessible storage structure, such as a register file, and makes it more manageable.
- **flexibility.** A sequential machine is more flexible in the sense that different transform sizes and even different transform families can be mapped onto it. A different transform translates itself to a different control sequence for its implementation. This control sequence can be seen as a "stored program" for the transform.

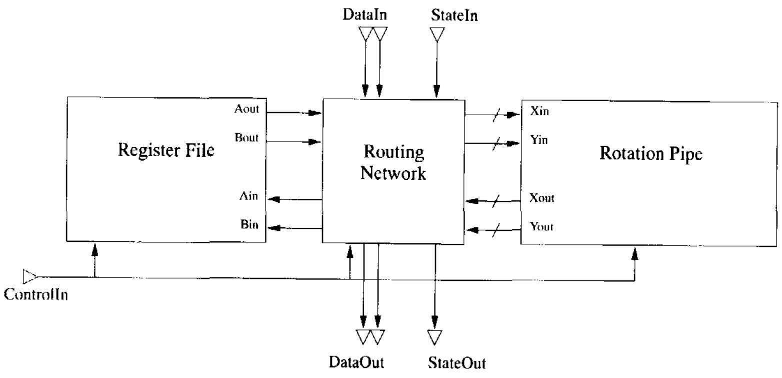


Figure 6.4. Architecture of a single transform engine (TFE).

The architecture of the TFE, as shown in Figure 6.4, is composed of the following subunits:

1. A pipelined rotation engine, capable of implementing any type of rotation, scaling or butterfly operation. A pipelined unit is chosen to greatly increase

the throughput of the machine, and also to increase the ratio of silicon area for computation versus that for storage and communication.

The pipeline is built out of identical stages, of which one is shown in Figure 6.5. These stages are designed such that they can implement any sequence of fast rotations of the types described in [8, 9], any scaling operation and butterfly operations.

2. A register file, capable of storing all necessary input, output and intermediate results. As all operations are of the type (2-in, 2-out), the storage demand stays constant throughout the execution of the flowgraph.

3. A simple routing network, which routes the data between the rotation pipe, the register file and the in- and output ports.

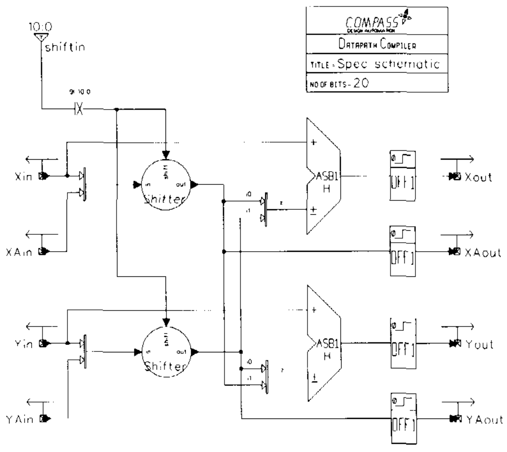


Figure 6.5. Schematic of a single pipe stage.

For all the above units it holds that they do not have any local control. All control signals in the TFE are bundled together and brought out as a single, very wide, control word. The removal of local control is possible as the flowgraphs of the transforms in question are static and data-independent, and therefore allow an implementation as a sequence of control words.

Using a *pipelined* rotation engine has serious repercussions on the factorization of the matrices. For one, all rotation operations must be constructed in such manner that they fit exactly in the number of available pipeline stages, or a multiple thereof, as not to waste any computational power. As well as this, precautions must be taken to guarantee that the data dependencies in the resulting flowgraph are such that the pipeline can be filled at all times without having to introduce too many dummy operations and hence losing performance.

The above problems have been solved in the **pipelined greedy** factorization algorithm, which produces a sequence of rotation operations q_t that permit a pipelined schedule. The matrix is factored according to the **greedy** algorithm under the additional constraint that, for any sequence of operations, the operation q_k has no conflict (data dependency) with consecutive operations $\{q_k, q_{k+1}, \ldots\}$ for the duration that it resides in the pipeline. A second additional constraint is that all the q_t fit in a multiple of the number of pipeline stages; the length of the q_t as expressed in stages, is also the time it resides in the pipeline.

Whereas in the non-pipelined **greedy** algorithm, the closest simple fast rotation is chosen for q_t, the pipelined version takes a *group* of fast rotations, such that their total length is a multiple of the pipeline depth. The resulting q_t is a better approximation, and hence the algorithm converges faster, requiring less steps. However, the nature of the fast rotations greatly limits the practical length of the pipeline, and with it, the achievable level of internal parallelism in the TFE. Practical experiments [4] have shown that, for the proposed transform and accuracy requirements, a pipeline depth of 4 stages is still considered as cost-effective.

The recursive flowgraph of the LOT contains parallel independent instances of the same (smaller) rotation matrices. These are scheduled in an interleaved manner onto the pipeline, thus weakening the conflict constraint, and allowing a less constrained factorization of the smaller matrices.

6.5.1 Architecture of the PTFE

The architecture of the system of parallel transform engines (PTFE) is shown in Figure 6.6. It consists of a linear array of transform engines (TFE's), as described in the previous section, working in parallel and which are governed by a global controller (Main Control). Connections between the TFE units (control and state signals) are of a highly local nature, minimizing routing overhead. All TFE units are connected to global busses for input and output of transform data. Both types of busses are implemented in double to increase throughput of data. Operation of the TFE units is *skewed* in time. Every TFE executes the same sequence of operations, as dictated by the control sequence, only with a fixed time difference.

