Implementation of the CORDIC Algorithm in a Digital Down-Converter

Chris K Cockrum
Email: ckc@cockrum.net

Fall 2008

Abstract

This paper shows that the CORDIC (**CO**ordinate **R**otation by **DI**gital **C**omputer) algorithm [6] gives significant efficiency gains over a Taylor approximation for calculating the sine and cosine functions to a given precision for many applications implemented in hardware or a microcontroller. In the case where the processor has no dedicated multiplier or trigonometric algorithm hardware, it is shown that the CORDIC is over 14 times more efficient when calculating the sine and cosine functions to 10 decimal digits of precision. This implementation is tailored towards use by Amateur Radio enthusiasts in the digital down-converter (DDC) of software defined radio (SDR) applications on dedicated hardware.

Keywords: CORDIC, SDR, digital down-converter, DDC, software defined radio, Amateur Radio.

1 Introduction

The need for a more efficient implementation of a CORDIC (**CO**ordinate **R**otation by **DI**gital **C**omputer) algorithm [6] arises from the desire to perform parallel demodulation of multiple narrowband signals from a single complex baseband input. Inexpensive methods of implementing high-performance radio-frequency (RF) receivers and analog to digital conversion have moved within the reach of experimenters and Amateur Radio enthusiasts which has spawned interest in software defined radio (SDR)[11, 12, 13, 14].

Many SDR receivers capture a much greater bandwidth than is required in order to perform tuning operations in software. For example, PSK31 modulation [5] uses 31.25 Hz of bandwidth while the Softrock40 lite [3] receiver combined with a Creative Labs SB1090 provides 96 KHz of bandwidth. This both allows for a more granular analog tuning step size and allows for hundreds (theoretically thousands but this is reduced by practical considerations) of signals to be received simultaneously. The difficulty lies in the limited processing power of the hardware that is performing the SDR functionality which in most cases is a microcontroller or field programmable gate array (FPGA).

2 System Description

2.1 System Overview

The receiver system used for this paper is the Softrock 40 lite with a Creative Labs SB1090 USB audio interface. This provides reception of a 96 KHz section of the 40 meter Amateur Radio band from 7008 KHz to 7104 KHz and provides 24 bits of (theoretical) resolution.

The signal is received from Gap Titan DX antenna [2] into the Softrock 40 lite where it is down-converted to a complex baseband signal, bandpass filtered, and amplified. The signal is then routed to the USB audio interface where it is digitized and sent to the computer as shown in Figure 1. The signal is captured in the PC with 24 bits of resolution at a 96 KHz sample rate for non-realtime processing.

Since this system has at best 24 bit accuracy, 10 decimal digits should be sufficient for this application since $2^{-24} = 5.96 \times 10^{-8} \gg 10^{-10}$.

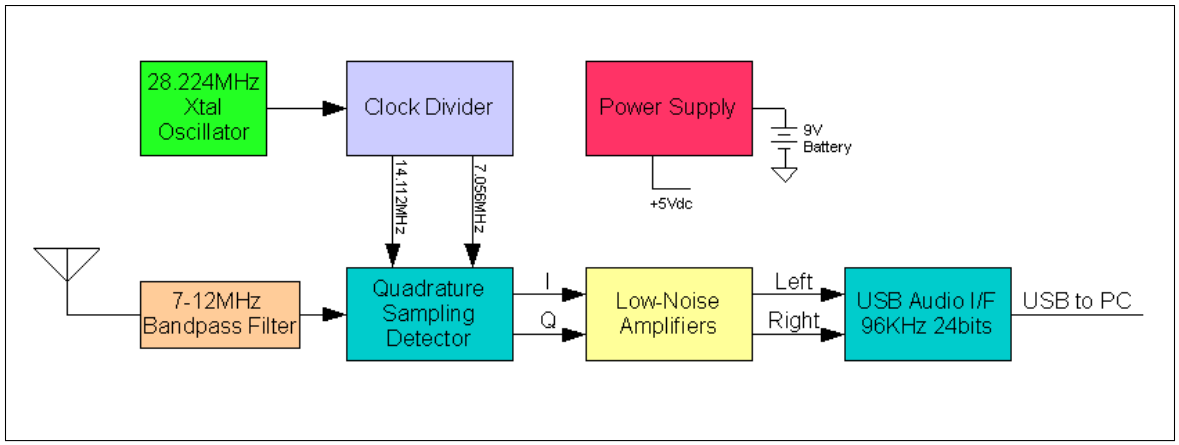

Figure 1: Block Diagram of Receiver System

2.2 Hardware

The Softrock 40 lite hardware for this project was constructed from a kit purchased from Tony Parks as shown in Figure 2. The design for this hardware is a collaborative effort in the Amateur Radio community with much of the work being done by the Softrock-40 interest group [3].

