

Compact and Efficient Generation of Trigonometric Functions using a CORDIC algorithm

Samuel Ginsberg

Cape Town, South Africa

feathers_mail@yahoo.co.uk

Introduction

Often trigonometric functions are used in embedded applications. Examples of this include motion control, filtering and waveform synthesis. For waveforms with few output points per cycle (for example one output point per degree) a lookup table will often suffice, and indeed this method is optimal in that it offers a reasonable compromise between speed and the need to use the microcontroller's memory efficiently.

For waveforms with many output points per cycle (consider a waveform with 4096 points per cycle) the lookup table approach is often unfeasible because of the memory requirements. In the case of a waveform with 4096 output points per cycle the lookup table alone would occupy at least 4kB of Flash, and more often the table would take 8kB of memory. This precludes the use of such a lookup table on the lowest cost microcontrollers with their limited Flash memory.

It would appear from the above that where many points are required on a waveform it would be more practical to compute points on the waveform in real time when required and output them as they are computed. This raises the question of how to compute trigonometric functions efficiently. Various methods exist. These include Taylor Series; various curve fitting algorithms and the CORDIC (COordinate Rotation Digital Computer) algorithm. The CORDIC algorithm often offers the most elegant solution to the problem, and it is astounding in its simplicity of implementation, efficiency and elegance.

Mathematical Basis of the CORDIC algorithm

The aim of the Algorithm is to compute the Sine and Cosine of a given angle, which we will call Θ (Theta). Suppose that we have a point on a unit circle, which may be illustrated as follows:

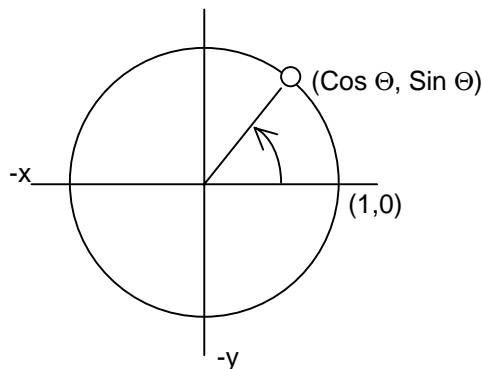


Figure 1: A point on the unit circle rotated by an angle Θ

The point on the unit circle that has a rotation of Θ has co-ordinates of $(\text{Cos } \Theta, \text{Sin } \Theta)$. This implies that if a point on the x-axis is rotated by an angle Θ then the Sine and Cosine of the angle of rotation may be read directly off the x and y axes.

The rotation may be achieved by rotating the point on the unit circle in a series of steps, which are smaller than Θ . In addition these steps may be either in an anti-clockwise direction (increase in Θ) or in a clockwise direction (decrease in Θ).

Example: Suppose we wish to achieve a total rotation of 25° . We may rotate our point 20° anti-clockwise, followed by 10° anti-clockwise, followed by 5° clockwise.

The co-ordinates of a point in a two dimensional space may be represented as a vector. If the co-ordinates of the point are (x,y) then the point may be equally well represented as (x,y)' where the inverted comma indicates a matrix transpose function. The rotation of a point in two dimensional space may be effected by multiplying the co-ordinates of that point by a rotation matrix. Thus:

$$\begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix} \dots\dots\dots(1)$$

Where (x', y') are the co-ordinates of (x,y) rotated by an angle of θ . This matrix operation may be expressed as follows:

$$x' = x \cos \theta - y \sin \theta \dots\dots\dots(2)$$

$$y' = y \cos \theta + x \sin \theta \dots\dots\dots(3)$$

For reasons that will become clear at a later stage these equations may be divided by $\cos \theta$ to yield:

$$(x'/\cos \theta) = x - y \tan \theta \dots\dots\dots(4)$$

$$(y'/\cos \theta) = y + x \tan \theta \dots\dots\dots(5) \quad (\text{Note that } (\sin \theta / \cos \theta) = \tan \theta.)$$

At this point we will present two examples to demonstrate the possible validity of the above mathematics.

