Numerical Simulation and Real-time Prototyping Platform

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Siddharth Raju

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Adviser: Prof. Ned Mohan

November 2017

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my adviser, Prof. Mohan for his invaluable guidance and support throughout my graduate life and for giving me the freedom to explore research areas outside my field of study.

I would like to thank my friend LN, for all his help in both personal and professional life and, keeping me motivated throughout this journey, even at times when I felt the project was a lost cause. I would like to thank my friend Saurabh, for all his sage advice on a wide range of subjects and always having left me more informed at end of every discussion. My time here at Minnesota wouldn't have been the same without, Shanker and Preeti and I would like to thank them for always looking out for me. For setting me up on this path, I would like to thank my mentor and friend Senthilkumaran and Ramesh. I would like to thank my friend Santhosh for all his help. I should also mention my colleagues, Gysler, Kartik, SaiPriya, Srikanth and Ashish for making my time here at the lab extremely memorable.

I would be remiss if I end without thanking my parents and my brother who have always believed in my dreams and have sacrificed their own to make mine come true.

Dedication

To my parents.

Abstract

Over the past decade, real-time control design has rapidly moved from the age-old process of developing code based firmware in embedded C, to platforms that auto generate the embedded C program from a higher level drag and drop, model based design. This has various advantages such as device independent system design, reduced possibility of user based code translation errors, limited effort needed in terms of code maintenance and most important of all, ability to design more complex systems without worrying about lower level implementation. Various softwares are commercially available such as Matlab, Plexim, PSIM, VisSim etc., for this purpose but they all have the same shortcoming, i.e., extremely unaffordable to individual users and small companies.

In this thesis, a new numerical simulation platform has been developed from scratch. To this end, a new programming language with inbuilt matrix support was created. The implementation details of the various compiler components, such as lexer, parser, sematic analyzer etc., has been presented. In addition to this code based platform, a model based platform was developed, to enable drag and drop based system development and, the design details including solver development, tools supported, real-time controller peripheral support etc., are presented in detail. A common platform independent intermediate language was created, to which all model and code developed in this platform gets translated into. Multiple up compilers are made available with the platform to enable translation to other languages.

One such supported language is C89 which is the most commonly supported language by real-time controller manufacture's standalone compilers. The auto generated code adds all necessary peripheral and system configuration code and control code, compiles it to device specific

hex code and programs it directly into the controller. This allows users to have no knowledge of the underlying device or any programming and makes the entire system more readable and easy to maintain.

For optimized running of the generated code, a high-performance math library was created using minimax polynomials and implemented directly in supported device specific assembly. In addition to this method, other commonly used methods are analyzed in depth. In addition to this, various other forms of optimizations were implemented that are not available on other platforms.

In the current version, only TI's TMS320F28335 is supported with plans to extend it to more controllers. Two low cost rapid prototyping platforms were developed to enable real-time control using model developed in simulation. The solution was tested on multiple motor control and power electronics applications, the results of which are presented.

This whole solution cuts down the cost of numerical simulation and rapid real-time prototyping to few thousand dollars which would have previously costed upwards of ten to hundred thousand dollar on a platform of similar capability.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past 10 years, motor/process control industry has rapidly moved from the age-old process of developing code based firmware in embedded C, to platforms that auto generate the embedded C program from a higher level drag and drop, model based design. This has various advantages such as device independent system design, reduced possibility of user based code translation errors, limited effort needed in terms of code maintenance and most important of all, ability to design more complex systems without worrying about lower level implementation. These advantages as significant as they are, comes at an extreme software licensing cost.

In the past, serious limitations on the performance of embedded processors both in terms of instructions evaluated per second as well as memory, meant having to write extremely concise code. To add on to this burden was the absence of embedded processors with dedicated floating-point math unit. It meant that any real-world mathematical control model developed had to be "translated" into an 8/16-bit controller instruction set and in many cases involved coding at assembly level to avoid translator introduced inefficiencies. This fixed-point optimization introduces significant quantization non-linearity in the systems that could lead to undesired system response, at worst system instability [1, 2]. With current day PC/mobile processors reaching up to 300K MIPS and far more efficient compilers, these issues might on first look seem trivial, but these advances have not penetrated as rapidly into the embedded motor control world where cost, thermal

dissipation and form factor play a far critical role than need for ever increasing computational performance. Nevertheless, there have been significant strides made in these by introduction of various processors by manufactures such as Texas Instruments, NXP, Atmel to name a few [3, 4].

The advent of these faster processor and more powerful compilers, allows for sufficient legroom to abstract away the lower level details of real-world controller implementation without losing too much on computational performance. Matlab, Plexim, PSIM, VisSim, Scilab are some of the software that have taken advantage of these developments to allow high-level model based design abstraction with automatic code translation. Of these, Matlab has been the de-facto tool for numerical simulation with capabilities far beyond any other competing products.

This chapter begins with an overview of various solutions available for designing motor control firmware. The various pros and cons of each of these solutions are analyzed in terms of their cost, engineering effort and performance limitations. A brief glimpse of various applications on which these were developed and tested is presented before finishing off with an introduction to the proposed solution.

1.1 Evaluation of existing solutions

There are a vast number of numerical simulation tools such as NumPy, Maple, Mathematica, Matlab etc., which have built in libraries for solving ordinary differential equations and most commonly used matrix operators. Of these, this section is dedicated to considering those platforms that support drag and drop, model based design environment with built in model to embedded C code translation. A list of solutions, model as well as code based and the device they were evaluated on for this analysis is listed below:

1. Matlab Simulink with
    a. dSpace ds1104
    b. Embedded Coder for TI TMS320F28335

c. System generator for Xilinx XC3S500E

2. Scilab with XCos
3. VisSim with TI TMS320F28335
4. TI Code Composer Studio with TMS320F28335
5. Microchip MPLAB with dSPIC33FJ128MC804

Matlab can convert model file into a generic C code using the separately sold Real-time workshop toolbox. Custom C code for device peripherals can be injected into the generated code, by means of s-function or can be separately added later to the generated generic code. This is the approach that dSpace ds1104 takes. The real-time controller is connected to the PCIe port of a PC and it uses 64-bit processor, MPC8240 with 250MHz clock to run the control code in real-time. This connects to an external interface board that break out the ADC, PWM, Encoder, Serial communication and digital IOs in the controller board and adds various driver and signal conditioning stages to it. For observing the feedback and computed data as well as dynamically changing any parameter values at run-time dSpace offers its own proprietary software called Control desk that ties with Matlab. Overall the combined solution makes an extremely powerful real-time prototyping solution. One of the short comings is that the simulation model and real-time model are not inter compatible, i.e. a real-time model developed using ADC, PWM blocks etc., of dS1104 cannot be simulated in Matlab before going to real-time mode. Thus, there is no way to validate the effects of real world non linearities such as saturation, slew etc., introduced by the digital peripherals, safely in simulation before going to real-time mode. The other serious limitation is that, this is just a prototyping platform. dSpace does not provide any out of the box solution to be able to use the same controller alone in the final product deployment. So, for final product design one the other methods must be used, and this leads to more effort and time to develop the system in another controller using another platform. Also of all the solutions explored in this section, this

is the most expensive one with dSpace alone costing upwards of $20,000 for non-academic users. Add on to this, the cost of Matlab licenses and all the associated tool boxes.

The second option with Matlab is using embedded coder toolbox. This is similar to Real-time workshop in the sense the model is translated into C code, but instead of just a generic code, it is translated into target specific code. Many device manufacturers offer peripheral tools for their device family which can be added to the developed model. These peripheral tools have a combination of selectable field and register value setting and for most part eliminate the need for user to know any C programming. It has a great cost advantage compared to dSpace since user need to only pay for Matlab licenses and most of these controllers are multiple orders of magnitude cheaper than dSpace. In addition to this both prototyping and large-scale deployment can use the same controller saving extensive development effort and time. Now coming to the limitations, even though the tool peripherals are drag and drop like dSpace, but these tool settings still require extensive knowledge of the device architecture and register information. With that regards this has additional learning curve and complexity. In addition, the developed code is highly unoptimized. This is terms of the math libraries that use generic development and not device specific assembly optimized code. The last shortcoming is this does not support data logging which was made possible in dSpace via their proprietary ControlDesk software.

The third option is the System generator toolbox offered by Xilinx for designing model based control algorithm for FPGA using Matlab. This is an extremely well developed and powerful tool but does not find real-world application for motor control, first due to cost and second due to extensive effort required in terms of designing a complete fixed-point system for a real-world control system. The time taken for code synthesis and programming is much higher than that for a simple motor control processor such as TI Delfino F28335 or dSPIC. This leads to much longer development time. Finally, the limited number of resources; since every operation uses a dedicated physical hardware unit the resource resources tends to run out rather quickly. For instance, the

Spartan 3E FPGA supports up to 20 multipliers. This is highly insufficient in most motor controls application. This can be overcome by time sharing these resources, but this leads to more design complexity. Alternatively, if a processor was used, there will be, in most cases just a single multiplier, but time sharing is inbuilt due to sequential execution.

There are other model based simulation platform alternatives to Matlab but all of these pale in their capability and ease of use in comparison to Matlab. The first among those is Scilab. This is a freely available platform. It has limited but sufficient tool blocks for developing motor control applications. The user interface is extremely primitive, for instance, it does not even support simple automatic routing between tool connections and the graphical elements are poorly designed. It has the capability to generate generic C code but cannot generate device specific code. So, the generated code needs to be further modified manually with device specific code. It also does not support any form of real-time data logging. Despite these limitations it remains widely used, due it being free, and all other similar platform is extremely expensive.

The other alternative is VisSim. These again are extremely limited in capability in comparison to Matlab but offers all necessary tools for motor control development. It is very similar to Matlab embedded coder but is limited to only TI's controllers. It suffers from all the shortcomings as that of Matlab's embedded coder. It cost as much as Matlab and hence, in choosing between Matlab and VisSim, Matlab seems to be the better choice. The final method that was evaluated was directly programming in C using manufacturer offered platform. This has the obvious shortcomings of design complexity, extreme effort need for code maintenance and inability to verify it in simulation before real-time deployment. The design is not portable to other platforms with ease. But, the major advantage is zero license cost compared to model based development platforms mentioned above.

Figure 1.1 shows the various solutions as function of cost vs design complexity with regards to motor control development. The capability of each solution is highlighted in the chart by means of circles or varying area, with larger indicating higher capability.

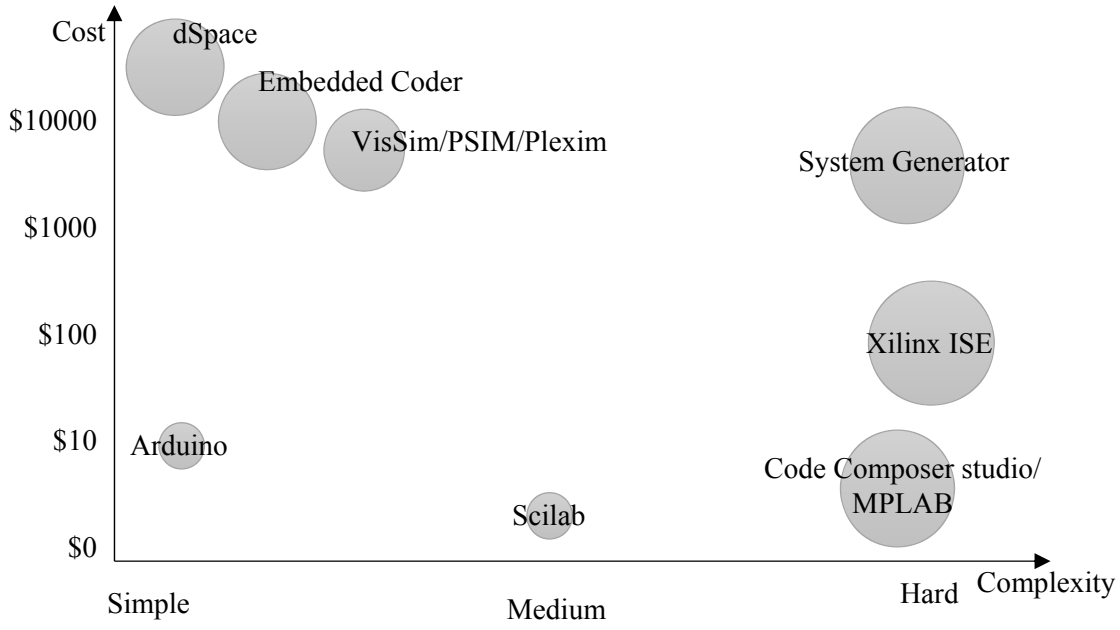


Figure 1.1: Cost vs design complexity vs capability of different real-time controller platforms

As part of this project development, the above platforms were tested on a variety of motor control and power electronics application and these are listed below:

i. Induction motor vector control using Matlab/dSpace.

ii. PMSM vector control using MPLAB/dSPIC33FJ128MC804.

iii. Back to back converter (active front end) control using MPLAB/dSPIC.

iv. BLDC motor control using MPLAB/dSPIC33FJ128MC804.

v. DC motor control using CCS/TMS320F28335.

vi. Direct matrix converter space vector control using XilinxISE/Spartan 3E xc3s500e.

vii. Multi-level matrix converter space vector control using System Generator/xc3s500e.

The above figure is generated based on data from the above-mentioned developments.

1.2 Organization of this thesis

This thesis provides the design and development details of a numerical simulation platform that is low cost, has all the capabilities of Matlab for motor control and power electronics application design, user interface as easily accessible and intuitive as Matlab as well as, a real-time platform that is as easy to use as dSpace but much cheaper and with the possibility of using the same prototyping controller for large scale deployment.

The following chapter explorers the methods for implementing a high precision high performance math library for elementary functions such as sine, cosine, logarithms, power etc. These are performance intensive functions and most of the processor execution time is spent in these function in most motor control applications. So, even a fractional optimization in these function leads to significant improvement in code performance. This allows for replacing a high cost, high speed controller with a lower cost lower speed version. Since cost reduction is one of the major constraints of this project, a significant effort is put into all feasible code optimization technique starting with assembly optimized device specific math libraries.

In the third chapter, a code based design platform is developed. A custom language with intrinsic matrix handling has been developed and the design details are explored. The components of compiler such as lexer, parser, semantic analyzer etc., are explored in detail with respect to the programming language developed. The language rules, keywords and operators are discussed in great detail in the Appendix associated with this chapter. In addition to user end programming language, a backend intermediate language has been developed. All model and code files are converted to this language and then translated to different languages such as C, C# and VB.Net as needed by the user. The runtime engine to run and display the result of the code file is briefly explored.

Chapter 4 explorers the other aspect of the developed platform, namely the model based design environment. It allows users to drag and drop tools from a pre-built suite of toolboxes and interconnect these to form complex time variant control algorithms. Various numerical solvers were implemented to solve integration and differentiation in the model file. This chapter explorers the various components of a model and their implementation in detail. It also briefly touches on how tools are translated into the intermediate language mentioned in chapter 3.

In the following chapter, real-time device specific C code generation from the model file is explored. This is followed by details of the two-low-cost hardware controller platform that were developed, one targeting motor control applications and other a more generic version with higher capabilities. Later the design of real-time data logging from DSP to computer and program code transfer from computer to DSP via USB is presented. Finally, all the above-mentioned applications developed using various platforms is redesigned, using the custom simulation and real-time prototyping platform and, is presented along with their results, showing the capability of the developed solution. Finally, the thesis is concluded with future work.

Chapter 2

High performance math libraries

The most task intensive operations of any control algorithms tend to be, the often-overlooked elementary math functions. All processors have dedicated arithmetic units that can perform series of addition and multiplications in a few instruction cycles. A few of them support dedicated instruction for inverse operation which may give a full precision final result or a partial first guess for speeding up computing and ensuring convergence of final result using other iterative methods such as Newton-Raphson. The number of microcontrollers/processors that supports dedicated instructions for trigonometric operations are just a hand full.

Considering these limitations, most processor manufactures distribute pre-built libraries for these functions usually written in C and down compiled to specific processors as needed. These are implemented in a variety of ways, with many of them using look up table based extrapolation, CORDIC, Taylor series or Remez polynomials. A few algorithms have evolved to perform extremely fast computation with good enough performance [5], but these are mostly of not much relevance within this context of examination for real-time control application.

Each of the above methods are well established and extensive literature available [6]. This chapter concentrates on the practical implementation on a single precision floating point processor with the aim of providing sufficient understanding to choose a suitable implementation based on application and weighing the cost and benefits of such solution. Finally, a consolidated set of

implementation methodology for these functions are provided by carefully weighing the cost of CPU performance over precision.

Each of the methods described below are evaluated based on the approximate instruction cycles to execute, program memory utilization, error in the result and a more relevant term, relative error. In a floating-point representation, same error at smaller values occupy more number of bits than at larger value where it may not even figure in the representation due to non-linear discretization of floating point numbers. The performance is evaluated for the most commonly used transcendental functions in embedded control applications [7]. Many of the transcendental function do not have a finite argument range but repeat within a certain interval. All the algorithms in the following sections work only within the specified interval and for arguments outside this interval, need to be reduced to an equivalent value within it. Theoretically, this usually entails a *modulus* operation. In practical implementation this leads to significant deterioration of precision. Hence, other contrived methods are used to achieve the same without loss of precision in the working range. These shall be described in section 2.4.

2.1 Look-up table based extrapolation

This method involves a finite set of block memory loaded with precomputed values of the function at mostly fixed intervals. For values in between these intervals, the results are linearly extrapolated. This is by far the fastest of all the algorithms. The only performance bottleneck is identifying the exact memory index of the closest function value, especially for inverse functions such as arc tangent.

a) sin(x): Consider the implementation for the simplest of trigonometric functions, namely *sin*. Consider N linear entries stored in look-up table (LUT) with value of *sin* pre- calculated at every $2\pi/N$ radians. To compute *sin(x)* where, $x \in [-\infty, \infty]$. The nearest entry in the memory table is computed by:

$$i = floor\left(reduce(x) * \frac{N}{2\pi}\right) \tag{2.1}$$

where, *reduce(x)*, reduces the argument to value between $[0, 2\pi]$ as described later in section 2.2. The result is computed from the LUT as follows and is shown in Fig. 2.1:

$$\sin_c(x) = LUT(i) + \left(x - (i-1) * \frac{2\pi}{N}\right) * \frac{(LUT(i+1) - LUT(i))}{2\pi/N} \tag{2.2}$$

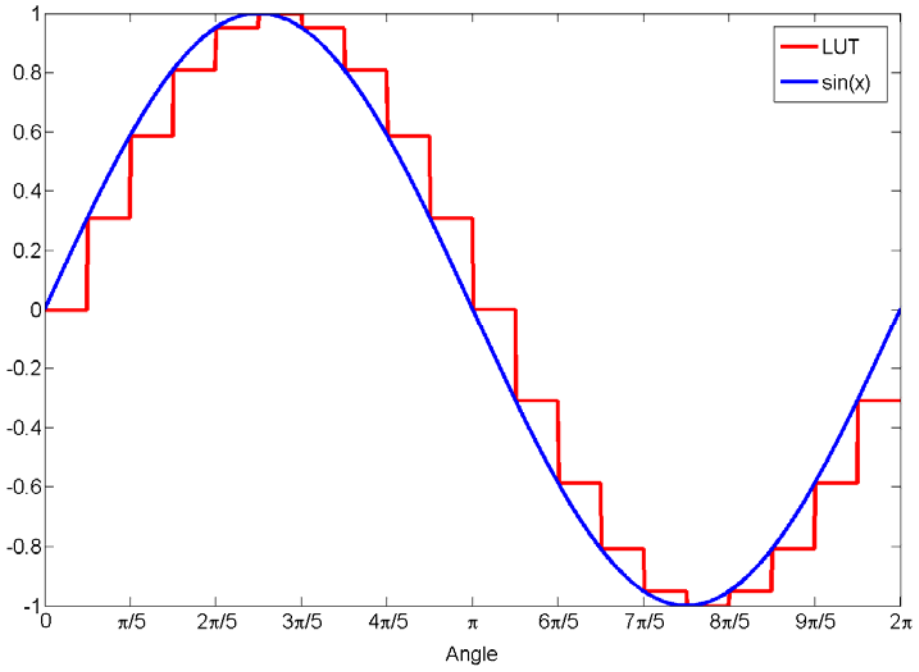


Figure 2.1: Sine values for 20 entries look up table and continuous Sine value over the range [0, 2π]

The relative error *eR* equals the ratio of the difference between the actual value and the computed value, and the actual value as given in Eq. 2.3 and plotted in Fig 2.2. As can be inferred from the figure, the minimum error occurs at zero crossing of sine, but it is also points of maximum relative error, due to large gradient. These are the limiting points and value at this point is obtained by $\lim_{x \to 0} eR(x)$ as expressed in Eqn. 2.4.

$$\mathrm{eR}(x) = 1 - \frac{(LUT(i) + (x * \frac{N}{2\pi} - i) * (LUT(i+1) - LUT(i)))}{\sin(x)} \tag{2.3}$$

$$ER = 1 - \frac{N}{2\pi}\sin(\frac{2\pi}{N}) \tag{2.4}$$

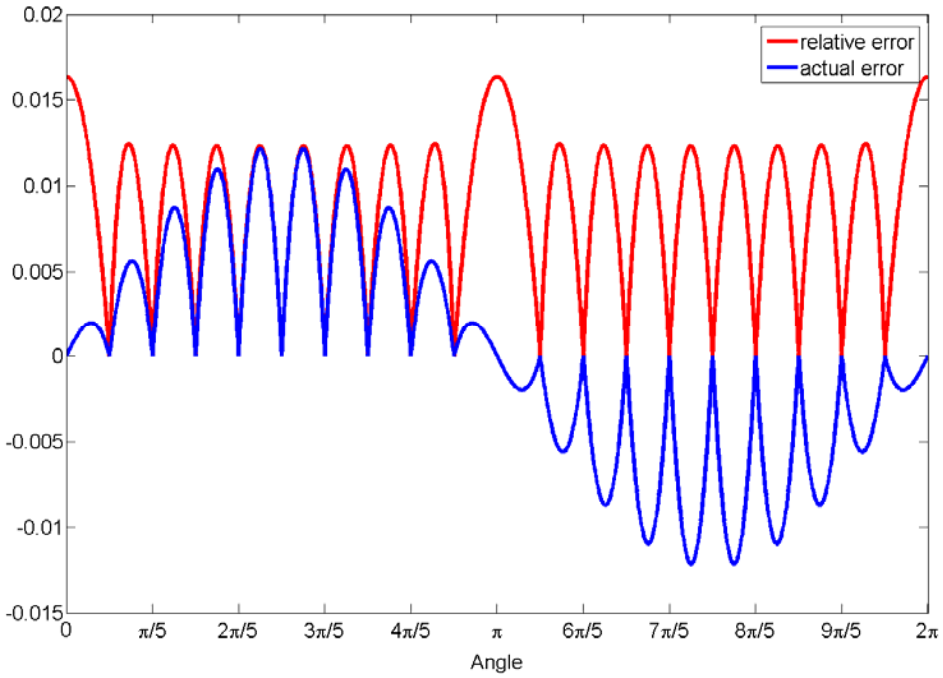


Figure 2.2: Actual error and Relative error of Sine computed from look-up table over the range [0, 2π]

Solution to Eqn. 2.4 yields the number of table entries needed for given relative error. In case of single precision floating-point number, the mantissa has 23 bits, which leads to a relative precision of $10^{-\log(2^{23})}$*. Any relative error is lost in discretization which is preferred. Figure 2.3 shows the number of entries N as a function of number of bits in mantissa. Even though single and double precision are the only IEEE standard implementation of floating point with 23 and 52 bits in mantissa respectively, the figure shows across values from 1 through 54 just to show the exponential trend, also where zero relative error is not necessary, the required bit precision can be reduced, and the entries needed can be inferred from the figure. Number of entries *N* for before mentioned relative errors for single precision floating-point turns out to N =7430.

**All references to 'log' is to the base 10 unless mentioned otherwise and 'ln' is natural logarithm*

This can be further reduced to a quarter, by reducing the interval from [0, 2π] to [0, π/2] and using a series of arithmetic operation to transform the result to appropriate sectors. This is a tradeoff between significant reduction in memory compared to minor increment in instruction cycles. Further reduction in table size can be introduced by changing the intervals from fixed to variable such that more table entries are present around high gradient. This will not lead to significant reduction in memory since the relative error at lowest gradient point, x = π/4, is very close to maximum error as seen in Fig 2.2. Also, this will render Eqn. 2.1 useless to find the nearest index. Rather an iterative method such as Binary Search algorithm needs to be used to find the nearest index to start from.

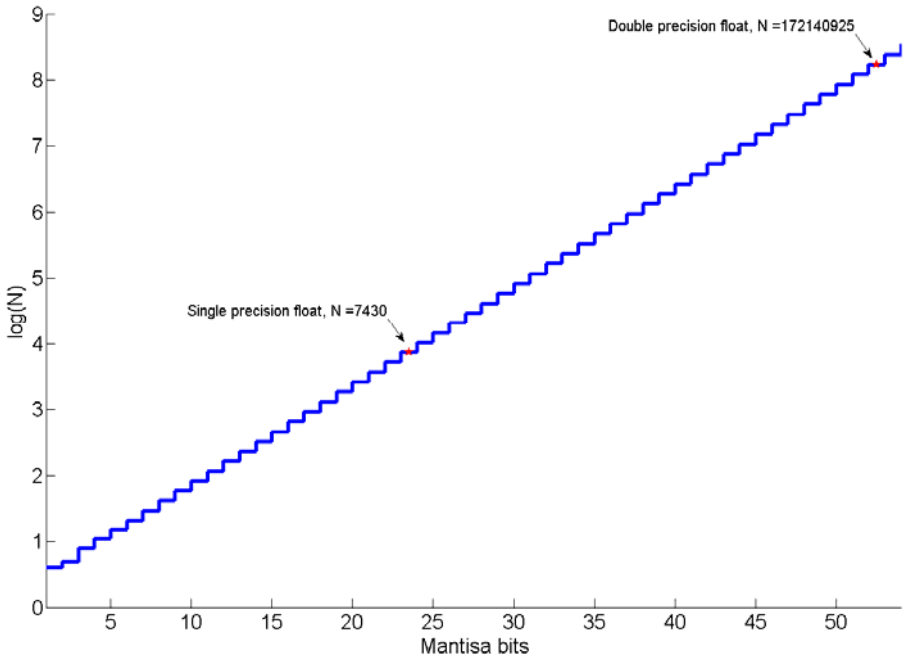


Figure 2.3: Number of table entries needed for zero relative error

As seen in Fig 2.3, this method does not scale well for double precision floating point since the number of fields required increase exponentially. For zero relative error for double precision floating point, it would require ≈ 1.28GB of data, which is impractical for anytime in the foreseeable future.

b) tan(x): The above analysis holds for *cos(x)* as well, which can be implemented as *cos(x) = sin(*π/2 - x*)*. tan(x) can be obtained by using the formula *sin(x)/cos(x)*. It must be noted though that this leads to loss of 2-bit precision in single precision floating-point system. The relative error introduced can be up to $3x10^{-7}$ as shown in Fig 2.4. This is due to multiple operations involved which do usually results in rounded number which by themselves are of proper precision for the operation involved, but when part of series of operation, the errors starts to accumulate. For example, consider a simple case of tan(π/3), which theoretically equals 1. In a floating-point machine implemented as mentioned above, $\sin\left(\frac{\pi}{4}\right) = \cos\left(\frac{\pi}{4}\right) = 0.7071068$ in its fullest precision. Now, $1/\cos\left(\frac{\pi}{4}\right) = 1.4142135$ on the same floating-point machine. The product of these two yields, 0.9999999 as opposed to the actual result of 1.000000. The error tends to be more at certain points.

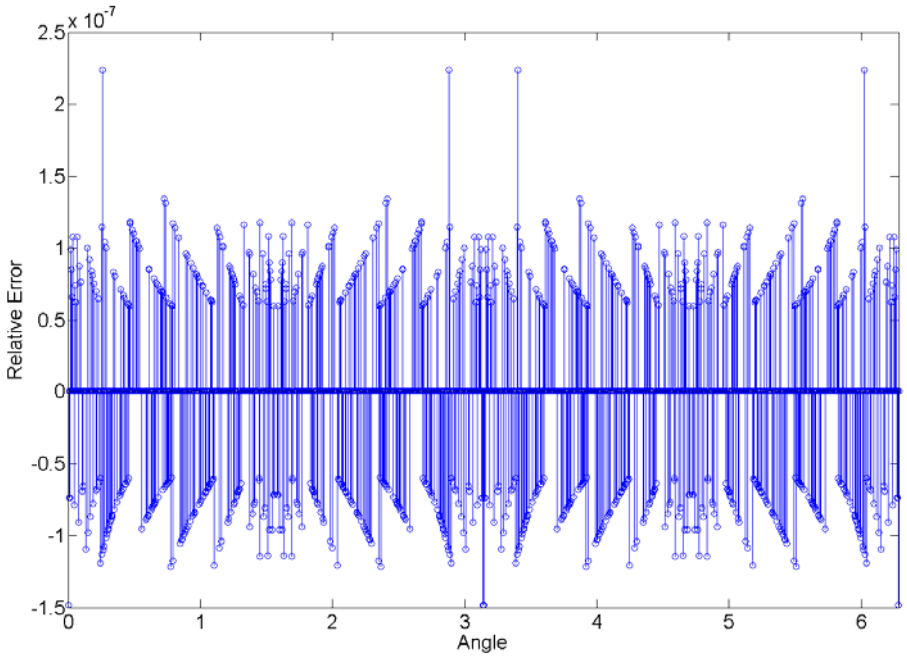
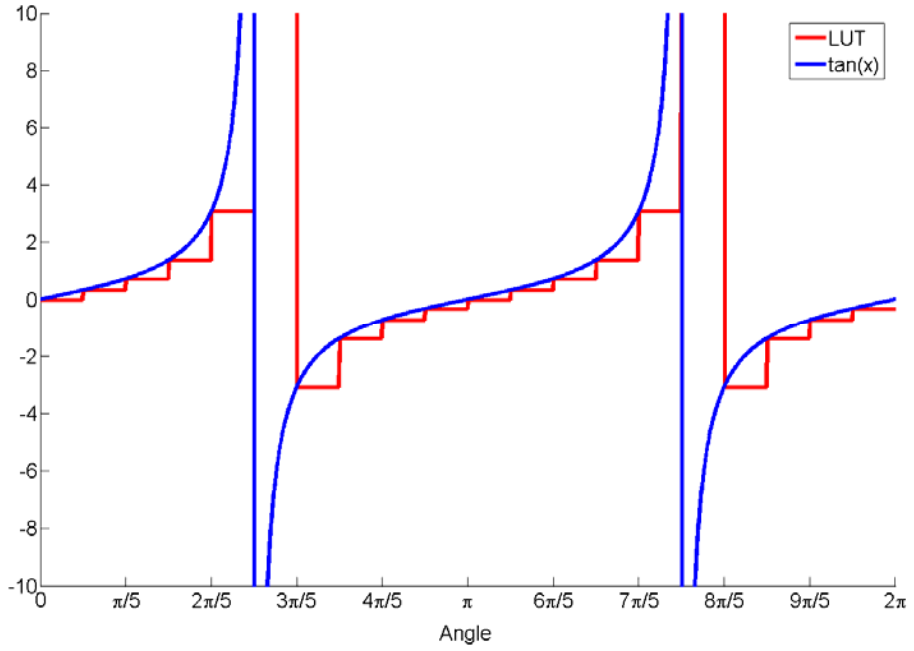
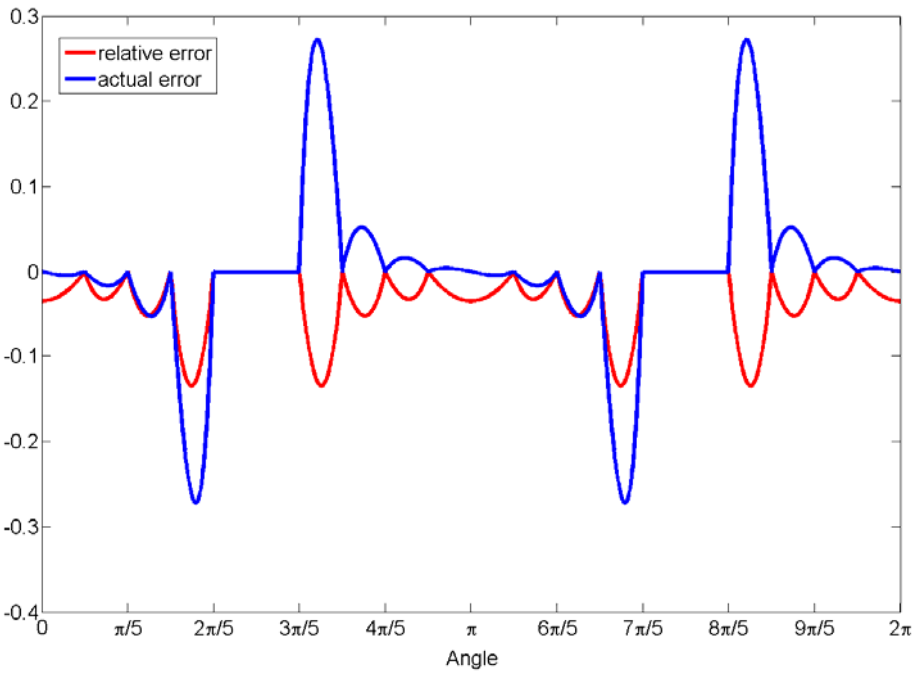


Figure 2.4: Relative error of tan(x) computed using sin(x)/cos(x)

To establish the number of entries required for zero relative error tolerance for tan(x), the same analysis as one for sin(x) is carried out, with the results of LUT for 20 entries and the computed error is shown in Fig 2.5.



(a)



(b)

Figure 2.5: Results for tax(x) LUT (a) tan(x) and LUT, (b) actual error and absolute error

In the interval where tan(x)→∞, the error→∞, since a linear extrapolation will have increasing error due to exponential nature of tan(x). In Fig. 2.5b, error in this region is set to 0, since this is handled by selecting interval such that at the edge of the final interval near x = π/2, the floating point reaches its maximum value, 3.402823e+38. To find the maximum error associated with a given LUT of size N, Eqn. 2.3 is replaced with *tan(x)* instead of *sin(x)* as in Eqn. 2.5 and the differential of Eqn. 2.5 equated to 0 as in Eqn. 2.6, solution to which gives the points local maxima.

$$\tan_c(x) = LUT(i) + \left(x - (i-1) * \frac{2\pi}{N}\right) * \frac{(LUT(i+1) - LUT(i))}{2\pi/_N} \tag{2.5}$$

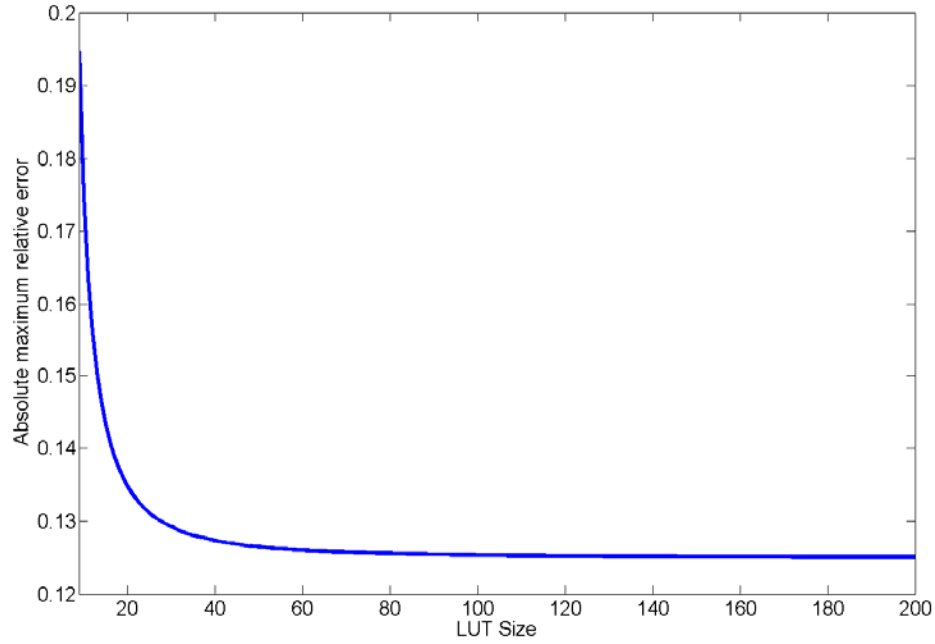
$$\sin(2x) - 2x - \left(\frac{LUT(i)}{LUT(i+1) - LUT(i)} - i - 1\right) * \frac{4\pi}{N} = 0 \tag{2.6}$$

where, $i = N/4\text{-}1$; which is the entry just before the interval where tan(x) →∞. For instance, for the above case of N = 20, I = 4, where LUT(4) = 1.3763819 and LUT(5) = 3.0776834. Above equation becomes:

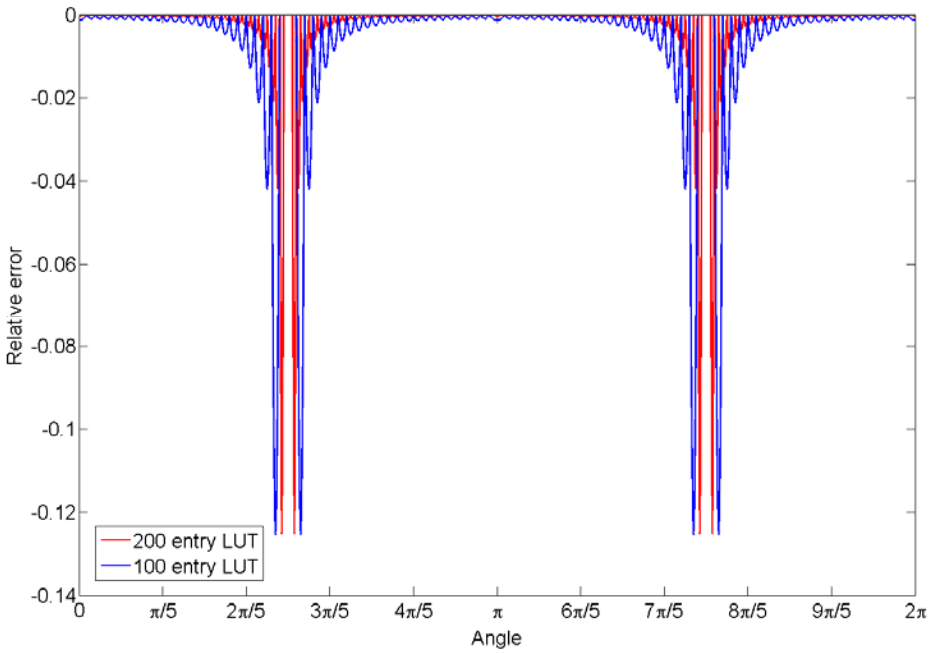
$$\sin(2x) - 2x + 1.3766353 = 0 \tag{2.7}$$

The solution to above equation gives x = 1.095266 at which point relative error = -0.1348208.

The maximum relative error for various values of LUT size $\boldsymbol{N}$, from 9 to 200 is shown in figure 2.6a. As it can be observed, the maximum relative error begins to converge to -0.125 as N→∞. The result is quite interesting because, one would expect as N→∞, it would be able to represent every single value accurately, but it does not because, even as *N* becomes large, within the final interval where tan(x)→∞, it can be thought of as tan(x) →∞ at much faster rate than N→∞. Thus, it is much more practical to use *sin(x)/cos(x)* as mentioned earlier and have the loss of up to 2-bit precision rather than use a LUT and have up to 20-bit precision loss for single precision floating-point system.