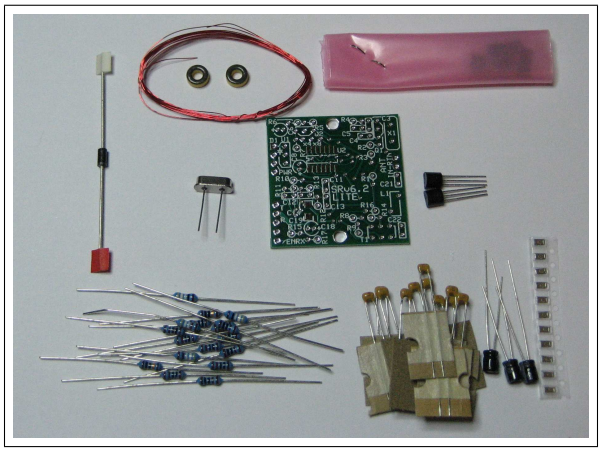

Figure 2: Softrock 40 lite parts

The assembled Softrock 40 lite along with its associated components were mounted in an radio-frequency (RF) shielded enclosure (an Altoids tin) as shown in Figure 3. The unit was then tested for functionality and performed as expected.

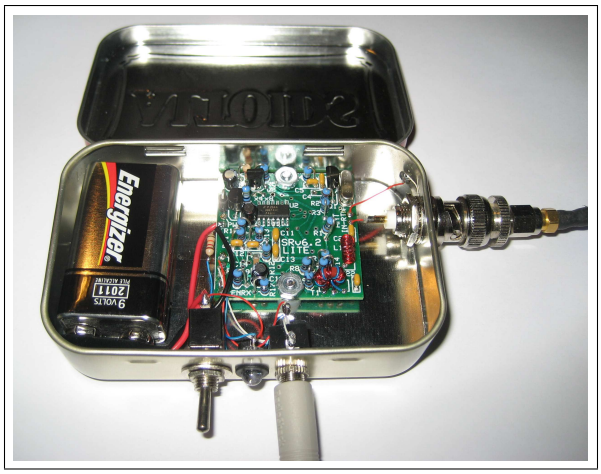

Figure 3: Softrock 40 lite assembled

2.3 Digital down-converter

The function of the digital down-converter software is to tune the receiver to a specific frequency. The input signal is 7056 KHz ± 48 KHz and the signal of interest may be centered anywhere within this range. The frequency shift is achieved by multiplying by $e^{i\omega t}$ were ω is the frequency shift in radians and t is time.

Let $x(t)$ be the input signal with $t = 0, 1, ...N$ samples at the sample rate which in this case is 96 KHz so each sample is $\frac{1}{96{,}000}$ seconds. For example, if we want to shift the frequency up by 9600 Hz then we need to multiply by $e^{i2\pi 9600t}$. This gives the output signal as:

$$y(t) = x(t)e^{i2\pi 9600t} = x(t)(\cos(2\pi 9600t) + i\sin(2\pi 9600t)) \tag{1}$$

In this example, a signal of interest at -9600 Hz at the input is now shifted to be centered at 0 Hz. At this point the signal would then be low-pass filtered and decimated leaving the signal of interest intact at a much lower sample rate that can be demodulated more efficiently.

3 Algorithm Description and Implementation

The CORDIC algorithm was originally described by Jack Volder in 1959 while he worked for Convair (a division of General Dynamics). The analog computer used on aircraft of the day was inadequate for the B-58 Hustler aircraft which was the first supersonic bomber used by the US Air Force. The solution was to replace analog computer by a digital computer that used the CORDIC algorithms to quickly perform accurate trigonometric calculations for navigation [7].