Example 1:

Suppose we wish to rotate the point (1,0) by 180° . Intuition states that the resultant point will be at position (-1,0).

Substituting into the LHS of equation 4 we get $(-1/\cos 180^\circ) = -1/-1 = 1$

Substituting into the RHS of equation 4 we get $1-0 \tan 180^\circ = 1$ as expected.

Substituting into the LHS of equation 5 we get $(0/\cos 180^\circ) = 0$

Substituting into the RHS of equation 5 we get $0+1 \tan 180^\circ = 0$ as expected.

Example 2:

Suppose we wish to rotate the point (1,0) by $+45^\circ$.

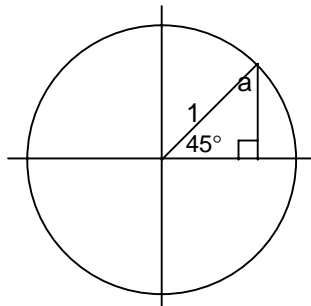


Figure 2: The point (1,0) rotated by 45°

Angle 'a' must also be 45° since the sum of the internal angles of a triangle must be 180° . This means that the triangle is isosceles and therefore the vertical side is the same length as the horizontal side. By Pythagoras's theorem the length of the vertical side and horizontal side must be $0.707 (1/\sqrt{2})$. This means that the co-ordinates of the rotated point are (0.707,0.707).

Substituting into the LHS of equation 4 we get $(0.707/\cos 45^\circ) = 1$

Substituting into the RHS of equation 4 we get $1-0 \tan 45^\circ = 1$ as expected.

Substituting into the LHS of equation 5 we get $(0.707/\cos 45^\circ) = 1$

Substituting into the RHS of equation 5 we get $0+1 \tan 45^\circ = 1$ as expected.

It is possible to break our rotation into many steps, each of decreasing size and have each step such that $\tan \theta$ is a power of 2. (Here θ refers to the rotation of each step). This would allow us to implement the

multiplication by $\tan \Theta$ as a very efficient bit shift operation. The first eight steps of the set of rotations are shown below to demonstrate the feasibility of this method.

Θ	$\tan \Theta$ in fractional form	$\tan \Theta$ decimal
45°	1	1
26.565°	$\frac{1}{2}$	0.5
14.036°	$\frac{1}{4}$	0.25
7.125°	$\frac{1}{8}$	0.125
3.576°	$\frac{1}{16}$	0.0625
1.789°	$\frac{1}{32}$	0.03125
0.895°	$\frac{1}{64}$	0.015625
0.4476°	$\frac{1}{128}$	0.0078125

Table 1: The first 8 rotational steps in the CORDIC algorithm

If the rotation steps are added up it may be seen that the maximum total rotation is almost 100° . Since the rotation may be in a clockwise or counter clockwise direction it is clear that these steps may be used to approximate any angle between (approximately) $+100^\circ$ and -100° . Further the decreasing size of the steps indicates that the approximation may be made arbitrarily accurate by increasing the number of steps of rotation. A small lookup table is used to store the angle Θ of each step. The size of this table increases with $\log_2 N$ where N is the number of steps of rotation.

Recall that we needed both $\tan \Theta$ and $\cos \Theta$ to effect the rotation of a point around the unit circle. Equations (4) and (5) are repeated as a reminder.

$$(x'/\cos \Theta) = x - y \tan \Theta \quad \dots\dots\dots(4)$$

$$(y'/\cos \Theta) = y + x \tan \Theta \quad \dots\dots\dots(5)$$

We have determined an efficient method of multiplying by $\tan \Theta$ and so all that remains is the issue of dividing by $\cos \Theta$.

If we assume that Θ is between -90° and $+90^\circ$ then the trigonometric identity $\cos(\Theta) = \cos(-\Theta)$. This implies that if our rotational steps are always the same angles it makes no difference if our rotation steps are clockwise or counter clockwise. This allows us to substitute a constant in place of the $\cos \Theta$ term. The constant depends only on the number of steps used to approximate the total angle.