(a)



(b)

Figure 2.6: (a) Absolute maximum relative error, (b) Relative error for LUT of 100 and 200 entries

c) e^x*:* The plot of e^x is shown in Fig 2.7 for the interval of [0, 5] with N = 4 samples per interval of size 1. Fig 2.8 shows the actual and relative error. The local maxima of actual error build up exponentially, but the relative error's local maxima stay constant because, the base function also increases exponentially.

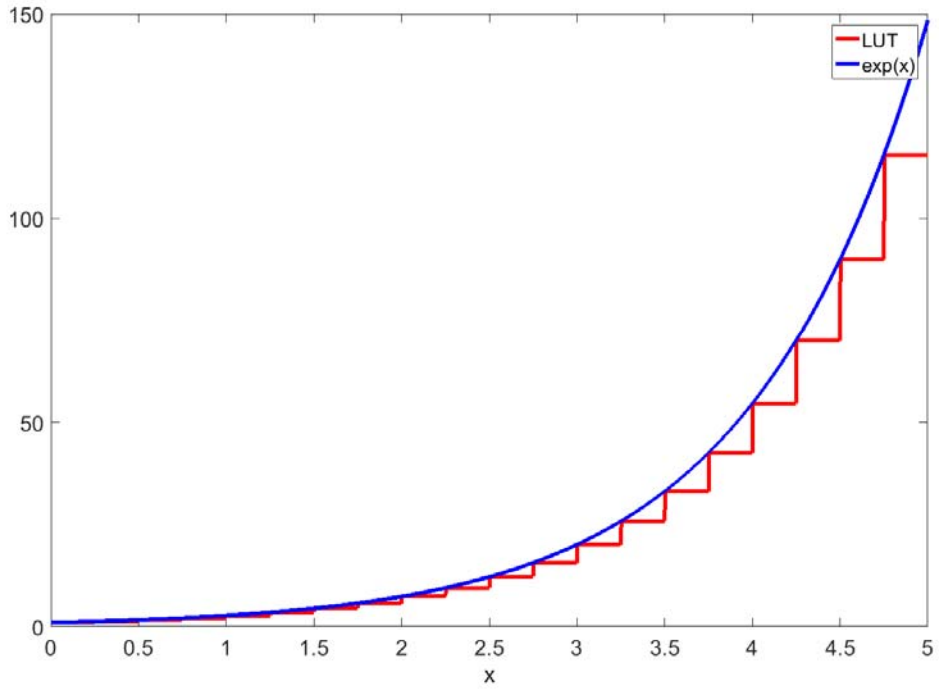


Figure 2.7: Exponent values for 20 entries look up table and continuous Sine value over the range [0, 5]

The estimate can be obtained by replacing *sin(x) in* Eqn. 2.2 with e^x. Since peak of relative error is constant across *x*, it can be computed for just the occurrence in the first interval as given in Eqn.2.8. The point of maximum relative error can be computed by solving for the differential of relative error equals 0 and the solution is given in Eqn. 2.9.

$$e^x = 1 + Nx * (e^{\frac{1}{N}} - 1) \tag{2.8}$$

$$x = 1 - \frac{1}{N(e^{\frac{1}{N}} - 1)} \tag{2.9}$$

where, *N* is the number of entries for the interval [0, 1].

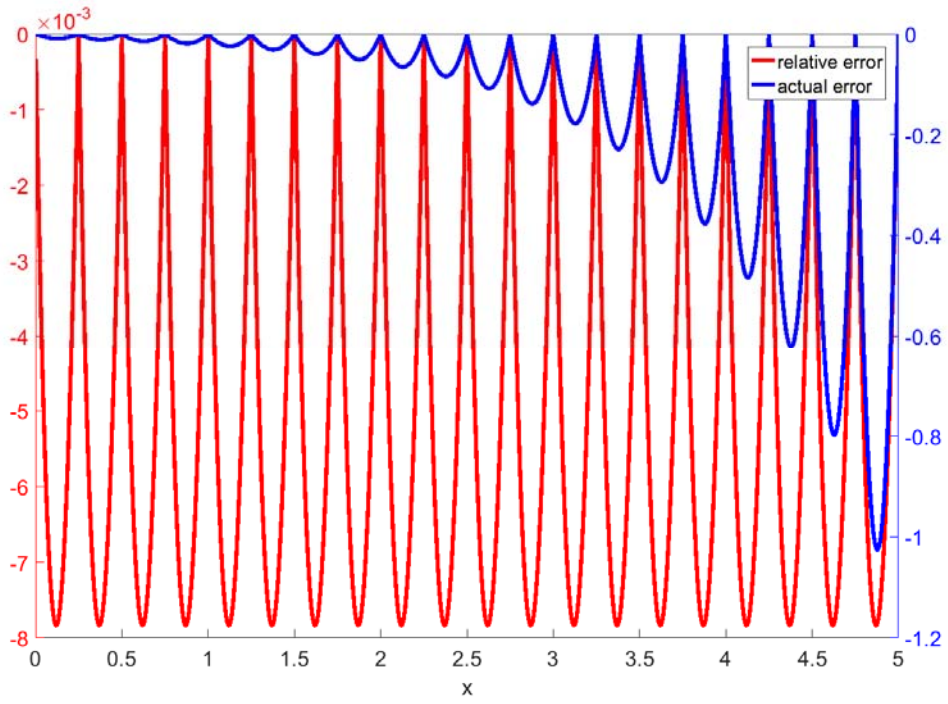
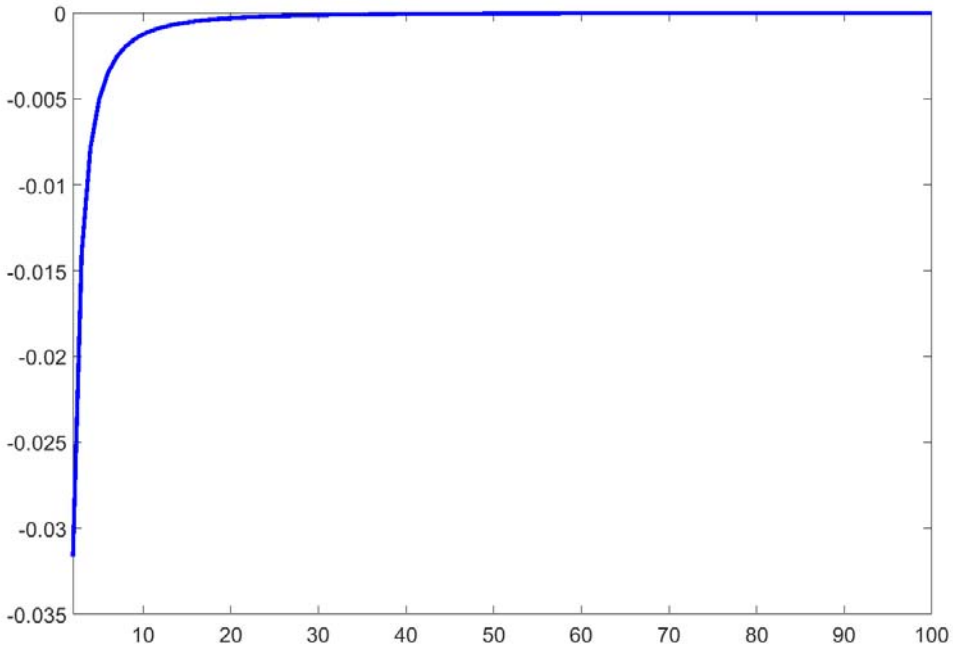


Figure 2.8: Actual error and Relative error of e^x computed from look-up table over the range [0, 5]

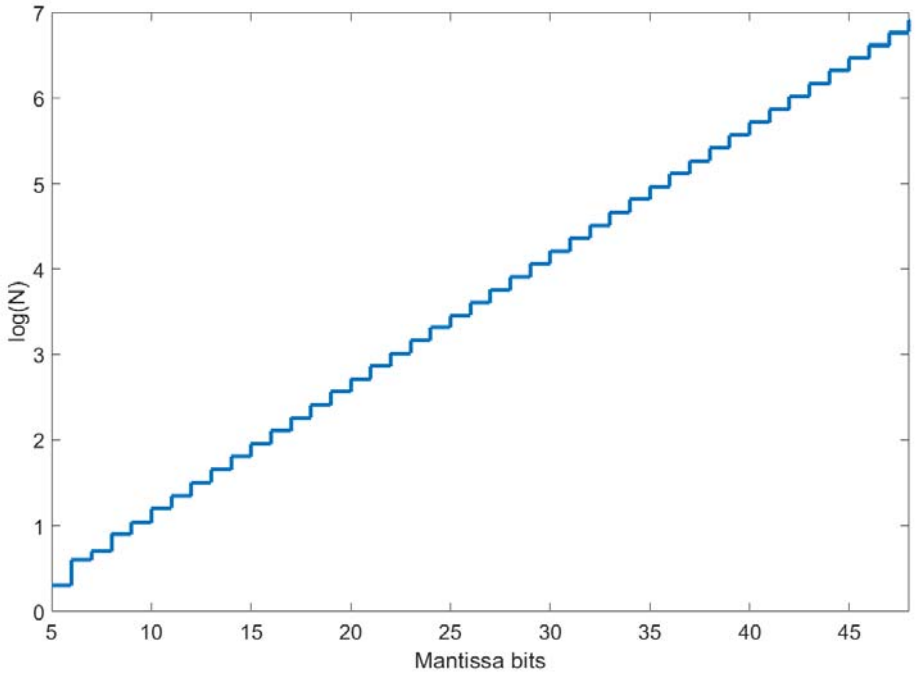
The maximum relative errors at points from Eqn. 2.9 for different values of *N* is plotted in Fig. 2.9. It converges to 0 as the number of entries increases. To translate this maximum relative error/precision to number of mantissa bits *x* in Eqn. 2.9a is modified as:

$$M = -log_2\left(1 - \frac{1}{N(e^{\frac{1}{N}} - 1)}\right) \tag{2.10}$$

where, *M* is the number of mantissa bits. The plot for *N vs M* is shown in Fig. 2.9b. It must be noted *N* is the number of fields for the interval [0, 1]. The full range of exponent, in case of single precision, is maximum *x* after which e^x overflows. This equals ln(float_max) = ln(3.402823e+38) ≈ 89. Exponent of *x* can be resolved as $e^x = e^a e^b$ where *b* is the fractional part of *x* and *a* is the integral part. The exponent values for integral *x* from -89, -88, …, -2, -1, 0, 1, 2, …, 89, a separate table is maintained. The result is product of exponent from the integral table and the fraction table.



(a)



(b)

Figure 2.9: (a) Maximum Relative error of e^x for N $\in$ [0, 100] (b) Number of LUT entries needed.

d) *ln(x):* Carrying out same procedure as done for *sin(x)* over a limited range of [0, 0.1] yields relative error as shown in Fig 2.10.

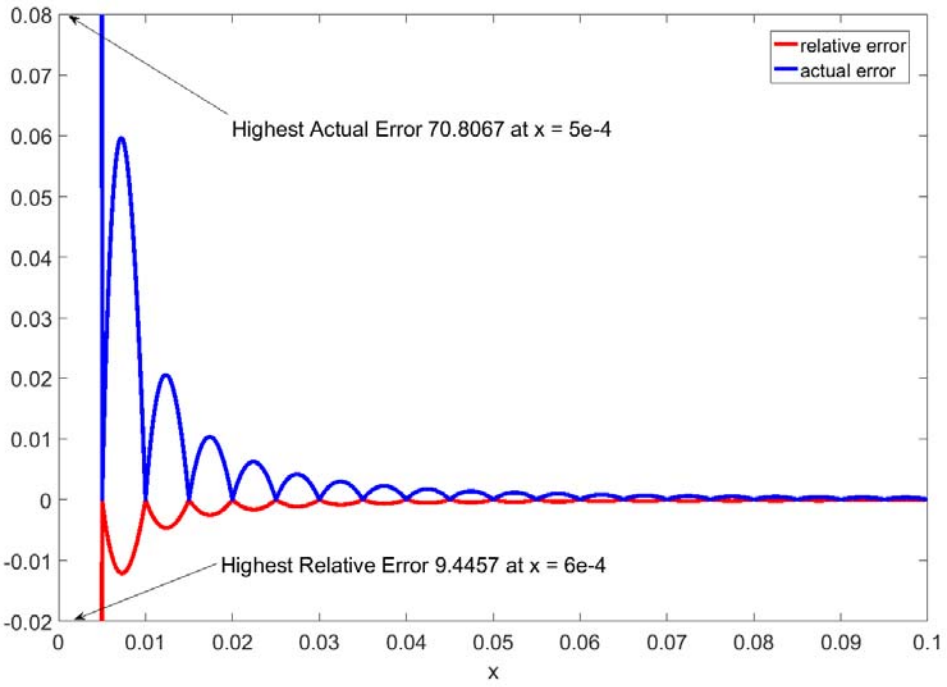


Figure 2.10: Actual error and Relative error of *ln(x)* computed from look-up table over the range [0, 5]

The relative error function is non-uniformly distributed, highly skewered towards 0. This implies uniformly spaced LUT will be an extreme waste of resource. In addition, unlike previous functions, *ln(x)* is not range limited, it produces valid results for all positive finite values of floating point. This does not mean that one entry will be needed for each possible *x* value because, *ln(x)* performs number line compression, multiple *x* will produce same result in a floating-point system. Hence the number of fields needed will not be 2 to the power of total bits used for float representation, but will still be an impractical value. To overcome this, a simple range reduction can be carried out. A floating-point number is represented as $m * 2^e$, where $m \in [0.5, 1)$ is the mantissa and, e is the exponent. This can be instantly reduced as:

$$\ln(m * 2^e) = (e + 1) * ln(2) + \ln(m/2) \tag{2.11}$$

It is expressed as in Eqn. 2.11 rather than as $e * ln(2) + \ln(m)$ because, in this case argument of ln ∈ [0.5, 1) while in Eqn. 2.11 it ∈ [0.25, 0.5). In the former, relative error increases rapidly as m→1, while in later, it converges as m→0.5 as seen in Fig. 2.11. The range can be further reduced to [0.125, 0.25) leading to reduced relative error, but this must be weighed against loss of single bit of argument precision every time range is reduced by half. In the first case, the relative error as m→1 was significant enough to justify the loss of precision while in the case of later, it is not.

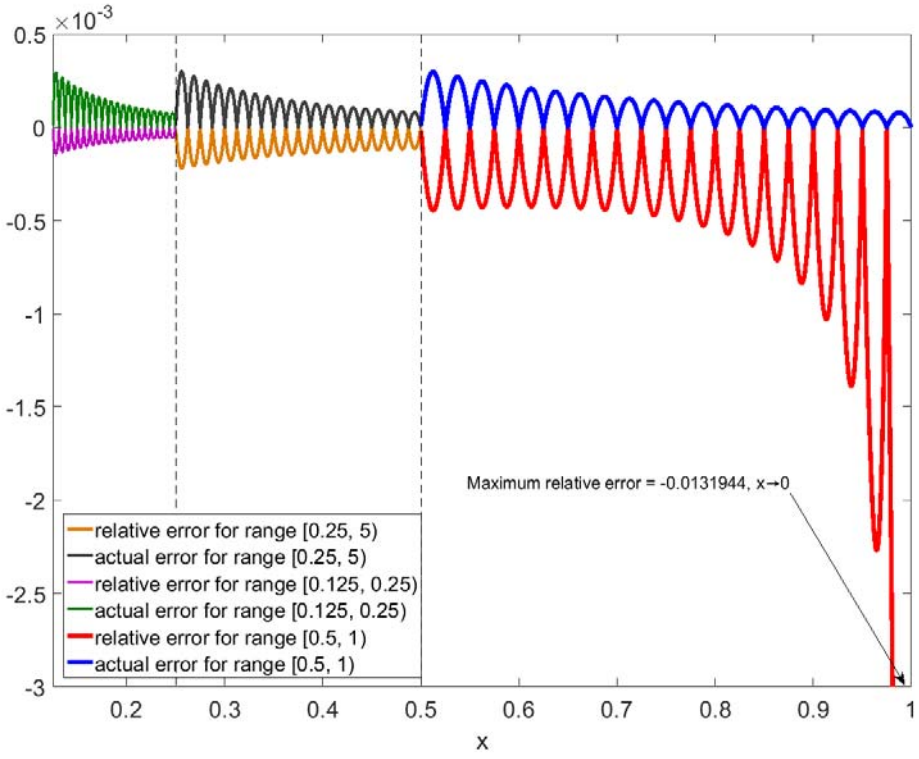


Figure 2.11: Actual error and Relative error of *ln(x)* using look-up table over the ranges [0.125, 0.25), [0.25, 0.5) and [0.5, 1)

Figure 2.12 shows the error for 20 entry LUT for *ln(x)*, $x \in [0.25, 0.5)$, a zoomed in version of Fig. 2.11. Following the same procedure as before to find the point of maxima in relative error, i.e. point where differential of relative error equals zero, gives Eqn. 2.12

$$x(\ln(x) - 1) = \frac{\ln(0.25)}{ln\left(1 + \frac{1}{N}\right)^{\frac{N}{0.25}}} - 0.25 \tag{2.11}$$

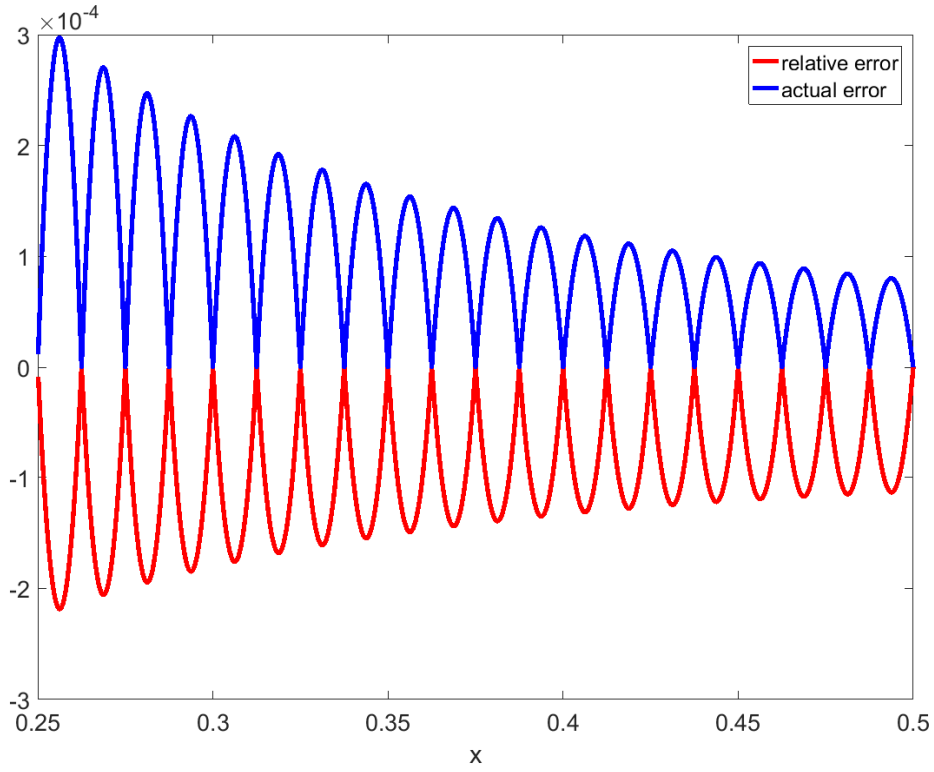


Figure 2.12: Actual error and Relative error of *ln(x)* using look-up table over the range [0.25, 1)

Equation 2.11 holds for the range [0.125, 0.25) as well, when 0.25 in the equation is replaced with 0.125. For the range of [0.5, 1), the limiting value occurs at 1 and does not require solving any equation as was the case in other two cases. The relative error is calculated at x as explained for other cases. This is equivalent to the relative precision of a floating-point number from which the number of mantissa bits M can be computed:

$$M = -log_2(relative\ error(x)) \tag{2.12}$$

The LUT size required for zero relative error in the computed range, calculated using Eqn. 2.11 and 2.12 is shown in Fig. 2.13. A reduction of range from [0.25, 0.5) to [0.125, 0.25) does not reduce the LUT size significant enough to justify the loss in argument precision. On the contrary, The LUT size for range [0.5, 1) increases double exponentially (note the plot is already in log(N))

and would require impractical amount of memory even for single precision floating-point number for zero or one-bit relative error.

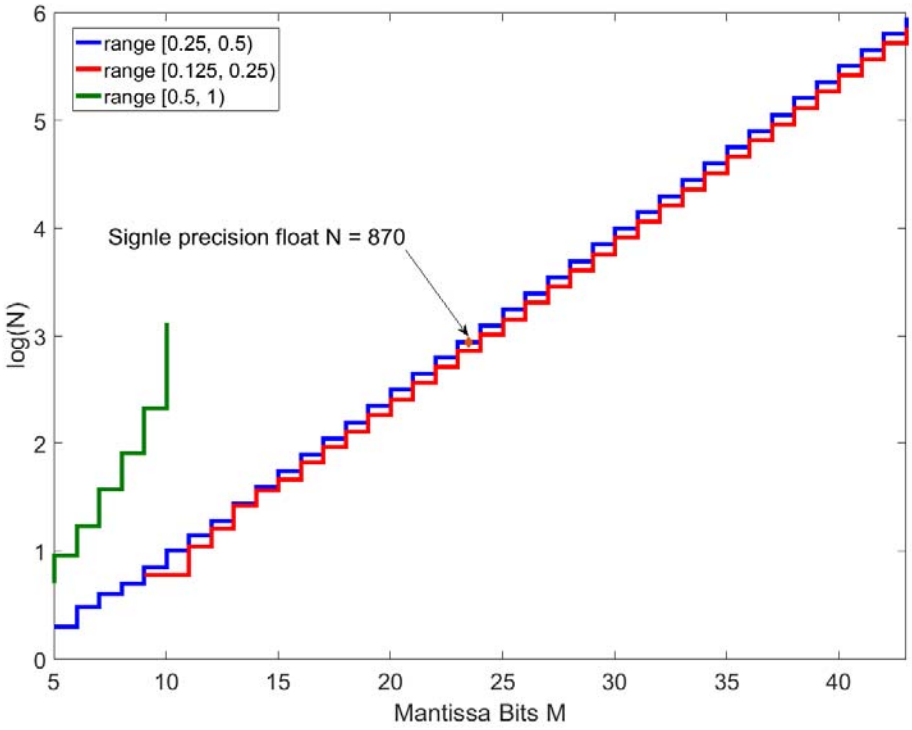


Figure 2.13: Number of table entries needed for the ranges [0.125, 0.25), [0.25, 0.5) and [0.5, 1)

e) *sinh(x):* This function is very similar to e^x, in the sense that the maximum relative error occurs at well define point, $x \rightarrow 0$ and the range is limited to (-90, 90) in single precision floating-point system due to exponential nature of hyperbolic function, where x outside this range causes *sinh(x)* to go to $\pm\infty$ in the limited bit representation. The relative error is shown in Fig 2.14, computed over the range [-2.5, 2.5] using a 20 entry LUT.

To compute the maximum relative error, consider a point δ, close to 0. The relative error at this point can be calculated by substituting *sinh* instead of *sin* in Eqn. 2.3 to get Eqn. 2.13. Relative error at 0 is computed by $\delta \rightarrow 0$ in Eqn. 2.13.

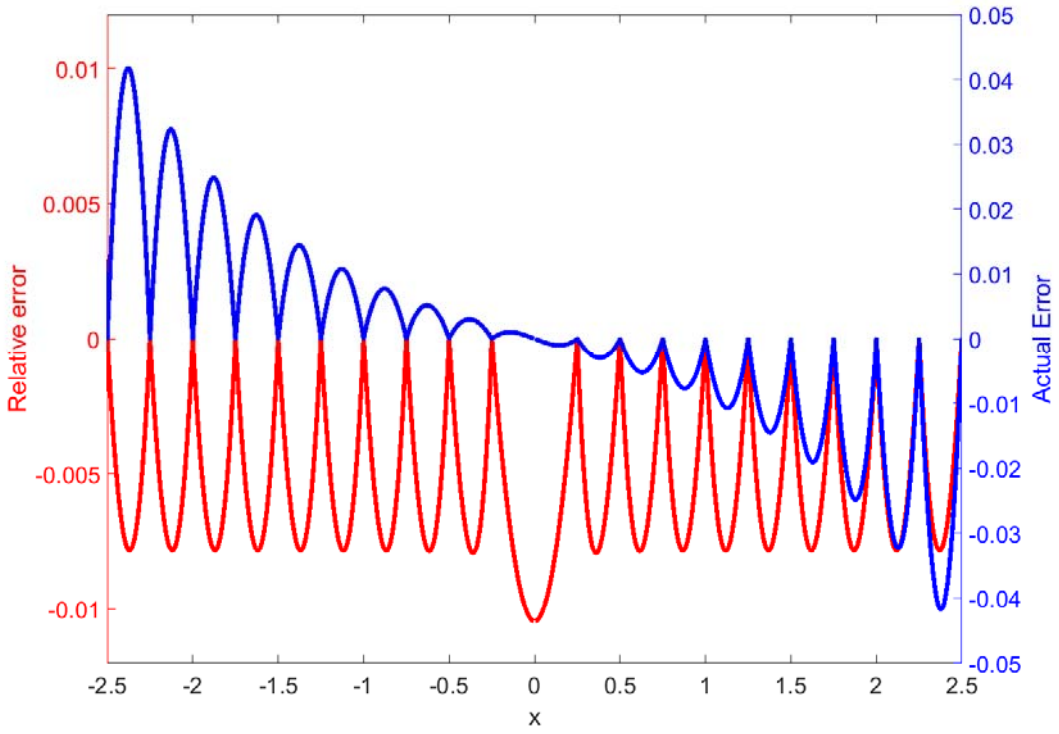


Figure 2.14: Actual error and Relative error of *sinh(x)* using look-up table

$$\mathrm{eR}(\delta) = 1 - \frac{\left(\sinh(0) + (\delta - 0) * \frac{\left(\sinh\left(\frac{1}{N}\right) - \sinh(0)\right)}{\left(\frac{1}{N}\right)}\right)}{\sinh(\delta)} = 1 - \frac{N\delta sinh\left(\frac{1}{N}\right)}{\sinh(\delta)} \tag{2.13}$$

$$\mathrm{eR}(0) = \lim_{\delta \to 0} eR(\delta) = 1 - Nsinh\left(\frac{1}{N}\right) \tag{2.14}$$

where, *N* is the number of LUT entries in the range [0, 1]. Like previous cases this relative error is converted to mantissa bits of a floating-point number as in Eqn. 2.12. The result is shown in Fig. 2.15. *N* is the length of LUT for the range [0, 1]. The maximum value of *x* above which *sinh(x)* overflows for single precision = *asinh(float_max)* = asinh(3.402823e+38) ≈ 90. Thus, total LUT length for single precision floating-point system for zero relative error = 90*1183 = 106,470. The negative range is computed by negating result for corresponding positive *x*. The results for *cosh(x)*

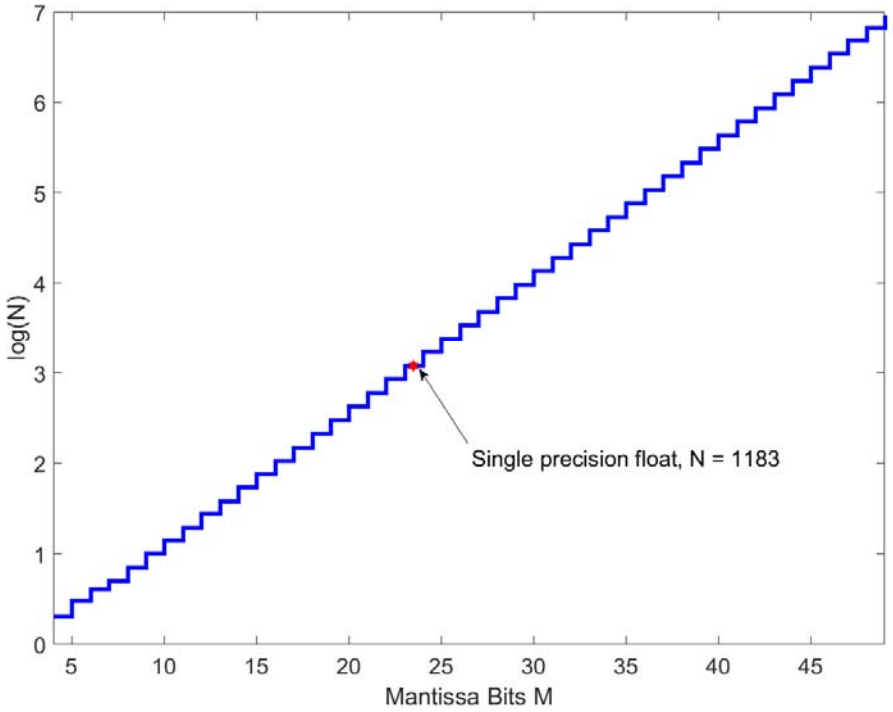


Figure 2.15: Number of table entries needed for zero relative error

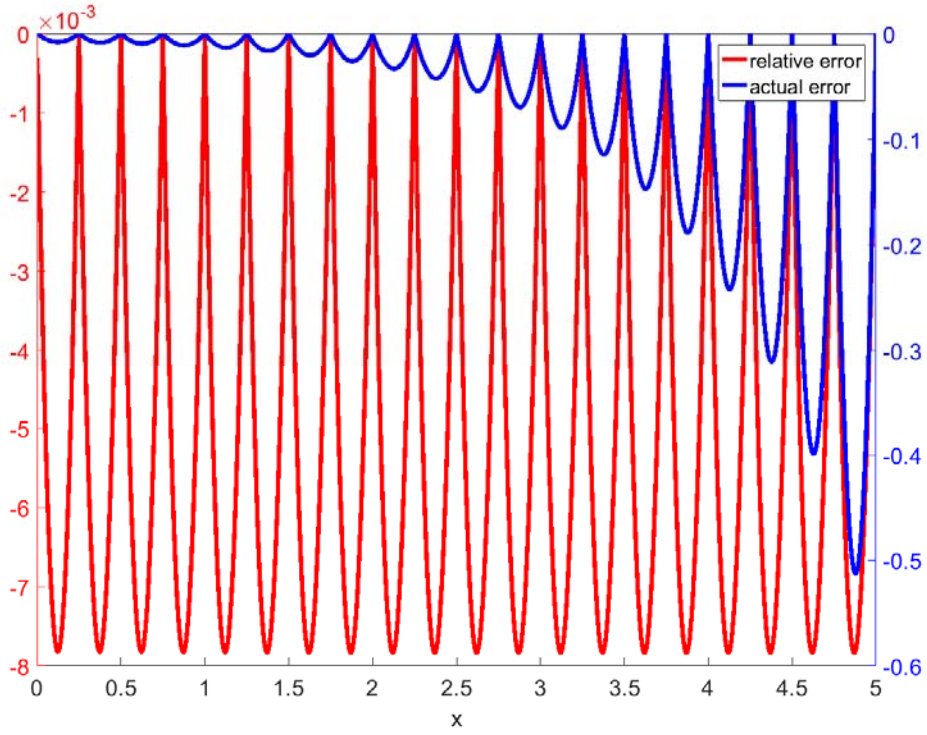


Figure 2.16: Actual error and Relative error of *cosh(x)* using look-up table

is shown in Fig 2.16 and it is similar to e^x discussed earlier and shall be skipped from further discussion.

The inverse of trigonometric functions can be computed doing a simple binary search of the value of their corresponding inverse functions. If only the inverse function is required and not their forward function, then similar approach can be used to generate their look up tables which will be of the same size as the forward function due to 1:1 mapping.

This sub section highlighted the memory requirement for all the functions for zero relative error as well as the size that would be required if these tolerances were to be relaxed. It gives a detailed picture of the feasibility of LUT based implementation based on available memory on a platform where it is to be deployed. In the subsequent sections for other methodologies, the constraint would pivot towards performance rather than memory.

2.2 CORDIC

CORDIC or Coordinate Rotation Digital Computer is an iterative algorithm based on rotational transformation to obtain result of trigonometric operation merely by use of addition and arithmetic shift operations. It was introduced when multiplication and division was not available on most processors or, were expensive both in cost and performance. These are particularly well suited for fixed point systems. For floating-point systems the task of dividing by 2 (or multiplying by 0.5) as required by this algorithm is not a simple shift operation as in case of fixed point. They involve a multiplication operation if a hardware floating-point multiplier is available or a series of logical and arithmetic operation on the exponent and mantissa and a fixed-point multiplication, simply replication the operations of a hardware floating-point multiplier. This method of solving above functions is advantage only for the fact their theoretical concept is simple to grasp. They are neither as fast as LUT discussed above nor are they precise or fast as Remez polynomial based algorithm discussed later. Despite this, it remains widely used in FPGA where dedicated hardware

multipliers are limited and in fixed point processor due to ease of implementation. In addition to the loss of accuracy there is a limit on the range of input argument over which this algorithm works but there have been multiple articles on addressing these issues usually one at the cost of the other [8, 9]. This sub section gives a brief overview of the method [10].

CORDIC algorithms are at its core, is manipulation of rotational transformation in orthogonal coordinate system in an iterative manner to yield the result of a desired function. These can be broken down into three main categories based on the rotational transformation used namely: circular, hyperbolic, and linear. Each of these can be further classified to rotational or vectoring mode. Consider the first case, i.e. circular coordinates in rotation mode. A point (x_1, y_1) in cartesian coordinates can be rotated by an angle θ around the origin to another point (x_2, y_2) using Eqn. 2.15. This can be modified into Eqn. 2.16 factoring out *cos(θ)*.

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \tag{2.15}$$

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \cos(\theta) \begin{bmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \tag{2.16}$$

Starting from (*1, 0*), rotating by an angle θ results in $x_2 = \cos(\theta)$ and: $y_2 = \sin(\theta)$. To achieve this result, Eqn. 2.16 is solved iteratively as explained below. $\tan(\theta)$ is computed at predetermined set of points. These points are chosen such that $\tan(\theta)$ equals inverse multiples of 2 as given in Table 2.1 such that the multiplication operation in above equation can be reduced to a simple shift operation in fixed-point system. In the first iteration, x_0 is set to 1, $y_0 = 0$ and initial angle $z_0$is set to θ. The values for each iteration is computed modifying Eqn. 2.16 to be evaluated at δ where $\tan(\delta)$ is inverse multiples of 2 as mentioned earlier:

$$\begin{aligned} x_{i+1} &= \cos(\delta_i)(x_i - \alpha * y_i * 2^{-i}) \\ y_{i+1} &= \cos(\delta_i)(y_i + \alpha * x_i * 2^{-i}) \\ z_{i+1} &= z_i - \alpha * \delta_i \end{aligned} \tag{2.17}$$

where, $\alpha = +1$ if $z_i > \theta$ else $\alpha = -1$.

i	$\tan(\delta_i)$	δ_i
1	1	45□
2	$\frac{1}{2}$	26.565□
3	$\frac{1}{4}$	14.036□
4	$\frac{1}{8}$	7.125□
⋮	⋮	⋮
n	$\frac{1}{2^{n-1}}$	$tan^{-1}\frac{1}{2^{n-1}}$

Table 2.1: $\tan(\theta)$ for each CORDIC iteration

What this does is, starting with a vector of unit length along X-axis, gradually rotates this by angles where *tan* is 'known' as given in Table 2.1. Every time the current rotated vector overshoots the desired final angle θ, the vector is rotated backwards by the angle corresponding to that iteration and if it undershoots, it is rotated forward in the following iteration. The result is to converge to the desired angle, like a binary search algorithm. At this point, the projection of this vector on *x* and *y* axis gives the *cos* and *sin* of the desired angle respectively which are byproducts of Eqn. 2.17. This equation can be further simplified by factoring out $\cos(\delta_i)$ in each iteration so that it can be multiplied in the end as a scaling constant *K*:

$$K = \prod_{i=0}^{n} \cos(\delta_i) = \prod \cos\left(tan^{-1}\left(2^{-i}\right)\right) = \prod \frac{1}{\sqrt{1+2^{-2i}}} \tag{2.18}$$

where, *n* is the total number of iterations. Above equation converges to $K = 0.6072153$ as number of iterations, $n \to \infty$. This scaling factor can be multiplied to *x, y* in the final iteration or in the first iteration in which case $x_0 = K$ instead of $x_0 = 1$, and $y_0 = 0$.

From above discussion, the maximum angle that a vector can be rotated by this method is given in Eqn. 2.19 and equals 99.883□ as number of iterations, $n \to \infty$.

$$\theta_{max} = \sum_{i=0}^{n} \delta_i = \sum tan^{-1}\frac{1}{2^i} \tag{2.19}$$

Within this range, given sufficient iterations all angles are *reachable* because angle corresponding to each iteration, δ_i can be synthesized by a combination of summation and difference of following iteration angles i.e.,

$$\delta_i = tan^{-1}\frac{1}{2^{i-1}} \leq \sum_{j=i+1}^{\infty} \delta_j = \sum_{j=i+1}^{\infty} tan^{-1}\frac{1}{2^{j-1}} \tag{2.19}$$

Satisfying this constraint is essential for convergence, since say during one the iteration if the rotated angle overshoots the desired angle, then Eqn. 2.19, ensures that the series of angle of following iterations will be sufficiently large enough to backtrack this overshoot, and vice versa in case of an undershoot. The maximum number of iteration needed for a floating-point system will be same as the number of mantissa bits. Any iteration after this point is unnecessary and the number would be so small that it will be lost in the addition operation's rounding. This does not mean the result will have zero relative error. Since in each iteration, every divide by 2 leads to loss of a least significant bit, which does not affect the precision of the current operation itself, but the cumulative loss affects the final result since $[x_1 + x_2] \neq [x_1] + [x_2]$ where, $[x]$ is x rounded to nearest integer. This effect can be seen in Fig. 2.17 where the actual error has almost constant peak across the computed range. Due to this relative error shoots upwards of 1 nearby where $\theta \rightarrow 0$. This lack of precision and an inability to control the relative error spectrum precisely is one of the main reason this method is not applicable in many applications requiring high performance, high precision math library. The other reason of limited range will become self-evident in the case of hyperbolic rotational transformation discussed next.

In the above method, the iteration started with a unit vector (which was later scaled by constant *K*) vector aligned along X-axis and this was rotated to the desired angle iteratively and the resultant *x* and *y* values represented the *cos* and *sin* of the desired angle. It must be noted at no point the magnitude of the initial vector was modified, i.e.:

$$\sqrt{{x_0}^2 + {y_0}^2} = \sqrt{{x_i}^2 + {y_i}^2} \tag{2.20}$$

where, *i* represents any of the iteration index.