Then in 1972, D.S. Cochran at Hewlett Packard used the algorithm in the HP-35 - the world's first pocket calculator [4] - shown in Figure 4. Without this algorithm, it would likely not have been possible for the HP-35 to perform trigonometric functions.

Figure 4: HP-35 Pocket Calculator [10]

The algorithm works by starting with a point in the plane

$$v = (1, 0) \tag{2}$$

and multiplying it by a rotation matrix (rotates counterclockwise by θ)

$$R_\theta = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \tag{3}$$

Now we multiply the point by the rotation matrix to get

$$\hat{v} = v \cdot R_\theta = (1, 0) \cdot \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} = (\cos(\theta), \sin(\theta)) \tag{4}$$

So the result $\hat{v}$ directly gives the cosine and sine values at θ
Now suppose that the angle θ is split up into multiple smaller rotations such that

$$\theta = \theta_1 + \theta_2 + \cdots + \theta_{n-1} + \theta_n \tag{5}$$

We now have that

$$\hat{v} = v \cdot R_\theta = ((((v \cdot R_{\theta_0}) \cdot R_{\theta_1}) \cdots R_{\theta_{n-1}}) \cdot R_{\theta_n}) \tag{6}$$

and since the rotations are additive we can split this up into any number of iterations as long as the sum of the angles is equal to θ.

Now we use the trigonometric identities

$$\cos(\alpha) = \frac{1}{\sqrt{1 + \tan^2(\alpha)}} \tag{7}$$

and

$$\sin(\alpha) = \frac{\tan(\alpha)}{\sqrt{1+\tan^2(\alpha)}} \tag{8}$$

to give

$$R_\alpha = \begin{pmatrix} \frac{1}{\sqrt{1+\tan^2(\alpha)}} & \frac{\tan(\alpha)}{\sqrt{1+\tan^2(\alpha)}} \\ -\frac{\tan(\alpha)}{\sqrt{1+\tan^2(\alpha)}} & \frac{1}{\sqrt{1+\tan^2(\alpha)}} \end{pmatrix} = \frac{1}{\sqrt{1+\tan^2(\alpha)}} \begin{pmatrix} 1 & \tan(\alpha) \\ -\tan(\alpha) & 1 \end{pmatrix} \tag{9}$$

If the values of $\tan(\alpha)$ are constrained to be powers of two, then the multiplications by the rotation matrix are simplified to additions, bit shifts, and a multiplication by a scale factor.

$$\tan(\alpha) = 2^{-n} \text{ for all } n = 1, 2, ...N \tag{10}$$

The scale factor is

$$K_i = \frac{1}{\sqrt{1+2^{-2i}}} \tag{11}$$

By constraining $\tan(\alpha)$ to be powers of two, the angles are then given by

$$\theta_n = \arctan(2^{-n}) \text{ for all } n = 1, 2, ...N \tag{12}$$

which are accumulated during each iteration and may be precomputed.

During each iteration, the accumulated θ value is compared against the desired value to determine the direction of the next rotation. An illustration of this is shown in Figure 5.

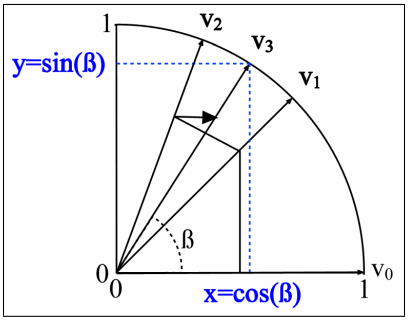

Figure 5: Illustration of Coordinate Rotation [9]

Since the value of K_i depends only on the number of iterations, it can be precomputed and applied at the end of the calculation.

$$K(n) = \prod_{n=0}^{N-1} K_i = \prod_{n=0}^{N-1} \frac{1}{\sqrt{1+2^{-2i}}} \tag{13}$$

The algorithm operates as follows:

Pseudocode:
Set initial value of vector $v = (1, 0)$
Set initial value of the current angle, $\beta = 0$
Set number of iterations, $i = 1$