If four steps are used to approximate the angle then the constant is calculated as follows:

$$\begin{aligned} \text{Constant} &= \cos(\text{Atan}(1/2^0)) * \cos(\text{Atan}(1/2^1)) * \cos(\text{Atan}(1/2^2)) * \cos(\text{Atan}(1/2^3)) \\ &= 0.6088 \end{aligned}$$

This same sequence may be used to calculate the constant for any number of steps.

As a final comment it should be noted that we may either rotate the point around the unit circle or we may rotate the reference axes. Rotating the reference axes allows easier comparison to determine the direction of the next rotation.

In summary

We need to calculate the Sine and Cosine of an angle Θ . We showed that rotating a point around the unit circle from an angle of zero to angle Θ would allow us to determine the Sine and Cosine from its co-ordinates. We demonstrated that the rotation might be made using equations (4) and (5). These equations require a term related to $\cos \Theta$ and another term related to $\tan \Theta$. We demonstrated that the total angle of rotation might be approximated by using a series of smaller rotations. Guidelines were given for the efficient calculation of the $\tan \Theta$ term. The $\cos \Theta$ term was shown to be constant, and therefore trivial to implement.

We are now ready to assemble the mathematics into an algorithm.

Pseudocode CORDIC algorithm

The algorithm is presented in its most basic form in a 'C' like pseudocode. We will assume a 12-step system. This will yield 12 bits of accuracy in the final answer. Note that the $\cos \theta$ constant for a 12 step algorithm is 0.60725. We also assume that the 12 values of $\text{Atan}(1/2^i)$ have been calculated before run time and stored along with the rest of the algorithm. If true floating-point operations are used then the shift operations must be modified to divide by 2 operations.

Initialisation:

```
Set A to the desired angle
Set Y=0
Set X=0.60725
```

//The initialisation specifies the total angle of rotation and sets the initial value of the point at (1,0) and multiplied by the constant 0.60725 which results from the Cosine terms in eqn (4) and (5).

Computation:

```
For (i=0;i<12;i++)
{dx=X shifted right by i places //effectively calculates X*tan θ for this step
 dy=Y shifted right by i places //effectively calculates Y*tan θ for this step
 da=Atan (1/2i) //From the small lookup table

if (A>=0) //decides if our next rotation must be clockwise or anti clockwise
{X=X-dy //compute X-Y*Tan θ as per eqn (4)
 Y=Y+dx //compute Y+X*Tan θ as per eqn (5)
 A=A-da //update the current angle
}
else //rotate in opposite direction
{X=X+dy
 Y=Y-dx
 A=A+da
}
}
```

The Sine of the desired angle is now present in the variable Y and the Cosine of the desired angle is in the variable X.

This algorithm as it stands requires the use of non integer numbers. This presents some inconvenience when translating the algorithm into assembler for implementation in a microcontroller. The algorithm may be modified to use only integers. The modified algorithm is given here. Bear in mind that we will be working with a 12 bit algorithm, and thus our output angle will range from -2048 to +2047. We will assume 16 bit calculations throughout.

Initialisation:

```
Set A to the desired angle*2048 //Note the change
Set Y=0
Set X=0.60725*2048 //Note the change
Setup the lookup table to contain 2048*Atan (1/2i)
```

Computation:

```
For (i=0;i<12;i++)
{dx=X shifted right by i places //effectively calculates X*tan θ for this step
 dy=Y shifted right by i places //effectively calculates Y*tan θ for this step
 da=lookup (1/2i) //From the small lookup table . Note the modification to the table

if (A>=0) //decides if our next rotation must be clockwise or anti clockwise
```

```

    {X=X-dy          //compute X-Y*Tan Θ as per eqn (4)
      Y=Y+dx        //compute Y+X*Tan Θ as per eqn (5)
      A=A-da        //update the current angle
    }
else //rotate in opposite direction
    {X=X+dy
      Y=Y-dx
      A=A+da
    }
}

```

The Sine of the desired angle is now present in the variable Y and the Cosine of the desired angle is in the variable X. These outputs are within the integer range -2048 to +2047.