A single *global* controller is used for the entire system of TFE's. This global controller (Main Control) stores the entire control sequence for the transform and delivers it to the first TFE unit in the chain when the input sequence arrives. The control is skewed in time and delivered to subsequent TFE units by means of FIFO's (ControlDelay). This architecture greatly reduces the control overhead as opposed to each TFE having its own, stored control sequence. Moreover, we have implemented the storage for the control sequence as a RAM, making the system (re)programmable for almost any type of transform. At power-up or in download

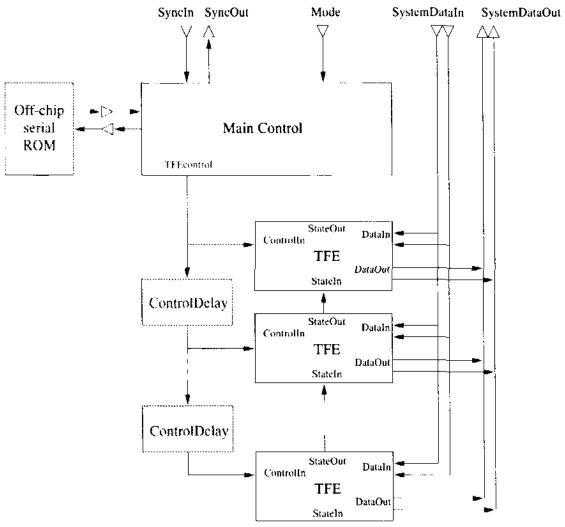


Figure 6.6. The architecture of the system of parallel transform engines (PTFE).

mode, this RAM is filled, either from a small serial ROM, or from a dedicated interface port.

The control at each TFE determines when it can read the input busses, and when it can send results over the (tri-state) output bus. This control sequence must be conflict-free. Figure 6.7 shows the typical schedule of the TFE's in time, and how the input and output phases of consecutive TFE's adjoin. The input and output

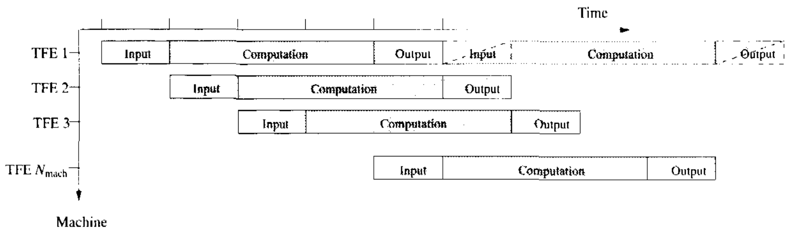


Figure 6.7. Time schedule of the TFE units.

phases are shown as partially overlapping with the computation phase. Special constraints are placed on the input and output sequence, as to allow a schedule where useful computation can start as early as possible, and can continue as long as possible. Additional stringent constraints can be placed on schedule and register allocation to even overlap the output phase with the input phase of the next transform on the same TFE unit.

The length of the input, output and computation phases is heavily dependent on the size and type of transform, the required accuracy, and the efficiency of the factorization. If a single PTFE chip does not have enough resources to provide the required throughput, more than one chip can be chained together in a system of PTFE's. Synchronization links between the PTFE chips schedule the system and prevent any system bus conflicts.

6.5.2 Mapping the flowgraph onto the architecture

Mapping the flowgraph of the transform onto the TFE architecture is basically a *list scheduling* problem. The schedule is greatly simplified by the fact that the natural order of the sequence of rotations, as produced by the **pipelined greedy** factorization algorithm, is of such nature that it can be used directly as a schedule. The problem remains of merging the (sub)schedules of the recursive network into one global schedule, with additional constraints on the sequence in which the input and output is produced, and how the state transfer between machines is synchronized.

6.5.3 Implementation of a prototype PTFE ASIC

A prototype of the PTFE, containing only 3 TFE's, has been designed in a 0.8μ CMOS process, using Compass tools. It has been fabricated at European Silicon Structures (ES2) in France.

The designed operation frequency of the ASIC is 40MHz. The die size of the PTFE measures $11 \times 10\text{mm}^2$. It is packaged in a 144 pin PGA package. In Figure 6.8 in Appendix 6.A we show a photograph of the layout of the prototype PTFE. Clearly visible in this photograph are the three TFE units, and the large storage for the instruction sequence that defines the transform. In Figure 6.9 we show a photograph of the detailed layout of a single TFE. Visible in the photograph are the 4 stage deep fast rotation pipeline, the register file for storage of intermediate variables, the memory to delay the incoming instruction sequence, and the local controller. Note the relatively small size of the controller. The PTFE has been designed by Zeng Zhiqiang [7, 6].

6.6 Conclusions

We have shown that the operators in Equation (6.1) can be implemented in an efficient way using fast rotations. The basic transform matrix **A** of dimension $N \times 2N$, where the transform size N can be as large as 32, is recursively decomposed into a network of rotation matrices and butterfly operators. The recursive decomposition of the block rows ensures that the structure of the transform matrix A is preserved under approximation of the submatrices, and using orthogonal operations preserves the orthogonality- that is the conditioning - of the transform. The rotation matrices that appear are approximated by a finite length greedy factorization into fast rotations. This leads to the flowgraph of an efficient realization in terms of fast rotation and trivial butterfly operations. The approximations are optimal in the sense that both cost and accuracy are no more than required by the application.