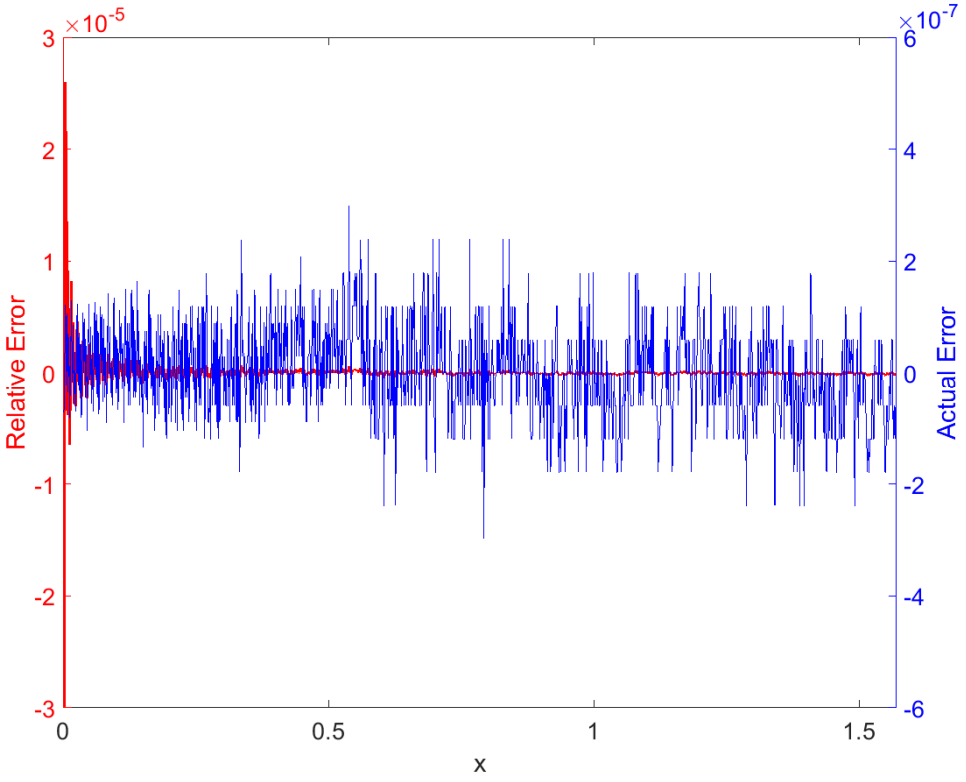


Figure 2.17: Actual error and Relative error of *sin (x)* using CORDIC

If instead, initial *x* and *y* start from a fixed value, i.e., non-zero magnitude vector at any angle, following the same procedure as above and rotating the vector to align along *x* will lead to:

$$\begin{aligned} x_{final} &= \sqrt{{x_0}^2 + {y_0}^2} \\ z_{final} &= tan^{-1}\left(\frac{y_0}{x_0}\right) \end{aligned} \tag{2.21}$$

This is the vectoring mode of CORDIC.

The method described was for circular coordinates which can be extended for hyperbolic and linear coordinates as well using Eqn. 2.22 and 2.23 respectively as shown in Fig. 2.18.

$$x_{i+1} = \cosh(\delta_i)(x_i - \alpha * y_i * 2^{-i}) \tag{2.22}$$

$$y_{i+1} = \cosh(\delta_i)(y_i + \alpha * x_i * 2^{-i})$$

$$z_{i+1} = z_i - \alpha * \delta_i$$

where, $\delta_i = tanh^{-1}(2^{-i})$.

$$\begin{aligned} x_{i+1} &= x_i \\ y_{i+1} &= y_i + \alpha * x_i * 2^{-i} \\ z_{i+1} &= z_i - \alpha * 2^{-i} \end{aligned} \tag{2.23}$$

Follow the same procedure as those for circular rotation, yields different results which are summarized in Table 2.2. The final values of *x* and *y* in the table must be scaled by appropriate scaling factor or the initial values can be pre-scaled to avoid a final multiplication as described earlier. So far, the discussion has centered around where one of the initial condition was 0, either *x* or *y*. Table 2.2 consists of functions such as e^x which are derived starting with non-zero initial conditions and these can be easily inferred from standard trigonometric identities and shall not be discussed in this section. In vectoring mode, Eqn. 2.20 is modified to Eqn. 2.24 and 2.25 to reflect 'constant distance' (points that lie along the curve that represents the rotational transformation) in hyperbolic and linear coordinates respectively.

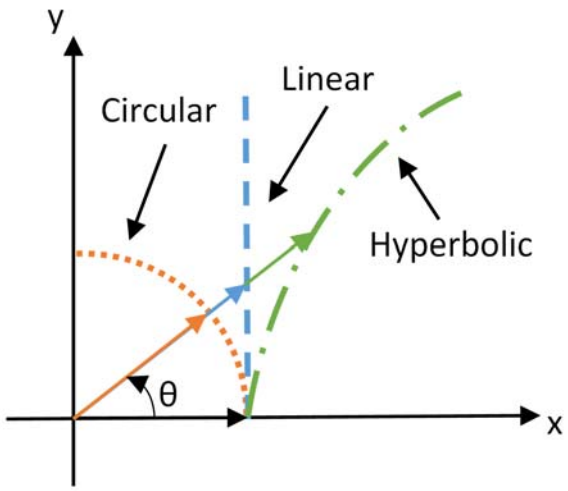


Figure 2.18: CORDIC for circular, linear and hyperbolic rotation

$$\sqrt{{x_1}^2 - {y_1}^2} = \sqrt{{x_i}^2 - {y_i}^2} \tag{2.24}$$

$$x_0 = x_i \tag{2.25}$$

Rotation	*Mode*	*Initial Value*			*Final Value*		
		x	*y*	*z*	*x*	*y*	*Z*
Circular	*Rotation*	1	0	0	*cos(θ)*	*sin(θ)*	*θ*
Circular	*Vectoring*	a	b	0	$\sqrt{a^2+b^2}$	0	$tan^{-1}\left(\frac{b}{a}\right)$
Linear	*Vectoring*	a	b	0	a	0	$\frac{b}{a}$
Hyperbolic	*Rotation*	1	0	0	*cosh(θ)*	*sinh(θ)*	*θ*
Hyperbolic	*Rotation*	a	a	0	ae^{θ}	ae^{θ}	*θ*
Hyperbolic	*Vectoring*	a	b	0	$\sqrt{a^2-b^2}$	0	$tanh^{-1}\left(\frac{b}{a}\right)$
Hyperbolic	*Vectoring*	$a+1$	$a-1$	0	$2\sqrt{a}$	0	$\frac{1}{2}\ln(a)$

Table 2.2: CORDIC functions

The final aspect for consideration is the range and convergence of CORDIC using hyperbolic and linear rotation. The maximum rotation angle for hyperbolic transformation is 60.4739□ as obtained from Eqn. 2.26. It must me noted, for hyperbolic transformation, unlike circular, the iteration index *i*, starts from 1 and not 0, because *tanh* inverse is ∞ when argument is 1.

$$\theta_{max} = \sum_{i=1}^{n} \delta_i = \sum tanh^{-1}\frac{1}{2^i} \tag{2.26}$$

The method does not converge at all points within this range because the sum of subsequent iteration angles adds up to less than the angle of current iteration. This means there are gaps within this range where CORDIC algorithm for hyperbolic function will not converge to.

$$\delta_i = tanh^{-1}\frac{1}{2^{i-1}} \geq \sum_{j=i+1}^{\infty} \delta_j = \sum_{j=i+1}^{\infty} tanh^{-1}\frac{1}{2^{j-1}} \tag{2.27}$$

This can be fixed by repeating certain iteration, namely 4, 13, 4j+1. Without these repetition, the scaling factor $K = 1.20514$ as number of iterations $n \rightarrow \infty$ and is obtained by solving Eqn. 2.28. With

repeating above-mentioned iterations for uniform convergence, scaling factor K is obtained by solving Eqn. 2.29 and equals 1.20753 as $n \to \infty$.

$$K = \prod_{i=1}^{n} \cosh(\delta_i) = \prod \cosh\left(tanh^{-1}\left(2^{-i}\right)\right) = \prod \frac{1}{\sqrt{1-2^{-2i}}} \tag{2.28}$$

$$K = \prod_{i=1}^{n} \frac{1}{\sqrt{1-2^{-2i}}} * \prod_{i=3*j+1}^{n} \frac{1}{\sqrt{1-2^{-2i}}} \tag{2.29}$$

This concludes the discussion on CORIC. As powerful as the algorithm is and with applications far reaching the ones discussed here, it does not suit well for the intended application of real-time control on a floating-point processor, which require guaranteed convergence for complete input argument range and high adherence to actual value, all which CORDIC fails to achieve. In the following sub section, polynomial based implementation for these transcendental functions are considered starting with Taylor series.

2.3 Polynomial Series

Polynomial series are finite sequence approximation of a function and are of the form, *P(x)/Q(x)* where, *P(x)* and *Q(x)* are equal to $\sum a_n x^n$. In their finite form, they approximate the actual function. When the number of terms in this sequence tend to infinity, they usually converge to the Taylor series of the function which is defined as follows for function *f(x) centered at x = a*:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{2.30}$$

where, $f^{(n)}$ is the n^{th} order differential of *f*. If the series is centered around *x = 0*, then it is called the Maclaurin series.

a) Taylor Series: The Taylor series unlike other finite series discussed later in this section, represents the actual function in a polynomial form and is not an approximation. But for practical implementation, the series is usually terminated in a finite number of terms and this

approximation leads to error in the result. Consider the Taylor series for *sin(x)* centered around $x = 0$, obtained from Eqn. 2.30:

$$f(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} - \cdots \tag{2.31}$$

Figure 2.19 shows the result of *f(x)* evaluated till, 1st, 3rd and 5th order along with *sin(x)*. As the order increases the function *f(x)* rapidly converges to *sin(x)*. The relative and actual error are shown in Fig. 2.20 for the range [0, π/2] and it increases as *x* moves farther from the center of the Taylor series polynomial which was at $x = 0$. Alternatively, the Taylor series could have been centered around the mid-point of the range at x = π/4. This would have led to reduction in peak error at the corner point, but no longer will the coefficients of even powered terms be zero. This leads to increased computations, effects of which will be same as increasing the order of polynomial evaluated in Eqn. 2.30. With the Taylor series centered at $x = 0$, the maximum relative error occurs

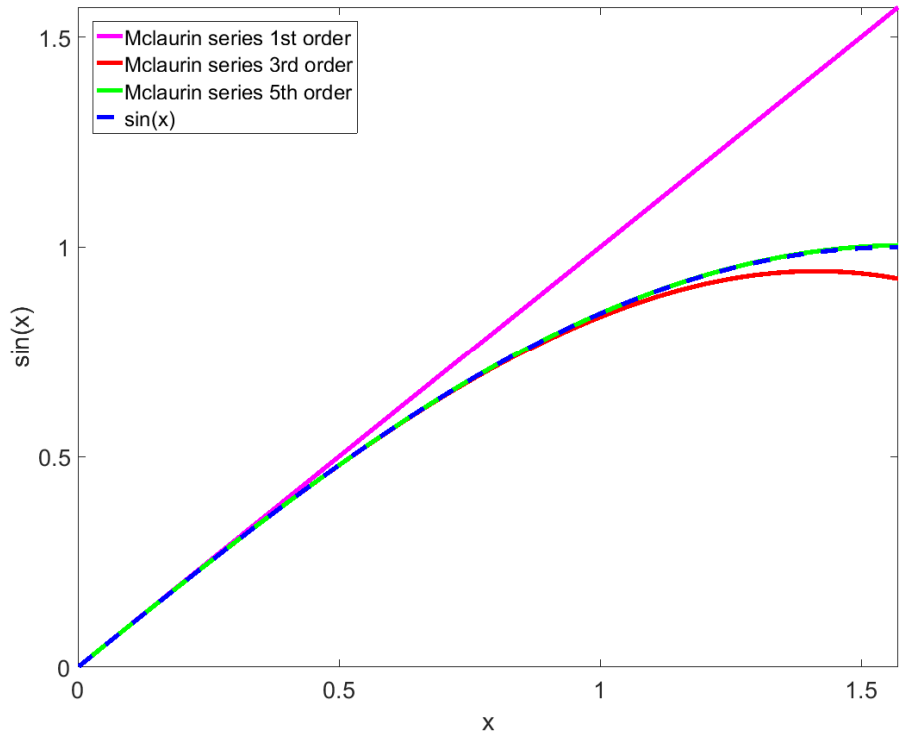


Figure 2.19: McLaurin series of *sin(x)* evaluated at 1st, 3rd and 5th order

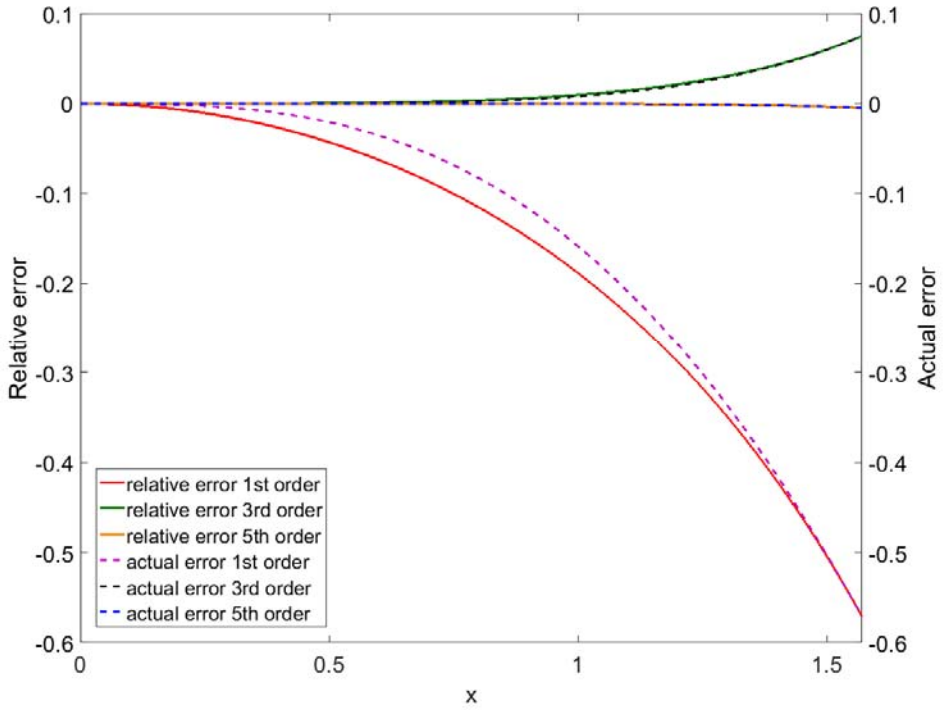


Figure 2.20: Relative and actual error of *sin(x)* for polynomial of order 1, 3 and 5

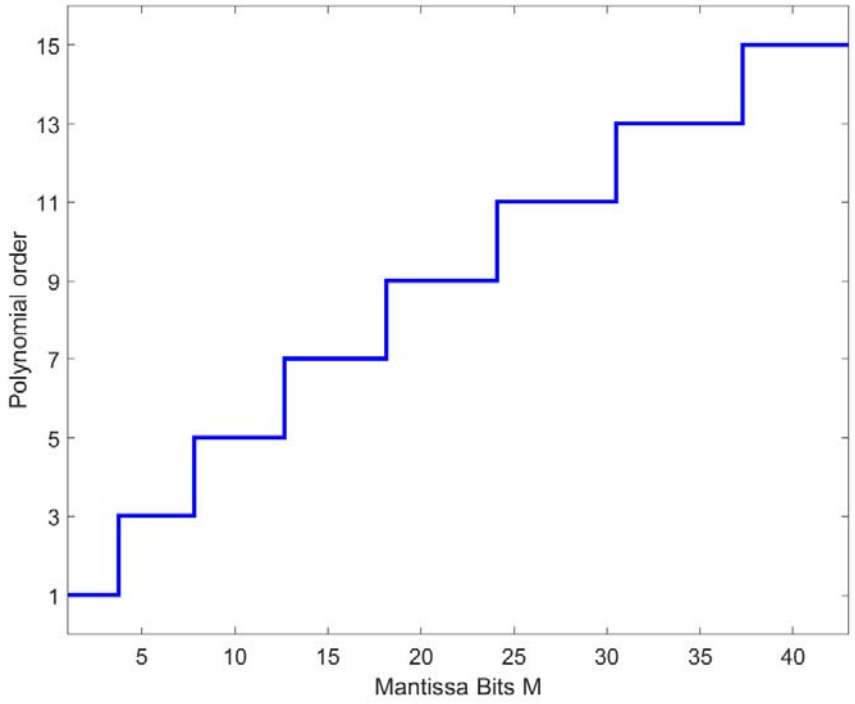


Figure 2.21: Order of polynomial vs Mantissa bits for zero relative error

at *x = pi/2*. Following procedure highlighted in earlier section, the order of polynomial of Eqn. 2.31 to be computed, such that relative error is zero as function of mantissa bits *M* is shown in Fig. 2.18. For single precision floating-point the polynomial needs to be evaluated till x^9 for zero relative error.

Extending the same analysis to e^x, Taylor series centered at *x = 0 is:*

$$f(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} \ldots \quad (2.32)$$

To limit the error, the range is limited to [0, 1]. For *x* outside this range it is split to integral and fractional component, integral part computed using LUT and fractional part using Eqn. 2.32, as mentioned in Section 2.2 subsection *e*. The McLaurin series evaluated at order 1 through 3, the relative and absolute error, and the Polynomial order as function of mantissa bits for zero relative error is shown in Fig. 2.22 through 2.24. For single-precision float its requires computing up to x^8 for relative error to be zero.

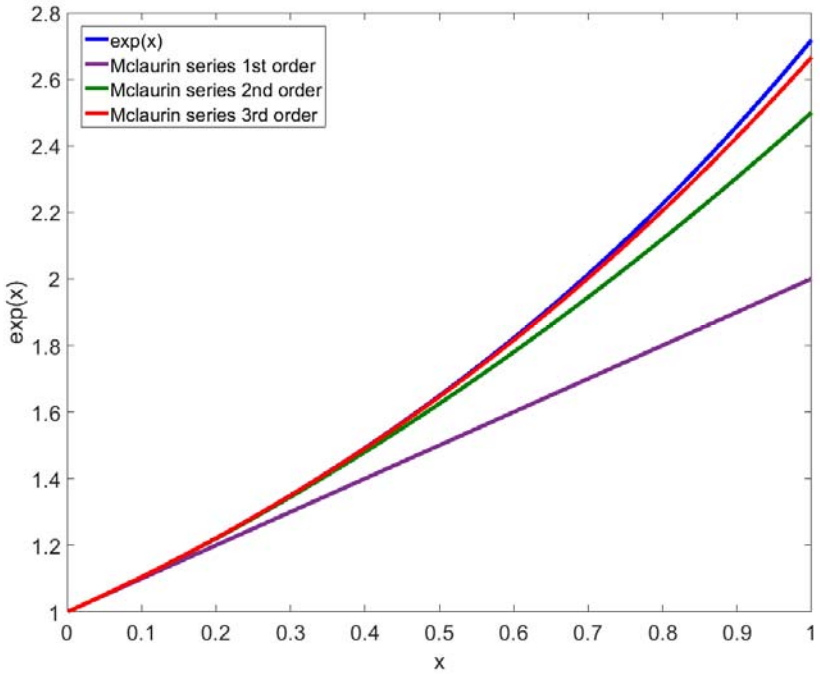


Figure 2.22: McLaurin series of e^x evaluated at 1st, 2nd and 3rd order

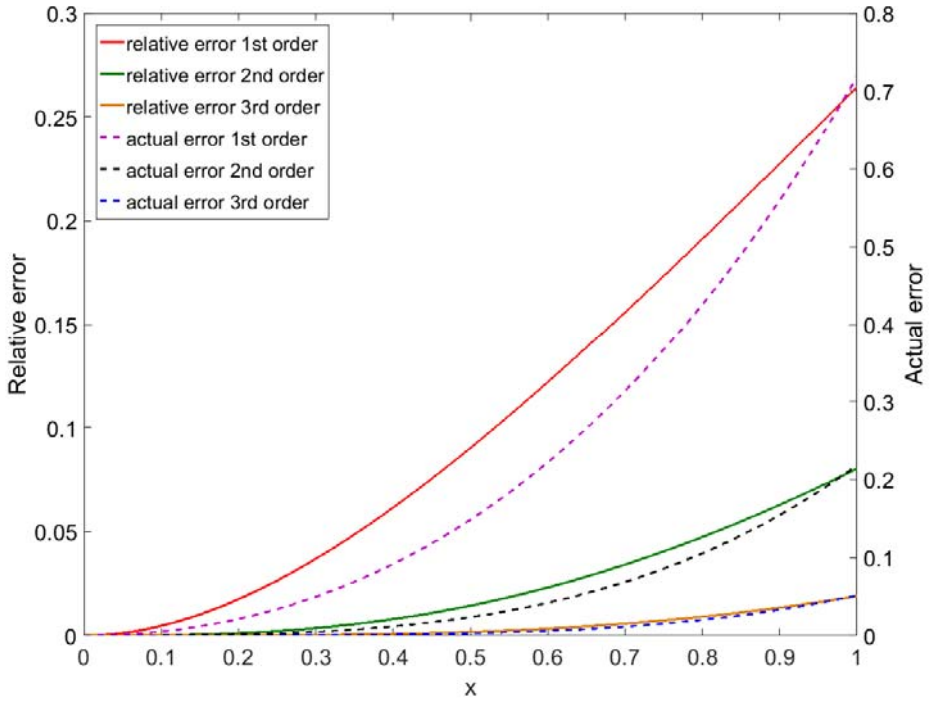


Figure 2.23: Relative and actual error of e^x for polynomial of order 1 through 3

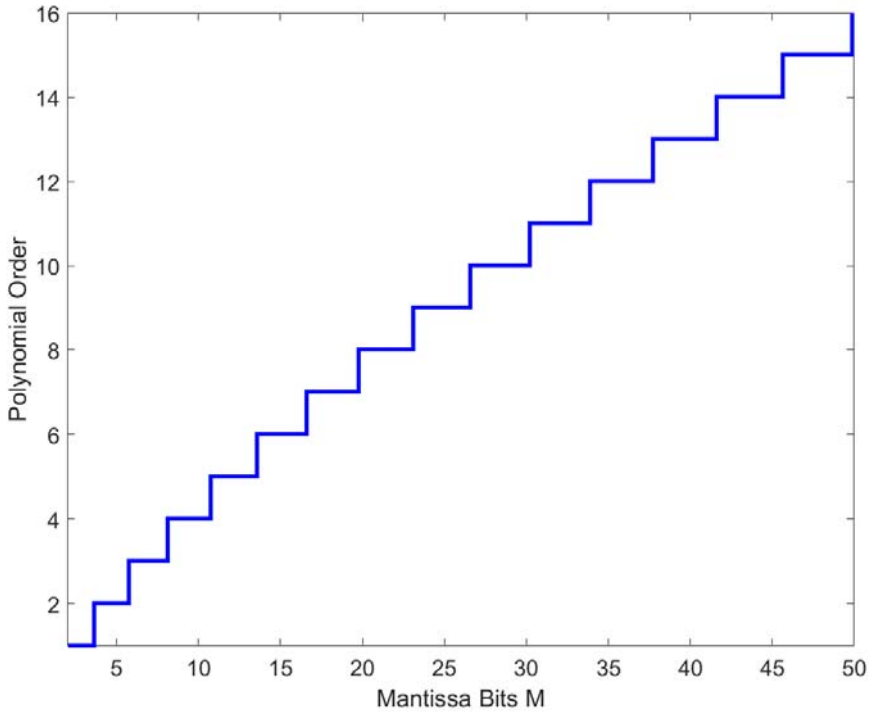


Figure 2.24: Order of polynomial vs Mantissa bits for zero relative error

For both *sin(x)* and e^x, the result of the polynomial converges to the actual result with a few orders of *x*. This is due to the exponentially increasing coefficients in the denominator of each term. This means as the order increases the effect of that order decreases since $x^n \ll n! \, as \, n \to \infty$. For functions whose effect of higher order terms in its Taylor series does not decrease rapidly the order of *x* required to be computed for zero relative error tends to be impractically larges as is the case with $tan^{-1}(x)$.

The Taylor series for $tan^{-1}(x)$ centered at $x = 0$ is:

$$f(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} \ldots \tag{2.33}$$

for $|x| \leq 1$. As $x \to \pi/2$, $tan^{-1}(x) \to \infty$. To avoid this region, the range is limited to $x \in [0, \pi/4]$. For $x \in (\pi/4, \pi/2]$, it is evaluated using the following identity:

$$atan(x) = \frac{\pi}{2} - \text{atan}(\frac{1}{x}) \tag{2.34}$$

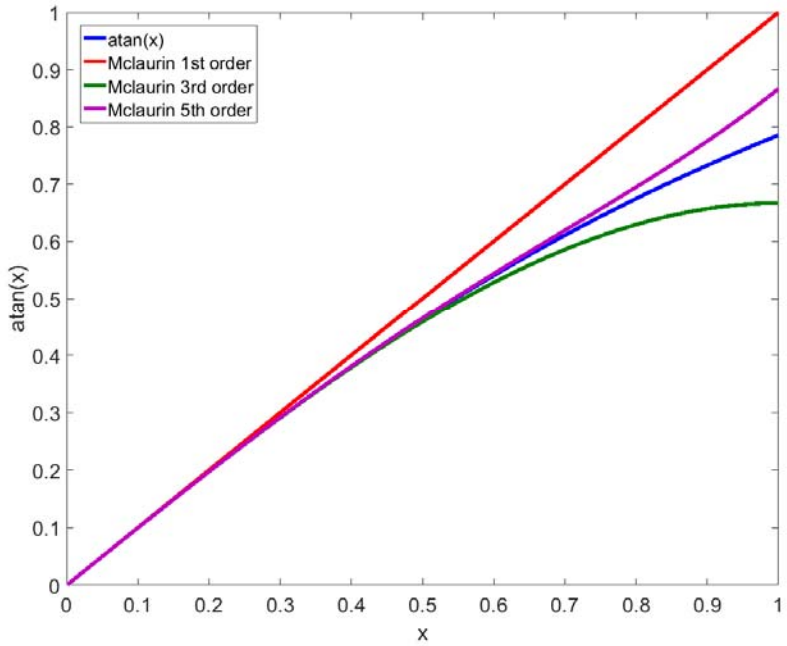


Figure 2.25: McLaurin series of $tan^{-1}(x)$ evaluated at 1st, 3rd and 5th order

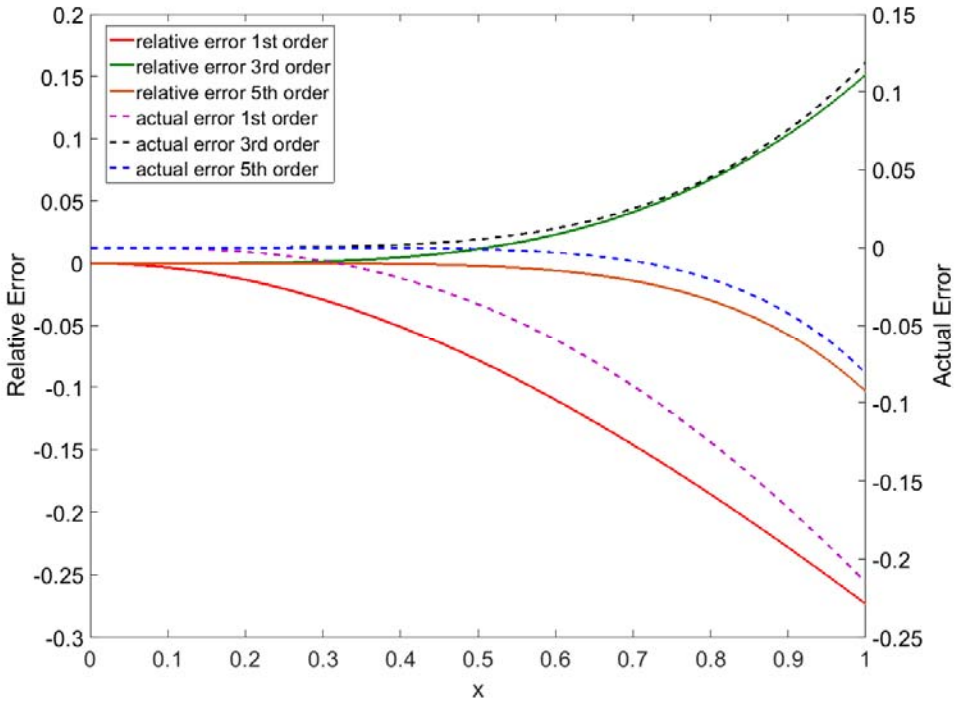


Figure 2.26: Relative and actual error of $tan^{-1}(x)$ for polynomial of order 1 through 5

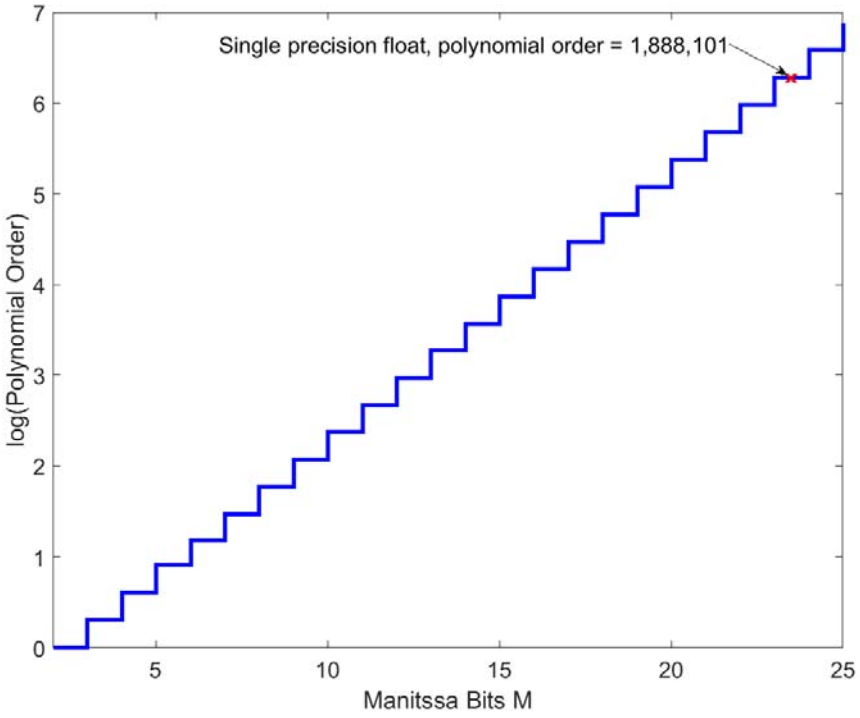


Figure 2.27: Log of order of polynomial vs Mantissa bits for zero relative error

The result of Eqn. 2.33 computed up to 1st, 3rd and 5th order is shown in Fig. 2.25. The relative and actual error for each of those series is shown in Fig. 2.26. The maximum relative error as with previous cases occur at the farthest point from where the series is centered around, i.e. at *x = 1*. The relative error reduces with increasing order, but the rate of reduction decreases exponentially. This is dues the fact that unlike the previous two cases of *sin(x)* and e^x the denominator does not increase at a rate faster than the numerator. The order of polynomial up to which Eqn. 2.23 needs to be evaluated for zero relative error for *M* mantissa bits floating-point system is shown if Fig. 2.27. It must be noted the vertical axis is in log scale. The order of polynomial for single precision floating-point is 1,888,101 which is impractical.

b) Chebyshev Polynomial: Finite order Taylor series is one of the many polynomials that can approximate the actual function and it is 'ill-conditioned' because the error builds up rapidly around the corner points. In the first method considered, LUT approach, a finite set of evenly spaced points were considered and for values in between, they were obtained by linear interpolation. Instead of using a linear interpolant, a cubic polynomial could have been used and it would have required roughly an eight of table size needed to implement *sin(x)* for the same precision. These are just two of an infinite number of polynomials that can reasonably approximate the actual function. Some of the commonly used interpolants such as Lagrange or Hermit interpolant could also be used.

Of all the polynomials whose coefficient of the leading order terms, i.e. the coefficient of highest order term is 1, Chebyshev polynomial has the least infinity norm $||x||_\infty$, i.e. the least absolute error over the range of the polynomial. Chebyshev polynomial comes under the category of orthogonal polynomials for which:

$$\int_{-1}^{1} w(x)P_i(x)P_j(x)dx = \begin{cases} k; & i = j \\ 0; & i \neq j \end{cases} \tag{2.35}$$

i.e. inner product of polynomials of two different function are zero for a weighting function w(x). If $P_i(x)$, is replaced with the actual function *f(x)* that is being approximated, coefficient *k* in Eqn. 2.35 indicates how closely the polynomial resembles *f(x)*, similar to projection using dot product for geometric vector.

Some of the important properties of this class of polynomial is that the higher order polynomials can be generated using recurrence formula, from the knowledge of lower order terms. In addition, these polynomials of order *N*, has *N* zeros and *N+1* extremas. The minimum error is achieved by instead of choosing uniformly distributed points as was the case with LUT, the points for Chebyshev polynomials are chosen at *x* given by Eqn. 2.36.

$$x = \cos(\phi) = \cos(\frac{k\pi}{N}) \tag{2.36}$$

The Chebyshev polynomials are given in Eqn. 2.37.

$$\begin{gathered} T_n(x) = \cos(n\phi) = \cos(n\cos^{-1}(x)) \\ T_0(x) = 1 \\ T_1(x) = \cos\left(\cos^{-1}(x)\right) = \mathrm{x} \\ T_2(x) = \cos\left(2\cos^{-1}(x)\right) = 2\cos^2\left(\cos^{-1}(x)\right) - 1 = 2x^2 - 1 \\ T_3(x) = \cos(3\phi) = 2\cos(\phi)\cos(2\phi) - \cos(\phi) = 4\cos^3(\phi) - 3\cos(\phi) = 4x^3 - 3x \end{gathered} \tag{2.37}$$

The above series can be written using recursive formula as:

$$\begin{gathered} T_0(x) = 1 \\ T_1(x) == \mathrm{x}T_0(x) \\ T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \end{gathered} \tag{2.38}$$

The desired polynomial that closely approximates the actual function is given by sum of projection of the function on each of the Chebyshev polynomial:

$$P(x) = \sum_{n=0}^{maximum\ order} k_n T_n \tag{2.39}$$

where the coefficient *k* in Eqn. 2.39 for Chebyshev polynomial is computed from:

$$k_n = \frac{\int_{-1}^{1} w(x) f(x) T_n(x) dx}{c_n}$$
$$where, c_n = \begin{cases} \pi & n = 0 \\ \frac{\pi}{2} & n \neq 0 \end{cases} and\ w(x) = \frac{1}{\sqrt{1-x^2}} \tag{2.40}$$

Consider *sin(x)*, over the range [-π, π]. The polynomial is defined only within the range x ∈ [-1, 1]. The function *f(x)* is defined as *sin(πx)*, to satisfy the range condition. Computing the coefficients *k*, for order up to 5 using Eqn. 2.40 yields Table 2.3.

n	k_n
0	0
1	0.5692307
2	0
3	-0.6669167
4	0
5	0.1042824

Table 2.3: Chebychev coefficients for sin(x)

Substituting these coefficients and Eqn. 2.38 in Eqn. 2.39 results in Eqn. 2.41 which is the least error polynomial for *sin(πx), x* ∈ [-1, 1]. The range is scaled back to *x* ∈ [-π, π] in Eqn. 2.42. The Taylor series polynomial for the same order is given in Eqn. 2.43. The two polynomial seems quite similar on first look. Figure 2.28 shows the plot of error of Taylor series polynomial of the 5th order and Chebyshev polynomial of the same order. The maximum error for Taylor series polynomial is up to two magnitudes higher than the Chebyshev polynomial.

$$P(\pi x) = 3.0913928x - 4.7533148x^3 + 1.6685184x^5 \tag{2.41}$$

$$P(x) = 0.9840208x - 0.15330167x^3 + 0.0054523x^5 \tag{2.42}$$

$$P(x) = x - 0.166666667x^3 + 0.008333333x^5 \tag{2.43}$$

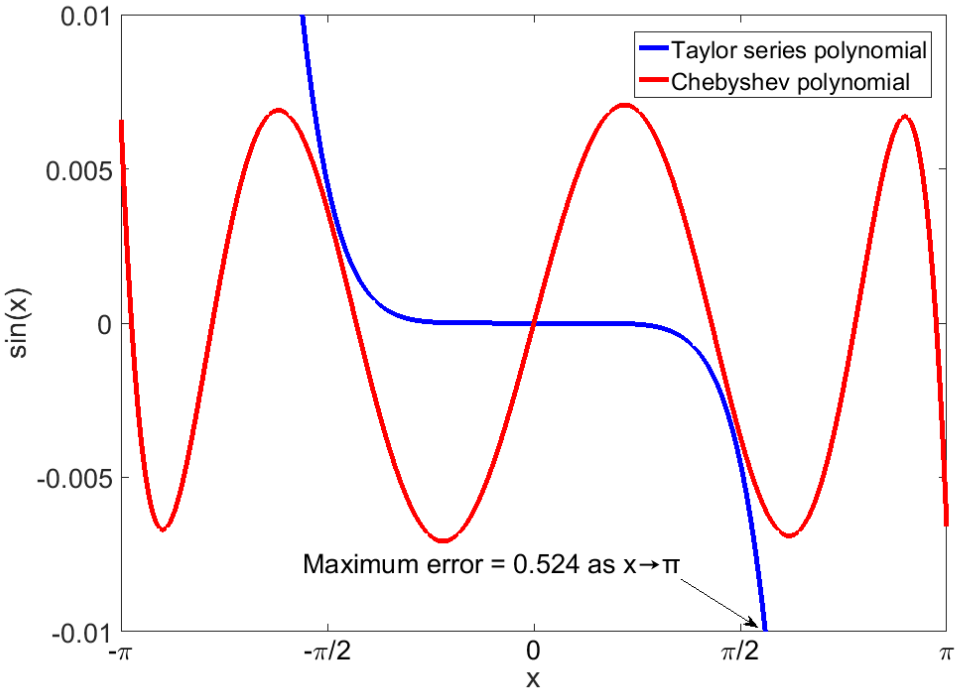


Figure 2.28: Error of Taylor series and Chebyshev 5th order approximation polynomial for *sin(x)*

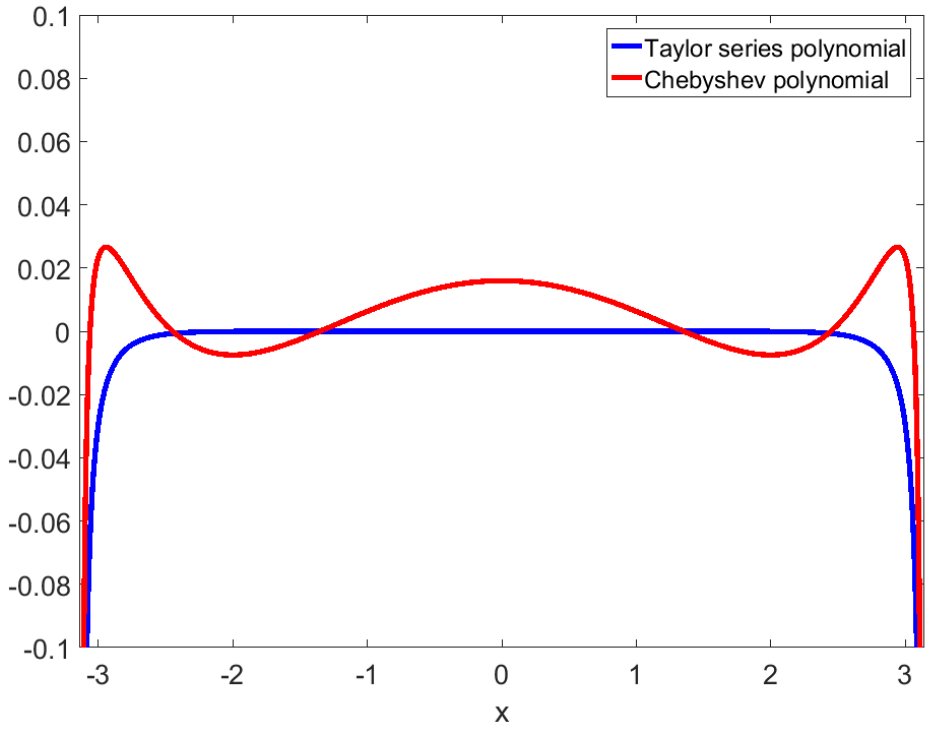


Figure 2.29: Relative error of Taylor series and Chebyshev 5th order approximation polynomial for *sin(x)*

Even though the maximum actual error for Chebyshev polynomial is significantly lower and the least of any polynomial of leading coefficient 1, the relative error is significantly high, higher than Taylor series polynomials as seen in Fig 2.29.