Establishing a Program

- Examination of the above algorithm shows that several basic procedures are needed. These are:
- 16 bit shift: A routine to shift a 16 bit number right by 1 places is needed. This must be an arithmetic shift, as it must include sign extension.
- Lookup routine: A routine is needed to retrieve a 16-bit number from a 12 entry lookup table.
- Negation routine: A routine to negate a number represented as a 16 bit 2's compliment number.
- Addition routine: A routine to add two 16 bit numbers.

The code is given below. Each routine is explained within the routine. There is no main function, as users would typically insert their own function here. An example of a function that outputs a full sine wave to a 12 bit DAC is given below the main program listing.

This program was programmed into a MC68HC908JK1 microcontroller (1.5kB Flash, 128B RAM) which was connected to a 12 bit DAC. The waveform that was generated has been captured and is presented here.

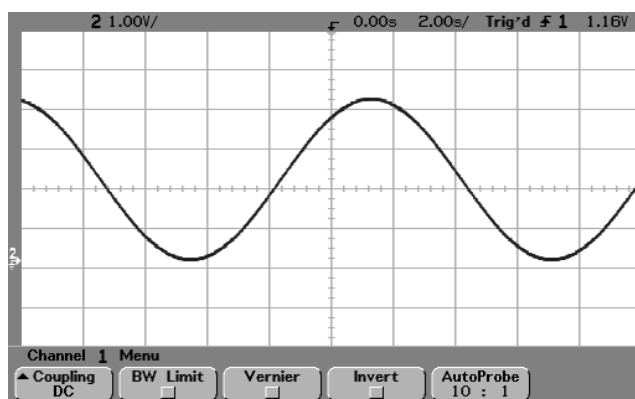


Figure 3: The waveform produced using a CORDIC algorithm. The screenshot includes oscilloscope setup parameters

```

----- HC08 code -----

;CORDIC demonstration program
;Written by Samuel Ginsberg
;January 2002

TOPRAM      EQU  $0080    ;Top of RAM area
TOPROM      EQU  $F600    ;Top of Flash area
TOPVECTORS  EQU  $FFDE    ;Top of Interrupt Vector table

$Include 'j13regs.inc'

;=====

```

```

    org TOPRAM
operand1 ds 2      ;these 16 bit operands are used by the addition routine
operand2 ds 2
Avar ds 2          ;cordic angle variable
Xvar ds 2          ;cordic x coordinate variable
Yvar ds 2          ;cordic y coordinate variable
dA ds 2           ;cordic angle modifier variable
dX ds 2           ;cordic x coordinate modifier variable
dY ds 2           ;cordic y coordinate modifier variable
loopcnt ds 1      ;cordic iteration counter
flags ds 1        ;variable containing various flags
Angle ds 2        ;only used in demonstration, not part of CORDIC

;bits within the flags byte
clockwise equ 0   ;rotation direction flag
;The following two flags are only used in the demonstration and are not part of CORDIC
even_quad equ 1   ;set if we are outputting from an even numbered quadrant
neg_cycle_sin equ 2 ;set if we are outputting a negative half cycle

;=====

    org TOPROM

Main:
;This is the point where code starts executing after a reset.
;Insert your code to use the cordic algorithm here.
;=====
cordic:
;This is the main CORDIC algorithm. The angle to be transformed is input
;via the 16 bit variable Avar. The angle is specified in radians and multiplied
;by 2048 before input. The maximum input range is  $-\pi/2 < A < \pi/2$ .
;The sine output appears in the 16 bit variable Yvar, which is scaled up by a
;factor of 2048.
;The cosine output appears in the 16 bit variable Xvar, which is scaled up
;by a factor of 2048.
;Note that Avar,Xvar and Yvar are packed in memory with the more significant byte in ;the lower
address.
    mov #$00,loopcnt      ;start at iteration zero
    mov #$00,Yvar        ;initialize Yvar variable
    mov #$00,(Yvar+1)
    mov #$04,Xvar        ;initialize Xvar variable
    mov #$DC,(Xvar+1)
iterate:
    ldhx Xvar            ;dX=Xvar shifted right by i places
    lda loopcnt
    jsr shiftright
    sthx dX

    ldhx Yvar            ;dY=Yvar shifted right by i places
    lda loopcnt
    jsr shiftright
    sthx dY

    lda loopcnt          ;dA=lookup (1/2^loopcnt)
    jsr lookup
    sthx dA

    bset clockwise,flags
    brclr 7,Avar,update_pos ;if Avar>=0 increase angle
    bclr clockwise,flags    ;else decrease angle

update_pos:
    ldhx Xvar            ;Xvar=Xvar-dY if clockwise=1. Xvar=Xvar+dY if clockwise=0
    sthx operand1
    ldhx dY
    brclr clockwise,flags,clockwisel
    jsr negate
clockwisel:
    sthx operand2
    jsr addition
    ldhx operand1
    sthx Xvar

    ldhx Yvar            ;Yvar=Yvar+dX if clockwise=1. Yvar=Yvar-dX if clockwise=0
    sthx operand1
    ldhx dX