The resulting flowgraph is mapped onto a powerful transform engine (TFE), which is centered around a rotation pipeline. Care has been taken to increase greatly the amount of silicon that is active with useful computations. A single chip implementation of the PTFE containing 8 of such TFE units working in parallel, would be the target for our application. A prototype PTFE, containing only 3 TFE's has been designed in a 0.8μ CMOS process. It has a (re)programmable controller that is downloaded with the scheduled control sequence of the transform. This flexibility in the programming makes it that the system can handle much more than one type of transform.

We have mapped many transforms on the architecture, including the DCT, LOT's which can be decomposed in terms of DCT's, and LOT's that cannot. Also for transforms that do not allow recursive decomposition can an approximating flowgraph be derived and mapped on the architecture. Very good results have been obtained with compression and coding of X-ray image sequences for storage purposes. In this application, we have used the LOT from [3, 4] which does not introduce blocking artifacts and which cannot be decomposed in terms of DCT's.

Appendix

6.A Images of the Parallel Transform Engine ASIC

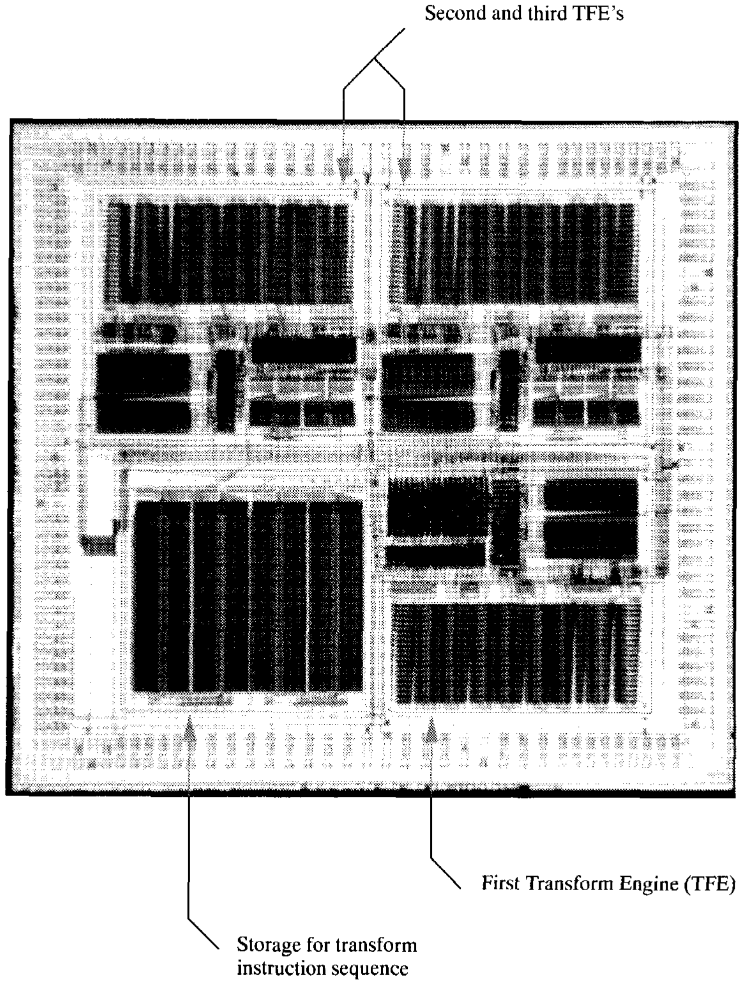

Second and third TFE's
First Transform Engine (TFE)
Storage for transform instruction sequence

Figure 6.8. Photograph of the layout of the PTFE chip, showing the TFE's and the storage for the instruction sequence.

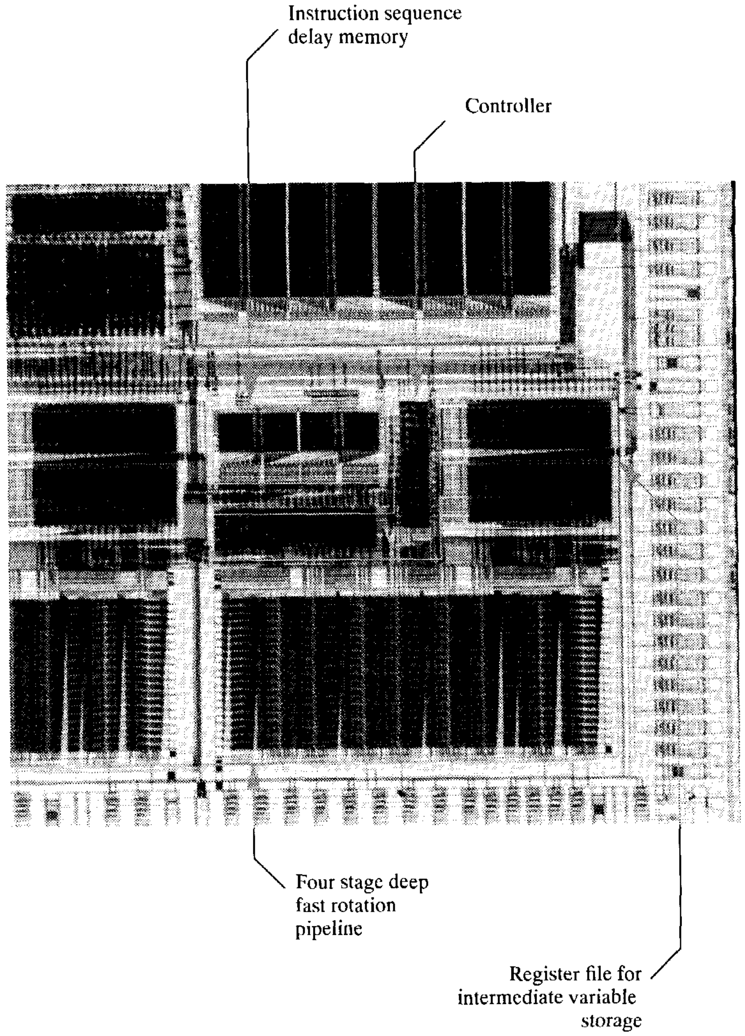

Instruction sequence delay memory
Controller
Four stage deep fast rotation pipeline
Register file for intermediate variable storage

Figure 6.9. Detail photograph of the layout of the PTFE chip, showing the fast rotation pipe, control, and storage of a single TFE.