Due to high relative error, these do not qualify as a good enough polynomial for high precision math library, but they do help as a starting point for generating minimax polynomial discussed next, using Remez exchange algorithm.

c) Minimax Polynomial: These are another series of orthogonal polynomials. This address the limitation of Chebyshev polynomial, i.e. high relative error. Let δ be the maximum allowable relative error. From Chebyshev alternation theorem, for a polynomial for best uniform approximation for function *f(x)*, the *p(x)-f(x)* alternates sign *N*+2 times, where *N* is the order of the polynomial. This can be seen in Fig 2.28 where the error of Chebyshev polynomial alternates 7 times for a 5th order polynomial. Using this principle, the relative error, for a polynomial with minimum relative error, must alternate sign *N*+2 times. This is expressed as:

$$W(x_i) * (f(x_i) - P(x_i)) = (-1)^i \delta \tag{2.44}$$

where, x_i are the points of extrema, $W(x_i)$ is the weighting function, in this case for minimized relative error, it is the inverse of actual function value, $f(x_i)$ and $P(x_i)$ is the desired polynomial. The RHS of the equation, is the maximum relative error tolerance and it alternates sign at every subsequent extrema.

Consider the function *sin(x)* for the range [0, π/2]. Eqn. 2.44 for this becomes:

$$\frac{1}{sin(x_i)} * (sin(x_i) - P(x_i)) = (-1)^i \delta \tag{2.45}$$

From Eqn. 2.42 and 2.43, the form of the final polynomial becomes self-evident and Eqn. 2.45 can be reduced such that $P(x_i)$ has all orders of *x*:

$$\frac{1}{sin(x_i)} * (sin(x_i) - x_i - {x_i}^3 P({x_i}^2)) = (-1)^i \delta \tag{2.46}$$

replacing ${x_i}^2$ by y and range of $y \in [0, \pi^2/4]$:

$$\frac{\left(\frac{sin(\sqrt{y_i})}{{y_i}^{3/2}} - \frac{1}{y_i} - P(y_i)\right)}{\frac{sin(\sqrt{y_i})}{{y_i}^{3/2}}} = (-1)^i \delta \tag{2.47}$$

Consider a 3rd order polynomial for $P(y_i) = a + by_i + c{y_i}^2 + d{y_i}^3$, where y_i are the points of extrema. If these points are known, the Eqn. 2.47 becomes a simple 5 variable equation in *a, b, c, d and* δ and there will be 5 equations according to Chebyshev alternation theorem. Since these points are unknown, they are determined iteratively using Remez exchange algorithm as follows:

1. Initial estimate of extrema points y_i can be *N*+2 equally spaced points in the interval of $y_i \in [0, \pi^2/4]$. Alternatively, these can be chosen as the points of Chebyshev extrema from Eqn. 2.36:

$$y_i = \frac{\pi^2}{4} \cos\left(\frac{(i-1)\pi}{N+1}\right);\ i \in [1, N+2] \tag{2.48}$$

2. Starting with these points solve Eqn. 2.47 for *a, b, c, d and* δ, 5 variables, 5 linear equation.
3. Compute the relative error function *R(m)*, i.e. LHS of Eqn. 2.47 for collection of *M* points $\{m_1, m_2, \ldots, m_M\}$, *M* much greater than order *N* using the polynomial with currently computed coefficients. Update the y_i using the following criterion:
    a. If relative error function at all points are lesser than δ or the sign alternation is less than *N*+2, terminate since it has achieved the best possible polynomial. If $\delta \leq$ expected relative error tolerance, this is the final polynomial, else increase the polynomial order and redo the whole procedure.

b. If not, for each point $m \in M$, set $y_i = m$, if *sign(m)* $\neq$ *sign*(y_{i-1})and |*R(m)*| is Max{*R(x)* $\forall\, x \in \{M, (y_{i-1}, y_{i+1})\}\}$, i.e. points within the relative error functions that are local extrema such that these extrema alternate signs between subsequent y_i.

Following above procedure, the coefficients obtained for minimum relative error for a single precision floating-point system for sin(x) is given in Table 2.4. The final polynomial resubstituting $y = x^2$ is given in Eqn. 2.49.

$$P(x) = x - 0.1666665668x^3 + 0.8333025139\text{x}10^{-2}x^5 - 0.1980741872\text{x}10^{-3}x^7 + 0.2601903036\text{x}10^{-5}x^9 \quad (2.49)$$

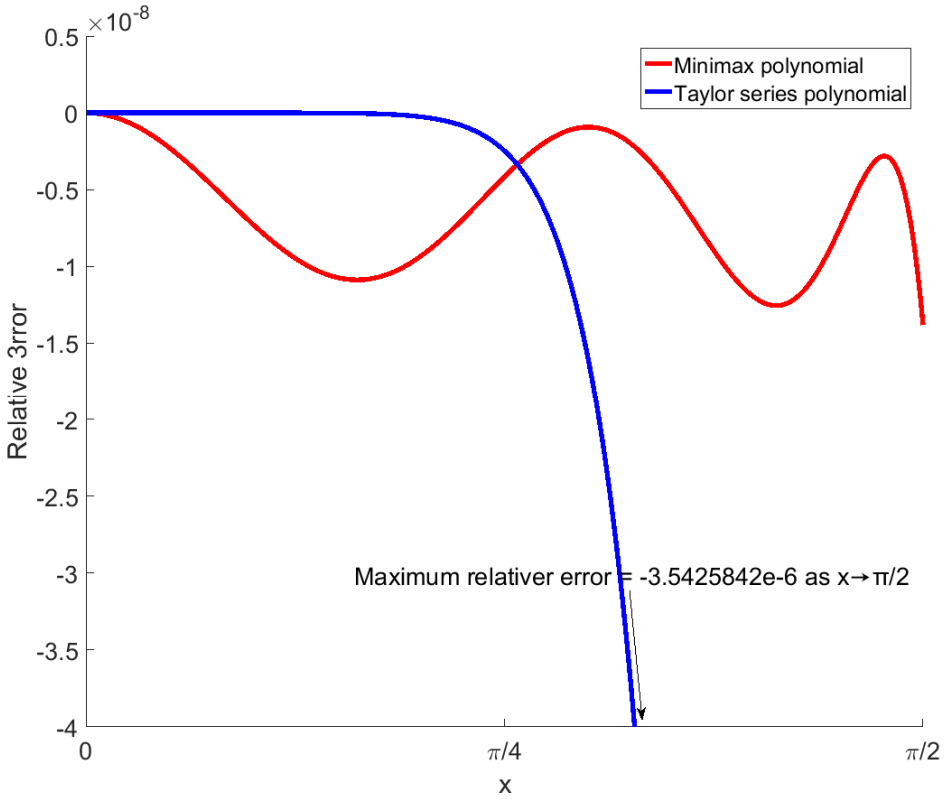


Figure 2.30: Relative error of 9th order Taylor series and Minimax polynomial for *sin(x)*

The minimax polynomial in Eqn. 2.49 closely resembles the Taylor series polynomial but the relative error is significantly smaller as shown in Fig. 30. This is one of the major limitation of minimax polynomial, i.e. it is highly sensitive to even minor changes to its coefficient. Thus, the Remez exchange algorithm must be carried out with extreme care, especially with regards to base of the floating-point system. For instance, the relative error performance for polynomial derived for double precision floating-point system will be drastically different to one computed specifically for single precision floating-point system.

Coefficient	*Value*
a	-0.1666665668E+0
b	0.8333025139E-2
c	-0.1980741872E-3
d	0.2601903036E-5

Table 2.4: Minimax polynomial coefficients for sin(x)

Function	*Maximum Execution Cycles on F28335*	
	Assembly	*Generic C*
sin(x)	77	331
cos(x)	80	348
tan(x)	71	743
e^x	70	792
ln(x)	89	1131
x^y	221	1705
sinh(x)	106	1235
cosh(x)	104	1241
tanh(x)	108	1425
asin(x)	93	1105
acos(x)	93	1107
atan(x)	103	1432
atan2(y, x)	99	1441

Table 2.5: Math Library performance

Using the method above, the minimax polynomials are derived for all the functions listed in [7]. The polynomial used, and the implementations details for single and double precision system, and details about input argument range reduction is available in [11]. The complete collection of Assembly optimized implementation for Texas Instruments C28x devices with single precision floating-point unit as well as generic C89 code for single and double precision system for these functions are made available as part of the downloadable software package. The performance of these functions in terms of number of instruction cycles on TI TMS320F28835 while running from RAM without being interrupted is presented in Table 2.5.

2.4 Conclusion

This chapter introduced the various methods of implementing common math functions. The look up table approach with linear interpolation is simplest of all the solutions as well as the fastest. This comes at a cost of extremely high memory requirements for the look-up table, in most cases being impractical. Despite this, it remains a powerful solution, if compromise can be made of the precision of the result. Following this, CORDIC algorithm was explored. These are useful in fixed point systems without a dedicated multiplier or limited and performance expensive multiplier. In case of floating-point system, they have an inherent loss of precision that cannot be done away with and unlike polynomial based algorithm increasing order (in this case iterations) does not reduce this problem. In addition to poor precision, this algorithm takes significant execution time on a sequential machine like a processor and hence are better suited for parallel architecture such as an FPGA. Finally, a series of polynomials were discussed, namely Taylor series, Chebyshev and Minimax polynomial. Taylor series has zero error, if all the elements of the sequence are computed but this is practically impossible since the series extends all the way to ∞, and the rate of convergence is in most cost not fast enough where higher order terms can satisfactorily be ignored. So, this finds absolutely no application in real world. With respect to minimizing error, it can be theoretically established that Chebyshev polynomial gives the smallest infinity norm of all

polynomials of same order with leading coefficient 1. The Chebyshev polynomial was derived for *sin(x)* and was show that the error performance was orders of magnitude superior to Taylor series. But the relative error was as worse or much worse than Taylor series. This was since Chebyshev polynomial distributes the error uniformly, that means at very small function values, the error is relatively much higher than the function value itself, even though the overall error is small. To overcome this, minimax polynomials were introduced along with Remez Exchange algorithm for iteratively solving for the coefficients of the minimax polynomial. As was observed, this polynomial though has more error than Chebyshev polynomial of same order, it has much smaller relative error. This was finalized as the most practical solution for high precision, high performance math libraries. All the general math functions were implemented for TI TMS320F28335 processor using fully-optimized assembly as well as a generic C and made available as part of the accompanying software.

Chapter 3

Programming language with matrix support

This chapter aims to provide an overall understanding of compiler without delving too much into the technical aspects for which this reference [12] would suffice. The proposed numerical simulation platform is model based, to ease the design process for real-time control. The model based approach is a powerful tool especially when dealing with computational loops involving memory based elements. It becomes extremely tedious to figure out the order of evaluation and to debug and update these when designed in a code based platform. But code based design does come in handy in case where iterative loops or recursive calls must occur. Arithmetic loops are rather easily resolved in code based platform as opposed to model based. In addition to this, even in a model based platform, there is always an element of coding involved for instance, the gain value could be an expression itself or the magnitude of constant block could have a matrix operation for its initialization. Though these are single line operations, the principle extends to multiline code editor as well.

A compiler is a tool that translates a piece of code from one language to another, usually from a higher-level language such as C, C++, C#, Java, Python etc., to a lower level language such as .NET CIL, Java Byte Code or platform specific machine code. This chapter covers the various aspects of a code compiler, which is one of the pillars of the simulation platform. It is followed by details of the lexical analyzer, parser, semantic analyzer, CIL generator, up compiler and the code

evaluator designed for this platform. Most compilers (the term compiler is loosely used here and encompasses all the components mentioned earlier), have a primary goal of generating optimized low-level code and these involve tremendous effort to build. For simplicity, in this platform, the aim of the compiler was to convert high-level code using the proposed programming language with inbuilt matrix support to a lower level common intermediate language (CIL) and then up compile this to C# or Java or other high-level language and use the compiler of those platform to down compile to machine code. This helps in three aspects, first it makes the compiler platform independent if the language it is up compiled to can run on that platform. Second, it makes it possible to provide translation to multiple other languages with ease. Third and most important of all, it allows taking a loose approach to optimization of generated code, since the target compiler developed with huge amount of resources is going to be able to generate well optimized code. A rigorous analysis all the functionality implemented in this software is beyond the scope of this thesis and it is recommended to refer to the complete reference manual.

3.1 General Compiler Architecture

Each compiler is unique and how each of these components are implemented vary drastically, often multiple components are combined and there is no clear demarcation between these components. In any case, just for a brief theoretical background the following section tries to highlight the major functions involved in the compilation of source code.

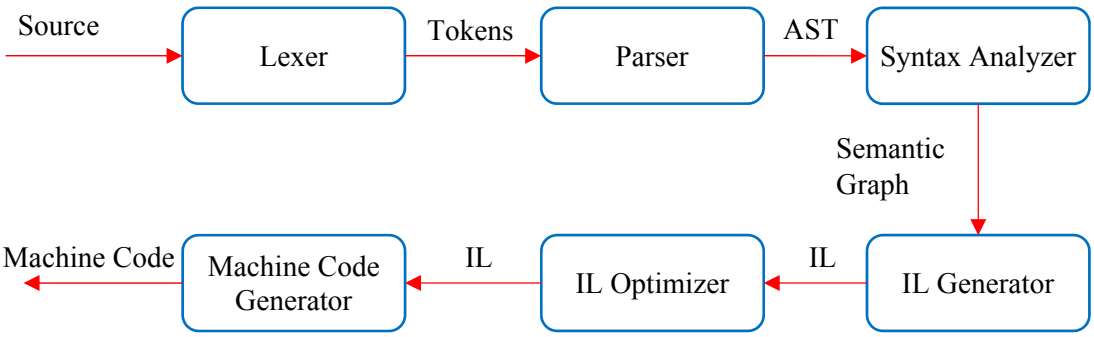


Figure 3.1: General Compiler Architecture

The major components for a general compiler is shown in Fig 3.1. Most of the components have evolved from our understanding of day to day processing of written language. When a written text such as this is read, the text is broken into components based on number of commonly understood demarcations such as period '.', commas ',', semi-colon ';', spaces etc. These act as a demarcation to identify components of a sentence that are different from one another. Similar to this, programming language follow these same patterns of demarcation, usually white space is used to signify separation between texts, brackets are used to signify separation between binary operators, semicolon is used in languages such as C to signify end of a line and in many cases their functionality is context based as will be discussed later. The purpose of a Lexical analyzer is to break all the "sentences" of the source code, into individual elements called 'tokens' based on a pre- determines delimiter. Figure 3.2 shows the output of passing a single line C source code through a lexer. It must be noted lexer is dependent on the language. The same characters might be broken differently for different languages. For instance, in C '==' i.e. two consecutive 'equal to' symbol is considered a single token and it performs the logical comparison of element on its right to element on its left, while '=', i.e. single 'equal to' is also considered a single token and it performs assignment of contents on the right to the element on the left. On the contrary in a language like VB.NET, both comparison and assignment use the same operator '=', i.e. single 'equal to'. So, using '==', i.e. double 'equal to' will be broken into two different tokens, though it will throw and error since it has no meaning in VB.NET.

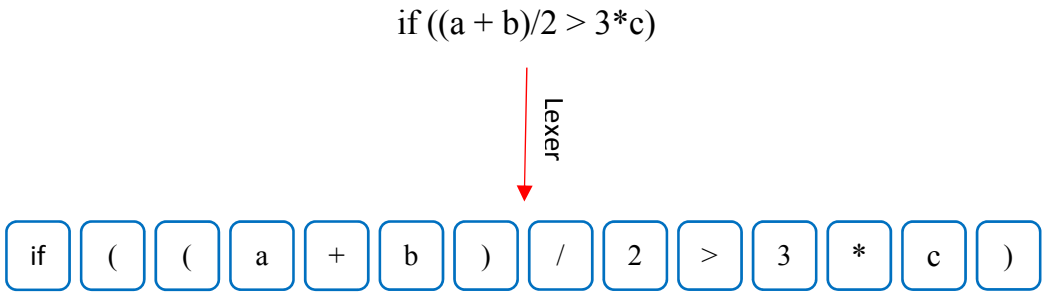


Figure 3.2: Output of Lexical analyzer for C

The tokens outputted by a lexer is not merely broken-down string. They have a bit more information about the type of the token such as what class does the operator fall under, such as arithmetic or logical or comparison operator, where a literal is a number or a reference to a variable etc. as shown in Fig. 3.3. This cursory classification helps speed the process of Parser, since it does not have to analyze the string anymore, but rather work on processed object.

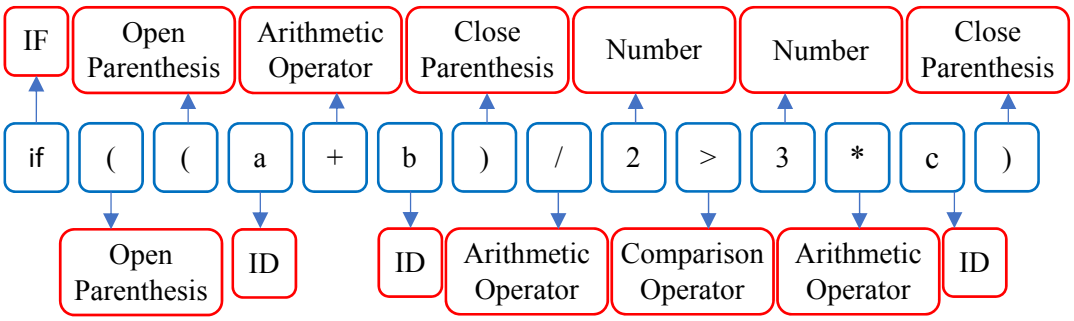


Figure 3.3: Lexer token association

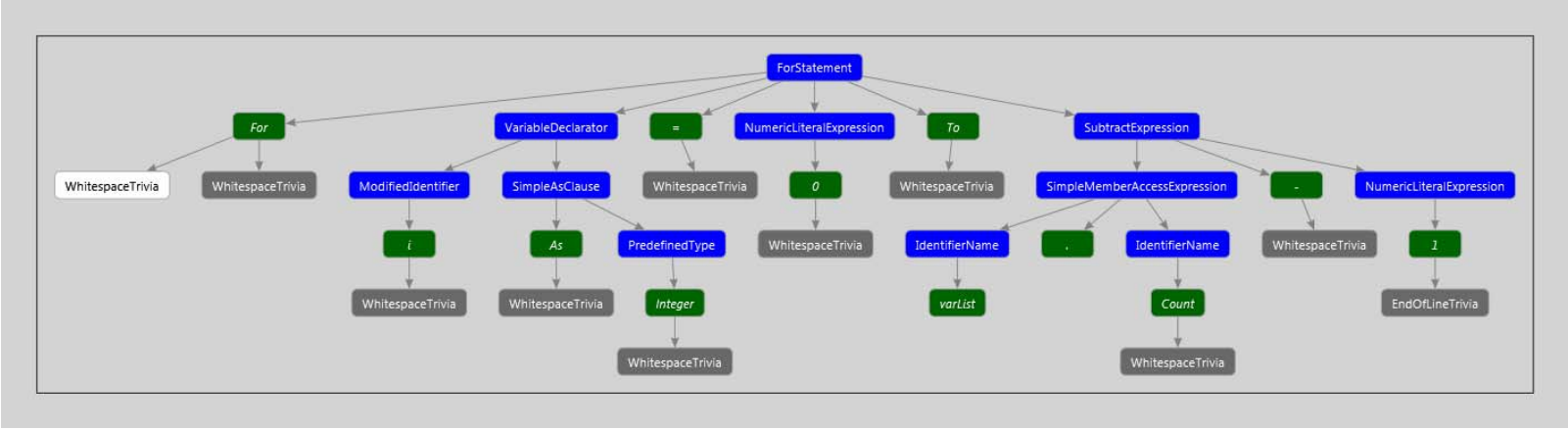
A lexer may not be able to resolve a token correctly in some cases where the function of a literal varies based on the context (context sensitive grammar). An example of this is the '=', single 'equal to' operator in VB.NET where it could either be a comparison operator as in the case: *if a = b then*. Here the '=' is a comparison operator as opposed to a line: *Dim a = b + c* where, '=' is an assignment operator. In these cases, the following parser resolves these tokens based on their context.

The primary task of the parser is to convert the linear token stream into a "grammatically sorted" hierarchy called the abstract syntax tree (AST). This is usually done by grouping collection of tokens and placing them under their logical blocks and the number of elements within a block are gradually reduced while traversing along the depth of the tree until just end terminal nodes remain, i.e. nodes that do not have any other terminal nodes that can follow it along the depth of AST. The rules of generating this tree are varied and detailed analysis can be found in any book on compiler architecture. This chapter will not cover these classical methods since the methodology

used later varies drastically from commonly used methods. AST for the VB.NET code line: *For i As Integer = 0 To varList.Count - 1* from Roslyn Syntax Analyzer is shown in Fig. 3.4. As it can be seen, the code line is broken down into multiple logical components, which in turn contains other components. For instance, the 'for' block is broken into 'for keyword', 'variable declaration', 'assignment operator', 'number literal', 'To keyword' and 'Arithmetic subtract expression'. The 'variable declaration' is in turn broken down to two components, 'variable identifier' and the 'As keyword'. The 'variable identifier' contains the name of the variable and with that this branch of the AST ends since the variable 'i' itself cannot be broken down into any further smaller components. In the same figure, the AST for code line: $f = a + b * c - d / e$ is shown. The right half is the binary expression tree of a simple arithmetic operation, with operators on top of the tree having the lowest priority and those in the bottom the highest. From above expression the product of 'b' and 'c' must occur first followed by adding the result to 'a' and the subtracting the result of 'd' divided by 'e'. The AST represents this order of precedence in a hierarchical manner. Simply tweaking the rules of the parse could lead to drastically different results. The whitespaces are of no significance in the AST and are usually ignored by the lexer when it converts input strings to tokens. Compilers which actively interact with the design environment to actively remove any extra spaces or add spaces between operator for more visual clarity etc., preserve these characters solely for reconstruction the standardized representation of input string from the AST. The functionality of input string can be reconstructed in most compiler from the AST by traversing it depth first and concatenating the string of each terminal node. It must be noted that depending on the parser, it may or may not chose to perverse tokens from the lexer. For instance, some parsers may choose to ignore grouping parenthesis since their information is embedded in the hierarchical structure of the AST.

AST in Fig. 3.4 was for just a single line of the source code, which itself is part of a bigger source code block. Figure 3.5 shows the syntax tree for a complete source code. Each line within

```
For i As Integer = 0 To varList.Count - 1
```




```
f = a + b * c - d / e
```

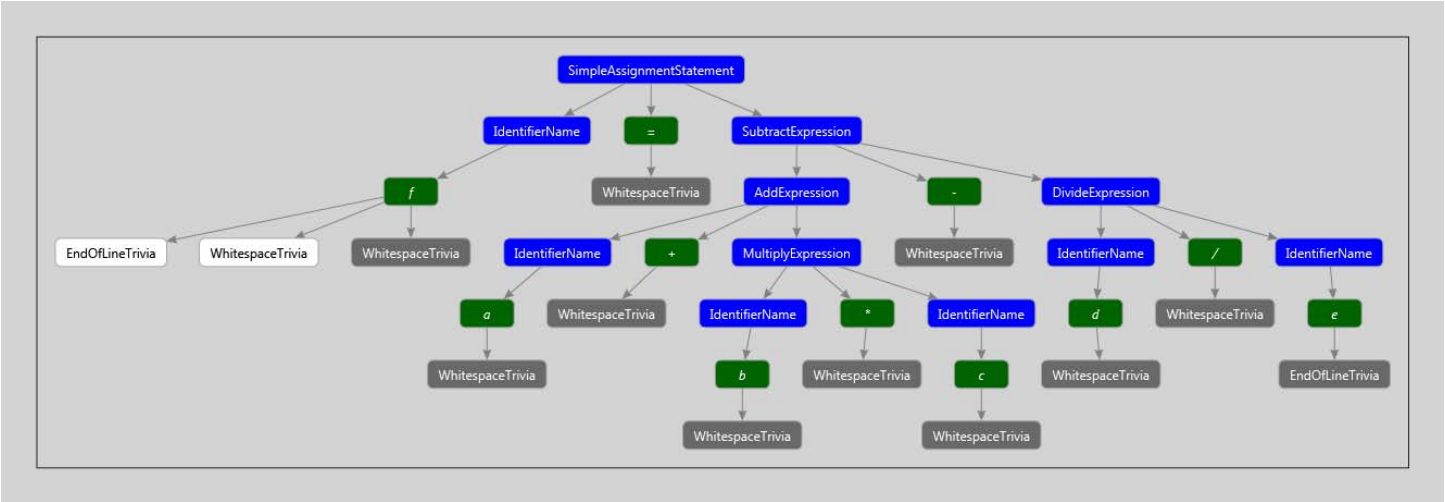


Figure 3.4: Directed Syntax Graph for single line of VB.NET Code

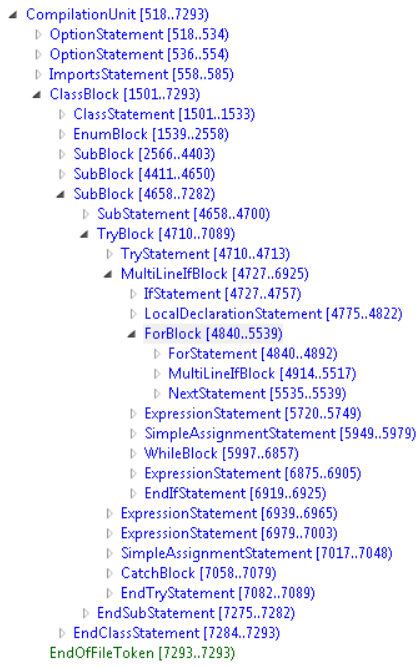


Figure 3.5: Syntax Tree

the syntax tree in Fig. 3.5 is comprised of its own AST. The 'for statement' for which AST was presented in Fig. 3.4 is part of the syntax tree in Fig. 3.5, enclosed within an 'if statement' which in turn is enclosed within a 'Try' block and that inside a 'function block' and so on till code entry point.

Each of the compiler units process its input and outputs a more refined code representation progressively moving closer to final machine code. In addition to this each unit emits other useful information. The lexer in addition to its primary output of tokens, in many cases outputs line and column number of each token in the source code. This information is used later to connect a source of compile-time or runtime error and warning to text within the source code. The semantic analyzer in addition to outputting semantic graphs also outputs any syntactic errors.

The semantic analyzer evaluates the legality of each connection in the AST. The conditions that are checked is dependent on the implementation, for instance a language that supports matrix operation can decide to throw an invalid size for an operation trying to add matrices of different sizes at compile time in which case it happens in this section or it could be postponed, to throw an exception at runtime. In strongly typed languages, AST performs type conformity. To do this all references are resolved to a definition. The output of semantic analyzer is semantic graph which is similar to AST, with add on information such as resolved references, resultant data type etc.

This semantic graph is taken as input by the IL (intermediate language) generator and it outputs IL Code. The reason for generating an IL rather than machine code directly is first it simpler to analyze and break down the code in multiple passes. But the most important is portability. In many languages such as Java or .NET framework, compilation of program usually ends with

generation of optimized IL code. This enables to developer to not worry about generating separate machine code for different platforms. Rather this task in done by the Just-In-Time (JIT) compiler. JIT down compiles IL to platform specific machine code when the program is first run. Even if this wasn't the intent, it is still better to compile to an IL for further optimization. This IL is passed through IL optimizer for generating optimized version of the code. The optimization unit is generally a multi stage multi pass system. This is then finally converted to machine code. The final stage is an oversimplified model, since it involves multiple stages such as linker, assembler, more stages of optimization etc. The following sections deals with developing a custom programming language with inbuilt matrix support using the components discussed above.

3.2 Language Components

Like any other programming language, it contains reserved keywords and operators listed in Appendix A. The language is built on two fundamental elements, native types and matrix types. By means of access modifiers it allows for data encapsulation and allows polymorphism by means of function overloading.

The top most unit in the language hierarchy is Modules. Each code file can contain any number of modules and each of these modules can in turn contain any number of variables and functions within it. Module is simply an enclosure for various ode components with similar behavior to be clubbed together into a single functional unit. For instance, the inbuilt Math Library consists of a Math module which has math constants such as PI, e and functions to compute sin, cos, round etc., and the Matrix module within the same library contains functions to generate identity matrix, get eigen values etc. Both are functionally different and hence encapsulated within a dedicated module. There is no requirement as such to maintain logically different components in different modules, just helps with readability and code maintenance. In some cases, there might be a necessity to create separate modules. For instance, consider the math libraries in Chapter and it user wants to have each of those methods of computing Trigonometric sine result available for

different scenarios. It would be preferable to have each of those implementations called as 'sin' rather than 'sin_LUT', 'sin_CORDIC' and so on since it is the most intuitive thing. To resolve this each of those function implementations can be paced in a separate module, namely Module LUT, Module CORDIC and so on and have same function 'sin' within them. For syntax for declaring a Module look at Appendix A.

A module can contain within it multiple variable declarations and functions. In this language all variables by default are matrices. Matrix sizes can be varied on the run and this means every time a matrix operation occurs the dimension compatibility is checked at runtime. This takes a significant performance hit. To alleviate this, if variable is just a single element, it can be declared as 'Native'. This eliminates need for compatibility checks in most operations and could speed up execution by up to 10 times. Also, all variables by default are Double data type unless declared otherwise. For instance, the following declaration: *Public A = 2*, here A is considered a matrix double. This is different from most other languages since in most of them, declaration like above would have made variable 'A' an integer since data on its right is an integer, but in this language, the data on the right in the above case is interpreted as Double. All numbers are by default Double, unless explicitly marked otherwise with a suffix following the number as given in Table 3.1.

Type	*Suffix*
Short	S
Integer	I
Long	L
Single	F
Double	*<none>*

Table 3.1: Datatype suffix

For declaring 'A' as integer matrix would be: *Public A As Integer = 2*I and to declare it as a non-matrix single element native: *Public A As Native Integer = 2*I. If on the right of assignment if '2' was used instead of '2I', 'A' would still be an Integer since it was declared as such but '2' would be a double which will be implicitly down casted to an Integer either at runtime or compile time. This must be noted since when declaring a Long type where all the values of this type cannot be fully represented in Double. So, the cast will in some cases lead to loss of data. It must also be noted when declaring Single type,

since the same real number might get rounded to different numbers in Double and Single precision representation. To declare an integer matrix for of size 2 rows and 3 columns, 2x3 Integer matrix: *Public A As Integer = [[3I, 4I, 6I], [5I, 9I, 1I]]*. Declaring A as Native in this case will throw an error since it cannot be represented using just a single memory element of integer type without loss of data. A matrix must be of uniform size i.e. in above case both row 1 and 2 has 3 columns, and they cannot be such that one of the row has different number of columns than the other. To create a matrix, just list the numbers and separate them by ','. If a collection of those need to be marked as a separate dimension, then put them with '[ ]'. Matrices can be of any dimension and can be simply formed by combination of ',' and '[ ]' as shown in Table 3.2.

Size	*Declaration*
Row matrix size 3	[a, b, c]
Column matrix size 4 (1x4)	[[a, b, c, d]]
Row matrix of size 1	[a] *or* a
Column matrix size 1 (1x1)	[[a]]
2D matrix; 2 rows, 3 column (2x3)	[[a, b, c], [d, e, f]]
3D matrix; 2 pages, 3 rows, 2 column (2x3x2)	[[[a, b], [c, d], [e, f]], [[g, h], [i, j], [k, l]]]
4D matrix of size 3 (1x1x1x3)	[[[[a, b, c]]]]

Table 3.2: Matrix declaration examples

where, {a, b, c, …., l} are any valid expression formed by collection of variables, functions and operators that produce a single value element. None of the elements within can be in turn a matrix of size greater than 1. The '[ ]' operator does not concatenate matrices. For that use the matrix library. Any sequence on elements can be extracted out of the matrix using 'To', 'Step' and 'All' keywords and index operators as shown in Table 3.3. The result is presented in reduced form, i.e. if a dimension has just single index then that dimension is removed. Consider the last case in the table, a 3x4x5 matrix where 2nd page is extracted, this yields a 1x4x5 matrix and since the 3rd dimension has just a single index, it is reduced to a 4x5 matrix. Same can be seen in the third

example where extracting 3rd column of 2x5 matrix yields a column matrix 2x1, but size it has just a single column it is reduced to a row matrix of size 2. If needed, the result can be converted back to a column matrix by using the matrix operator '[ ]': *[A(All, 3)]*.

Size	*Element*	*Code*	*Result*
2x5	1st row 2nd column	A(1, 2)	Row matrix size 1
2x5	1st row	A(1, All)	Row matrix size 5
2x5	3rd column	A(All, 3)	Row matrix size 2
2x5	2x2 matrix of both rows and column 1 and 2	A(All, 1 To 2)	2x2
5x4	3x3 matrix of rows 1, 3, 5 and 1, 2, 3	A(1 To 5 Step 2, 1 To 3)	3x3
3x4x5	2nd page of the 3D matrix	A(2, All, All)	4x5

Table 3.3: Extracting matrix elements from matrix A

In addition to variables a module can contain functions which may take parameters and may return a value. The format for function declaration and usage are similar to other languages and it supports overloading, where multiple functions can have the same name if their parameters are different either in type or count or order. All parameters and returned data are by value and not reference so modifying it within a function does not affect the actual value. All declarations are declared as either Private or Public. Former is accessible only within the scope on enclosing entity, while the latter is discoverable outside the enclosing entity. All other concepts except for intrinsic matrix support are similar to other programming language and are not discussed any further and the complete feature and functionality is listed in Appendix A.

3.3 Compiler Structure

The programming platform is supported by an almost instantaneous compiler, which parses every single character as it is being typed and parses it and forms the AST on the main thread. So, if a series of text is entered, the compiler goes through multiple calls of lexer and parser to dynamically modify the AST. When text no longer changes in that line for a particular period, the

AST gets pushed for semantic analysis on a secondary thread which analyzes the tree for error and warnings and also forms a conditioned string that adheres to the various standards imposed by the language such as spacing, alternative symbol recognition, keyword coloring, maintaining case etc. If at any point the lexer is called on the line that is being analyzed, it halts till new changes are committed, else the conditioned text replaces the user written text and all error and warning if any are shown or cleared. After this the function containing the recently modified, parsed and analyzed line is added to a ternary thread which generates the IL for the function, if there are no errors and if it is not interrupted by any other changes to the function. The final stage, machine code generation is replaced by up-compilation in case of simulation to C# and in case of real-time control to C and happens on a separate thread when user explicitly calls for code compilation.

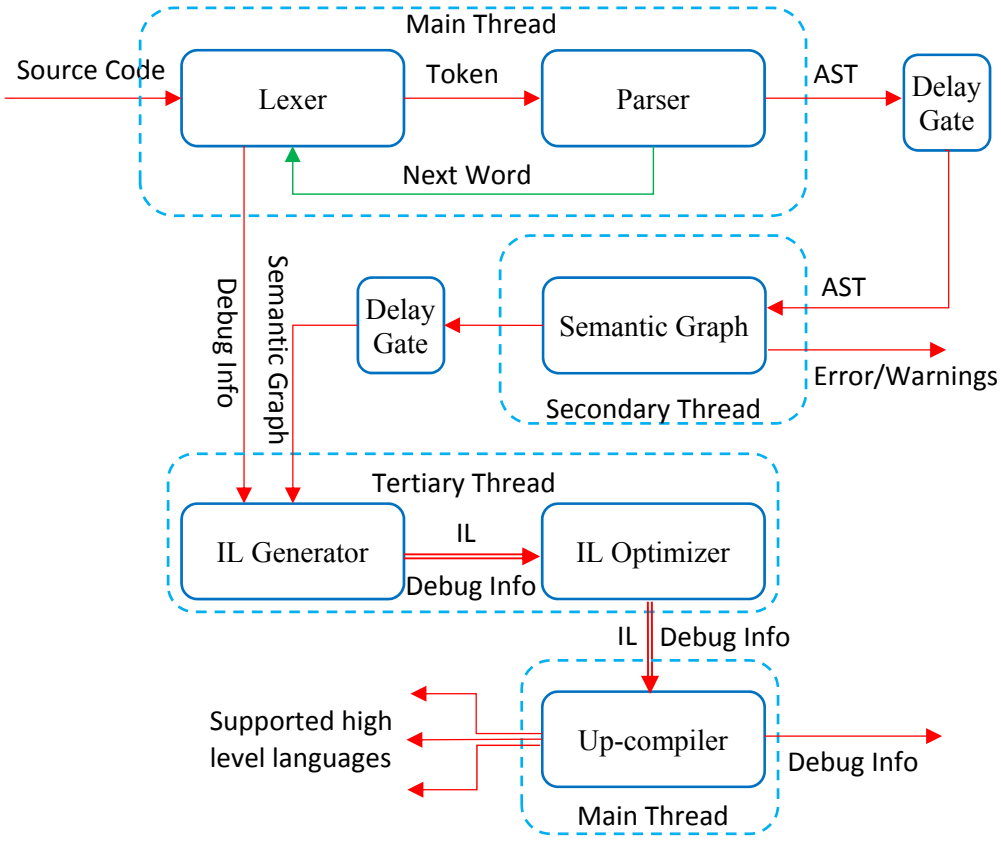


Figure 3.6: Compiler Architecture

The compiler architecture shown in Fig. 3.1 does not strictly apply in this case and the architecture for above mentioned compilation sequence is shown in Fig. 3.6. The output of parser, AST is a binary tree, i.e. each node has either no child node or a left and/or right child node. The AST is formed from the token using the following two rules:

i. If a token T1 is on the left of token T2, then the node N1 corresponding to T1 will be on the left of node N2 corresponding to T2 in the AST.
ii. If priority of node N1 is greater than node N2 in the AST, then N1 will never be a child node of N2 or any of N2's children node.

The first constraint ensures that regenerating the string from AST reproduces the original string. This rule does not ensure that a unique string can produce only a single AST since the string has just one degree of freedom where a word can be only in a particular position along a horizontal line, while AST has two degrees of freedom where a node can be anywhere in the horizontal (width) space as well as the vertical (depth) space. Thus, with just rule 1, multiple AST can be synthesized for a unique string but with the assurance that such ASTs will always reverse translate back to the string that generated the AST. Uniqueness of AST is important because it makes rules required, a finite set. Uniqueness is achieved by imposing additional constraint on to the parser. Rule 2, ensures that for a unique string there is one and only one AST. Thus, these two rules together ensure, for every string there one and only one AST and for every AST there is one and only one string.