While $i <$ MaxIterations
    If $\theta > \beta$ (counterclockwise rotation)
        $\beta = \beta + \arctan(2^{-i})$
        $v = v \cdot R_{\arctan(2^{-i})}$
    Else (clockwise rotation)
        $\beta = \beta - \arctan(2^{-i})$
        $v = v \cdot R_{\arctan(2^{-i})}{}^{T}$
    End If
End While

$v = v \cdot K(n))$ (Apply scale factor)

The precision (round-off) error can be ignored in this case since we are seeking precision to 10 decimal digits and this is much larger than the machine epsilon.

$$10^{-10} \gg 2.22 \times 10^{-16} = \varepsilon_{machine} \tag{14}$$

The significant error in this method is determined by the number of iterations used. The approximated angle converges to within $\arctan(2^{-n+1})$ in n iterations [8]. This angular error is most detrimental where the sine and cosine functions have the greatest slope which is at 0 and $\frac{\pi}{2}$ respectively. The slope of these functions is 1 at these points and therefore the error is bounded by

$$\text{error} \leq |\arctan(2^{1-n})| \text{ after } n \text{ iterations} \tag{15}$$

To guarantee 10 digit accuracy, 35 iterations are required since

$$\text{error } \leq |\arctan(2^{1-n})| \leq 5.82 \times 10^{-11} \text{ after 35 iterations} \tag{16}$$

4 Results

4.1 Efficiency and accuracy using a CORDIC

Calculation of sine and cosine to 10 decimal digits of precision using the CORDIC algorithm with 35 iterations requires the following operations when optimally implemented:

Comparisons: 35
Bit shifts: 35
Additions: 105
Multiplications: 1

The implementation in Matlab is far from optimal because of the interpretive nature of Matlab. Although this isn't the most efficient implementation, the cordic.m Matlab function and test.m Matlab script shown in Appendix A and Appendix B is functionally implemented as described in this paper.

4.2 Efficiency and accuracy using a Taylor approximation

Calculation of the sine function using a Taylor series approximation to 10 decimal places requires that 16 terms be used as shown below.
Taylor series approximation of $\sin(x)$ between $x \in [0, 2\pi]$ using N terms gives

$$\sin(x) = \sum_{i=0}^{N} (-1)^i \frac{x^{2i+1}}{(2i+1)!} \tag{17}$$

The truncation error is bounded by the $N+1$ term evaluated at 2π which is

$$\frac{(2\pi)^{2N+3}}{(2N+3)!} \tag{18}$$

and bounds the error by

$$\frac{(2\pi)^{2(15)+3}}{(2(15)+3)!} \approx 2.52 \times 10^{-11} \text{ for } N = 15 \tag{19}$$

Precision error is neglected since $2.52 \times 10^{-11} \gg 2.22 \times 10^{-16} = \varepsilon_{machine}$
This gives

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots + \frac{x^{29}}{29!} - \frac{x^{31}}{31!} \tag{20}$$

The number of operations required for the calculation of $\sin(x)$ to 10 decimal places using a Taylor approximation is

Exponential: 15
Additions: 15
Divisions: 15

Calculation of the cosine function using a Taylor series approximation to 10 decimal places requires that 17 terms be used as shown below.
Taylor series approximation of $\cos(x)$ between $x \in [0, 2\pi]$ using N terms gives

$$\cos(x) = \sum_{i=0}^{N} (-1)^i \frac{x^{2i}}{(2i)!} \tag{21}$$

The truncation error is bounded by the $N+1$ term evaluated at 2π which is

$$\frac{(2\pi)^{2N+2}}{2N+2!} \tag{22}$$

and bounds the error by

$$\frac{(2\pi)^{2(16)+2}}{(2(16)+2)!} \approx 4.66 \times 10^{-12} \text{ for } N = 16 \tag{23}$$

Precision error is neglected since $4.66 \times 10^{-12} \gg 2.22 \times 10^{-16} \varepsilon_{machine}$
This gives

$$\cos(x) = x - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots + \frac{x^{30}}{30!} - \frac{x^{32}}{32!} \tag{24}$$

The number of operations required for the calculation of $\cos(x)$ to 10 decimal places using a Taylor approximation is

Exponential: 16
Additions: 16
Divisions: 16

Calculation cost of both $\sin(x)$ and $\cos(x)$ to 10 decimal places using a Taylor approximation is

Exponential: 31
Additions: 31
Divisions: 31

4.3 Comparison of the CORDIC algorithm to the Taylor approximation

Although the code shown in Appendix A and Appendix B cannot match the realtime speed of the Matlab built-in functions in this implementation, the accuracy and functionality of the CORDIC algorithm is as predicted. The actual error versus the Matlab built-in functions fall just within the predicted error bound as shown in Figure 6.