Bibliography

[1] R. Veldhuis and M. Breeuwer, *An Introduction to Source Coding*, Prentice Hall, New York, 1993.

[2] H.S. Malvar and D.H. Staedlin, "The LOT: Transform coding without blocking effects," *IEEE Trans. on ASSP*, vol. 37, no. 4, pp. 553–559, 1989.

[3] Richard Heusdens, "Design of lapped orthogonal transforms," *IEEE Trans. on Image Processing*, to appear.

[4] Richard Heusdens, *Overlapped Transform Coding of Images: Theory, Application, and Realization*, Ph.D. thesis, Delft University of Technology, 1996.

[5] Gerben J. Hekstra, Ed F. Deprettere, Richard Heusdens, and Monica Monari, "Recursive approximate realization of image transforms with orthonormal rotations," in *Proceedings International Workshop on Image and Signal Processing*, Manchester, UK, November 1996.

[6] Gerben J. Hekstra, Ed F. Deprettere, Richard Heusdens, and Zhiqiang Zeng, "Efficient orthogonal realization of image transforms," in *1996 IEEE Workshop on VLSI Signal Processing*, San Francisco, November 1996.

[7] Gerben J. Hekstra, Ed F. Deprettere, Richard Heusdens, and Zhiqiang Zeng, "Efficient orthogonal realization of image transforms," in *Proceedings SPIE*, Denver, Colorado, US, August 1996.

[8] J. Götze and G. Hekstra, "An algorithm and architecture based on orthonormal μ-rotations for computing the symmetric EVD," *Integration, the VLSI Journal*, vol. 20, pp. 21–39, 1995.

[9] Gerben J. Hekstra and Ed F.A. Deprettere, "Fast rotations: Low-cost arithmetic methods for orthonormal rotation," in *Proceedings of the 13th Symposium on Computer Arithmetic*, Tomas Lang, Jean-Michel Muller, and Naofumi Takagi, Eds. IEEE, July 1997, pp. 116–125.

Index

GLOSSARY OF SYMBOLS

General, Chapters 2 and 3

α	angle of rotation (general).
$\tilde{\alpha}$	approximated angle of rotation.
$\alpha(x)$	angle of rotation.
α_{h}, $\alpha_{\mathrm{h}}(x)$	hyperbolic angle of rotation.
$\alpha_{\mathrm{u}}(x)$	angle of rotation of a unified fast rotation.
$\tilde{\alpha}_i$	base angle.
Γ	overall angle domain of convergence.
γ	angle domain of convergence.
δ, δ_i	angle resolution.
ε	error (general). error in magnification (specific).
ε_α	error in approximation of an angle.
ε_K	error in the overall magnification.
$\varepsilon_{\mathrm{round}}$	error in rounding towards nearest.
$\varepsilon_{\mathrm{chop}}$	error in rounding towards minus infinity.
κ	angle exponent.
σ, σ_i	digits of the angle representation.

K	overall magnification factor (Cordic).
$\dot{K}$	finite-precision inverse of K.
L	cost of a fast rotation.
N_{exp}	number of bits in the representation of the exponent.
N_{mant}	number of bits in the representation of the mantissa.

c, $c(x)$	cosine part of a (polynomial) sine-cosine approximation pair.
c_{h}, $c_{\mathrm{h}}(x)$	cosine part of a (polynomial) hyperbolic sine-cosine approximation pair.
$c_{\mathrm{u}}(t.x)$	cosine part of a polynomial unified sine-cosine approximation pair.
q	accuracy of a fast rotation (in bits).
s, $s(x)$	sine part of a (polynomial) sine-cosine approximation pair.

s, $s(x)$	sine part of a (polynomial) hyperbolic sine-cosine approximation pair.
$s_{\mathrm{u}}(t,x)$	sine part of a polynomial unified sine-cosine approximation pair.
m, $m(x)$	magnification factor.
m_{h}, $m_{\mathrm{h}}(x)$	magnification factor of a hyperbolic fast rotation.
$m_{\mathrm{u}}(t,x)$	magnification factor of a unified fast rotation.

$\mathbf{E}_{\alpha}$	rotation matrix for the error ε_{α}.
$\mathbf{F}$, $\mathbf{F}_i$	fast rotation matrix. *also*
$\mathbf{F}_i$	factor matrix of a factored fast rotation.
$\mathbf{R}$	rotation matrix.
$\tilde{\mathbf{R}}$	approximate rotation matrix.

$\mathcal{A}$, $\mathcal{A}_{\kappa}$	angle base.