For above statement to be true, no component of rule 2 must in turn be a function of rule 1. In a context based language such as this, it is not possible to ensure this. Consider the following statement: $a = b = c$. Here the $b = c$ is a comparison operation that compares if b equals c, if so the result is *True* if not *False*. This result is assigned to a. So, the first '=' is assignment while the second is comparison. In the AST, assignment operation has higher priority than comparison operation as highlighted in Appendix A.2. To determines if an '=' operator is assignment or comparison operator, it must know if previously any other operators occurs in the line. So, the

priority of the node becomes a function of position. The same issues occur with multiple keywords that are used in more than one context and each of those have different priorities based on context. To resolve this issue while maintaining two-way uniqueness, additional constraints are imposed on the AST synthesis:

i. Priority of node must be resolved prior to adding the node to AST.
ii. Newly added node to the AST must strictly abide by rule 1 mentioned in previous paragraph and the parser will correct for rule 2.

The first constraint is obvious, the second implies, that every time a new token is parsed, it is added initially at a random point in the existing AST and the parser moves it around to satisfy all constraints. The initial position as per constraint 2 cannot be random but must follow the rule that token on left must translate to node on left. Also, there can be only one wrongly positioned node in the AST at any time that the parse can correct. The result of parsing code lines in Fig. 3.4 for this language using the above-mentioned parser is shown in Fig. 3.7. All ASTs begin with line entry/ start of line node as the first node and it takes only right child node since there cannot be any tokens before beginning of line. As can be seen the AST is a binary tree and abides by all the rules set earlier. One of the main advantages of using a binary tree as opposed to more complex structure shown in Fig. 3.4 is ease of generating rules. Each node needs to simply check if node on its left and right are acceptable if not error is thrown. A single AST might have multiple errors and they may all be due a single root cause, hence to control the amount of data thrown at the user, the error of highest priority node is reported while the rest are suppressed.

Figure 3.8 is the flow chart of procedure to position a newly added token in an existing AST. It's a recursive function that correct one node at a time till all the rules laid out earlier are satisfied. A simple case is analyzed in Fig. 3.9 where '*' in the expression is added. Node with red colored outline are those that have an error because, either its right or left child is not what is expected. For instance, in Fig. 3.9 a, the variable reference node 'c' throws an error because it does

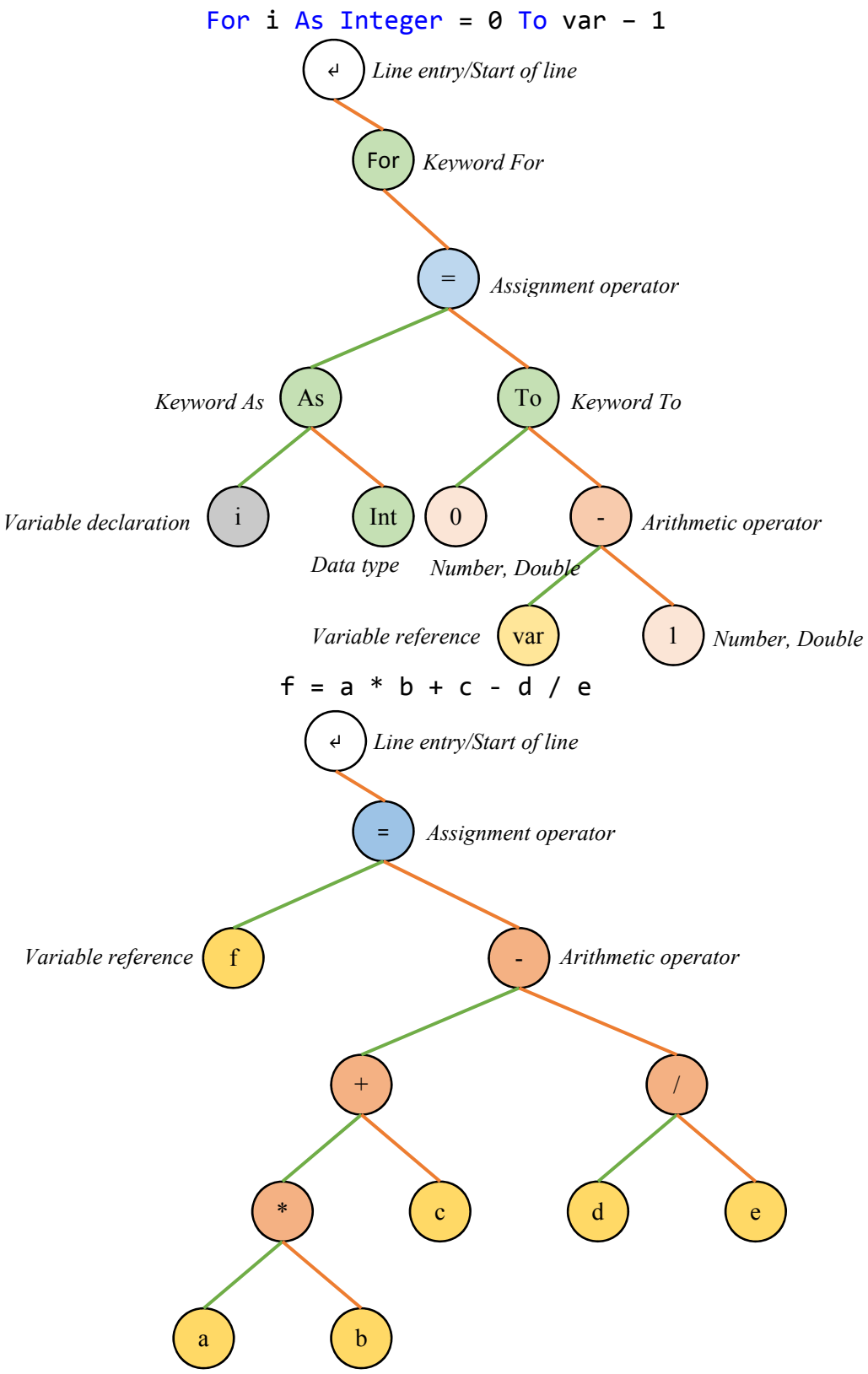
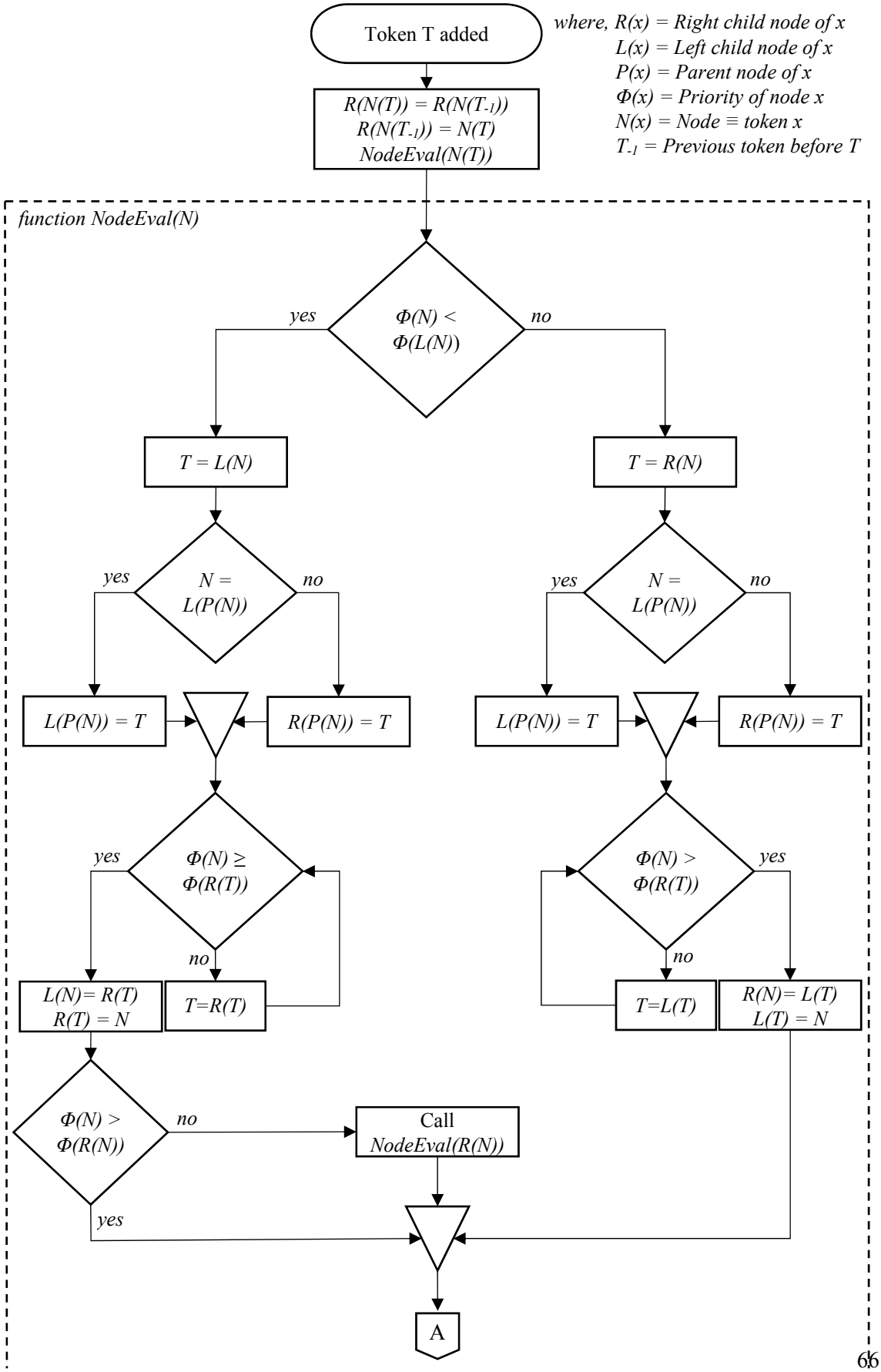


Figure 3.7: AST for single line of code


Token T added
where, R(x) = Right child node of x
L(x) = Left child node of x
P(x) = Parent node of x
Φ(x) = Priority of node x
N(x) = Node ≡ token x
T-1 = Previous token before T
R(N(T)) = R(N(T-1))
R(N(T-1)) = N(T)
NodeEval(N(T))
function NodeEval(N)
Φ(N) < Φ(L(N))
yes
no
T = L(N)
T = R(N)
N = L(P(N))
L(P(N)) = T
R(P(N)) = T
Φ(N) ≥ Φ(R(T))
Φ(N) > Φ(R(T))
L(N)= R(T)
R(T) = N
T=R(T)
T=L(T)
R(N)= L(T)
L(T) = N
Φ(N) > Φ(R(N))
Call NodeEval(R(N))
A

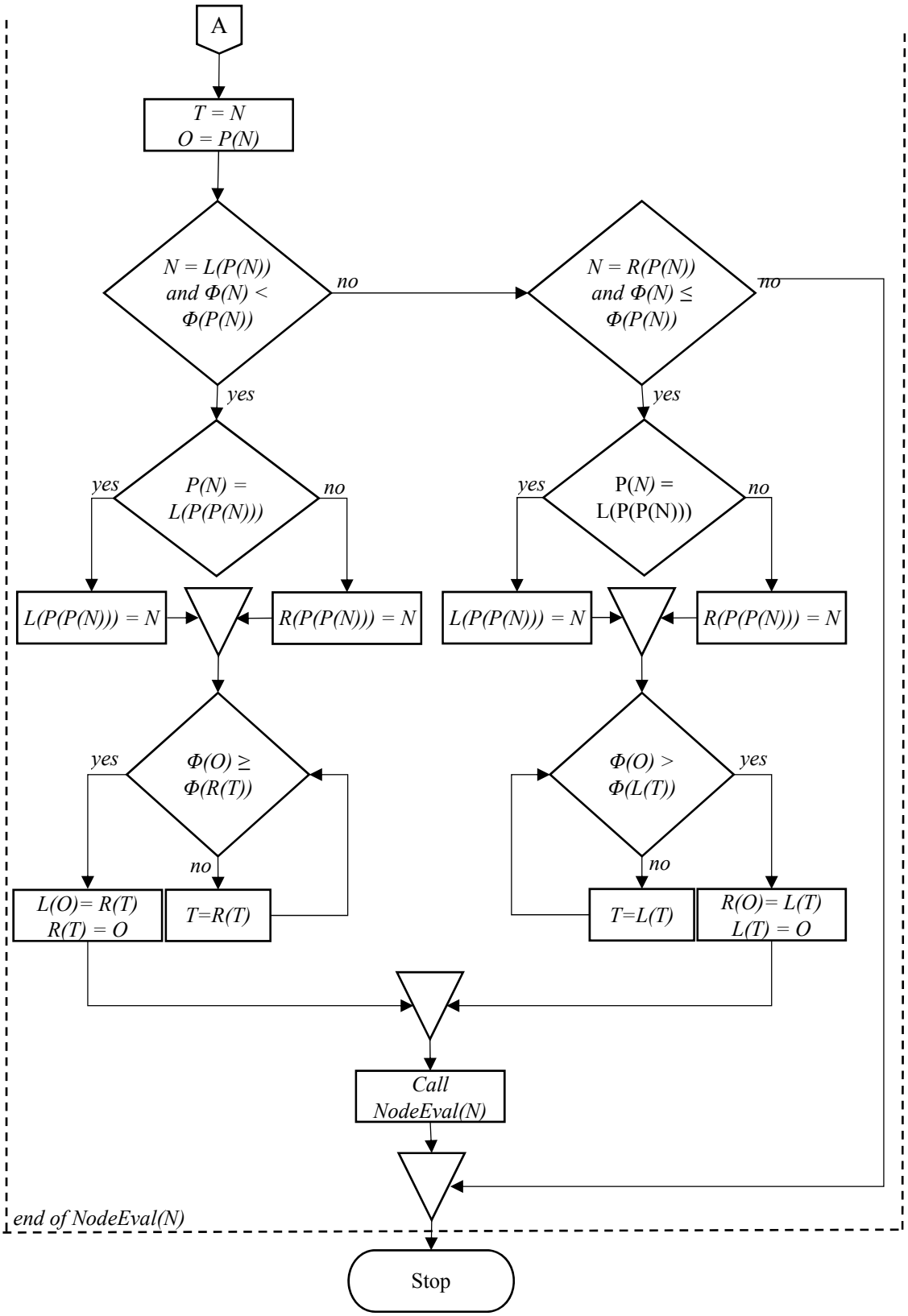
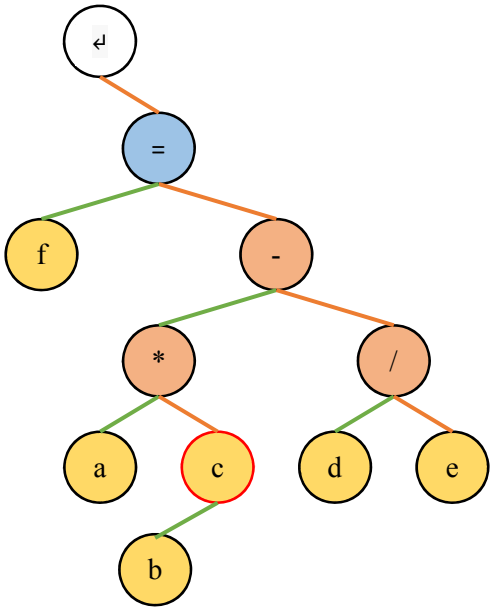
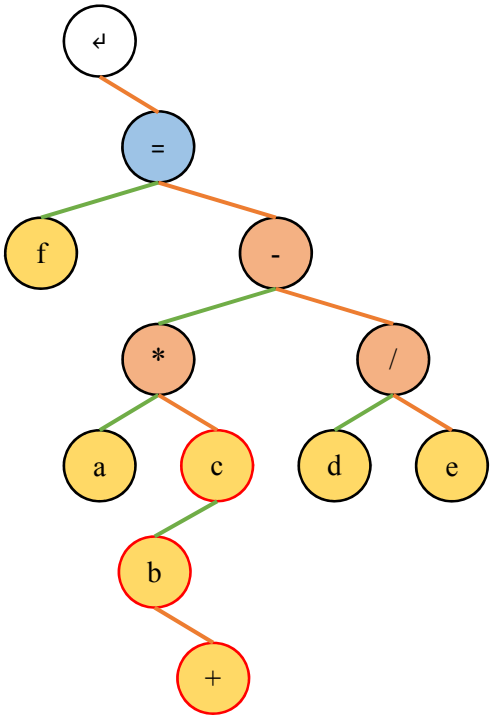


Figure 3.8: AST synthesizer flow chart

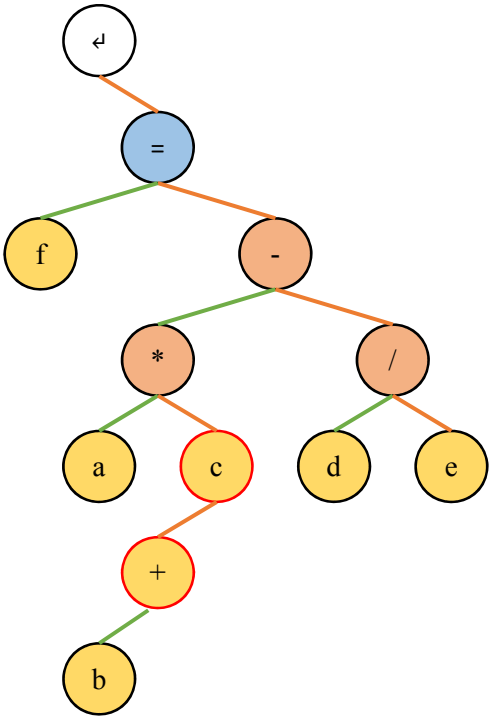
f = a * b c - d / e → f = a * b + c - d / e



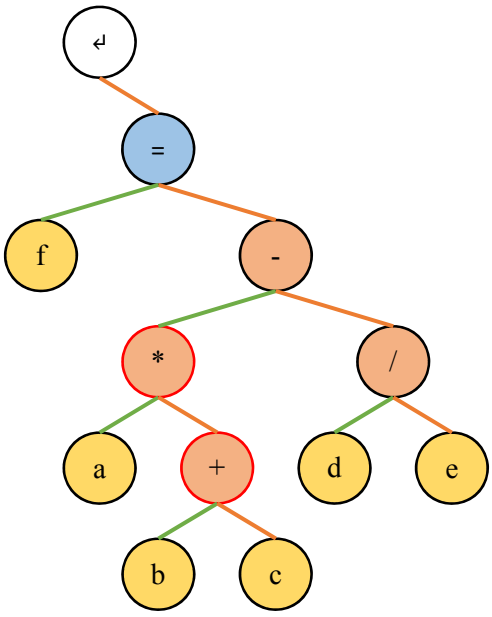
(a)



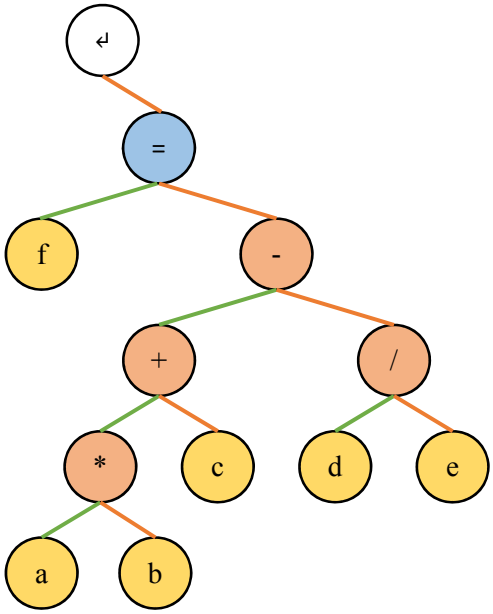
(b)



(c)



(d)



(e)

Figure 3.9: AST synthesis

not accept another identifier as it child node. Similarly, Fig. 3.9 b, throws and error because arithmetic operator '*' node expects both left and right child which are its operand and variable reference node 'b' does not accept an arithmetic operator as it child. The final AST in Fig. 3.9 e abides by all the rules highlighted in Appendix A.3 and does not throw any error. In addition to this the following conditions are checked as well in semantic analyzer:

i. Each of the reference variable node is associated with a definition that is accessible.

ii. Data types of each children is compatible, the resultant datatype is propagated upwards.

Variable reference nodes that have a corresponding definition are attached to the definition so that if any changes to the definition is made or another definition is added that associates more closely to the reference, then the association is reevaluated. If no definition exists for a reference, then the reference node is attached to the project start node and throws an error. Any new definition checks for all variable reference nodes in project start node to see if it can form an association.

So far, AST of a single line of code has been analyzed. Once a line is parsed, it is classified as one of following lines listed in Table 3.4 and an example is shown in Fig. 3.10:

Line Priority	*Line Type*	*Comment*
0	Project Start	This is top most line and every project have this line node. Any undefined variables are attached to this node. All module definition and model definition are attached to this node.
1	Module definition End module	This encompasses definition of a module. This is contained within a project start line node. Within it can either be data member definition or function definition line node.
2	Data member definition	This is definition of variable within a module. It is always contained within Module definition line and cannot enclose any other lines.
2	Function definition End function	This encompasses definition of a function. This is contained within a module definition line. Within it can either be any of the following lines in the table. All the following lines in the table can exist only inside a function except for last two lines.
3	If ElseIf Else End If While End While For End For Select End Select	These set of line always exists within function definition line or within one of the current lines. Each line must be followed by its corresponding end line. Within it can be any other lines from this table starting at current priority. Priority of all lines are within these lines are increased by last priority (10), similar to what enclosures such as parenthesis, brackets and braces did in ASTs.
4	Case Default	This always exist only within Select line and must follow right after select statement and within it can contain any of the following lines.
5	Variable declaration	This is a local function variable declaration. It can be defined anywhere within the function, preferably at the start of function but this is not a requirement. Variables declared within an enclosing unit is accessible only within that enclosing unit. This line cannot enclose any other lines within it.
6	Assignment	This is an assignment line and in most programs, ends up being the most used line. This line cannot enclose any other lines within it.
6	Function call	This represent a function call. This line cannot enclose any other lines within it.
7	Function return	This represent a return function call. This line cannot enclose any other lines within it.
8	End statement	This represent End of program execution. This line cannot enclose any other lines within it.
9	Unknown	Unknown character has the highest priority in an AST next only to line entry node. So, if a line has any unknown character the node is pushed to top of AST and is immediately discovered during error analysis. Any line that contains such as character is marked as unknown line. This line cannot enclose any other lines within it.
10	Blank	Lines with no character other than spaces is marked as blank lines and are ignored for all purposes except for formatting.

Table 3.2: Code line nodes

```
1    Public Module Math
2        Public highestValue As Integer
3        Public Pure Function factorial(num As Integer) Returns Integer
4            Local result As Integer = 1
5            If num > 1I
6                result = result * factorial(num – 1)
7            End If
8            Return result
9        End Function
10   End Module
```

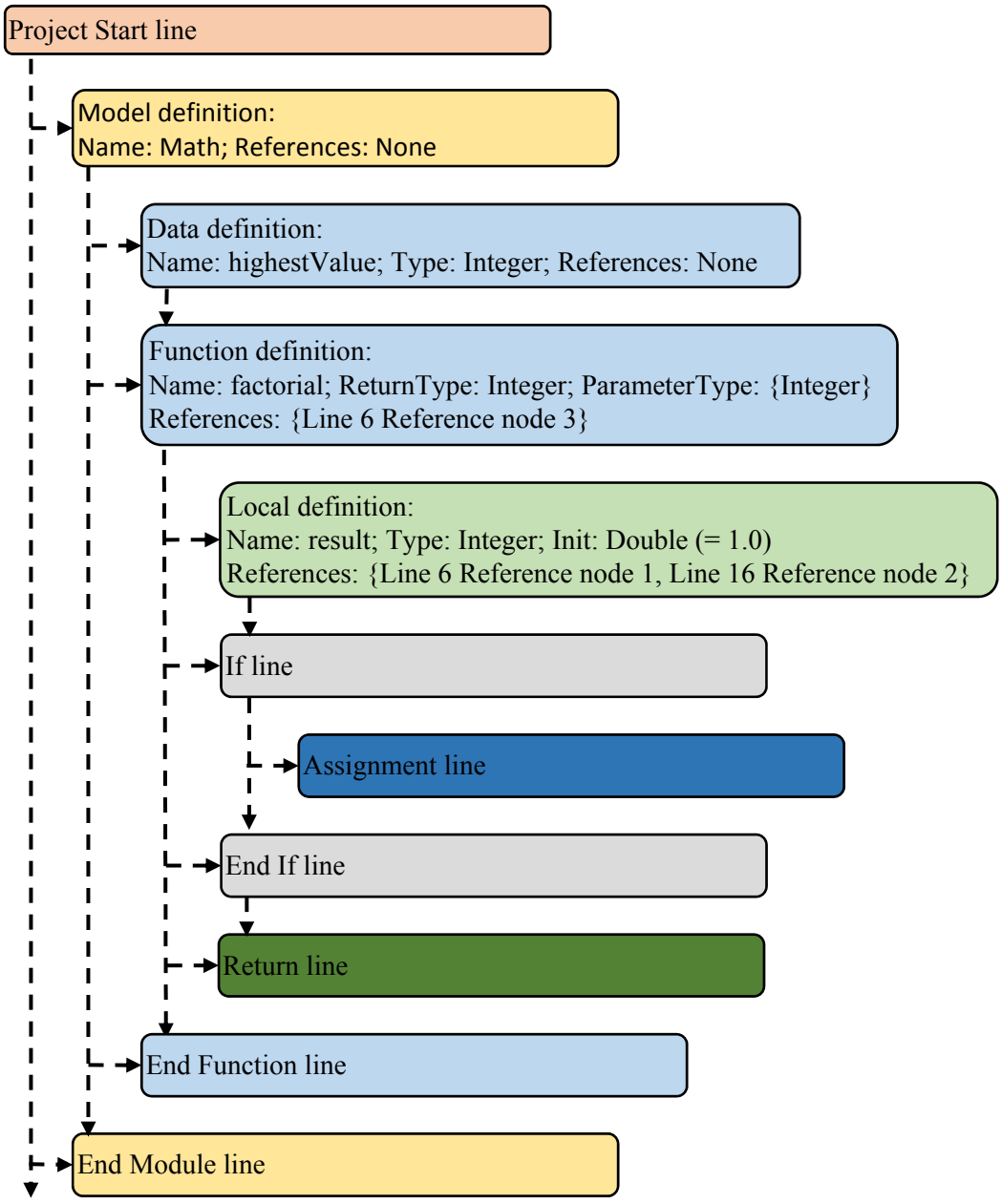


Figure 3.10: Line parser

If any line is enclosed by a line or followed by a line that does not abide by rules in Table 3.4 then, line error is thrown. Every definition is parsed into a definition packet that list all the details that are needed by reference and all references to that definition are added to it. It must be noted that each line in the above figure has its own AST. Error of line nodes has lower priority than errors of AST node. For example, if module definition format is wrong, it will throw a AST node error and if 'End Module' is missing, it will throw Line error, of these the AST node error gets higher preference for display.

The semantic analyzer is followed by translation of semantic graph into IL code. The IL language commands and the functions are listed in Appendix A.4. Each function generates a block of IL. All data member initialization is clubbed as a single function unit of the enclosing module. This initialization is lazy invoked, i.e. all members are initialized when any one of the variable reference is invoked. Each line node within the parsed function is traversed and sequentially translated. During this stage multiple optimizations occurs. For instance, if there is a conditional block, whose condition is always false, the whole block is ignored from translation, if a variable is assigned but never used it will be ignored during increased optimization setting and so on. For the complete list of optimizations check the manual included in the software. Each AST of each line node is traversed depth first and each node is requested to generate its corresponding IL. Each line terminates its IL with 'EOL' instruction. This is used to build map of debug information to tie a line to the error generated during runtime. Also during AST IL translation various optimization take place such as elimination of redundant casts, changing instructions to eliminate type casting, etc. The generated IL for *factorial* function in above example is listed in Table 3.5 and the generated hex is: *0x08, 0x0F, 0x1A, 0x52, 0x07, 0x00, 0x0F, 0x2F, 0x19, 0x0F, 0x04, 0x52, 0x08, 0x08, 0x07, 0x00, 0x0F, 0x20, 0x08, 0x09, 0x21, 0x1A, 0x52, 0x50, 0x52, 0x08, 0x03, 0x52*. This is similar to any assembler output. It must be noted this is just for a single function. Outputs of multiple functions are combined, and references are replaced with memory index location in the final table.

Line Number	*Instruction count*	*Instruction stack*	*Reference count*	*Reference stack*	*Comments*
4	1	LRF	1	result	Load variable for initialization
	2	LI1		*<no change>*	Load integer 1. Even though RHS is double it is optimized.
	3	ASV			Assign 1 to ‘result’
	4	EOL			End of line
5	5	LOA			Load function argument
	6	0			First argument
	7	LI1			Load integer 1.
	8	GT			Compare ‘num’ > 1
	9	BRF			Branch if false
	10	15			Jump instruction stack by
	11	4			Jump reference stack by
	12	EOL			End of line
6	13	LRF	2	result	Load assign to variable
	14	LRF	3	result	Load first operand of multiply
	15	LOA		*<no change>*	Load first operand of subtract
	16	0			Load first function argument
	17	LI1			Load second operand of subtract
	18	SUBT			Subtract ‘num - 1’
	19	LRF	4	factorial	Load function call
	20	LMD		*<no change>*	Call function
	21	DMUL			Multiply result of function call with ‘result’
	22	ASV			Assign product to ‘result’
	23	EOL			End of line
7	24	BRP			Mark branch to point for end of ‘If’
	25	EOL			End of line
8	26	LRF	5	result	Load return value
	27	RET		*<no change>*	Return value
	28	EOL			End of line

Table 3.3: Generated IL code

The final stage is up translated to the desired language. In case of simulation, it is up translated to C# and compiled using CodeDOM [13]. In real-time mode it is translated to C89 and compiled using device specific compiler. For all information related to matrix math implementation refer to WorkbenchMathCore.dll within the software folder.

3.4 Conclusion

In this chapter, a brief introduction of general compiler architecture was explored. This was followed by an over simplified description of foundational components of the instantaneous compiler that was implemented for the custom developed language. The code is broken into tokens and then reconstructed in to abstract syntax tree and references associated with definitions, errors and warnings generated, followed by translation to custom intermediate language. Finally, this intermediate code is translated to higher level language and down compiler using platform specific compiler.

The language has inbuilt matrix operation support and all functions supported in simulation are supported in real-time as well in many cases extremely optimized assembly level functions have been implemented specific to de device in use. It does not rely on any 3rd party math libraries like GNU Octave used by many other platforms. The generated code can refer to external 3rd party components. To reduce the chances of mishaps associated with running untrusted code, the generated code is run in a Sandbox which has its own challenges especially with regards to huge data log transfer between the AppDomain [14]. As mentioned earlier, discussion on all the components of the language are beyond the scope of this thesis and it is recommended to refer to the complete reference manual.

Chapter 4

Model based design platform

In the previous chapter, design of a code based numerical simulation platform was discussed. In this chapter, another major aspect of the platform, namely model based design is discussed. Even though theoretically, anything that can be implemented in code based platform must be possible in model based platform and vice-versa, since a model based platform in the end effectively translates each model entity into equivalent code, each of these are more suitable under different contexts. Code based platform is much easier in case of a system that can be easily represented using recursive and iterative units; also, systems that can be broken down into time independent functional components. A model based platform is more suitable for systems that are time dependent, streamlined and sequential without much conditional elements. For most part it is a user preference, similar to choosing between coding in high level language vs in assembly (assuming that both cases efficiency is not a constraint).

A code based platform is more complex from user end but extremely simple in terms of its backend implementation compared to a model based platform which is easier on the user end but far more complex due to various number of deeply interlinked components. In this chapter a cursory analysis of the various functional components of model based design is explored. For a more detailed analysis refer to the complete reference manual.

4.1 Platform components and features

The model based design platform consists of 5 major components:

i. Drag and drop tools
ii. Links and connections
iii. Model foundation
iv. Subsystems
v. Data logging

Each of these in turn have multiple sub components and functionalities. The first component, the drag and drop tools are components that can have series of inputs and operates on these inputs to produce an output which can be numerical or a boolean value. The output of a tool can be input of other tools like arguments of a function, where each tool is a function that returns an output and takes inputs as arguments. This connection between tools is established using links by clicking on the tool ports and dragging the wire to another port and the link handler will determine a minimum weighted path. The two components are built on the model foundation that does the following, including but not limited to: data type and size determination, sample time propagation, establishing compilation order, undo/redo and file handler, tool error and warning consolidation, maintain tool IL code stack, real-time and conditional tool determination and so on. Some of these components will be discussed in further detail in the following section.

The subsystem in some respects a tool since it has its own implementation in the toolbox and can have inputs and outputs. But in contrast, it does not generate any code, does not have any rules and is merely an enclosure for more tools within it which is where it shares similarities to a model foundation. But unlike model foundation, it does not do any model functions mentioned earlier. Rather it bypasses the call to enclosing model file. Other than visual element it is transparent for all other functionality. The final major component is data logging. This has two sub components, simulation logging and real-time logging which shall be addressed in the next chapter.

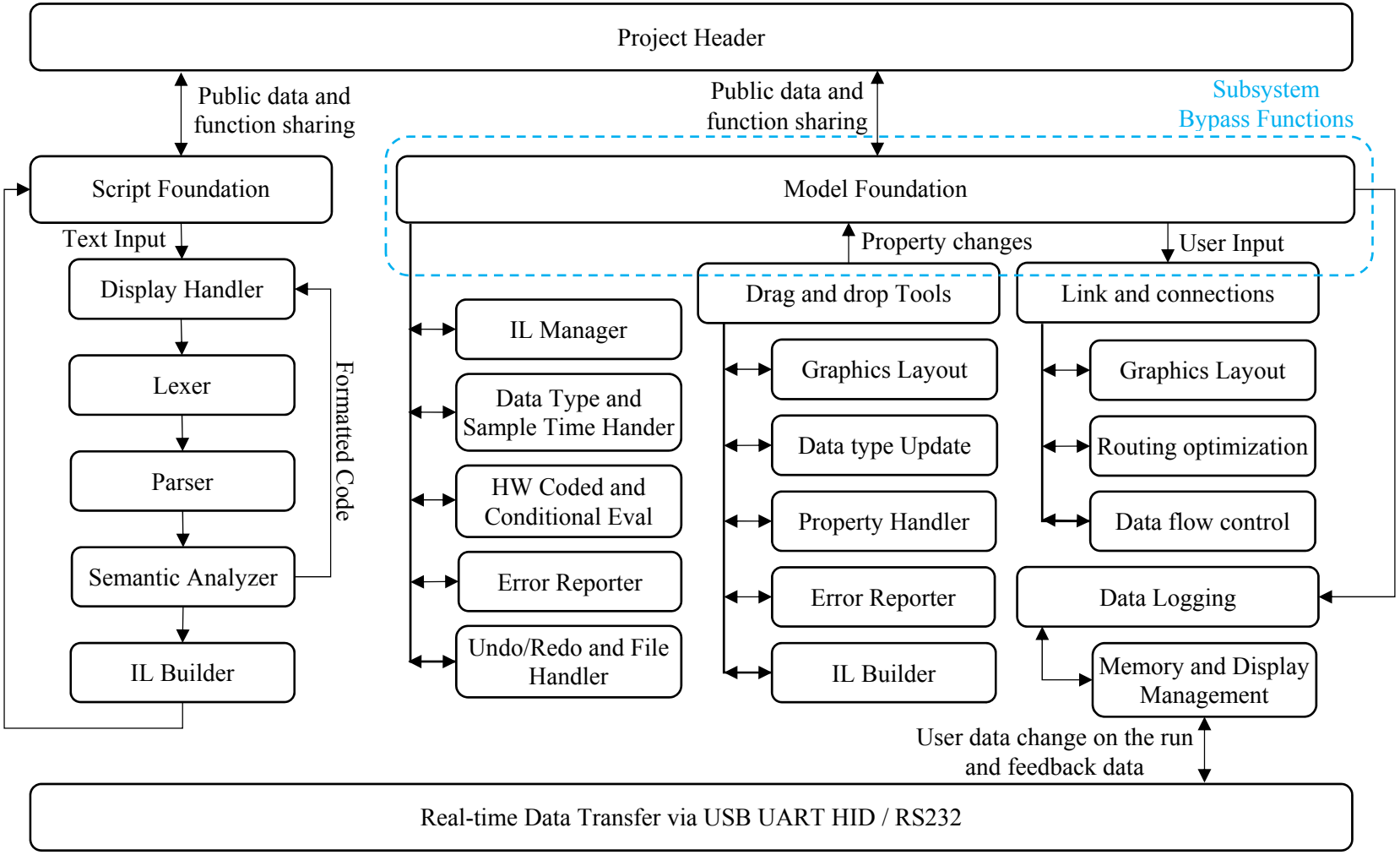


Figure 4.1: Platform components overview

4.2 Platform components and features implementation

A brief implementation overview of some of the important components highlighted in Fig. 4.1 is discussed in this section starting with Link and connections handler. For each of these sub sections, the graphical layout implementation is not discussed since it is highly development environment specific. The other major components are routing and data flow.

4.2.1 Links and Connections – Routing Handler

Routing handler consists of multiple methods that yields an "aesthetically" weighted shortest path between two points which in many cases is from output of a tool to input of another tool. The term aesthetically weighted is used to indicate given an option of multiple routes that of closely matched lengths, the route that is more likely to be used if it were manually routed by majority of test cases is the one that is considered. Of all the possible route options between two points in a screen of X, Y pixels (factorial of (X pixel count * Y pixel count)), only a very few are to be likely preferred generally due to easier readability. Figure 4.2 shows some of those possible options for the test case considered. Sine source tool is already connected to Scope as shown and the figure shows different options for routing from Constant to Gain block, around it or through existing connections. In general, it has been observed that there is an innate preference to have lines that run parallel, avoid passing through existing blocks, limited intersection of other links and lesser the corners the better; all the while maintaining as short a path as possible. Of 4 cases shown in figure above, case (b) and (c) have the shortest of path, while (a) and (d) avoid any form of intersections. Based on various user preference test cases, option b is the shortest aesthetically weighted path. The weights of such a path in most cases, is given in Table 4.1. From the table, it is obvious that given an option for routing algorithm to pass through a tool or cut across a link, the latter is preferred due to lower cost. Similarly, a straight link is preferred twice as much as a link that has a corner (turns).