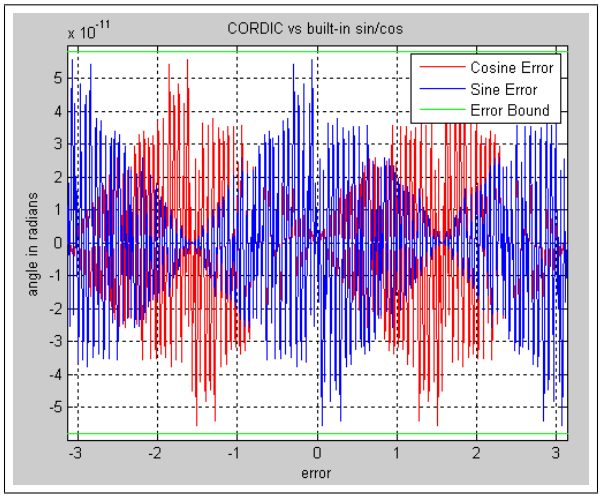

Figure 6: CORDIC error

This particular implementation doesn't demonstrate a speed increase due to the Matlab implementation but on many platforms the cost of a single multiply, divide, or fast exponentiation on a 32 bit word is over 32 times that of an addition, bit shift, or comparison. By taking this increased cost into account we arrive at the following efficiency comparison

Taylor Series Approximation of sin and cosine to 10 decimal digits
Exponential: 31 x 32 = 992 Cycles
Additions: 31 x 32 = 992 Cycles
Divisions: 31 x 32 = 992 Cycles
Total: 2,976 Cycles

Calculation of sine and cosine to 10 decimal digits of precision using the CORDIC algorithm

Comparisons: 35
Single Bit shifts: 35
Additions: 105
Multiplication: 32
Total: 207 Cycles

When optimally implemented in hardware, the complexity of a CORDIC rotator is equivalent to that of a single multiplier of the same word size[1].

4.4 Digital down-converter performance

When the CORDIC algorithm is used as part of the DDC system, its performance is nearly indistinguishable from that of Matlab's built-in sine and cosine functions since the CORDIC algorithm's accuracy is significantly greater that the resolution of the USB audio interface's analog-to-digital converters. Figure 7 shows a frequency spectrum plot of the 96KHz input from the USB audio interface. Figures 8 and 9 show the down-converted spectrum (shifted right by 30 KHz). Notice that the signal levels remain constant and there is no increase in the noise floor. A change in the signal levels or a rise in the noise floor at this point would indicate that the down-conversion process has introduced a noticeable error into the system.