Chapter 4

Δ_{φ}, Δ_{θ}	angle resolution for hemisphere coordinates.
$\varepsilon_{(n)}$	overall magnification error of n consecutive fast rotations.
φ	angle, azimuth in the hemisphere.
θ	angle, elevation in the hemisphere.
φ_i, θ_j	hemisphere coordinates of a ray.
κ_{φ}, κ_{θ}	angle exponent.
λ	ray parameter.

A_i	area of a patch.
dA_i	differential area on a patch.
F, $F_{i,j}$	form-factor.
HID	occlusion parameter.
N_{φ}, N_{θ}	number of rays along the φ, θ axes in the ray index space.
N_r	number of rays.
P_i	vertex of a patch.
T_i	triangle.
$T_{L \to G}$	geometry transformation, local to global coordinate system.
$T_{G \to L}$	geometry transformation, global to local coordinate system.
X	intersection point.

f, f_i	delta form-factor of a ray
$q_{(n)}$	overall accuracy of n consecutive fast rotations.
r	ray
$t_\Delta, u_\Delta, v_\Delta$	patch coordinates.
$u_\square, v_\square$	patch coordinates.
$\mathbf{R}_{xy}()$	rotation matrix (azimuth).
$\mathbf{R}_{zx}()$	rotation matrix (elevation).
$\mathbf{R}_{L\to G}, \mathbf{R}_{G\to L}$	rotation matrices for geometry transformations.
$\mathbf{e}_x, \mathbf{e}_y, \mathbf{e}_z$	local coordinate system basis vectors.
$\mathbf{n}$	normal of the source patch at the sample point.
$\mathbf{n}_{i,j}$	bounding plane normal.
$\mathbf{p}, \mathbf{p}_i$	vector to a patch vertex.
$\mathbf{r}, \mathbf{r}_{i,j}$	ray direction vector.
$\mathbf{s}, \mathbf{s}_i$	sample point on the source patch.
$\mathbf{x}$	vector to the intersection point. *also*
$\mathbf{x}$	vector to a point on the patch, or to a point on the ray.
$\mathcal{P}$	set of patches in the environment.
$\mathcal{R}$	ray frustum.
$\mathcal{S}$	set of sample points.

Chapter 5

Λ	matrix of Eigenvalues.
λ	Eigenvalue.
γ_κ	limit angle.
$\tilde{\gamma}_\kappa$	approximate limit angle.
E	off-diagonal energy
d	reduction factor.
g_κ	limit tangent.
$\tilde{g}_\kappa$	approximate limit tangent

$\mathbf{M}$	iteration matrix.
$\mathbf{Q}$	orthogonal matrix.

$\mathcal{A}$	set of fast rotation angles.
$\mathcal{F}$	set of fast rotations.

Abbreviations

Cordic	Coordinate Rotation Digital Computer
SBB	Spherical Bounding Box
ISBB	Index Space Bounding Box
ICU	Intersection computation unit
GTU	Geometry Transformation Unit
CTU	Cell Traversal Unit
DSP	Digital Signal Processor
FIFO	First In, First Out memory
QR, QRD	QR Decomposition
iQR	Inverse QR Decomposition
EVD	Eigenvalue Decomposition
SVD	Singular Value Decomposition
RLS	Recursive Least Squares
MVDR	Minimum Variance Distortionless Response
DCT	Discrete Cosine Transform
DST	Discrete Sine Transform
LOT	Lapped Orthogonal Transform
MLT, MLOT	Modified Lapped Orthogonal Transform
TFE	TransForm Engine
PTFE	Parallel TransForm Engine
CMOS	Complementary Metal Oxide Semiconductor
LSI	Large Scale Integration
VLSI	Very Large Scale Integration
ASIC	Application Specific Integrated Circuit
PGA	Pin Grid Array
FLOPS	FLoating-point Operations Per Second
lsb	least significant bit.
msb	most significant bit.

SUMMARY

High performance numerical computations are required in quite many applications in the field of digital signal processing (DSP). We use the term "high performance" here in a dual context: that of very high speed, high throughput and low latency of the computations, and/or that of very high accuracy and large dynamic range of the data. For example, current and planned future applications exist in processing of radar signals or radio astronomy signals, that require a computational power in the order of Giga FLOPS (10^9 Floating-Point Operations Per Second) through to Tera (10^{12}) FLOPS and even on to Peta (10^{15}) FLOPS. It is therefore essential that computer arithmetic techniques are developed that enable the efficient implementation of high performance numerical processors.

Most of the applications that we consider use matrix algebra algorithms for the main core of their signal processing. Examples of such algorithms are: QR decomposition (QRD), Singular Value Decomposition (SVD), Eigenvalue Decomposition (EVD), Minimum Variance Distortionless Response (MVDR), Recursive Least Squares (RLS). The Cordic (COordinate Rotation DIgital Computer) algorithm, which performs a 2×2 rotation of a vector, plays an important role in the efficient and robust realization of these matrix algebra algorithms.

Implementations of the Cordic algorithm exist, which employ a combination of a pipelined architecture, the use of redundant arithmetic, a special logic and circuit design, or even a high-speed VLSI process, such that an extreme high throughput is achieved. Throughputs have been reported in excess of 500 million rotations per second [1]. The classical Cordic algorithm, on which these high-throughput implementations are based, has a number of shortcomings. One of these shortcomings is that it inherently only works well on fixed-point data, and not on floating-point data. The applications that we consider, however, more and more often encounter a large dynamic range in signals and hence require floating-point operations for efficient implementation. Another shortcoming is that the cost of realizing a rotation is still relatively high, compared to, say, realizing an addition or multiplication.