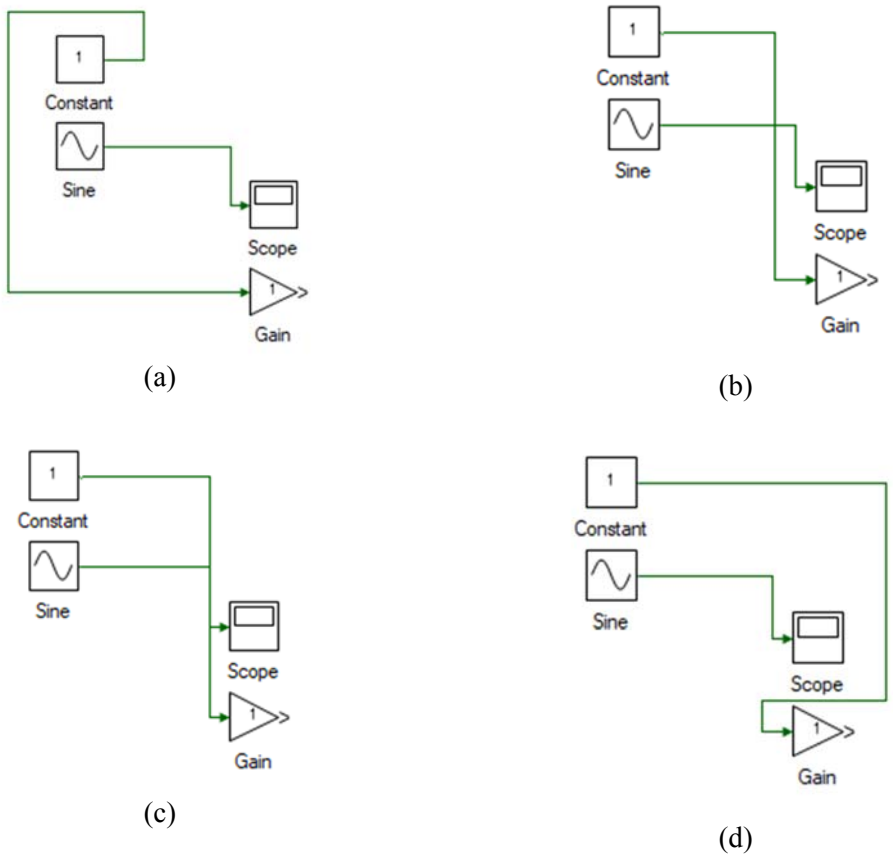


Figure 4.2: Some of the possible Route options

Case	*Weight*
Straight Line	1
Corner	2
Intersecting line	10
Intersecting tool	100
Intersecting text	1000
Form border	*10e6*

Table 4.1: Routing weights

The problem is similar to any path finding algorithm in a directed weighted graph and several solutions are available that address this [15]. The commonly used Dijkstra's algorithm starting from point 1, travels through every path of equal length till end point 2 is reached by any one of the equal length wave-front. Of all these paths if just the ones that are closest to the end point is first explored, it is the A* algorithm and it converges much faster in most practical cases. Dijkstra's is a special case of A* algorithms. Both these algorithms yield the best possible paths. There are other heuristic routing algorithms that converge much faster but do not always give the shortest path and hence are not considered for this implementation. The runtime for Dijkstra's is given by $O(E +$

$Vlog(V)$) implemented as given in [16] where V is the number of vertices in the directed graph in this case equals the number of pixels in the model form given by X*Y and is the E is the number of edges between the vertices. Given that only vertical or horizontal routes are considered that equals 4 times the number of vertices. The worst-case runtime for A* equals the runtime for Dijkstra's but for average cases it runs much faster. The average time taken for 300 different routing conditions considered was 11.3 seconds for Dijkstra's vs 2 seconds for A*. Though the exact numbers make no sense outside the context of the test case it shows the general real-world performance of these implementation for this particular application. Though A* is faster, it is still not suitable since 2s is a noticeable computation time and will defeat the target of seamless user interface. The increased time is due to this application where in general the number of path are much more than the number of obstacles (tools, texts etc.) limiting these paths.

To reduce the time further the A* algorithm is modified as described which limits the number of points considered as possible paths. The graph is considered as a 2D matrix, filled with weights based on values in Table 4.1. For instance, if there is a tool starting at point (10, 20) of size (15, 45), the whole rectangle is filled with the weight for tool, i.e. 100 in the 2D matrix W. This is done for all the components in the model form. Every time a new component is added this matrix is updated. In addition to this, the partial differential of this matrix along X and Y is computed as well, which yields a column matrix C and row matrix R respectively as given in Eqn. 4.1.

$$C(i) = \sum_{j=1}^{X-1} W(j,i) - W(j+1,i)$$
$$R(i) = \sum_{j=1}^{Y-1} W(i,j) - W(i,j+1) \tag{4.1}$$

A change in the differential vector, indicates change from blank space to a model component such as tool or links, if it is positive and vice-versa if negative. Now, in the A* algorithm the search direction which was initially in all four directions from every possible point, is limited to just a

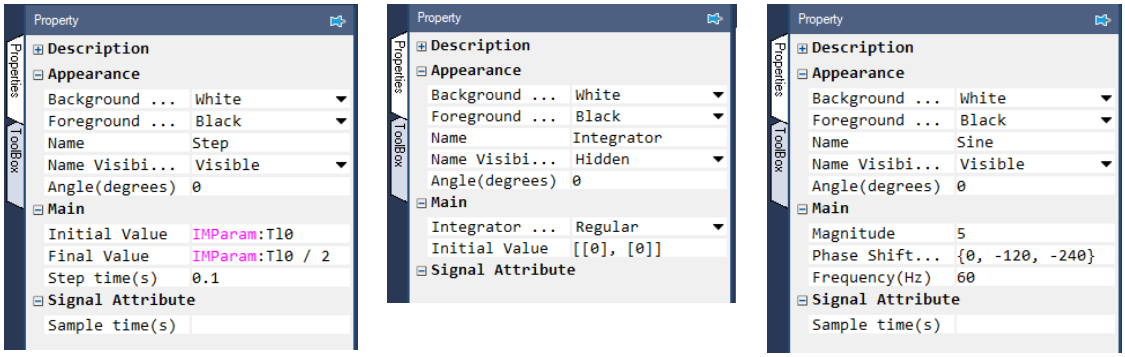
straight line. A search in other direction is carried out only if there is a change in differential along the perpendicular direction, i.e. if a search is progressing along X axis, at point P along this direction, additional search node along Y from point P is added if the differential along Y is different, i.e. $R(P.Y) <> R(P.Y+1)$. With this added constraint the worst-case runtime is still same as Dijkstra's runtime, but the average is much better than A*. For the same 300 test cases run under similar conditions, the average runtime is 97msec as opposed to 2s for A* and 11.3s for Dijkstra's. This significant reduction is due to elimination of most of the parallel paths.

4.2.2 Drag and drop tools:

Functionally these are similar to Script functions described in previous chapter. In that case, the parser broke down each functional line into a structured binary tree where information such as datatype, IL code etc., was propagated from the child node to parent node. In Model file, there is no need for a lexer or parser since the components are inherently structured into logical tools and tree structure formed via links. The complexity lies in the fact that the number of inputs or output is different for different tools and cannot be resolved into a binary tree structure. Also, the flow of information is not always from inputs to outputs as one might expect. For instance, while datatype of tools is forward propagated, i.e. from output of a tool to input of tool it connects to, information related to conditional evaluation is backward propagation. These and other difference prevent having a unified foundational layer that can be shared by both model and script files. This sub section explores some of the key aspects of a tool, a critical component of any model based design environment.

All the tools and their functionalities are listed in Appendix A.5. These tools, similar to script operators, can takes as input any matrix or muxed arrays. In addition to time varying inputs, some tools can have initialization value that are not changed during model runtime such as, gain value of gain block, levels of saturation block, initial value of integrator block etc. These values can be set via Property window. Each tool is simply a class and the properties decorated with

custom properties attributes are discovered and linked using reflection and displayed in the property window. Any changes to these properties are relayed back to the containing tool.



(a) (b) (c)

Figure 4.3: Tool properties of: (a) Step Source, (b) Integrator, (c) Sine Source

Figure 4.3 shows properties of some of the tools. The step source has an initial value, final value, and time when this transition happens. These fields extend the Script editor and compiler described in previous chapter. As seen, it supports all operators and instantaneous compiler that report error and warnings immediately. The only difference between the compiler for Script editor and Property fields is that while the former has a structured multi line format, the latter just accepts single line and that too limited to AST that can occur on the RHS of an Assignment ("=") node. In Fig. 4.3b, the property of an integrator is shown, and the output is initialized to a two-element column matrix of 0's and in Fig. 4.3c, the property of sine source block is shown where, the properties are set to generate a three-phase sine wave of 60Hz. In this case, the phase is mentioned as a muxed array as opposed to a matrix. This leads to the function call being generated three separate times one for each phase shift value. Though this as opposed to matrix increases the code size, it makes the generated code extremely efficient and reduces a huge burden on the user end of having to repeat the same tools multiple times with different properties. The array is merely a shorthand expression that shall be expanded during compilation.

The datatype of a tool is updated anytime its inputs or properties datatype changes. How these affect the datatype is tool dependent. For instance, in case of add block, the output datatype is the highest of input datatype with the order from lowest to highest being: Boolean, Short, Integer, Long, Single and Double. So, if one of the input is a Single and other is Double, the output is inherited as Double. In case of integrator, the output is always Double irrespective of the input datatype. Most of the tools only accepts numeric types, i.e. types other than Boolean. This is so because unlike languages such as C, C#, VB.net etc., there is no fixed value for Boolean type, its value is function of the platform to which it is translated to with False = 0 and True = 1 in C and C# and False = 0 and True = -1 in VB.net. To keep the result of running generated code across all supported languages the same, the platform does not support any implicit boolean to numeric conversion. To achieve this, a boolean cast tool is available in the toolbox that requires the user to set explicitly the values of False and True. The size of all the datatypes are fluid as well and dependent on the compiler used. For instance, Integer in C# generated code is 32 bits while the same model compiled to generate C code to run on a TI F32833x processor will of size 16 bits. There are options to modify these as well and shall be discussed later.

Similar to code lines in Script, each tool reports errors and warnings instantaneously at compile time. All the tools share the same functional foundation and therefore share same of the common error and warning reporting such as, all tool throw a warning if any of the tool inputs or outputs are unconnected, throw error if sample time is invalid and so on. In addition, each tool reports specific errors as well such as, an integrator block throws error if its initial value is not proper expression, add block throws error if any of its input is boolean, selector block throws implicit down cast warning if its index is not an integer etc. Similar to code lines, each tool consolidates these errors and warnings and reports the most severe one to the user and reports the next one once the highest error is cleared and so on till all the errors are cleared. These compile time errors are caught as and when they happen, and the model cannot be run without fixing these.

The other class of error, runtime exceptions are ones that cannot be resolved by the compiler and occurs when a model is run. For instance, if 2x3 matrix is added to 3x2 matrix, this is caught during runtime and the compiler does not catch this since all the sizes are resolved only at runtime. Similarly, if input of ADC block crosses the limits supported by the real-time controller selected, it throws out of range exception at runtime and this cannot be obviously known at compile time. All these exceptions are reported to an error, warning, exception and message (EWEM) handler that consolidates and maintains all the EWEMs. It maintains a pointer to the tool or code line that reported them, and constants checks if all the EWEMs are still active and removes those that are not. These are maintained on a separate thread much slower than the main UI thread. Figure 4.4 shows the EWEM handler reporting compile time errors and runtime exceptions.

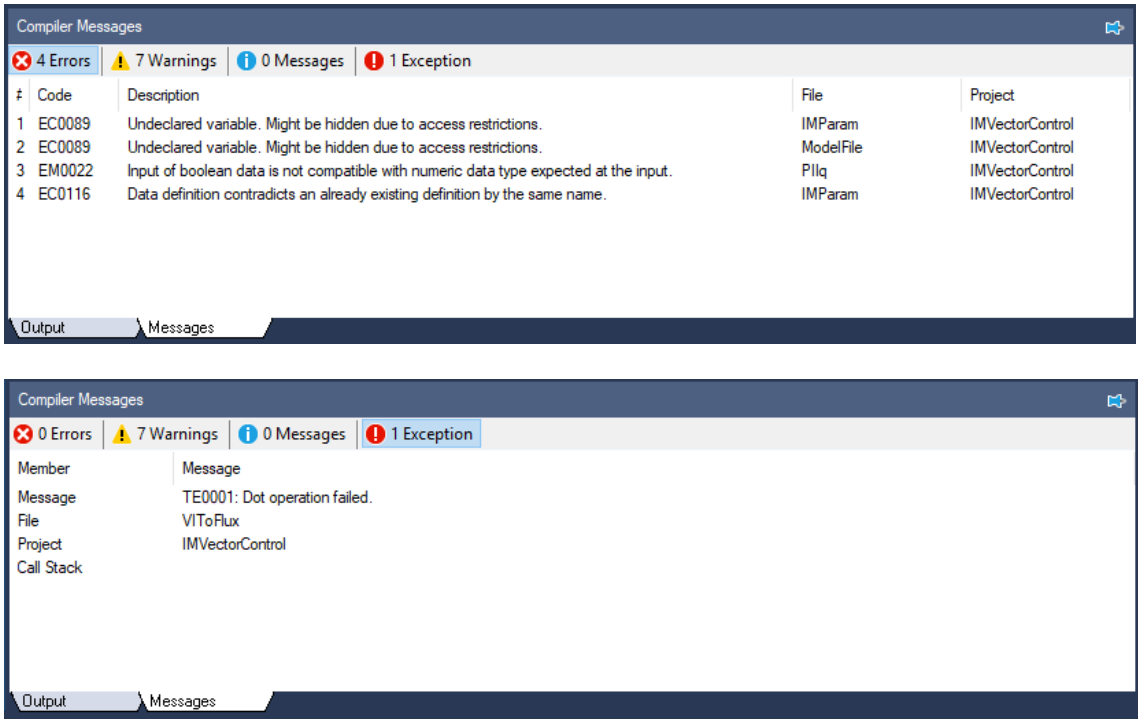


Figure 4.4: Error, warning, exception, and message Reporter

The final component of a tool is IL code generation. Each tool generates two set of code, initialization code that is run just once at time step 0 and the runtime code that is run at every time step. Each tool code like code line, end with *EOL* instruction. In addition, *EOL* instruction is used

to mark the separation between initialization and runtime code. Some tools such as addition, multiplication, elementary functions etc., may not have an initialization section in which case it has just an *EOL* instruction in its stack for initialization code and likewise tools such as Constant which has only initialization code and no runtime code, has just an *EOL* instruction in its runtime instruction stack. The IL generated for a gain block with a gain value of *Vdc*/2 and two muxed inputs is given in Table 4.2.

Instruction count	*Instruction stack*	*Reference count*	*Reference stack*	*Comments*
1	LRF	1	gain	Load gain reference
2	LRF	2	Vdc	Load reference variable Vdc
3	LVI		<no change>	Load integer value
4	2			Load integer 2, used to divide Vdc
5	DDIV			Vdc/2
6	ASV			Assign result to gain
7	EOL			End of Gain block initialization code
8	LRF	3	gainOp_1	Load first of two muxed gain outputs
9	LRF	4	gainIp_1	Load first of two muxed gain inputs
10	LRF	5	gain	Load gain reference variable
11	DMUL		<no change>	gain * gainIp_1
12	ASV			Assign result to gain output
13	LRF	6	gainOp_2	Load second of two muxed gain outputs
14	LRF	7	gainIp_2	Load second of two muxed gain inputs
15	LRF	8	gain	Load gain reference variable
16	DMUL		<no change>	gain * gainIp_2
17	ASV			Assign result to gain output
18	EOL			End of Gain block runtime code

Table 4.2: IL Code for Gain block

Above case was for one of the simpler tool, for most of the tools the generated IL code changes drastically based on selected options are inputs. Consider the Conditional If-Else Selector tool. One of the properties of this tool is whether conditional evaluation is enabled or not. If it was

then connected input tool on True is evaluated only if condition is True and False input if only condition is false. This optimization gives extreme performance boost especially if inputs are some complex functions such as series of trigonometric calls. It also helps in that the user from does not to use separate 'if-else' blocks as is with other platforms which leads to flow discontinuity and does not have to manually determines how to organize the blocks since the compiler auto determines this. The details of how the compiler determines which tools to evaluate conditionally and when is discussed in later section. Table 4.3 shows the code generated with optimization disabled and enabled respectively. In both cases, the inputs are all non muxed inputs.

Instruction count	*Instruction stack*	*Reference count*	*Reference stack*	*Comments*
1	EOL	0	<null>	No initialization code
2	LRF	1	Op	Load selector block output reference
3	LRF	2	Ip1	Load condition (which is first input)
4	LRF	3	Ip2	Load if True value (which is second input)
5	LRF	4	Ip3	Load if False value (which is third input)
6	TIF		<no change>	Ternary If instruction
7	ASV			Assign result to output

Table 4.3: IL Code for If-Else Selector block with conditional evaluation optimization disabled

The code for these two conditions with same inputs are drastically different. The code when optimization is disabled is a simple ternary if instruction which is translated to C as: *Op = condn ? Ip1 : Ip2*. The code when optimization is enabled is an if-else block where the assignment of output to input 2 happens if condition is true else to input 3 if false. During final compilation, with the 'if' block the code for tools that need to be evaluated only if condition is true is injected and similarly for false condition. Also, what inputs are accepted changes, as seen in first case, condition input can be a matrix in which case an element by element if true else false value assignment takes place. But when conditional evaluation is enabled, it makes no sense to matrix condition since condition determines functions to compute. Using the IL operators in Appendix A.4 all the tools in A.5 can be implemented as shown in the above examples. For complete implementation details refer to the

reference manual. In the following sub section, the model foundation is discussed which maintains the overall tool structure, file handling and undo redo stack.

Instruction count	*Instruction stack*	*Reference count*	*Reference stack*	*Comments*
1	EOL	0	<null>	No Initialization code
2	LRF	1	temp	Load a temporary variable to cast condition from matrix to native element. In conditional evaluation, condition being matrix has no physical meaning and hence must be native.
3	LRF	2	Ip1	Load condition (which is first input)
4	CTE		<no-change>	Cast matrix to native, ignored if already native
5	ASV			Assign to temporary variable
6	LRF	3	temp	Load condition (which is in 'temp')
7	BRF		<no-change>	Branch if condition is False
8	9			Jump by instruction count
9	2			Jump by reference count
10	LRF	4	Op	Load output reference variable
11	LRF	5	Ip2	Load if True value (which is second input)
12	ASV		<no-change>	Assign this to output
13	BRU			Branch to end of code
14	7			Jump by instruction count
15	2			Jump by reference count
16	BRP			Branch to point marker for condition false
17	LRF	6	Op	Load output reference variable
18	LRF	7	Ip3	Load if False value (which is third input)
19	ASV		<no-change>	Assign this to output
20	BRP			Branch to point marker for end of code
21	EOL			End of runtime code

Table 4.4: IL Code for If-Else Selector block with conditional evaluation optimization enabled

4.2.3 Model Foundation:

One of the major functions of model is to maintain the structured flow of tools. In case of script file, the structure was inherent due to sequential nature, where each code line is executed once the previous line is, the branches were more obvious, and blocks of code was demarcated by

enclosing within functions, modules, and conditional blocks. The same is not the case with model files where tools can be interconnected in almost any order forming an intricately connected web. The main task of model is to maintain data flow between this web such that it can be broken down into a sequential code structure. Before delving into details of how this is done, the other component of model file namely, file and undo/redo handler is discussed.

The model file class contains all the properties of such as solver used, step time etc., and a list of all tools and links within the model. Within the list of tools could in turn be subsystems which contain more tools and links within it. A subsystem is tool only in the sense it has input and output ports unlike a model file. But unlike a tool it does not generate IL code or has any datatype. Any reference to its input and output ports get directed to tools within the subsystem that are connected to those ports. Similar to a model, a subsystem encloses other tools which can be subsystems and links within it. The similarity end there, unlike model file subsystem does not maintain tool structure or undo/redo stack nor does file handling. With regards to those it is a tool and bypasses such task to enclosing model file. Starting with a model file, every other element of the model is directly linked to it via a web of references. Each tool and link class is decorated with DataContract attribute and file save and open are handled using the .Net data contract serializer [17].

The undo/redo (UR) stack is built by registering every action that occurs. With regards to links and connection this includes, changes such as deleted link, added links, branches, resizes and modified sources/sink tools. For tools these includes but not limited to tool resize, tool name change, tool property changes that leads to tool port changes such as changing the port count of adder block sets of a cascade of events starting with port count change followed by tool resize followed by liking or unlinking connected/nearby open links etc. Each user action that causes any change in the model file is registered in the UR stack. Each element of an UR stack in turn consists of a bundle of sub actions. For instance, in the example above, where user changes the port count

of adder block, this is registered as a single action in the UR stack and the cascade of events it set forth in the background such as property change leading to tool resizing and removed port connections are registered as sub events within it. These are noted in the exact order in which they occur for reversing them if Redo is called. Properties of tools that do not affect the physical aspects of a tool such as changing the initial value of an integrator, magnitude of sine source etc., are stored in a separate sub event and are ignored during UR unless explicitly mentioned by the user to include those as well.

In many cases, a series of actions are carried out and these are undone followed by another set of actions and this process could be done multiple times. It may be necessary to restore back to the state that was during one of these multiple undo, update actions. To allows for this the UR stack is modified into a tree where any undo action followed by new action is spun off into a new branch. When user undoes back to this branch point has the option to redo any of the stored branches. The depth and breadth of these branches is dynamically adjusted based on memory and performance limitations. As the tree depth increases older actions are compressed into single actions, such as multiple sequential moves of the same tool is compressed into a single mode to reduce memory usage. Figure 4.5 shows an example of an UR tree where a gain tool is added followed by a constant tool and a link between the two is established along straight line. Then the last two actions are undone followed by adding a Multiplier tool, link from Multiplier output to Gain input. This is followed by number of multiplier ports increased from 2 to 3, this leads to other sub events as seen where the tool height is increased to accommodate the added port. Then again, these last two actions are undone, and a sine block is added instead, and its Magnitude is changed. The change in magnitude does not cause any visual changes to the model and hence is marked as a secondary event and ignored during undo/redo unless mentioned otherwise. This contrasts with property change of Multiplier which was registered as a primary event since its sub event causes tool size to change which is observable. Finally, this is followed by connecting output of Sine to input of Gain.

As seen the UR is built into a tree and when undone it reverses every action sequentially in the current active branch. At point of branching, in this case after Gain tool is added, the user may choose any of the three branches to redo or can add a new branch to the tree by choosing none of them and instead doing a new set of actions.

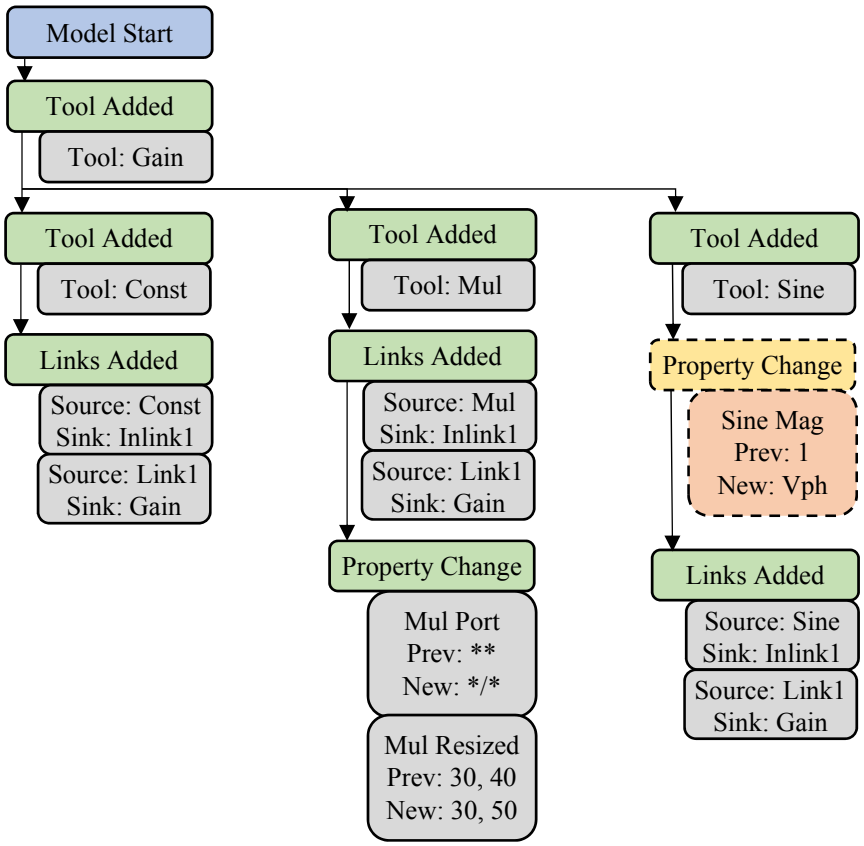


Figure 4.5: Undo/Redo Tree

In case of subsystem, a local UR tree is maintained and events within a subsystem are locally registered. The only exception occurs when an in-port or out-port tool is added to or deleted from the subsystem. In these cases, the event affects the within the subsystem and registered locally as well as outside the subsystem where previous existing links may be removed due to deleted ports or size might have changed and moved connected links due to added ports and so on, which are registered with the enclosing model/subsystem. This leads to continuity between the global UR tree and the local trees.

The other major component of model file is handling tool structure. This includes, datatype propagation, hardware coded determination, compilation ordering, conditional evaluation, and IL stack maintenance. Any changes to tool property or connections could affect the output datatype as indicated earlier. All these events add an update datatype call to a secondary evaluator thread (SET) stack. The SET delays these calls to eliminate redundant calls and reduce load on the CPU by consolidating these calls. When a tool's data type update function is called, it updates its type and if it different from earlier type, adds all output tools connected to it to the SET stack. Also it issues a destroy IL code call to IL manager in the model foundation and adds an event to SET stack to generate updated IL code for the tool.

One of the optimizations done by the model is elimination of IL code generation for tools that are not observed. Tools whose output do not connect to any tool or connects to tools that in turn down the line connect to no other tools are ignored since any computation with regards to these are useless because it is not observed. The same feature is extended to optimize generated code in real-time mode. The tools in the Peripheral toolbox section consist of blocks that get data from or send data to element outside the processors. For instance, the ADC tool, converts the analog signal at its ports to a digital value that a processor can operate on. Similarly, PWM tool takes digital duty signal generated by the processor and converts to ON/OFF pulses in real-time. All peripheral tools are single input single output (SISO) tools. In simulation mode all the peripheral blocks act as 1:1 blocks bypassing whatever is in the input to the output in normal operation, with introduction of real world behavior. For instance, if in simulation ADC input is sine wave and if its value were to go above what the selected processor can handle it will throw and exception so that user know if the same were to happen in real-time it will destroy the device. If it is within bounds it bypasses it to output. In real-time mode, the simulated sine wave is not needed since the actual signal is being fed in real-time to the ADC port. The ADC code for the device in use is injected instead of a 1:1 response function, the details of which will be discussed in later section. Same logic as optimizing

out unobserved tool in simulation mode, is done in real-time where tools that do not connect directly or indirectly via other tools to a peripheral tool that affects real world variable is ignored from code generation. This is determined by starting from peripheral output tools such as PWM, GPO, SPI Write etc., and tracing back all the connected input tools either one of the two conditions occurs:

i. Current tool has no more input tools connected to it.

ii. Current tool is a peripheral input tool such as ADC, GPI, AQB, SPI Read etc.

The result of this shown in Fig. 4.6.

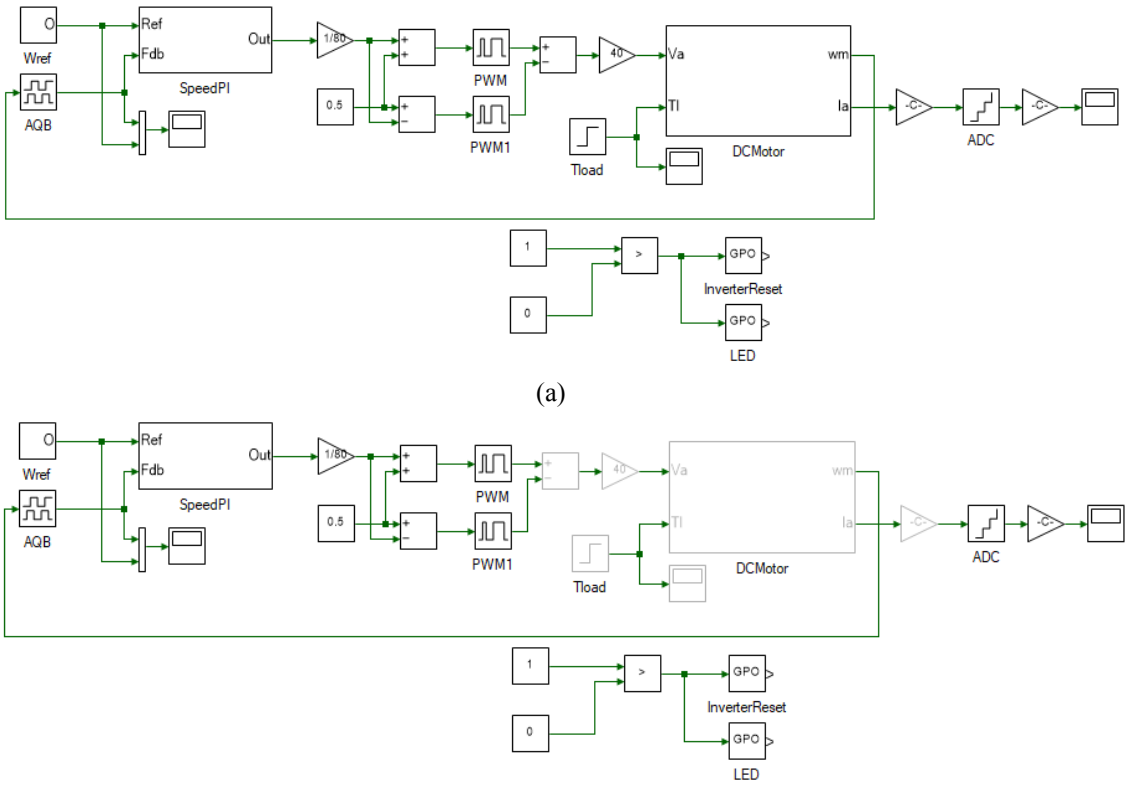


Figure 4.6: DC motor model: (a) Simulation mode, (b) Real-time mode

Figure 4.6a shows the DC motor control model in simulation mode and real-time mode. On the left is the PI block that takes reference speed and speed feedback from AQB block and generates required output voltage which is converted to duty and sent to PWM block. The output of PWM block is converted to voltage and injected into DC motor numerical model on the right.

This model takes input voltage and load torque and computes the speed at which the motor rotates. This speed is feedback to the AQB block. In real-time mode, the speed comes from running of an actual motor and hence the DC model need not be executed in real-time to get the speed. As seen in Fig 4.6b, using above algorithm, the model automatically eliminates code generation for the model in real-time mode as indicated by graying out those blocks that are not coded. This automatic identification allows users to use same model for simulation as well and real-time mode and also see the effects of real world nonlinearities such as saturation, in play in simulation mode. This is referred to as Hardware coded optimization and as shown unlike datatype update it propagates backward.

As mentioned earlier, the whole model file is a web of interlinked tools. To convert this to sequential lines of code, the order of compilation of each tool must be determined. Similar to datatype update this is determined by forward propagation. Source tools have no inputs and have a compilation order of 1 always. Any newly added tool which has none of its inputs connected has a compilation order of 1 as well. If this tool T was connected to multiple other tools $T_1, T_2, \ldots, T_i, \ldots, T_n$ of compilation order $C_1, C_2, \ldots, C_i, \ldots, C_n$ then its compilation order C is computed as given in Eqn. 4.2

$$C = \begin{cases} min\{C_j\} - 1 & where\ T, \{T_j \subseteq T_i\} \in storage\ tools \\ 0 & where\ T \in delay\ tools\ and\ T_i \notin storage\ tools \\ max\{C_1, C_2, \ldots, C_n, 0\} + 1 & where\ T, T_i \notin storage\ tools \end{cases} \quad (4.2)$$

where, storage tools are tools whose output are a function of previous outputs such as integrator, delay block, transfer function etc. If a storage tool's input is not connected to any other storage tool, then its compilation order is 0, i.e. it is coded before even source blocks are coded. If the storage tools input is combination of other storage and non-storage tools, then the compilation order is one less than the minimum of all input connected storage tool compilation order. If the current tool is not a storage tool, then its compilation order is one more than the maximum of all connected input tools. The final rule is quite obvious since it basically states, inputs of a tool must be computed

before the tool itself is computed. For instance, if an adder block has two inputs one from a sine source and another from a constant, then it is clear that both sine and constant's output must be established prior to computing their addition. This rule leads to an infinite loop if any of the input tool is in turn connected to other tools that leads to current tool's output as seen in Fig. 4.7. This is known as arithmetic loops and the platform throws a compile time error in these cases. To prevent getting locked in a loop, a stack is maintained of all the tools that have already been visited for compilation order update and if it is revisited in the same loop twice then all the elements on the stack are marked as forming arithmetic loop and computing compilation order is discontinued for current iteration.

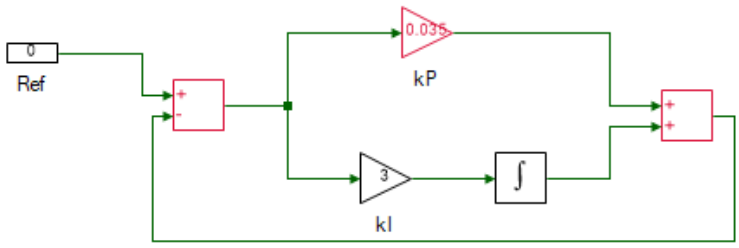


Figure 4.7: Compilation order loop arithmetic loop

As seen in figure above an arithmetic loop is formed since the compiler cannot resolve which of the three tools highlighted in red needs to be evaluated first. It must be noted that same does not occur in the bottom loop containing the integrator, since compilation order of integrator is independent of a its input due to it being a storage block it does not form an arithmetic loop. The compilation order for DC motor model is shown in Fig. 4.8. As seen, the order of non-storage tool is one greater than input order. For storage blocks it is 0, except for the 'Integrator2' which is preceded by a storage block hence between these two the latter is to be evaluated first and hence its order is one less that the input integrator order. Similar to other updates, any changes to a tool compilation order, causes committing all connected output tools to the SET stack for order reevaluation.

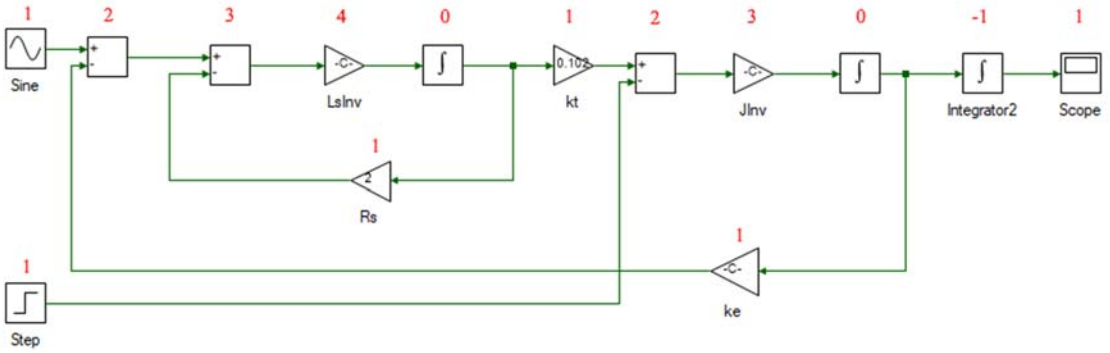


Figure 4.8: Compilation order for DC motor model

Tool sample time is forward propagated with any changes to a tool's sample time leads to committing all its connected output to SET stack which calls sample time update on each of those. The sample time S of a tool T is computed from the sample time $S_1, S_2, \ldots, S_i, \ldots, S_n$ of connected input tools $T_1, T_2, \ldots, T_i, \ldots T_n$ using Eqn. 4.3:

$$S = \begin{cases} gcd(S_1, S_2, \ldots, S_n) & where\ S_i > 0\ and\ T \notin continuous\ tools \\ 0 & where\ any\ S_i \leq 0\ and\ T \notin continuous\ tools \\ -1 & where\ T \notin continuous\ tools \end{cases} \quad (4.3)$$

where, *gcd* is greatest common divisor and *continuous tools* are tools that invoke integration solver, i.e. any tools whose IL code has ING instruction. This happens for only Integrator and Transfer function tool in the current version. Continuous tools are evaluated at step time determined by model step time S_m, selected solver, and error tolerance. Non-continuous tools are evaluated only when any of its input changes, this happens at a maximum of the least sample time of connected input tool. But selecting just a minimum of those value will not work. Consider the case where adder's input is connected to two tool whose sample times are 0.3 and 0.2 second respectively, then simply choosing the minima of 0.2 means output will be computed at 0, 0.2, 0.4, 0.6, 0.8 and so on. As can be seen, it misses steps 0.3, 0.9 etc., where the first input may change values. Hence the sample time of adder is chosen as the gcd of the input sample time which will be 0.1 and this does not miss changes in either of the inputs. If the computed sample time is not an integral multiple of set solver step time, then a compile time error is thrown. Any tool with sample time of 0, is

evaluated at every model step time which is the same for continuous tools if a fixed step solver is used instead of variable step.

The final component to be analyzed is the conditional evaluation tool. In the current version it consists of two tools namely, Index selector and If-Else selector. The IL code for the later was investigated in earlier section. This is a backward propagated update like hardware coded update discussed before. Consider the model in Fig. 4.9, just as an example without any real implication.

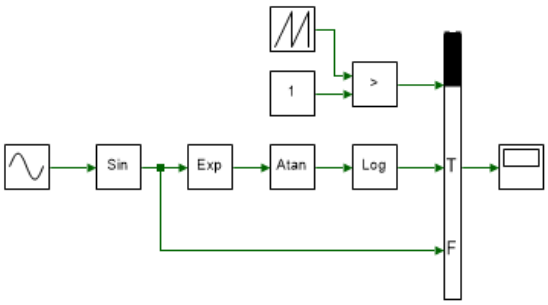


Figure 4.9: Example of Condition tool

The output of condition tool is selected from port marked ‘T’ (True) if triangle > 1, else it is selected from port marked ‘F’ (False). If conditional optimization is disabled, then irrespective of whether condition is True or False, all the tools connected to the input are evaluated at every time step. As can be seen, if most of the time the condition is going to be False, it unnecessarily computes a slew of clock intensive functions (Exp, Atan and Log). If conditional evaluation is enabled, the tools Exp, Atan and Log are computed only if condition is True else it is ignored. The Sin and Source block are evaluated always because irrespective of the condition it is always needed. The code for the conditional blocks are injected into the branch statements of the conditional blocks IL code by the model’s IL handler. This involves a complex set of operations which includes updating the reference and instruction jumps and marking the delta increment using CEL instruction.

When a tool output changes, it reevaluates the tools and their ports on which the running of current tool depends on. If it has changed, the it loads all the connected input tools to the SET stack for update as well. If a tool T1 is dependent on another tool T2, then tool connected to T1’s input, T3 will be dependent on T2 as well as any other conditional tools connected to its output on that as If a tool depends on all the conditional port of a tool as was the case with Sine block in

above figure, then it becomes unconditional of the selector tool. If the selector tool itself was connected to another selector tool, then this condition is backward propagated.

Conditional evaluation is applicable only for non-storage tool as these update their states at every time step and cannot be conditional upon another tools input. Also, if a tools sample time is not a integral multiple of the selector block it is conditional upon, then it is removed from its conditionality since the updates of the two tools are not compatible and occur at different intervals. For instance, if in above example if sample time of Exp block was 0.3 and sample time of Selector was 0.29 then even if condition was True always, Exp is evaluated only at multiples of 0.3*0.29 since 0.3 is not a multiple of 0.29. Hence for conditional optimization to occur sample time of tools connected to input must be multiple of selector block and must be non-storage block. If not, code will still compile and run as expected but the optimization for that tool will be ignored as it should be.