```

```

        brset clockwise,flags,clockwise2
        jsr negate
clockwise2:
        sthx operand2
        jsr addition
        ldhx operand1
        sthx Yvar

        ldhx Avar          ;Avar=Avar-dA if clockwise=1. Avar=Avar+dA if clockwise=0
        sthx operand1
        ldhx dA
        brclr clockwise,flags,clockwise3
        jsr negate
clockwise3:
        sthx operand2
        jsr addition
        ldhx operand1
        sthx Avar
        bra next_iteration

next_iteration:
        inc loopcnt          ;next iteration of the loop
        lda #$0B
        cmp loopcnt
        bne iterate
        rts

;=====
output:
;This function outputs data calculated by the CORDIC algorithm
;Data is input to DAC1 via the variable Xvar
;Data is input to DAC2 via the variable Yvar
;Implementation depends on the DAC being used and its configuration.

        rts
;=====
lookup:
;This routine looks a 16 bit number up from the lookup table.
;The lookup table index comes in via the accumulator and the looked
;up value comes out via H:X.

        lsla          ;needed because each lookup entry is 2 bytes
        tax
        clrh
        lda lookup_atan,X ;retrieve upper byte from lookup table
        psha          ;H reg used for addressing so not accessible
        aix #1        ;move to next position in table
        ldx lookup_atan,X ;retrieve lower byte from lookup table
        pulh
        rts
;=====
negate:
;This routine calculates the 2's compliment negative of a 16 bit number.
;The input comes in via H:X and output is returned via H:X
        pshh
        pula
        coma          ;Acc now holds 1's complimented high byte
        comx          ;X reg holds 1's complimented low byte
        psha
        pulh          ;H reg holds 1's complimented high byte
        aix #1        ;Add 1 to H:X to form 2's compliment from 1's compliment
        rts
;=====

addition:
;This routine adds two 16 bit numbers. The overflow bit is discarded,
;as this routine is meant for two's compliment numbers.
;The inputs are in the variables operand1 and operand2 and the output is
;in operand1. Operand1 and operand2 are packed in memory with the more
;significant byte in the lower address.
        lda (operand1+1)
        add (operand2+1)
        sta (operand1+1)