[1]Not even considering the computation of the necessary sine and cosine of the angle, a rotation is already the equivalent of 6 floating-point operations for the matrix multiplication of the rotation matrix with the vector.

This thesis is made up of two parts. The first part is computer arithmetic oriented and the second part is more application oriented.

In the first part, we focus on what we see as the two major shortcomings of the Cordic, namely the problem of the limited accuracy and low dynamic range due to fixed point operation, and the problem of a relatively high cost of implementation. To overcome the former shortcoming, we propose a full floating-point Cordic algorithm, *i.e.* one that employs a floating-point representation for the vector components and for the angle, that allows an efficient implementation for both a sequential architecture, as well as for a parallel, pipelined architecture. The latter ensures a high throughput of computations. A full floating-point Cordic algorithm was presented before by Walther, and indeed has been used successfully in early desk and hand-held calculators, all the way through to INTEL's 80x87 series of numerical co-processors. However, this algorithm does *not* have an efficient implementation for a parallel architecture, so no high throughput is guaranteed.

To overcome the second shortcoming, we propose a new, Cordic related technique that we have called "Fast rotation". This is a technique to perform rotations at an extremely low cost in implementation. Even though fast rotations only exist for certain "nice" angles only, the technique has proven to be powerful enough to be applied with success in a large range of signal processing applications.

So much, in fact, that in the second part of the thesis, we focus on a number of applications in which fast rotations have been applied with success.

The first application which we consider, from the field of computer graphics, is the photo-realistic rendering of artificial scenes. This application, in fact, has led to the development of fast rotations: techniques to rotate over certain fixed angles at a very low cost. The use of fast rotations in this application has led to the feasibility and the actual implementation of the Intersection Computation Unit (ICU) ASIC, a dedicated VLSI chip that computes intersection points with a high accuracy, and at a high throughput. We show that a reduction in the cost of computation of a factor of around 20 is gained over an implementation with multipliers, and a factor of around 15 over an implementation with Cordic.

The second application is the realization of orthogonal matrix algebra algorithms, in this case Eigenvalue decomposition (EVD), by means of fast rotations. We consider the problem of on-line computation of the EVD, which implies that both the calculation of the angles of rotation and their application must be computed at low cost and high-speed. We show that both operations can indeed be done using fast rotations. A reduction of a factor of 7 to 8 is gained in the cost of computation over the use of the classical Cordic technique for EVD.

The third and last application is the realization of an orthogonal filterbank, used in transform coding for high quality compression of medical images, by means of fast rotations. The difference with the previous application is that the analysis phase of how to realize the filterbank with fast rotations is performed off-line. This allows

a greater freedom in the methods of realization, and we can use a greater computational effort in this phase to search for a realization with the lowest achievable cost. Compared to a direct implementation of the filterbank with multipliers, we achieve a reduction in the cost of computation of roughly an order of magnitude. However, other efficient techniques exist, which are also applicable to our filterbank, such as the fast implementation of the Discrete Cosine Transform (DCT). Our technique is comparable in cost, but has other desirable properties that make it more attractive to use. We present a prototype ASIC, equipped with a number of parallel operating pipelined fast rotation units, which is capable of implementing such filterbanks at low cost and high flexibility.

SAMENVATTING

Veel toepassingen in het gebied van de digitale signaalverwerking vereisen het gebruik van high performance numerieke berekeningen. Wij gebruiken de term "high performance" (hoge prestatie) in een dubbele context: die van een hoge berekeningssnelheid en een hoge verwerkingscapaciteit van data, en die van een hoge berekeningsnauwkeurigheid en een groot dynamisch bereik van de data. Bijvoorbeeld, er bestaan al (plannen voor) toepassingen voor het verwerken van radarsignalen en radio-astronomie signalen die een rekenkracht vereisen die in het gebied ligt van GigaFLOPS (10^9 FLOPS = zwevende komma bewerkingen per seconde) tot TeraFLOPS (10^{12}), en zelfs door tot PetaFLOPS (10^{15}). Het is daarom van essentieel belang dat er technieken in de computer aritmetiek worden ontwikkeld die het verwezenlijken van efficiënte high performance numerieke processoren bewerkstelligen.

Een groot deel van de toepassingen die wij in beschouwing nemen, berusten op matrix algebraïsche algoritmen voor de kern van hun berekenigen. Voorbeelden hiervan zijn de QR ontbinding (QRD), de singuliere waarde ontbinding (SVD), de Eigenwaarde ontbinding (EVD), Minimum Variance Distortionless Response (MVDR) en de recursieve kleinste kwadraten methode (RLS). Het Cordic (COordinate Rotation DIgital Computer) algoritme, welk een 2×2 rotatie uitvoert van een vector over een hoek, speelt een belangrijke rol in de efficiënte en robuuste uitvoering van deze matrix algebraïsche algoritmen.

Er bestaan realisaties van het Cordic algoritme, die gebruik maken van een combinatie van een pipeline architectuur, van redundante rekentechnieken, van een specifiek logisch of circuit ontwerp, of zelfs van een hoge-snelheid VLSI proces, om een extreem hoge verwerkingscapaciteit te bereiken. In de literatuur zijn meldingen gemaakt van een verwerkingscapaciteit die de 500 miljoen rotaties per seconde[2] overstijgt. Het conventionele Cordic algoritme, welk ten grondslag ligt aan deze realisaties, heeft echter een aantal tekortkomingen. Een dezer tekortkomingen is dat het algoritme inherent alleen geschikt is voor vaste komma berekeningen, en niet geschikt is voor zwevende komma berekeningen. Bij het soort

[2]Afgezien van de complexe berekening voor de sinus en cosinus van de hoek, is een rotatie vergelijkbaar met tenminste 6 zwevende komma berekeningen.

van toepassingen die wij beschouwen, worden wij in meerdere mate geconfronteerd met een groot dynamisch bereik van de data en van de signalen, zodat een zwevende komma representatie en bewerkingen een vereiste zijn.