The IL handler splits the IL code into its initialization and runtime code and stitches these two components with other tool's IL code sequentially in the compilation order, with lowest first. In addition, it adds conditional code to evaluate a tool only at multiples of its step time. Tools of same compilation order might be moved around such that same sample times are clubbed together to limit the number of conditional blocks added. Finally, the codes of tools that evaluate conditionally are injected into corresponding controlling tool's code segments. Every time a code is moved around, empty pockets are created in the instruction and reference stack. The instruction at the start of this location is filled with EMP instruction followed by number of discarded fields. If the ratio of used fields vs discarded fields approaches 0.1, i.e. less than 10% stack utilization, the stack is compressed which eliminates all EMP instructions and forms contiguous instruction and reference stack. A force compression is called even if utilization ratio is greater than 0.1 when runtime engine is called. This concludes the sub-section on Model foundation. In the following section, a brief description of data logging is presented.

4.2.4 Data Logging:

Data logging encompasses all interface that display to user runtime value, in form of charts, gauges, color codes or changing texts. This includes both simulation and real-time data. Discussion real-time data logging requires knowledge of the architecture in place for data transfer and conversion, hence is postponed till those are dealt with in following chapter. With regards to simulation data logging, dealt in this section, it could either be synchronous or asynchronous.

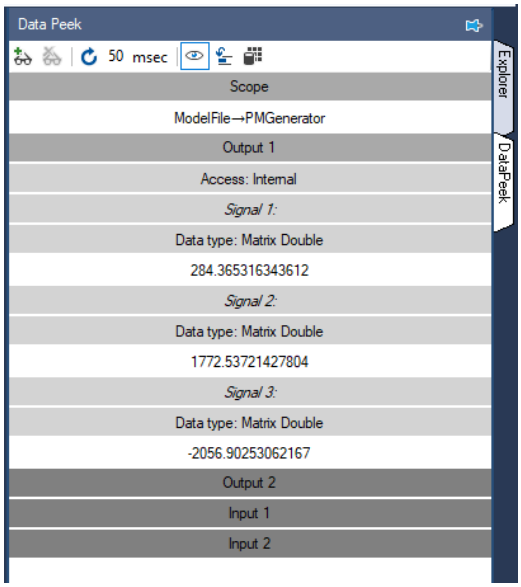


Figure 4.10 Data peek window

Scope tool listed in Appendix A.5 is an example of synchronous logging, where each input data to the scope is added to a list as it becomes available during runtime. The data peek window is the case of asynchronous data logging. A scope data log achieved by DLG instruction in IL code is a blocking performance expensive instruction, and slows down runtime considerably. A scope data is plotted using XY charts. A data peek on the other hand is non-intrusive that does not alter runtime performance by any measurable metric of significance. It uses reflection to get runtime value of requested variable and are updated at user observable rate which is very slow and readily yields CPU to any critical task and hence unlike scope it updates data asynchronously. Figure 4.10 shows the result of a data peek and Fig. 4.11 shows the result of a scope. For a data to be logged into a scope it must be connected to a scope tool. On the other hand, all tool data and script variable can be observed using data peek. It is useful when just final value or trend needs to be observed especially during debugging or testing where multiple run needs to be done as fast as possible and performance hit associated with using a scope becomes unfeasible. The graphics handling of scope and its functions are beyond the scope of this thesis and is not discussed any further.

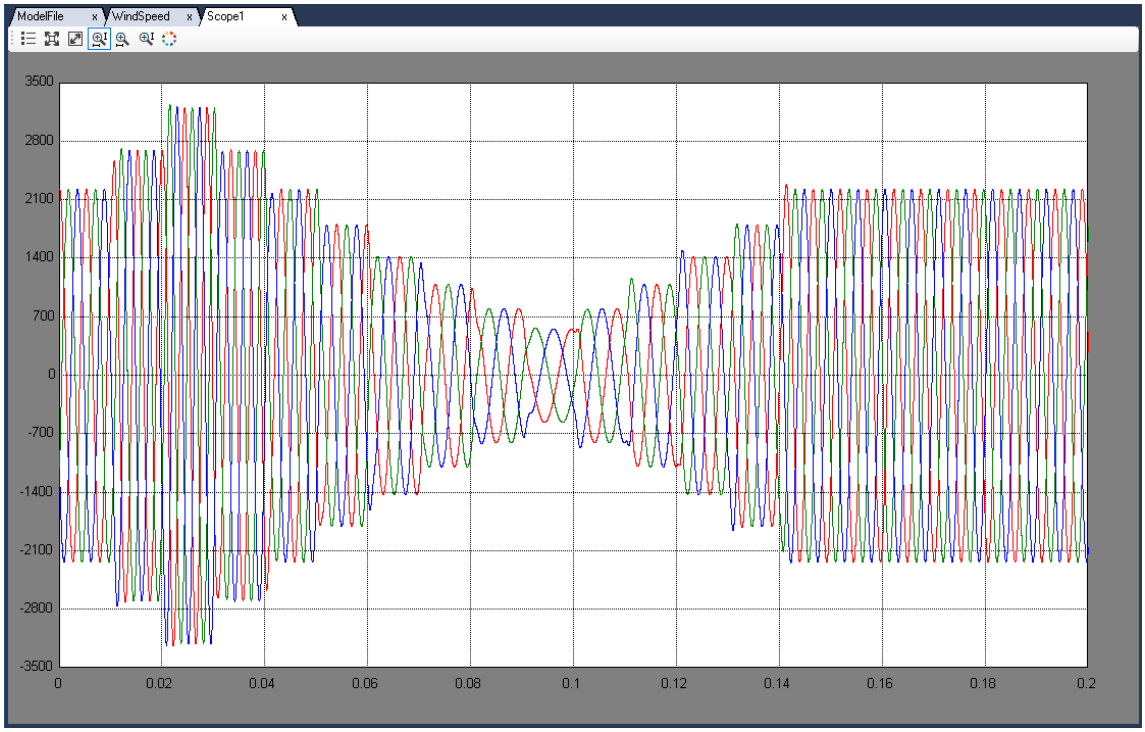


Figure 4.11: Result of scope

This concludes the section on components of model based platform. In the next section, the implementation of solver for solving integration and differentiation is discussed.

4.3 ODE solvers

Compared to code based implementation, model based implementation is advantages only when time based component is introduced. Else, it is simply just a sequence of interconnected function calls. Tools that are function of time are sources such as sine, triangle etc., and integrator, differentiator and transfer function. Any transfer function whose order of numerator is less than or equal to denominator can be resolved into series of integrals using the controllable canonical form [18]. The current version mainly targets real-time code generation where variable step solver which computes output of integration for a particular time step and if error is greater than tolerance, reduces the time step till error is within tolerance, is unsuitable. With that in mind, only fixed step solvers were implemented. The solvers implemented, and the accuracy order is listed in Table 4.5 and their implementation details and analysis can be found in [19].

Solver	*Order of accuracy*	*Equation*
Euler	1	$y_n = y_{n-1} + \Delta t * x_{n-1}$
Heun	2	$y_n = y_{n-1} + \frac{\Delta t}{2} * (x_n + x_{n-1})$
Bogacki-Shampine	3	$y_n = y_{n-1} + \frac{\Delta t}{24} * (3 * x_n + 8 * x_{n-1} + 6 * x_{n-2} + 7 * x_{n-4})$
Runge-Kutta 4th order	4	$y_n = y_{n-1} + \frac{\Delta t}{6} * (x_n + 4 * x_{n-1} + x_{n-2})$

Table 4.5: Numerical implementation of integration

Differentiation is implemented using one of the six possible solvers using backward difference methods and the formula is listed in Table. 4.6.

Order of accuracy	*Equation*
1	$y_n = \frac{x_n - x_{n-1}}{\Delta t}$
2	$y_n = \frac{\frac{3}{2}x_n - 2 * x_{n-1} + \frac{1}{2}x_{n-2}}{\Delta t}$
3	$y_n = \frac{\frac{11}{16}x_n - 3 * x_{n-1} + \frac{3}{2}x_{n-2} - \frac{1}{3}x_{n-3}}{\Delta t}$
4	$y_n = \frac{\frac{25}{12}x_n - 4 * x_{n-1} + 3x_{n-2} - \frac{4}{3}x_{n-3} + \frac{1}{4}x_{n-4}}{\Delta t}$
5	$y_n = \frac{\frac{137}{60}x_n - 5 * x_{n-1} + 5x_{n-2} - \frac{10}{3}x_{n-3} + \frac{5}{4}x_{n-4} - \frac{1}{5}x_{n-5}}{\Delta t}$
6	$y_n = \frac{\frac{49}{20}x_n - 6 * x_{n-1} + \frac{15}{2}x_{n-2} - \frac{20}{3}x_{n-3} + \frac{15}{4}x_{n-4} - \frac{6}{5}x_{n-5} + \frac{1}{6}x_{n-6}}{\Delta t}$

Table 4.6: Numerical implementation of differentiation

where, y_n is the output and x_{n-i} is the input, i time steps before current time *t* and Δt is the time step.

4.4 Conclusion

In this chapter, a model based design platform was explored. This is well suited over code based implementation for time based functions such as time varying signal sources, integration,

differentiation etc. Though these can be implemented in a code based platform as well, it takes much more effort and, reduces readability and maintenance significantly. The advantage of abstracting away the code functionalities into discrete tools is countered by the backend compiler development effort for model based platform. Unlike a code based platform where code flow is sequential and within a code line, the data flow is always from child nodes in AST to parent nodes, in case of model based platform there is no clear structure to code flow on first sight. Various data flows between tools in both directions. For instance, which tools are coded in real-time is determined by if tools connected to its output are hardware coded thus leading to backward propagation of data. On the other hand, tool datatype is function of the datatype of input tools and this leads to forward propagation of data. Resolving these and various other functionalities were explored including IL code generation for tools.

In addition to this, synchronous and synchronous data logging was discussed with focus on simulation mode. Finally, the solvers used for solving integration and differentiation in model was discussed.

Chapter 5

Real-time mode and results

This chapter delves into generating highly optimized C code using libraries developed in Chapter 2 from a model file using concepts highlighted in Chapter 4. In previous chapter, the hardware mode was introduced that allows transitioning from simulation model to real-time without any additional user input and the compiler will automatically figure out which tools are to be converted to real-time code and those that must be skipped during this translation. In relation to this, the peripheral tool blocks were introduced, which in simulation mode for most part are 1:1 tools. In hardware mode, it is replaced by actual code for peripheral update specific to the device. The real-time code generated is highly device specific. In the current version and for this chapter, the implementation solely targets TI's TMS320F28335 DSP.

In the first section, the overall structure of the generated C is described. This is followed by discussion on the two hardware platforms that was developed namely, the three-inverter and the extended DSP. The former is geared towards prototyping all motor control developments and the latter is more applicable to general power electronic prototyping requiring custom peripherals and high number of IOs. In previous chapter, datalogging was addressed with respect to simulation mode and in this chapter, it will be addressed with respect to hardware mode. In the concluding section, all the motor control and power electronics applications on which this was tested is present along with test results.

5.1 Generated C Code structure

To generate C code for hardware mode, the model is first run in simulation mode but just for the 0^{th} time step, i.e. just initialization code of model file is evaluated and the runtime code is ignored. This fixes all the inputs, outputs and property variable sizes in the model. Unlike code file, a model file variable sizes do not change after initialization. This is followed by up compiling the IL into C code and stored in main.c file. During this up compilation, all matrices are assigned as fixed size array and operation are catered to each matrix of different sizes and dimension. This is not the case in simulation mode where, the sizes are established during runtime and hence all matrix operation must be handled by size independent matrix operation that checks for compatibility at runtime and determines which specific function to call. In addition, all tool properties are initialized as constants in C code from the values determined from running the initialization code in simulation mode. All these values are retrieved using data peek feature discussed in previous chapter. The structure of resultant generated file is shown below in Fig. 5.1.

The first section is the include file and each of those files are explained further in the next paragraph. This is followed by declaration and initialization of data log variables which is followed by declaration and initialization of all tool variables. In the next section is the model runtime code that is executed at every step time, followed by function for logging data. Final section contains the *main()* function call, which at entry calls the device configuration function discussed later. In current project setting, the run mode has been set to forever, hence the *modelRun()* function is added in a forever while loop. After execution of the functions, the peripheral update is called which takes the computed duty cycles, GPO signals etc., and updates the peripheral registers with these values. Also, model variable corresponding to peripheral inputs such as ADC, AQB etc are copied from the respective registers to these variables. This is followed by call to *logData()* function if there is data to be logged and only if current code has run before the timer set to step time has not yet reached the set time. The available vacancy is used to log data. If even after this the timer is

still pending, then the code waits for it to complete and the flag is reset and the whole process starting with *modelRun()* is repeated again and again.

```
#include "rt_types.h"
#include "rt_tieups.h"
#include "rt_math.h"
#include "rt_log.h"

LogStruct dLogStruct = {0, 0, 0, 0, 0};
LogField dLogData[16];
Double stepTime = 0.0;
// declaration and intialization of variables of all tools
....

void modelRun()
{
   // model runtime code
   ...
}

void logData()
{
   // data logging code
   ...
}

void main()
{
   DeviceConfig();
   dLogStruct.Data = dLogData;
   dLogStruct.TotalSizeInBytes = 32;
   while (true)
   {
      modelRun();
      stepTime = 5.0E-05;
      UpdatePeripherals();
      if((dLogStruct.SendingSizeInBytes == 0) && !loopRestart) LogData();
      while (loopRestart == false);
      loopRestart = false;
   }
}
```

Figure 5.1: Structure of IL up-compiled C code

The datatypes such as Integers, Short and in some cases even standard floating-point types have different size interpretation based on the compiler used. The sizes of these on .Net platform and on TI's F28335 compiler is shown in Table 5.1. For fixed point integral type, F28335 only supports 16-bit arithmetic core. This means any fixed-point datatype that exceeds this must have software library for all the arithmetic, logical and shift operations using instructions available for

16-bit type. This is what is done if those types are used and they have a huge performance hit. A multiplication that takes 1 instruction cycle and assuming 2 cycles for loading the operands and 1 for unloading the result, it takes a total of 4 cycles. The same for 32-bit using software library will take 5 times more, i.e. 20 cycles and for 64-bit will take 10 times more which is 40 cycles.

Datatype	.Net	F28335
Short	16-bit Integer	16-bit Integer
Integer	32-bit Integer	16-bit Integer
Long	64-bit Integer	32-bit Integer
Long Long	-	64-bit Integer
Single	32-bit Float	32-bit Float
Double	64-bit Float	32-bit Float
Long Double	-	64-bit Float

Table 5.1: Data type sizes

Similarly declaring a variable as 'double' in .net environment is interpreted as 64-bit IEEE floating-point, while in C using F28335 compiler, the same keyword is recognized as 32-bit IEEE floating-point. For 64-bit version, the keyword is 'long double' and this is software implemented unlike the 32-bit float which is hardware implemented. The performance cost of even addition is for 64-bit variant is far greater than 32-bit and this becomes unfeasible for operations such as division. This performance hit for most basic of operation is unacceptable in most cases. So, a compromise is made where loss of precision and range is tolerated for better performance. To allow for this 'down-cast' and still allow users to choose higher precision if needed at cost performance, the project properties allows choosing what size the type in real-time mode needs to be as shown in Fig. 5.2.

As seen in the figure, the Integer can be chosen to either be 16-bit or 32-bit. If 32-bit is chosen a message is thrown by the EWEM handler that this leads to significant performance hit. Now, if all the sizes where set to default values leading to highest performance, there is a limit of range that was not anticipated during simulation. For instance, if a system was modelled using 32-bit integer in simulation mode whose range in between -2^{31} and $2^{31} - 1$ and transitioning to 16-bit in real-time mode which has a range -2^{15} and $2^{15} - 1$ would lead to issues if it went outside this range. To manually inspect this would be burdensome, especially in case of types such as double

in simulation being 64-bit and in real-time in this case being 32-bit not only has range limit but also loss of precision, effects of which can be almost impractical to manually quantize.

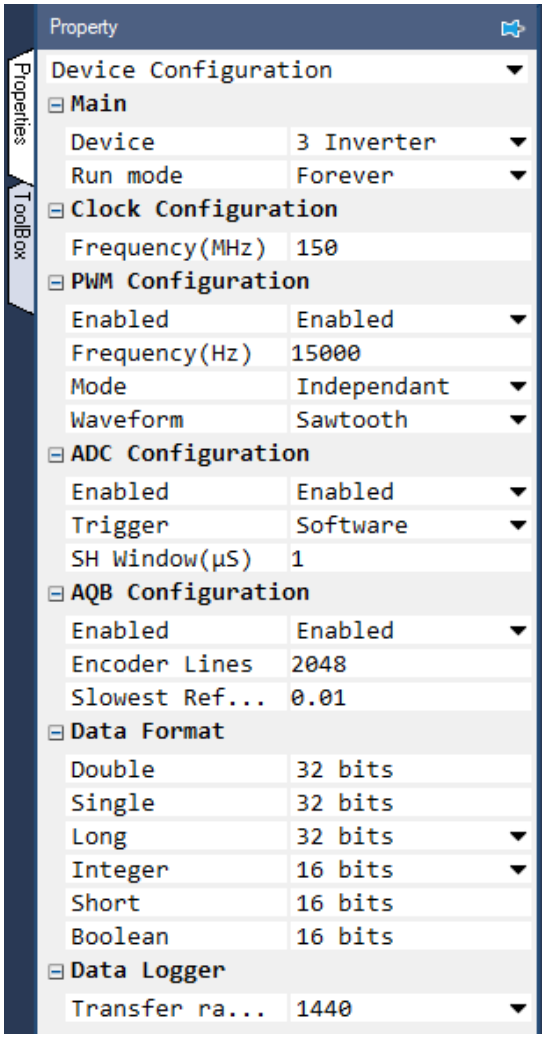


Figure 5.2: Project settings, Real-time datatype

To allow user to verify if this down-casting affects the systems, the platform allows interpreting the datatypes in simulation as same sizes as those used in real-time. To maintain the fluidity to jump between different sizes, in the real-time generated code, custom types are *typedef*ed in the first included file in Fig. 5.1 are defined and these are used across all the generated code. To change the size of any type, modification to the definitions needs to be made just in this file and will be reflected everywhere else. The content of this file is shown in Fig. 5.3. Boolean type is intrinsic to C99 but not supported in C89. To keep all generated code functional across both C89 and C99, all generated code is C89 compliant which among other things does not support boolean and complex types, does not support runtime determined array sizes and so on.

The other elements of include section in main file, is *rt_tieups.h* which is simply a file that defines all the model variables that are used outside the *main.c* file. These includes, the model runtime which is updated by a timer, pwm duty, adc read value and other peripheral related value, and finally the data log which is transmitted serially. These variables are used in both *main.c* and *rt_config.c* which handles all device related functions. It is portioned such that, one of these files

only writes into a variable that is only read by the other file. No variable is written in both these files.

```
#ifndef rt_types_H_
#define rt_types_H_
typedef int Boolean;
#define true (1 == 1)
#define false (1 == 0)
typedef float Double;
typedef float Single;
typedef long Long;
typedef int Integer;
typedef int Short;
#define isTrueDouble 0 /* Size of Double is 64 then True, else False */
typedef int Integer16;
typedef long Integer32;
typedef long long Integer64;
typedef unsigned int UInteger16;
typedef unsigned long UInteger32;
typedef unsigned long long UInteger64;
typedef UInteger16 LogField;
#endif
```

Figure 5.3: Contents of header file rt_types.h

The *rt_math.h* contains the declaration for all the fundamental math function in [7] as well as for matrix functions such as matrix inverse, eigen values etc. All math functions are implemented in generic C and these are substituted by assembly or C files that are more optimized for the device in use if available, else just the generic version is used. For instance, F28335 has a dedicated instruction for first guess of an inverse operation that leads to convergence within 2 iterations and real-time mode this used to take advantage of performance instead of the generic inverse function. On the other hand, there is no instruction that computes *sqrt* and for this the generic C file containing this code which uses Newton-Raphson to find the value. In some cases, combination of these are used as in the case of *mod* where the *mod* is carried out using generic C code, but that requires a division call for which the optimized version is used. All the elementary math functions are implemented using minimax polynomials discussed in chapter 2.

The final component of the include section is *rt_log.h*. This includes the definition for a log packet. The structure of log is shown in Fig. 5.4. This is used across the platforms for anywhere that requires compressing multiple data together sequentially and transferred. Similar to other *rt_tieup.h* variables of this structure is set either in *main.c* or in *rt_config.c* and the other just reads the data. In case of data logging, *main.c* loads all the variables to be logged into the log array, the

```
#ifndef rt_log_H_
#define rt_log_H_

#include "rt_types.h"

typedef struct {
    Integer TotalSizeInBytes;
    Integer SentSizeInBytes;
    Integer SendingSizeInBytes;
    void (*SendData)(void);
    LogField *Data;
}LogStruct;

#endif
```

Figure 5.4: Log Structure

'Data' variable in the structure as show in Fig. 5.5a. This is transferred serially via UART in *rt_config.c* as shown in Fig. 5.5b. Once transfer is completed the synchronous lock on the variable is released and the *main.c* can upload next batch of data. The direction is reversed in case of SPI read, where *rt_config.c* writes to the 'Data' array and *main.c* reads from it. Figure 5.5a shows where double variables, *ia, ib, ic, vdc, id, iq and wm* and integer variables *step and iteration* are packed together into sequential memory fields. This is achieved by using the PKG instruction in IL code and UPK instruction does the reverse for SPI read where data is received as a sequential field and it needs to be assigned to different variables.

```
LogStruct dLogStruct = {0, 0, 0, 0, 0};
LogField dLogData[16];
void LogData()
{
*(UInteger32*)&dLogData[0]=*(UInteger32*)&rtv_ia;
*(UInteger32*)&dLogData[2]=*(UInteger32*)&rtv_ib;
*(UInteger32*)&dLogData[4]=*(UInteger32*)&rtv_ic;
*(UInteger32*)&dLogData[6]=*(UInteger32*)&rtv_vdc;
*(UInteger16*)&dLogData[8]=*(UInteger16*)&rtv_step;
*(UInteger16*)&dLogData[9]=*(UInteger16*)&rtv_iteration;
*(UInteger32*)&dLogData[10]=*(UInteger32*)&rtv_id;
*(UInteger32*)&dLogData[12]=*(UInteger32*)&rtv_iq;
*(UInteger32*)&dLogData[14]=*(UInteger32*)&rtv_wm;
dLogStruct.SentSizeInBytes = 0;
dLogStruct.SendingSizeInBytes = 0;
dLogStruct.SendData();
}
```

(a)

```
#pragma CODE_SECTION(dataLogSent, "ramfuncs");
void dataLogSent(void)
{
  int i;
  int remainingBytes;
  dLogStruct.SentSizeInBytes += dLogStruct.SendingSizeInBytes;
  remainingBytes = dLogStruct.TotalSizeInBytes - dLogStruct.SentSizeInBytes;
  dLogStruct.SendingSizeInBytes = (remainingBytes > 16) ? 16 : remainingBytes;
  if (remainingBytes > 0)
  {
    for (i = dLogStruct.SentSizeInBytes >> 1; i < (dLogStruct.SentSizeInBytes + dLogStruct.SendingSizeInBytes) >> 1; i++
    {
      SciaRegs.SCITXBUF = dLogStruct.Data[i];
      SciaRegs.SCITXBUF = dLogStruct.Data[i] >> 8;
    }
    SciaRegs.SCIFFTX.bit.TXFFINTCLR = 1;
    PieCtrlRegs.PIEACK.all = 0x0100;
  }
}

#pragma CODE_SECTION(isr_dataLogSent, "ramfuncs");
__interrupt void isr_dataLogSent(void){dataLogSent();}
```

(b)

Figure 5.5: (a) Write to log structure data, (b) Read/transfer from log structure

The structure of *rt_config.c* is shown in Fig. 5.6. The #include section has been discussed except for *Devices.h* which contains the definition for all device registers and shall not be delved into since it changes with device. This is followed by the bios timer that maintains and throw interrupt at every step time. During this interrupt, all the values from input peripherals are read and loaded to the respective variables since this is immediately followed by code that uses the value to compute what outputs should be. Then is the *dataLog* function discussed earlier followed by output

```
#include "rt_types.h"
#include "rt_tieups.h"
#include "rt_log.h"
#include "Devices.h"

// declaration for local variables
....

__interrupt void isr_biosTimer()
{
   // update all input peripherals such as ADC, AQB etc.
   // Restart timer.
   ...
}

void dataLogSent()
{
   // data transfer code. Keep sending till all data has been transferred after which release the variable
   ...
}

// transfer complete interrupt..
__interrupt void isr_dataLogSent() {dataLogSent}

void UpdatePeripherals()
{
   // update all output peripherals such as PWM, GPO etc.
   ...
}

void DeviceConfig()
{
   // Clock, interrupts and IO pin configurations.
   // Critical code, transfer from Flash to RAM.
   // Initialize flash.
   // Peripheral configuration.
   // Enable peripherals and bios timer, and initialize all shared variables.
   ...
}
```

Figure 5.6: Structure of rt_config.c

peripheral update function call. This is called right after the control code has been executed and values for update are ready in *main.c*. Finally, the *DeviceConfig* function which is called only once at the start of device to initialize all device register with appropriate settings and to transfer critical code such as interrupt code from the Flash where it was programmed to RAM since it runs faster in the latter. This concludes the cursory analysis of the generated code structure.

5.2 Three-Inverter hardware design

At present only F28335 device is supported in real-time mode with plans to extend it to more controllers. Using F28335, two prototyping platforms have been designed, one of them is the three-inverter platform that as the name suggest contains three 3-phase inverters that allows it to control up to 3 motors in tandem. This board has the F28335 controller onboard along with circuitry for data transfer via USB. In addition, it contains all other components needed such as power management to generate multiple voltage level, connectors etc. The other platform is a more generic platform that is not limited to motor controls and it has an onboard F28335 processor coupled to a Xilinx xc3s500e FPGA.

The three-inverter is shown in Fig. 5.7 with all the major components highlighted. It consists three 3-phase DRV8322 50V, 12A MOSFET module with inbuilt driver and over current protection and uses bootstrap capacitor for gate signal isolation for upper switches. The output of the inverter is connected to 50Ω, 12A ferrite bead to limit current rise time in case of short so that over-current protection has sufficient time to detect and take action. Hall-effect current sensor ACS722 is used to measure the output currents. The outputs of these sensors are sent to F28335 ADC channels. The F28335 also takes Fault signal from the inverter module as input. The motor speed is measured using A quad B encoder. The input can either be single ended or differential and this signal is converted to single-ended and conditioned before being sent to F28335 encoder peripheral input. With these three inputs, the F28335 runs the control code and generates as output, PWM signals and reset signal which are sent to the inverter module. The input for powering the

motor is from a DC supply and voltage is limited by the inverter module, to a maximum of 40V. The DC bus is decoupled using two 4700µF capacitors. The value is decided based of amount of inertial energy that would be dumped back from the motors used while abruptly stopped while running at full speed. The value was chosen such that at worst case scenario, the DC voltage rises to maximum of 48V (2% below the 50V tolerance of module) when the complete inertial energy is dumped back. The capacitors must absorb this since most DC power supplies do not allow regeneration. Also, there is a diode on the input side that prevents power flow back and it was added to protect the board in case of wrong polarity connection of input DC. The input DC voltage is stepped down to 12V for gate drive supply in the power management circuit. It further derives 5V for the A quad B encoder and, 3.3V and 1.9V for the F28335 peripherals and core respectively.

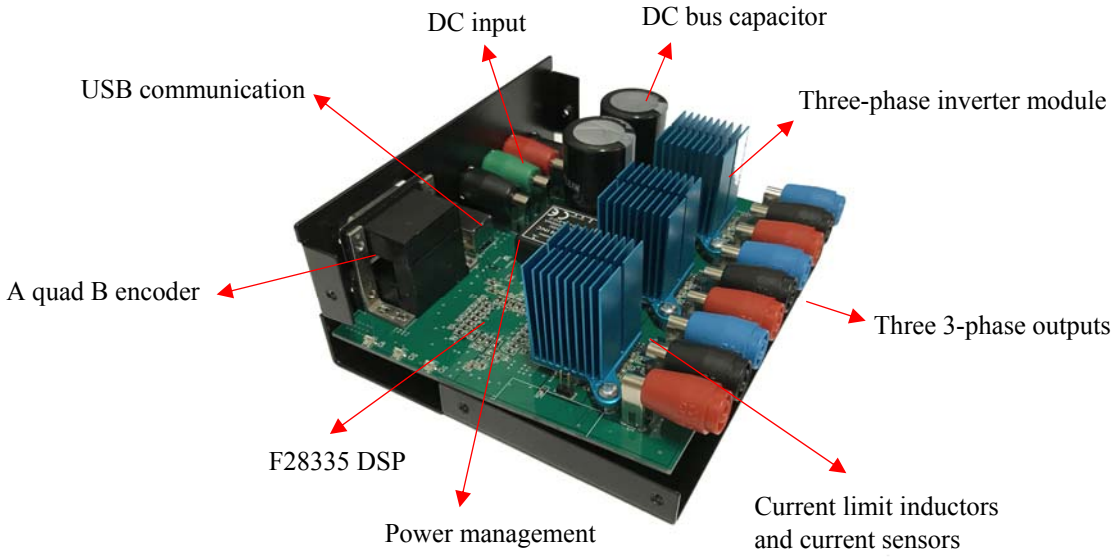


Figure 5.7: Three inverter open enclosure

The extended DSP is shown in Fig. 5.8 with all the major components highlighted. It consists TI's TMS320F28335 DSP which is coupled to Xilinx xc3s500e FPGA via 32-bit parallel channel that can be individually configured as either inputs or outputs or be used as 16/32-bit external memory access for high speed data transfer. Among those 32-bit link, is the SPI link as

well which can be used to transfer at rate of up to 10Mbps between the FPGA and DSP. In addition to this, 12-bit PWM channel is directly connected to the FPGA which can be used as is by simply bypassing them to outputs or can be modified to add in commutation or error checking and so on or it can be used as a IO's or for data transfer.

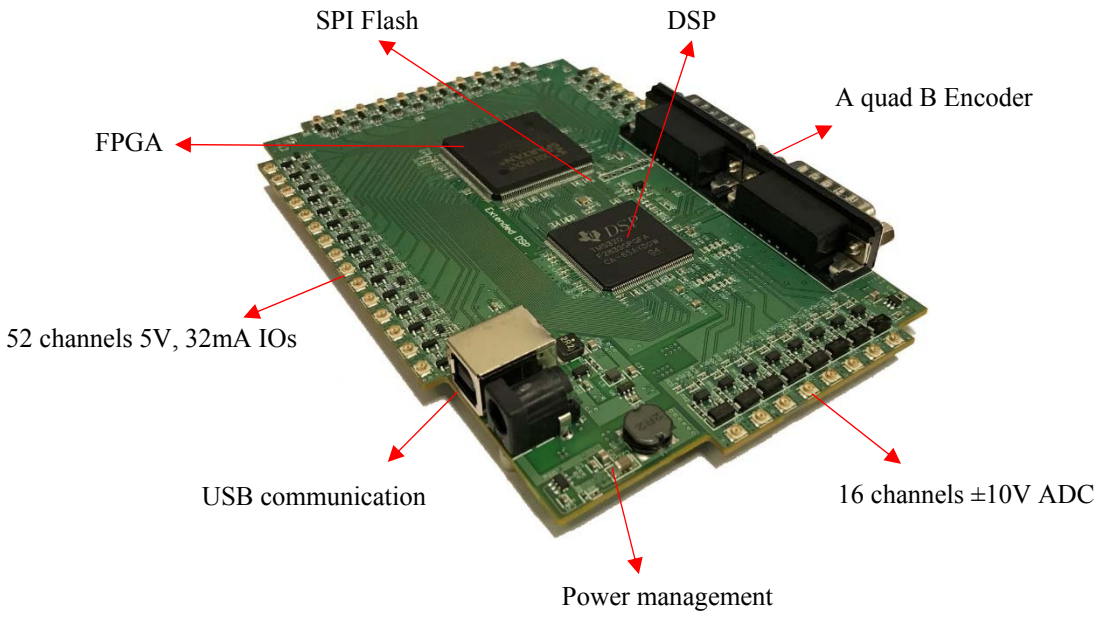


Figure 5.8: Extended DSP

The DSP also accepts 2 A quad B encoder inputs. The A quad B signal can be differential or single ended and the input circuitry on board, converts to single ended and conditions it before sending to DSP. The DSP also accepts 16 ADC inputs with range of 0-3V. The analog inputs to the board can be ±10V. On board circuitry filters this and scales range to ±1.5V and centers around 1.5V before sending to DSP. All the inputs are protected against ESD by bidirectional TVS diode.

The digital inputs and outputs are all connected to the FPGA, 52 of them. They can be either operated as inputs or outputs in groups of 16-bits, 32-bits, 16-bits allowing for all possible combination in group of 16bits. The direction of any block can be modified by the FPGA. The FPGA output of 0-3.3V and max 4mA current is scaled via drivers to 0-5V and source up to 32mA.

The power management is done by series of buck converter and switched capacitor converter. The input voltage can be 9-15V/3A. A buck converter steps it down to 5V. A switched capacitor circuit generates -5V/200mA. The ±5V/200mA is used in ADC input conditioning circuit and +5V/2A is used for IO drivers connected to FPGA and the A quad B peripheral connected to the DSP. Input is also stepped down to 3.3V/3A. Of the 2A capability, 0.6A is required by the DSP and 0.6A by the FPGA IO peripherals, and 1.8A in further step-down circuitry. The 5V is stepped down to 2.5V/0.6A to generate FPGA configuration component voltage rail. The 3.3V is further stepped down to 1.2V/1A for FPGA core voltage and 1.9V/1A for DSP core voltage.

The programming of the DSP from the computer is done via on board USB HID IC FT260 same as the one used in the three-inverter board. The same channel is used to send data back for logging. The board also has a 32MB external SPI flash IC to store the FPGA configuration code. The configuration code is first transferred from computer to the DSP vis FT260 link and then from DSP to external Flash IC via SPI and finally this is transferred to FPGA from external Flash via SPI during FPGA bootup. This eliminates the need for additional port such as JTAG for configuring the FPGA.

5.3 Real-time data transfer

The firmware aspects of data logging in C was discussed in the section 5.1. In this section, the hardware elements as well as firmware on the computer end is discussed. This section considers only the USB HID protocol for communication and the details of this can be found in [20].

On the compute side, all OS have inbuilt low-level drivers for common human interface devices (HID) such as keyboard, mouse, printer etc. Starting with Windows 7 HID support was added for UART and I^2C communication protocol. First all the connected USB devices are enumerated to identify the device of interest. This is done by using native methods available in *setupapi.dll* library. The *SetupDiGetClassDevs* is called with the following parameters:

SetupDiGetClassDevs(hidClass, Nothing, IntPtr.Zero, DiGetClassFlags.DIGCF_PRESENT Or DiGetClassFlags.DIGCF_DEVICEINTERFACE)

This returns the device information set of all currently connected HID device that support device interface. Then iterate through each of the returned device info set using *SetupDiEnumDeviceInfo*. Each device can have multiple interface and each of these is extracted using *SetupDiEnumDeviceInterfaces* function. From the interface information, the device path and description are extracted.

Having enumerated through all connected HID devices, the next step is to identify the particular USB device. Each USB device product family has a unique identification code, namely the VID (vendor specific) and PID (product family specific). For the current device, FT260 it is 0x0403 and 0x6030 respectively. This is obtained by polling each device and requesting for the device attribute information. The driver functions for these are available in the native *hid.dll* library. The *HidD_GetAttributes* function returns the device attribute which contains the following data: the size of the attribute structure (32-bit), VID (16-bit), PID (16-bit) and Version (16-bit). The VID and PID of each of the enumerated interface is compared again the values of FT260 and the one that matches is selected. It must be noted that FT260 supports two interfaces, one for UART and other for I^2C. But in the current version only the UART is hardwired to be discovered and hence if only one FT260 is connected only one device attribute will be discovered that matches the VID and PID mentioned.

This is followed by device open, which creates a file to read and write to the HID port. This can happen synchronously (blocks code execution till read/write is complete) or asynchronously (does read/write in a parallel thread, in a non-blocking manner). The choice depends on the action being carried. In case reading information about the device the former is used since the information is necessary to proceed but in case of data logging or program transfer, it is done via latter since while a block of data is being transferred other operations can be carious out.

The functions to handle file creation and creating events to notify on task completion, cancelling a task after some time etc., are available in *kernel32.dll* library.

The HID communicates in form of reports which of three categories: feature report, input report and output report. The feature report is transferred using Control transfer which is not time controlled. It can happen any time after a report has been requested. The input and output report happens via interrupt transfer, which happens every 1ms for USB 2.0 full speed device and has a maximum payload of 64 bytes.

The report format to get and set various features of FT260 is given in [21]. Using this the UART baud rate, IO pin for LED signals, the clock, I^2C disable and UART format is set by writing to the file attached the enumerated USB HID interface. These occur using feature report. Once the device is configured the read and write is reconfigured for asynchronous operation so that the large data transfer that follow does not block UI. The actual data transfer happens through input and output reports. As mentioned earlier each of these reports can be maximum of 64 bytes. The report ID and report size take some of these bytes and remaining is actual data. For out (write from USB to FT260) report, it leaves 60 bytes of data payload and for in (read from FT260 to USB) report it leaves 62 bytes of data payload. Once a packet of data is written to FT260 from USB it is immediately transferred to the DSP via SCI and, the DSP must accept this data right away since there is no storage memory to accumulated more than one packed of data in FT260. The same holds true when DSP writes to FT260, the computer must retrieve this data right away else it will be lost. So, the asynchronous read file operation is carried out continuously to check for data and read then write away. In spite of this the read thread could be blocked and might miss out a packet which will lead to data corruption. To avoid this a simple check sum is implemented where if corruption is detected, computer issues a command to F28335 to restart transfer from first data.

Having established the mechanism for data transfer, method to program the DSP is explored. The generated C code is compiled and linked using manufacturer provided standalone

tools. The resultant .out file is converted into hex code which lists the data at each memory location in the DSP. F28335 RAM can be boot loaded via SCI [22], but this method is available only for the RAM. Loading program into RAM has two issues, first the memory is limited, and code is not retained on power cycle. To avoid this code needs to be loaded into the Flash. This is done by initially loading the Flash API into the RAM using the USB→FT260→SCI link. Once this code is loaded it begins to execute and it accepts data via the SCI port and transfer those data to the Flash. At this point the actual program code is transferred via the same link and it is loaded by the Flash API running on the DSP, into the flash [23].

Once the code is loaded the controller is reset and boot loader is configured to Flash, which is controlled via FT260 IOs which are connected to the reset port and the boot loader pins of DSP. The complete code is available as part of the software package.

So far data was transferred from the computer to DSP with just handshake and feature description package flowing from DSP to computer. If the roles were reversed, it leads to data logging. As mentioned earlier, interrupt transfer happens every 1ms and within that interval 60 bytes can be transferred both ways using in out reports. This leads to a transfer rate of 480Kbps. Which is 30,000 words per seconds where a word is 16 bits, which in turn is 15,000 single precision IEEE floating-point data per second. If a 100 Hz sine wave is logged, then both its magnitude and time is logged which results in 750 samples for every interval. If more data is logged, then due to limited bandwidth resolution starts to approximately halves every time a new signal is added to the log. This approach of data logging is only suitable for observing a few moderately varying signals or a lot of slow time varying signals and completely unsuitable for observing fast time varying signal such as switching frequency components.

Finally, the received data on the computer end is just a collection of bytes. This must be unpacked to respective variables as discussed earlier with respect to DSP firmware. In addition to

unpacking, the data type must be casted to platform specific type as mentioned earlier, where same 'double' keyword is implemented as 32-bit float in DSP and 64-bit float in .Net.