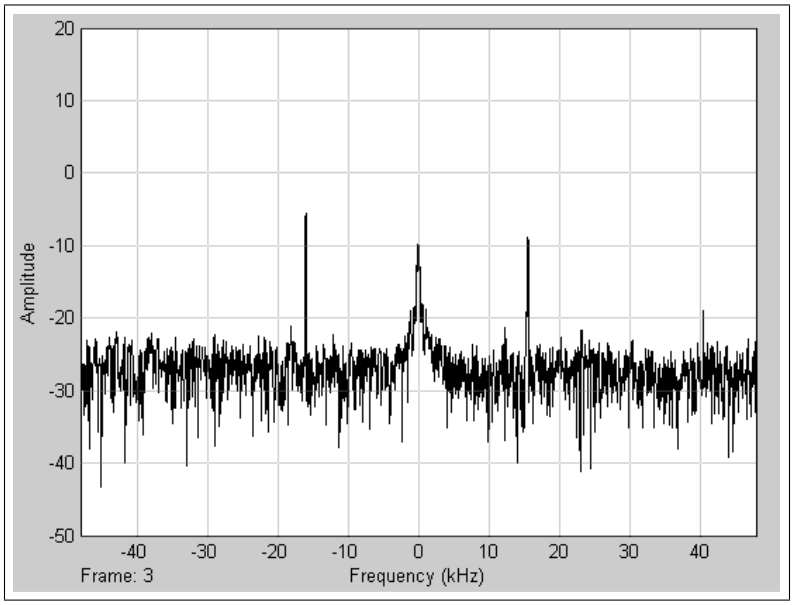

Figure 7: Input signal to DDC

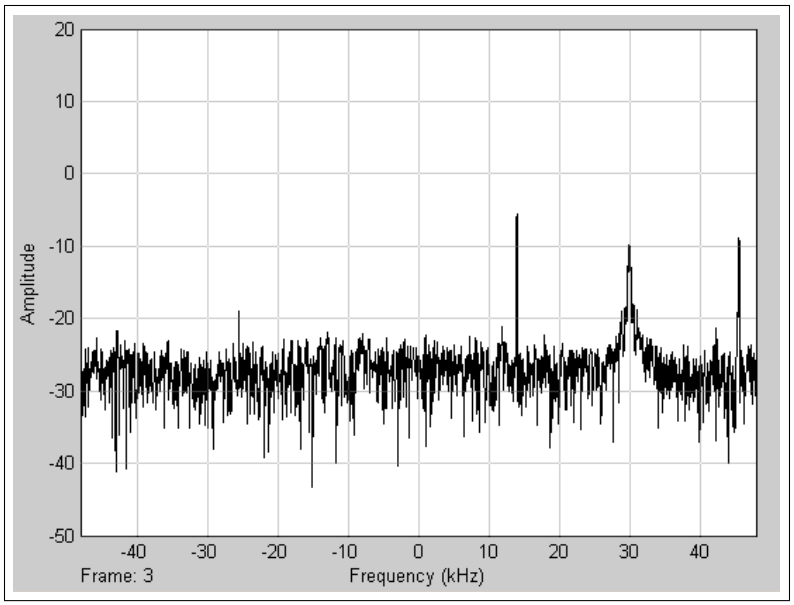

Figure 8: Frequency shifted output signal using CORDIC

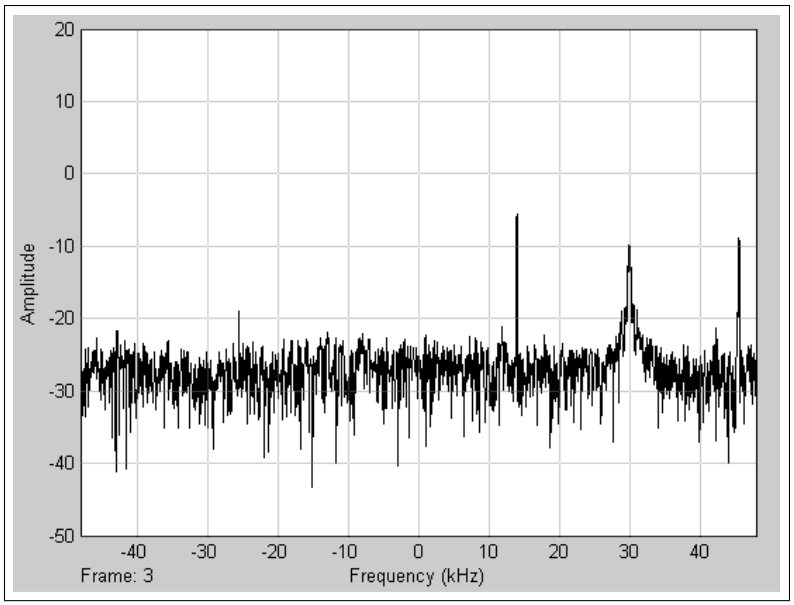

Figure 9: Frequency shifted output signal using Matlab's built-in sine and cosine functions

5 Conclusion

The CORDIC algorithm is a good compromise of accuracy versus speed for this and many other applications. For platforms with built-in multipliers or dedicated hardware for trigonometric functions, there is no benefit to using the CORDIC algorithm but on many platforms, such as many microcontrollers and field programmable gate arrays (FPGAs), the CORDIC algorithm is over 14 times more efficient.