Een andere tekortkoming is dat de rekenkosten om een rotatie uit te voeren, relatief hoog uitvallen vergeleken met die voor bijvoorbeeld vermenigvuldigen of optellen.

Dit proefschrift bestaat uit twee delen. Het eerste deel is gericht op de computer aritmetiek (rekenkunde), terwijl het tweede deel meer is gericht op de toepassingen.

In het eerste deel concentreren wij ons op wat wij zien als de twee grootste tekortkomingen van het conventionele Cordic algoritme. Deze zijn de beperkte nauwkeurigheid en laag dynamisch bereik door de inherente vaste komma bewerkingen, en de relatief hoge kosten voor het uitvoeren van een rotatie. Om de eerste tekortkoming teniet te doen, stellen wij een volledig zwevende komma Cordic algoritme (een die een zwevende komma representatie voor zowel de vector als voor de hoek gebruikt) voor, die een verwezenlijking toestaat op zowel een sequentiële als op een parallelle, pipeline architectuur. Deze laatste garandeert een hoge verwerkingscapaciteit. Een volledig zwevende komma Cordic algoritme was al eerder gepresenteerd door Walther, en dit algoritme is met succes gebruikt vanaf de eerste rekenmachines tot aan de gehele lijn van de INTEL 80x87 numerieke co-processoren. Echter, dit algoritme staat geen efficiënte verwezenlijking toe op een parallelle architectuur, welk nodig is om een hoge verwerkingscapaciteit te garanderen.

Om de tweede tekortkoming teniet te doen, stellen wij een nieuwe, aan Cordic gerelateerde techniek, voor die wij “fast rotations” (snelle rotaties) hebben genoemd. Dit is een rekenkundige techniek om met extreem lage kosten een rotatie uit te voeren. Ook al bestaan snelle rotaties alleen voor bepaalde “toepasselijke” hoeken, heeft de techniek zich al met succes bewezen in een breed scala van signaalverwerking toepassingen.

In zoverre zelfs, dat wij het tweede deel van het proefschrift wijden aan een aantal toepassingen waarin snelle rotaties met succes zijn toegepast.

De eerste van de toepassingen komt uit het gebied van de computer graphics, en betreft het foto-realistisch weergeven van kunstmatige scènes. Het is juist deze applicatie die de drijfveer is geweest voor de ontwikkeling van de snelle rotatie techniek. Het gebruik van snelle rotaties binnen deze toepassing heeft geleid tot de haalbaarheid en de verwezenlijking van de Intersection Computation Unit (snijpunt berekenings eenheid, afgekort ICU), een geïntegreerd circuit dat toegespitst is op het met een hoge nauwkeurigheid en hoge verwerkingscapaciteit berekenen van snijpunten. We laten zien dat een vermindering in de berekeningskosten met een factor van ongeveer 20 is gehaald, ten opzichte van een uitvoering met enkel vermenigvuldigers, of met een factor van ongeveer 15 ten opzichte van een uitvoering met Cordic.

De tweede toepassing is die van het verwezenlijken van matrix algebraïsche algoritmen, in dit specifieke geval een Eigenwaarde ontbinding (EVD), met behulp van snelle rotaties. Wij beschouwen het geval waarbij zowel het uitrekenen van de hoek alswel het uitvoeren van de rotatie “on-line” (op de plek) gebeuren. Dit houdt in dat beide berekeningen lage kosten met zich mee moeten brengen. Wij laten zien dit inderdaad kan, en dat beide berekeningen met uitsluitend snelle rotaties bewerkstelligd kunnen worden. Een vermindering van de berekeningskosten met een factor 7 tot 8 is gehaald voor het EVD probleem, ten opzichte van een uitvoering met Cordic.

De derde, en tevens laatste toepassing is het verwezenlijken van een orthogonale filterbank, die wordt gebruikt voor de hoge kwaliteit compressie van medische beelden, met behulp van snelle rotaties. Het verschil met de vorige toepassing is dat de analyse fase, van hoe de filterbank met een netwerk van snelle rotaties wordt gerealiseerd, nu “off-line” (niet op de plek) plaatsvindt. Dit geeft ons een veel grotere vrijheid in aanpak, en geeft tevens de mogenlijkheid een veel grotere rekenkracht te benutten om een realisatie te vinden die de laagst haalbare rekenkosten met zich meebrengt. Vergeleken met een naïeve uitvoering met behulp van vermenigvuldigers, halen wij een orde verschil in de vermindering van rekenkosten. Er bestaan echter ook andere efficiënte technieken, die gebruik maken van de ontbinding van de Discrete Cosinus Transformatie (DCT), en die ook op onze filterbank van toepassing zijn. Deze zijn van vergelijkbare kosten als onze techniek, ware het niet dat de realisatie met snelle rotaties andere gewenste eigenschappen heeft die het voordeliger maken. Wij tonen tevens een prototype geïntegreerd circuit die uitgerust is met een aantal, in parallel werkende en ge-pipeline’de snelle rotatie eenheden, waarmee het mogelijk is om orthogonale filterbanken te realiseren met lage rekenkosten, en een hoge flexibiliteit.