```

```

lda operand1
adc operand2
sta operand1
rts

;=====
lookup_atan:
;This is the arctangent lookup table. Each entry is 16 bits wide. There
;are 12 entries to give 12 bit resolution.
    db $06,$48,$03,$B5,$01,$F5,$00,$FE,$00,$7F,$00,$3F
    db $00,$1F,$00,$0F,$00,$07,$00,$03,$00,$01,$00,$00

;=====
No_irq:
;This interrupt handler catches interrupts that happened when no
;interrupt should have occurred. For this reason the processor is halted
;and user intervention is required. This routine is only called under severe
;fault conditions.

    stop        ; Stop the processor and wait for a reset.
    rti         ; return

;=====

org TOPVECTORS

    dw No_irq    ; ADC Conversion Complete interrupt
    dw No_irq    ; Keyboard interrupt
    dw No_irq    ; Unused
    dw No_irq    ; Unused
    dw No_irq    ; Unused
    dw No_irq    ; Unused
    dw No_irq    ; Unused
    dw No_irq    ; Unused
    dw No_irq    ; Unused
    dw No_irq    ; Unused
    dw No_irq    ; TIM1 Overflow interrupt
    dw No_irq    ; TIM1 Channel 1 interrupt
    dw No_irq    ; TIM1 Channel 0 interrupt
    dw No_irq    ; Unused
    dw No_irq    ; interrupt line
    dw No_irq    ; Software interrupt
    dw main      ; Reset Vector

;=====

main:
;This example of a main function puts a full sine wave into a 12 bit DAC.

    mov #$03,CONFIG1 ;Disable the watchdog timer, enable the stop instruction
    mov #$FF,DDRB    ;ports output for DAC connection
    mov #$FC,DDRD
    mov #$00,flags   ;start status flags in known state
    mov #$00,Angle   ;start angle in known state
    mov #$00,(Angle+1)
    mov #$00,Xvar
    mov #$00,(Xvar+1)
    mov #$00,Yvar
    mov #$00,(Yvar+1)

;This section of code generates a full sine wave from four quarter waves. The
;CORDIC system is used to generate sinusioids between 0deg and 90deg. The angle
;that is put into the algorithm is ramped up and then down to get each half cycle.
;Every second half cycle is inverted to create negative half cycles.
;There are 12868 points per cycle.

loop:
    ldhx Angle
    brset even_quad,flags,dec_angle ;decide whether to ramp the angle up or down
    aix #1                          ;ramp angle up
    sthx Avar                        ;unless at the peak of the waveform
    lda #$0C                        ;which is at (pi/2)*2048=3216decimal=$0C90
    cmp Avar                         ;if we are at the peak then at the next
    bne load_angle                  ;point we must start ramping the angle down.
    lda #$90
    cmp (Avar+1)
    bne load_angle

```

```

        bset even_quad,flags
        bra load_angle
dec_angle:
        aix #-1                ;ramp the angle down
        sthx Avar
        clra
        cmp Avar                ;unless the angle is zero, in which case
        bne load_angle         ;prepare to start ramping the angle up.
        cmp (Avar+1)           ;in addition when we reach zero we must
        bne load_angle         ;change from a positive half cycle to a
        bclr even_quad,flags    ;negative half, or visa versa
        brset neg_cycle_sin,flags,negative_half
        bset neg_cycle_sin,flags
        bra load_angle
negative_half:
        bclr neg_cycle_sin,flags
load_angle:
        sthx Angle             ;load the updated angle into the angle
        sthx Avar              ;variable and feed it into CORDIC
        jsr cordic
        mov #$07,operand2      ;add a DC offset to the output because our
        mov #$FF,(operand2+1) ;DAC is unipolar.
        ldhx Yvar
        brclr neg_cycle_sin,flags,no_bias_negative_sin ;if the current half cycle
        aix #-1                ;is negative then we need
        jsr negate             ;to invert it to produce a
no_bias_negative_sin:         ;full wave.
        sthx operand1
        jsr addition
        ldhx operand1
        sthx Yvar

        jsr output             ;send the output to the DAC
        jmp loop

```

References:

I am indebted to the following sources for information contained in this document.

Dr N Morrison, who introduced me to the CORDIC algorithm and taught me the basics of the system.
 Ingo Cyliax, Author of the article "CORDIC the swiss army knife for computing math functions" which may be found on the internet at www.ezcomm.com/~cyliax/Articles/Rob/Nav/sidebar.htm
 This article is highly recommended to those wishing to pursue the subject.