6 Acknowledgements

This report was prepared as a final project for Math 620 Introduction to Numerical Analysis at University of Maryland, Baltimore County. Thanks to Professor Matthias Gobbert (gobbert@umbc.edu) for his support in this course and on this paper.

References

[1] R. Andraka. A Survey of CORDIC Algorithms for FPGA Based Computers. Andraka Consulting Group. *Inc. Copyright*, 1998.

[2] Multiple Authors. Gap titan antenna. http://www.gapantenna.com/titan.html. [Online; accessed 23-November-2008].

[3] Multiple Authors. Softrock-40 interest group. http://groups.yahoo.com/group/softrock40. [Online; accessed 23-November-2008].

[4] D.S. Cochran. Algorithms and Accuracy in the HP-35. *Hewlett-Packard Journal*, pages 10–11, 1972.

[5] S. Ford. PSK31: Has RTTYs replacement arrived. *QST (May, 1999)*, pages 41–44, 1999.

[6] J.E. Volder. The CORDIC trigonometric computing technique. *IRE Trans. Electron. Comput*, 8(3):330–334, 1959.

[7] J.E. Volder. The Birth of Cordic. *The Journal of VLSI Signal Processing*, 25(2):101–105, 2000.

[8] JS Walther. A unified algorithm for elementary functions. In *Spring Joint Computer Conf*, volume 38, pages 379–385, 1971.

[9] Wikipedia. Cordic. http://en.wikipedia.org/wiki/CORDIC. [Online; accessed 23-November-2008].

[10] Wikipedia. Hp-35. http://en.wikipedia.org/wiki/HP-35. [Online; accessed 23-November-2008].

[11] Gerald Youngblood. A Software-Defined Radio for the Masses, part 1. *QEX*, pages 13–21, Jul/Aug 2002.

[12] Gerald Youngblood. A Software-Defined Radio for the Masses, part 2. *QEX*, pages 10–18, Sep/Oct 2002.

[13] Gerald Youngblood. A Software-Defined Radio for the Masses, part 3. *QEX*, pages 27–36, Nov/Dec 2002.

[14] Gerald Youngblood. A Software-Defined Radio for the Masses, part 4. *QEX*, pages 20–31, Mar/Apr 2003.