ACKNOWLEDGEMENTS

The research presented in this dissertation has been supported by the Dutch Technology Foundation STW (Stichting voor de Technische Wetenschappen) under contracts DEL99.1982 and DEL00.2331. The work on the Radiosity Engine was also partly funded by the ESPRIT program, under project BRA 3280 (NANA).

I would hereby like to thank all those, past and present, at the Circuit and Systems (CaS) group (former Network Theory group), faculty of Electrical Engineering, Delft University of Technology, for providing a pleasant and stimulating environment during my stay there.

Of all these, in the first place, I would like to thank my supervisor and co-promotor Ed Deprettere, for giving me the opportunities and especially the freedom to perform this research. Also for a never-ending source of challenging problems, all in the field of high-performance computing, to work on. His presence and wholehearted involvement in all the projects were inspiring.

I especially would like to thank my promotor, Prof. Patrick Dewilde, for the opportunity to carry out Ph.D. research at the Circuits and Systems group. Also for the careful proof-reading of the draft version of this thesis.

For their cooperation on the CORDIC project (DEL99.1982), I would like to thank Mark Bloemendaal and Remco de Zwart.

For their cooperation on the Graphics project (DEL00.2331), I would like to thank Gerrit Boersma, Ji-Chun Bu and Henk Hakkesteegt (all at TNO-TPD, Delft), Remco de Zwart and Patrick Vervoorn (formerly both at CaS). Also for their help and cooperation on the VLSI design of the Graphics chip-set I would like to express my thanks to André van der Avoird, Luc Vidal and Gerald Touzé. Of all members, I am most indebted to Li-Sheng Shen, on leave from ITRI, for a very close and good cooperation on the system design of the Radiosity Engine. Also for many valuable discussions, his patience to enrich me about Taiwanese culture and to extend my knowledge on Chinese characters and ideograms.

I would like to thank Richard Heusdens (then at Philips Research Labs, Eindhoven), for involving me on the Medical Image Compression project, and expressing the faith that an algorithm and architecture based on fast rotations might provide a cost-effective alternative to known solutions. Also to Monica Monari for her contributions on optimization of the algorithm, and to Zhiqiang Zeng for his work on the PTFE chip.

I would like to thank all members of the Jacobium project, Edwin Rijpkema, Jun Ma, Xu Yijun, Simone Fantini, Alco Looye and Daniël van Loenen, for their interest, commitment, and the incredible amount of ground that has been covered by them towards the specification and the design space exploration of a "Jacobium" processor. You have made the last half year at university a very memorable experience for me. Connected to this, I would like to thank those that have stood at the cradle of the Jacobium project, in particular Hylke van Dijk. Furthermore thanks to Marc Moonen and Filiep Vanpoucke (both at KU Leuven), and Alle-Jan van der Veen for many interesting discussions on this topic.

For helping me with testing the variety of the VLSI chips from all of the projects, I would like to thank Kees Schot and Jan Koopmans (both at DIMES).

For the shared interest in applications of fast rotations, and close cooperation in the initial work on them, I would like to thank Jürgen Götze (then at TÜ München).

Furthermore, I would like to thank all those in the computer arithmetic community and the digital signal processing community, for a shared interest, for inspiration, and for providing a sounding board for this research. In particular: Miloš Ercegovac, Tomás Lang, Jeong-A Lee and Keshab Parhi.

For providing me with a wealth of (long lost) information about the earliest work on Cordic and its applications, I would like to thank Jack Volder and Bernhard Kuchta (then both at Convair / General Dynamics).

I would especially like to thank both Corrie Boers-Boonekamp and Marion de Vlieger for their involvement with, and professional handling of, all other tasks connected to this thesis. Likewise, I would like to thank Jean-Pierre Veen of STW, for showing his personal interest in the research and the projects, and expressing his faith in them by funding them.

Last but, in the least, not least, I would like to express my greatest gratitude to my wife, Ewa, for her love, patience, and help.

ABOUT THE AUTHOR

Gerben Johan Hekstra was born on December 9th, 1965 in Rotterdam, the Netherlands. In June 1984, he received the Atheneum-β diploma at the S.G. "De Krimpenerwaard", Krimpen a/d IJssel, the Netherlands. Prior to this he followed part of his secondary school at Oban High School, Oban, Argyll, Scotland, until June 1981. In August 1984 he started with his studies in Informatica at the Delft University of Technology, Delft, the Netherlands. Towards the end of his studies, he started his specialization at the Network Theory section, faculty of Electrical Engineering, Delft University of Technology. His Master's thesis concerned the efficient implementation of parallel multipliers for VLSI (Very Large Scale Integration). In December of 1990, he received the Informatica Ingenieur diploma from the Delft University of Technology. From then on, in January 1991, he started on his Ph.D. studies, also at the Network Theory section. The work spanned several projects, of which the main ones were the re-design of a high-performance floating-point pipelined Cordic, and the specification, design and implementation of a chipset for a high-performance parallel system for photo-realistic rendering, known as the "Radiosity Engine". The essence of all the projects was the efficient VLSI implementation of complex arithmetic functions, and the methodology, design, implementation and testing of high-performance parallel systems.

In July 1997, he joined the TV Systems group of the Philips Research Labs (Nat. Lab.) in Eindhoven. His current work concerns video applications such as "Natural Motion", their efficient implementation on a TriMedia processor, and the benchmarking and design space exploration of future TriMedia processors.