7 Appendix A. Matlab Code for cordic.m

```
% Math 620 Final Project
% Chris K Cockrum
% December 15, 2008
% CORDIC Algorithm
% theta = angle in radians
% -2pi <= theta <= 2pi
% n = number of iterations (36 for 10 digit precision)
% v = [cos(theta) sin(theta)]
function v = cordic(theta, n)

    % Limit size of n
    if (n>39)
        n=39;
    end

    % Get the lenght of theta
    len=length(theta);

    % If input is an array
    if (len>1)
        % Iteratively calculate theta
        for i=1:len
            % Since the CORDIC algorithm works for -pi/2 to pi/2
            % If the point is in quadrant 2 or 3, we move it and
            % negate the result
            if (theta(i) > pi/2)
                t=theta(i)-pi;
                neg=-1;
            elseif (theta(i) < -pi/2)
                t=theta(i)+pi;
                neg=-1;
            else
                t=theta(i);
               neg=1;
            end

            v(i,:)=cordicf(t, n)*neg;
        end
    else
        % Since the CORDIC algorithm works for -pi/2 to pi/2
        % If the point is in quadrant 2 or 3, we move it and
        % negate the result
        if (theta > pi/2)
            t=theta-pi;
            neg=-1;
        elseif (theta < -pi/2)
            t=theta+pi;
            neg=-1;
        else
            t=theta;
            neg=1;
        end
```

```
        v = cordicf(t, n)*neg;
    end
end

function v=cordicf(theta, n);

% Generate table for atan in increments of negative powers of 2
%rot_ang=atan(2.^-([0:n-1]));

% Precomputed table for rot_ang
rot_ang = [ ...
  0.7853981633974828000,     0.4636476090008060900,     0.2449786631268641400, ...
  0.1243549945467614400,     0.0624188099959573500,     0.0312398334302682770, ...
  0.0156237286204768310,     0.0078123410601011111,     0.0039062301319669718, ...
  0.0019531225164788188,     0.0009765621895593194,     0.0004882812111948982, ...
  0.0002441406201493618,     0.0001220703118936702,     0.0000610351561742088, ...
  0.0000305175781155261,     0.0000152587890613158,     0.0000076293945311019, ...
  0.0000038146972656065,     0.0000019073486328102,     0.0000009536743164059, ...
  0.0000004768371582031,     0.0000002384185791016,     0.0000001192092895508, ...
  0.0000000596046447754,     0.0000000298023223877,     0.0000000149011611938, ...
  0.0000000074505805969,     0.0000000037252902985,     0.0000000018626451492, ...
  0.0000000009313225746,     0.0000000004656612873,     0.0000000002328306437, ...
  0.0000000001164153218,     0.0000000000582076609,     0.0000000000291038305, ...
  0.0000000000145519152,     0.0000000000072759576,     0.0000000000036379788, ...
  0.0000000000018189894,     0.0000000000009094947,     0.0000000000004547474, ...
];

% Function for K_i
%Ki= @(i) (1+2^(-2*i))^(-1/2);

% Generate table for K
%k(1)=1;
%k(2)=Ki(0);
%for i=3:n+1
%    k(i)=k(i-1)*Ki(i-2);
%end

% Precomputed table for K
k = [ ...
  1.0000000000000000000,     0.7071067811865474600,     0.6324555320336757700, ...
  0.6135719910778962800,     0.6088339125177523200,     0.6076482562561681400, ...
  0.6073517701412959300,     0.6072776440935260300,     0.6072591122988927300, ...
  0.6072544793325623800,     0.6072533210898751800,     0.6072530315291343500, ...
  0.6072529591389448400,     0.6072529410413971500,     0.6072529365170102900, ...
  0.6072529353859135200,     0.6072529351031393800,     0.6072529350324458200, ...
  0.6072529350147724000,     0.6072529350103540400,     0.6072529350092494800, ...
  0.6072529350089733700,     0.6072529350089043200,     0.6072529350088871000, ...
  0.6072529350088827800,     0.6072529350088816700,     0.6072529350088814400, ...
  0.6072529350088813300,     0.6072529350088813300,     0.6072529350088813300, ...
  0.6072529350088813300,     0.6072529350088813300,     0.6072529350088813300, ...
  0.6072529350088813300,     0.6072529350088813300,     0.6072529350088813300, ...
];
```

```
% Starting point x=1 y=0 and current angle=0
v=[1;0];
cur_angle=0;
iter=1;
power=1;

% While the angle isn't exact or max iterations have been reached
while (iter<=n)
    if (theta > cur_angle)
        % Choose Direction to rotate
        cur_angle=cur_angle+rot_ang(iter);

        % Rotate by this amount
        v=[1, -power;power, 1]*v;
    else
        cur_angle=cur_angle-rot_ang(iter);

        % Rotate by this amount
        v=[1, power;-power, 1]*v;
    end

    % Divide tan value by 2 (bit shift right)
    power=power/2;

    iter=iter+1;
end

% Multiply by scale factor
v=v*k(iter);

end
```

8 Appendix B. Matlab Code for test.m

```
% Math 620 Final Project
% Chris K Cockrum
% December 15, 2008
% Test function for CORDIC Algorithm

% Set number of iterations
N = 35;

% Set theta vector
theta=-pi:pi/256:pi;

% Call CORDIC algorithm
v=cordic(theta,N);

% Call Matlab built-in cos and sin
w=[cos(theta)' sin(theta)'];

% Plot error
figure(1);
plot(theta',v(:,1)-w(:,1),'r',theta',v(:,2)-w(:,2),'b',theta', ...
    atan(2^(-N+1)),'-g',theta',-atan(2^(-N+1)),'-g');
xlim([-pi pi]);%ylim([-pi pi]);
legend('Cosine Error','Sine Error','Error Bound');
axis on;
grid on;
title({'CORDIC vs built-in sin/cos'});
xlabel('error');
ylabel('angle in radians');
```