5.4 Hardware and simulation results

The functioning of simulation and the real-time controller platform was evaluated for the following application as proof of concept and are made available as part of the software package:

i. Induction motor vector control: Simulation and real-time.

ii. DC motor closed loop speed control: Simulation and real-time.

iii. Permanent magnet synchronous motor vector control: Simulation.

iv. Wind turbine emulation: Simulation.

v. Direct three phase matrix converter: Simulation and real-time.

vi. Direct three level three phase matrix converter: Simulation and real-time.

This section does not delve into theoretical details of these examples which can be found in the following literatures [24, 25, 26, 27, 28].

The overall view of the induction motor (IM) vector control model developed using this platform is shown in Fig. 5.9. The systems consist of an induction motor run in speed control mode and this is mechanically coupled to a DC motor which emulates a load as shown in Fig. 5.10. On the top left is the inner dq current PI control loops and the outer speed control loop of the IM. In the bottom left is the torque/current control loop of the DC motor. The induction motor speed is step from rest to 100 rad/sec at time =1s. At time t = 3s, the DC motor is loaded to 0.75A from 0A to emulate a step change in IM load. In the center is the IM and DC linearized numerical model on the top and bottom respectively. Above the IM model is the estimator model which estimates motor slip from measured speed and current. This slip speed is added to actual speed to get the speed of rotor flux along which the dq frame is aligned. The motor parameters listed in Table 5.2.

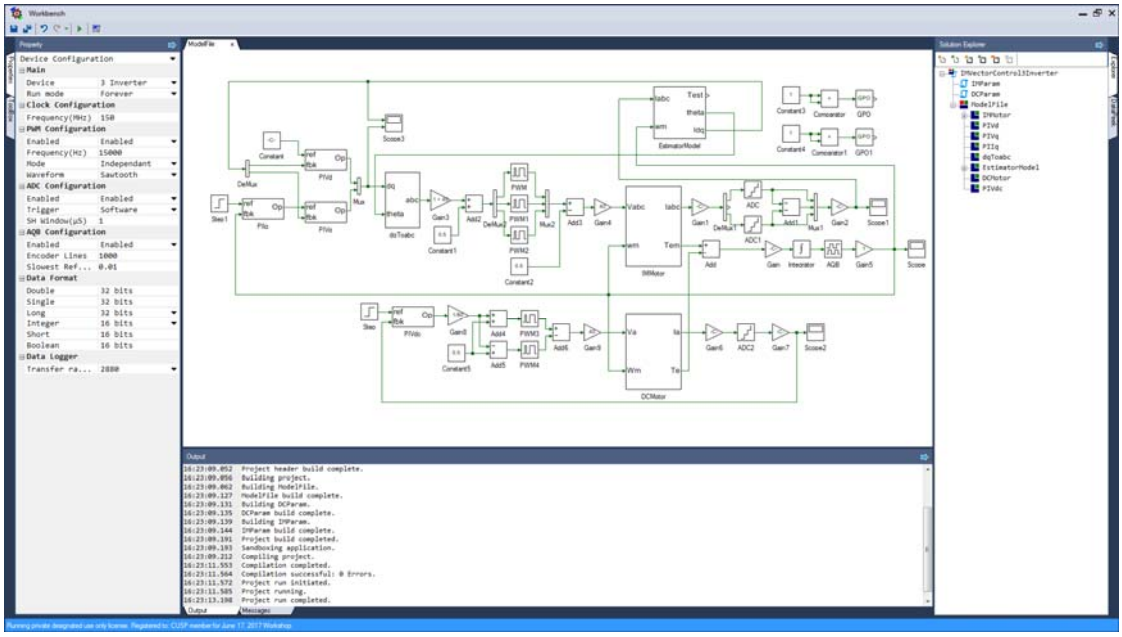


Figure 5.9: IM vector control overall model

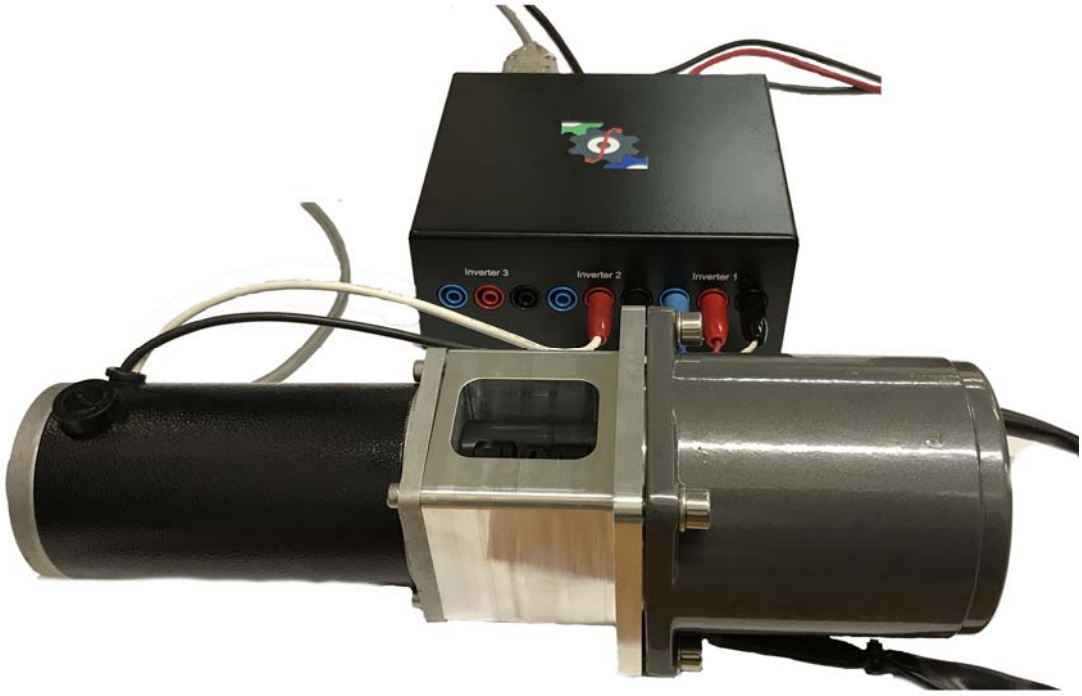


Figure 5.10: Hardware setup

The current loop is tuned for 60□ phase margin and 1250 rad/s cross-over frequency and, the outer speed loop is tuned for the same phase margin and 125 rad/s cross-over frequency. These are computed in the script file shown in Fig. 5.11.

Induction motor

Parameter	*Value*
Stator phase resistance	1.79Ω
Rotor phase resistance	1.05 Ω
Stator leakage reactance	1.85 Ω
Rotor leakage reactance	1.85 Ω
Stator magnetizing reactance	9.42 Ω
Rotor inertia	150µ kg.m^2
Number of Poles	4
Rated line-line RMS voltage	14.7
Rated slip	0.1
Rated frequency	50Hz

DC motor

Parameter	*Value*
Stator resistance	2Ω
Stator inductance	15mH
Rotor inertia	150µ kg.m^2
Back emf contant ke	0.102Nm.A

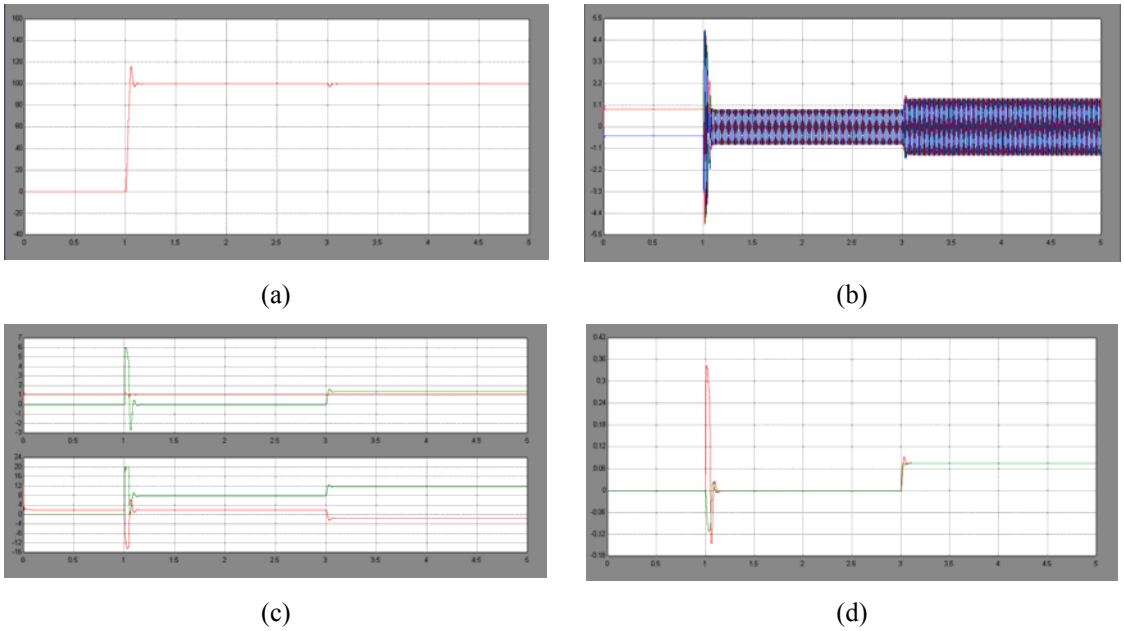
Table 5.2: IM and DC motor parameters

ModelFile | IMMotor | IMParam

```
Public Module IMParam
    Public sqrt2by3 As Native Double = (2 ÷ 3) ^ 0.5

    Public f As Native Double = 120.0 ! Rated frequency
    Public Rs As Native Double = 0.35 ! Stator Resistance
    Public Rr As Native Double = 0.1670 ! Rotor Resistance
    Public Xls As Native Double = 0.4524 * 2 ! Stator leakage Reactance
    Public Xlr As Native Double = 0.6786 * 2 ! Rotor leakage Reactance
    Public Xm As Native Double = 1.84726 * 2 ! Mutual Reactance
    Public Jeq As Native Double = 0.0000221 ! Rotor Inertia
    Public p As Native Double = 4 ! Number of poles

    Public Ls As Native Double = (Xls + Xm) ÷ (2 * Math:PI * f) ! Stator Inductance
    Public Lm As Native Double = Xm ÷ (2 * Math:PI * f) ! Mutual Inductance
    Public Lr As Native Double = (Xlr + Xm) ÷ (2 * Math:PI * f) ! Rotor Inductance
    Public VLLrms As Native Double = 20 ! Rated line to line voltage
    Public s As Native Double = 0.1 ! Rated slip
    Public Wsyn As Native Double = 2 * Math:PI * f ! Synchonous speed at rated frequency
    Public Wm As Native Double = (1 - s) * Wsyn ! Speed at rated slip
    Public Va As Native Double = VLLrms * sqrt2by3 ! Phase voltage peak
    Public Vs0 As Native Double = 1.5 * Va
    Public ThetaVs0 As Native Double = 0
    Public Thetada0 As Native Double = 0

    ! Initial stator dq Voltages
    Public Vsd0 As Native Double = sqrt2by3 * Math:Abs(Vs0) * Math:Cos(ThetaVs0 - Thetada0)
    Public Vsq0 As Native Double = sqrt2by3 * Math:Abs(Vs0) * Math:Sin(ThetaVs0 - Thetada0)

    Public taur As Native Double = Lr ÷ Rr
    Public A As Double = [[Rs, -Wsyn * Ls, 0, -Wsyn * Lm], [Wsyn * Ls, Rs, Wsyn * Lm, 0], [0, -s * Wsyn * Lm, Rr, -s * Wsyn * Lr], [s * Ws
    Public Ainv As Double = 1 ÷ A
    ! Initial dq stator and rotor voltages (as row matrix)
    Public Vdq0 As Double = [[Vsd0], [Vsq0], [0], [0]]
    Public Idq0 As Double = Ainv * Vdq0 ! Initial dq stator and rotor currents
    ! Retriving individual current from row matrix
    Public Isd0 As Native Double = Idq0(1, 1)
    Public Isq0 As Native Double = Idq0(2, 1)
    Public Ird0 As Native Double = Idq0(3, 1)
    Public Irq0 As Native Double = Idq0(4, 1)
```

Figure 5.11: IM vector control parameter initialization

The results of running the simulation is shown in Fig. 5.12. As expected the speed steps to 100 rad/s at t = 1s and at t = 3s, the load torque is steped. The motor speed falls due to increased torque but is corrected quickly by the speed loop.



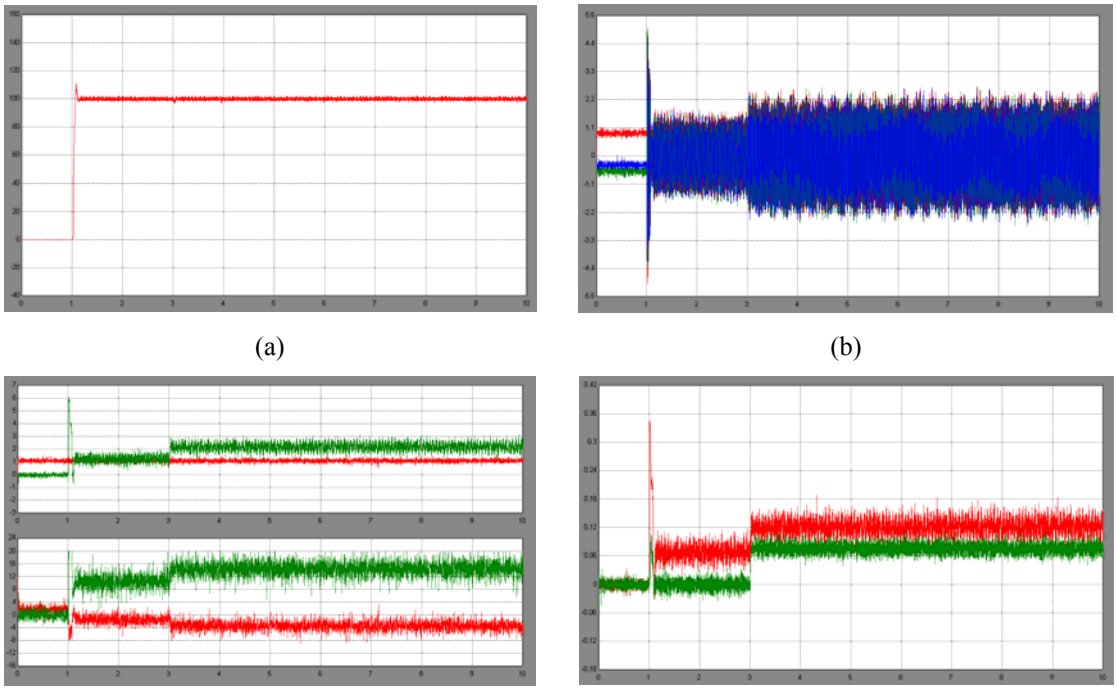
(a) (b)

(c) (d)

Figure 5.12: IM vector control simulation results: (a) motor speed, (b) IM abc currents, (c) IM dq currents and voltages, (d) IM torque vs DC motor torque

The same results obtained in real-time mode is shown in Fig. 5.13. The data is acquired and logged by the method discussed in the earlier section. In real-time mode, the three-inverter coupled to actual IM and DC motor is used. As mentioned earlier, the same simulation model is automatically converted to real-time model and motor models are eliminated from the real-time code. As seen the results of real-time run matches closely with simulation. In simulation IM and DC motor torque are equal at steady state, but in real-time mode there is a steady state difference. This is due to frictional losses which is not modeled in simulation.

The next example is a C motor with just a speed control loop and no current control loop. The simulation/real-time model combination is shown in Fig. 5.14. The real-time results are shown in Fig. 5.15. Of interest are the current and DC voltage waveforms. Since there in no inner current loop, the motor current is not limited in firmware. It is solely limited by the current limit feature



(a) (b)

(c) (d)

Figure 5.13: IM vector control real-time results: (a) motor speed, (b) IM abc currents, (c) IM dq currents and voltages, (d) IM torque vs DC motor torque

explained earlier and is set to limit at 5.7A, which is lesser than the 12A rating because the module is completely enclosed and even though it is force air cooled it gets too hot for currents upwards of 7A. The DC bus voltage as seen rises from 40V to 47V when the motor is stopped abruptly while running at full speed. As mentioned earlier, the whole inertial energy is dumped back to the capacitor leading to its voltage rise and the capacitors are chosen such that even in this extreme operating point, the DC voltage does not exceed the module rating of 50V.

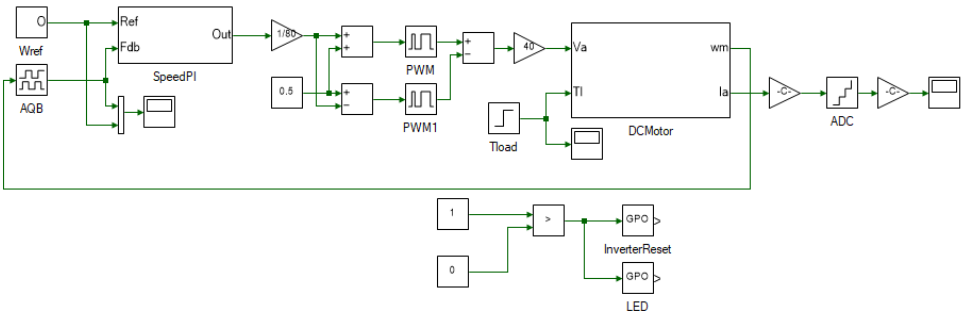
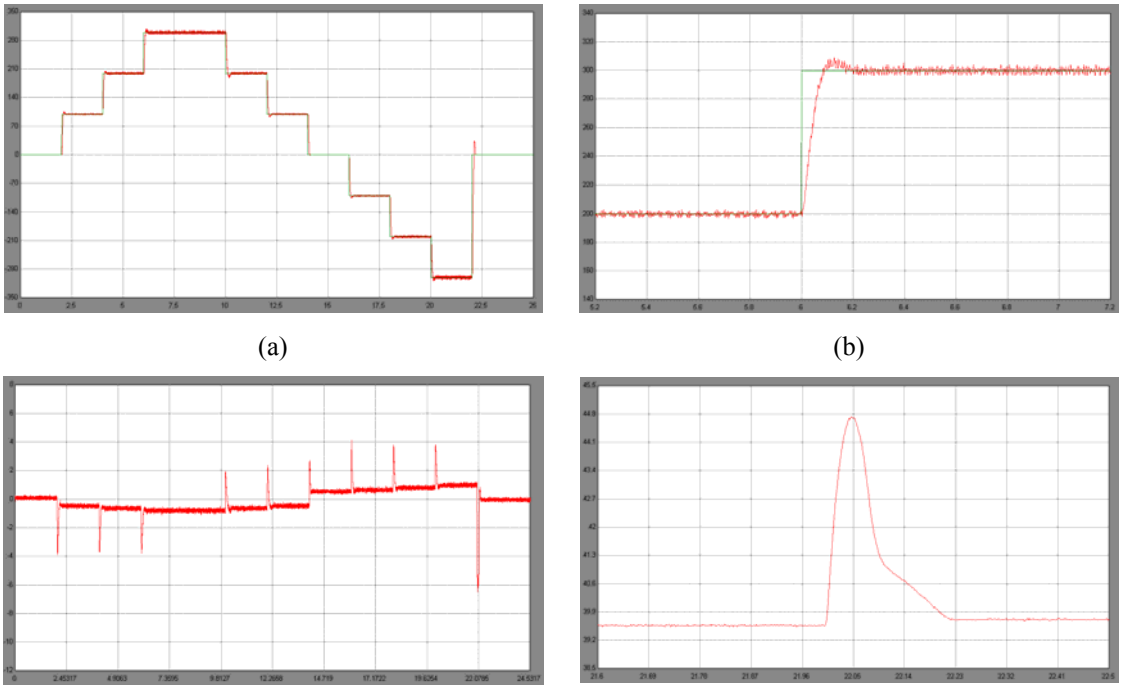


Figure 5.14: DC motor closed loop speed control model



(a) (b)

(c) (d)

Figure 5.15: DC motor speed control real-time results: (a) motor speed, (b) motor speed single step, (c) motor current, (d) DC bus voltage

Example 3 and 4 are simulation only. The permanent magnet synchronous motor (PMSM) vector control model is shown in Fig. 5.16 and it is very similar to IM vector control but there is no need to estimate synchronous speed for rotor flux alignment since the rotor flux rotates vector is always aligned with rotor axis position obtained by integrating rotor speed.

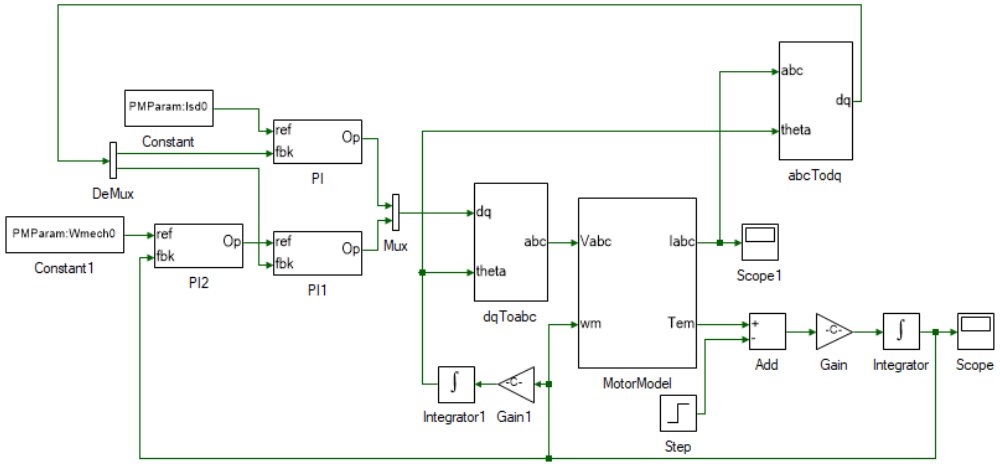
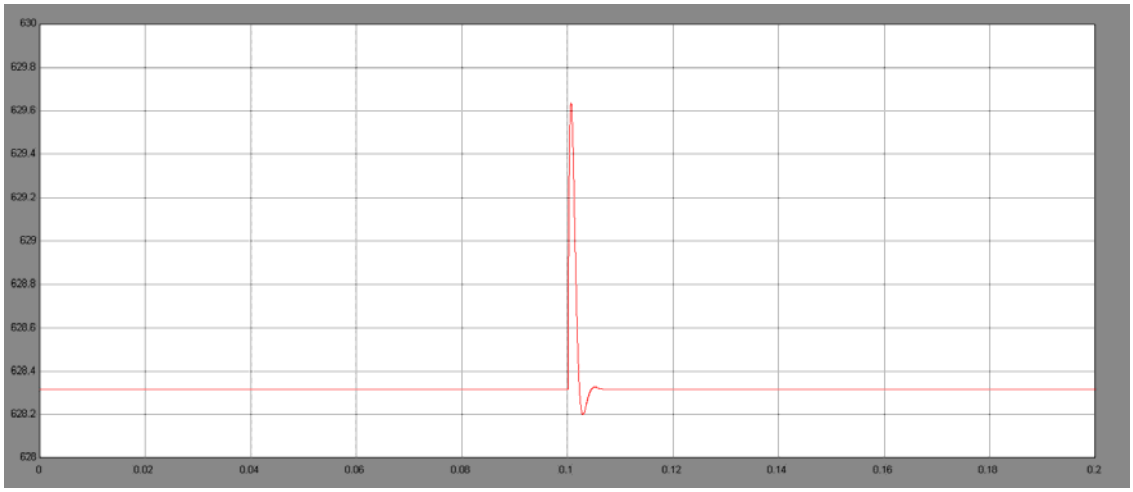
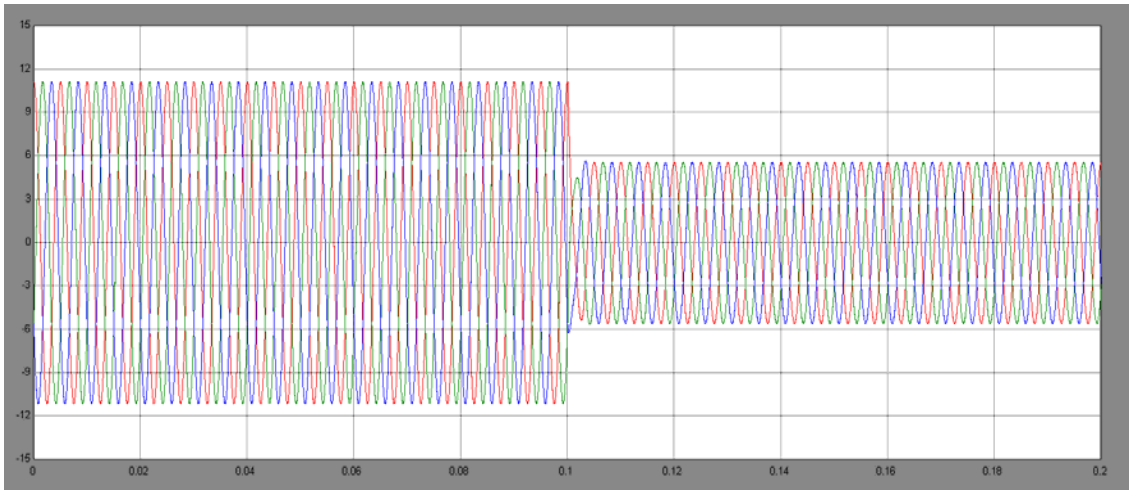


Figure 5.16: PMSM vector control simulation model

`The model starts off from steady state operating point of 6000RPM and full load torque. At t= 0.1s, the load torque is halved. This reals to rise in speed which is actively compensated by the PI loops to bring the speed back to 6000RPM. The results of running the simulation is shown in Fig. 5.17.



(a)



(b)

Figure 5.17: PMSM vector control simulation: (a) speed, (b) currents

Example 4 is a wind turbine emulation which was implemented in [27] using Matlab. The same is reproduced here in this platform. The model is shown in Fig. 5.18a. On center top, is the PMSM model which represents the generator connected to the turbine. In the bottom is the DC motor model that emulates wind turbine torque. The DC motor is run in torque control mode and the generator is run in current control mode, with torque axis current continuously adjusted to

extract maximum energy from the wind. On the left is the Cp coefficient calculator for turbine in consideration. The underlying subsystem components is shown for the all the components discussed. The script file computes the maximum tip speed ratio for given blade angle and the Cp for given tip speed ratio and blade angle as shown in Fig. 5.19.

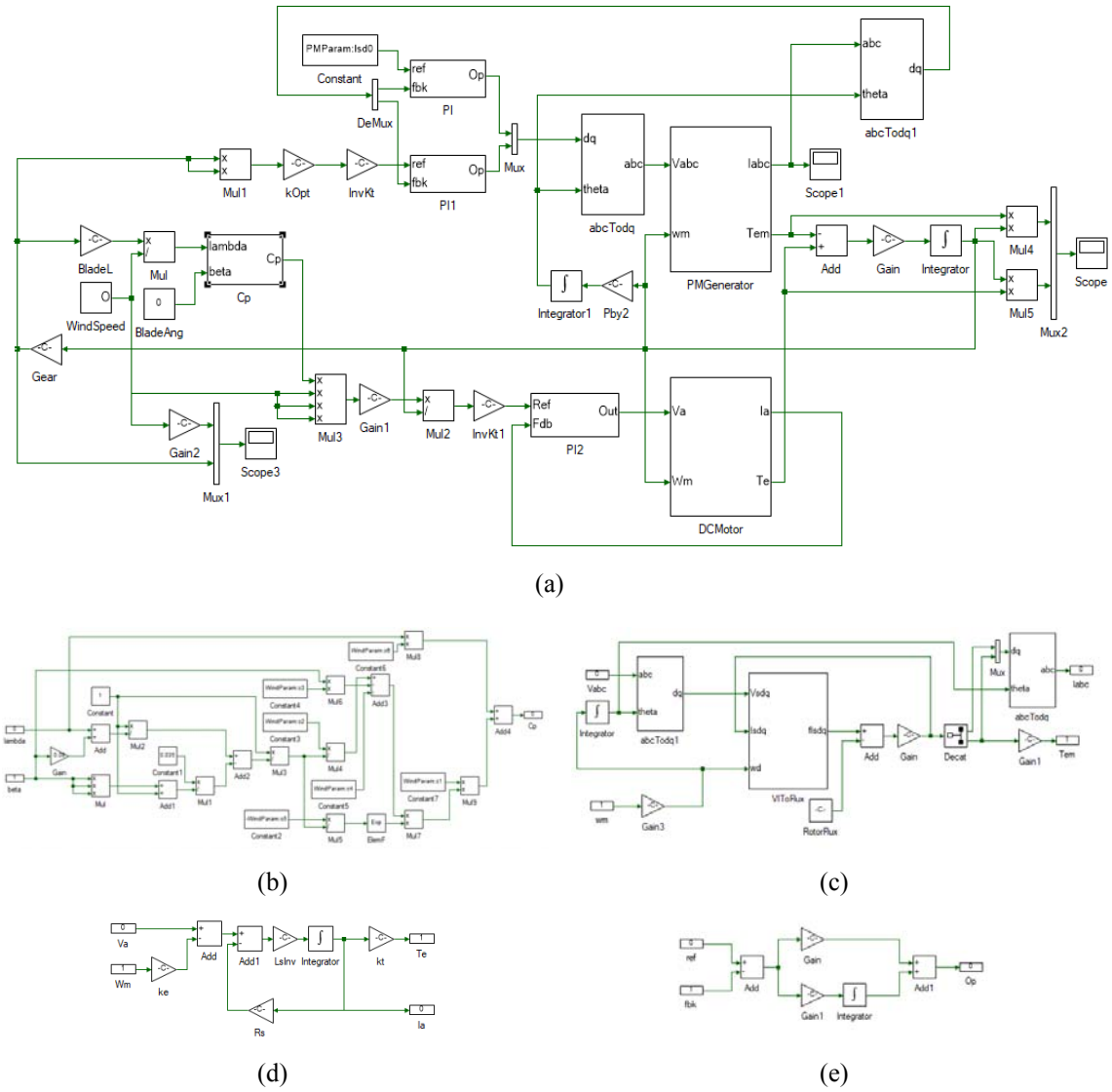
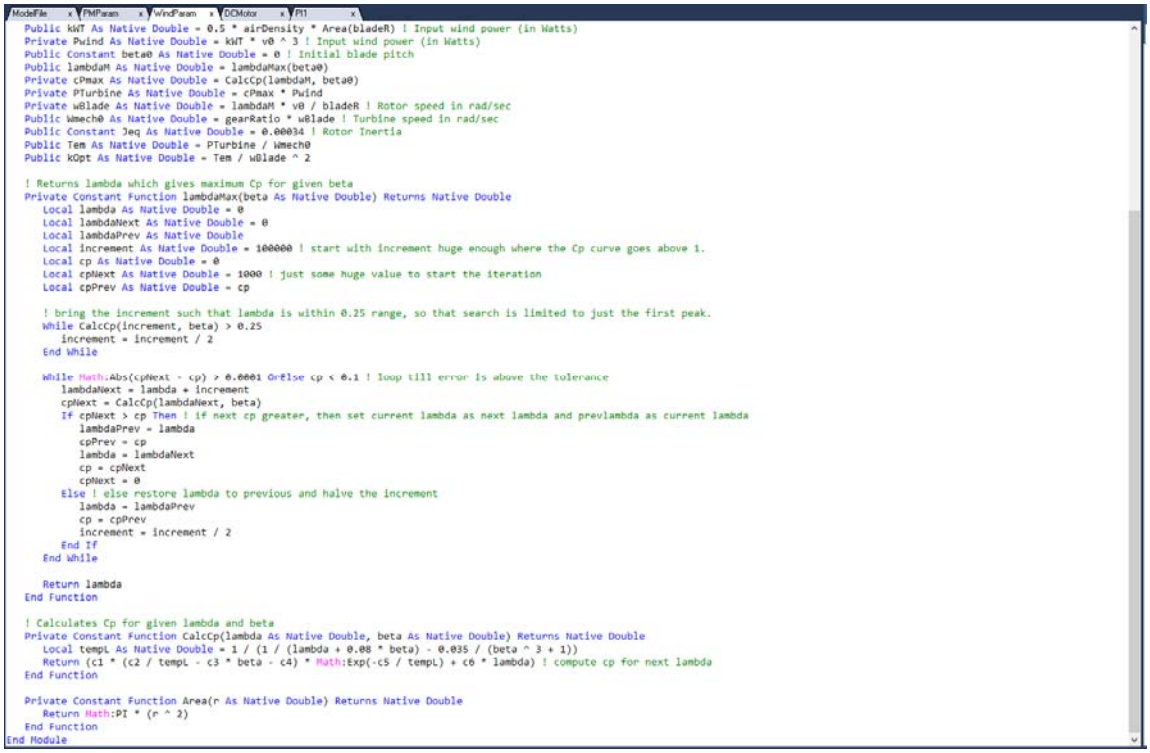


Figure 5.18: Wind turbine emulation: (a) overall mode, (b) Cp subsystem, (c) PM generator subsystem, (d) DC motor subsystem, (e) PI loop subsystem

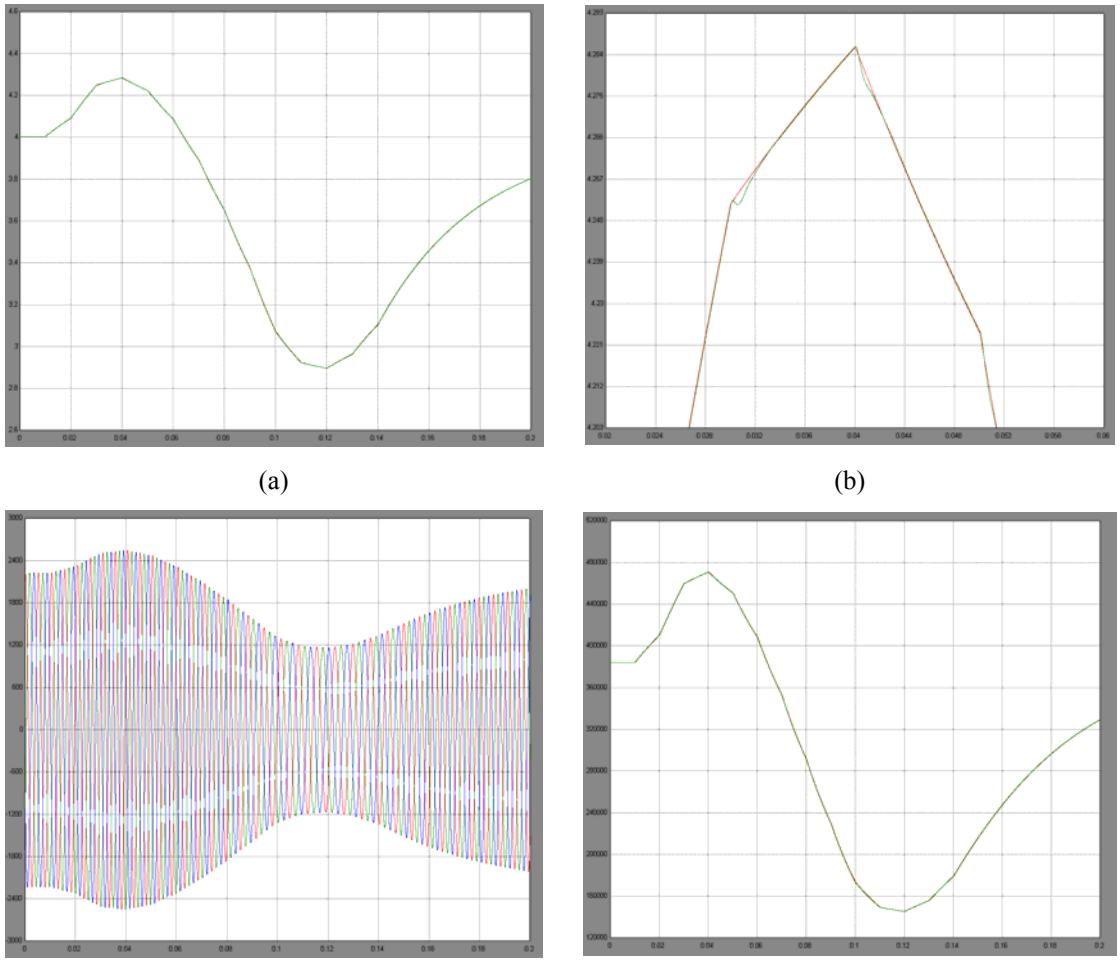


```
ModelFile x PMParam x WindParam x DCMotor x PI1 x
   Public kWT As Native Double = 0.5 * airDensity * Area(bladeR) ! Input wind power (in Watts)
   Private Pwind As Native Double = kWT * v0 ^ 3 ! Input wind power (in Watts)
   Public Constant beta0 As Native Double = 0 ! Initial blade pitch
   Public lambdaM As Native Double = lambdaMax(beta0)
   Private cPmax As Native Double = CalcCp(lambdaM, beta0)
   Private PTurbine As Native Double = cPmax * Pwind
   Private wBlade As Native Double = lambdaM * v0 / bladeR ! Rotor speed in rad/sec
   Public wmech0 As Native Double = gearRatio * wBlade ! Turbine speed in rad/sec
   Public Constant Jeq As Native Double = 0.00034 ! Rotor Inertia
   Public Tem As Native Double = PTurbine / wmech0
   Public kOpt As Native Double = Tem / wBlade ^ 2

   ! Returns lambda which gives maximum Cp for given beta
   Private Constant Function lambdaMax(beta As Native Double) Returns Native Double
      Local lambda As Native Double = 0
      Local lambdaNext As Native Double = 0
      Local lambdaPrev As Native Double
      Local increment As Native Double = 100000 ! start with increment huge enough where the Cp curve goes above 1.
      Local cp As Native Double = 0
      Local cpNext As Native Double = 1000 ! just some huge value to start the iteration
      Local cpPrev As Native Double = cp

      ! bring the increment such that lambda is within 0.25 range, so that search is limited to just the first peak.
      While CalcCp(increment, beta) > 0.25
         increment = increment / 2
      End While

      While Math:Abs(cpNext - cp) > 0.0001 OrElse cp < 0.1 ! loop till error is above the tolerance
         lambdaNext = lambda + increment
         cpNext = CalcCp(lambdaNext, beta)
         If cpNext > cp Then ! if next cp greater, then set current lambda as next lambda and prevlambda as current lambda
            lambdaPrev = lambda
            cpPrev = cp
            lambda = lambdaNext
            cp = cpNext
            cpNext = 0
         Else ! else restore lambda to previous and halve the increment
            lambda = lambdaPrev
            cp = cpPrev
            increment = increment / 2
         End If
      End While

      Return lambda
   End Function

   ! Calculates Cp for given lambda and beta
   Private Constant Function CalcCp(lambda As Native Double, beta As Native Double) Returns Native Double
      Local tempL As Native Double = 1 / (1 / (lambda + 0.08 * beta) - 0.035 / (beta ^ 3 + 1))
      Return (c1 * (c2 / tempL - c3 * beta - c4) * Math:Exp(-c5 / tempL) + c6 * lambda) ! compute cp for next lambda
   End Function

   Private Constant Function Area(r As Native Double) Returns Native Double
      Return Math:PI * (r ^ 2)
   End Function
End Module
```

Figure 5.19: Wind emulation initialization script file

The input wind speed is changed in the model and the generator rapidly adjust its speed to extract maximum energy from the wind. This is seen in Fig. 5.20a where the wind speed and the scaled turbine speed, Fig. 5.20b shows the zoomed in version to show the close correlation. In figure c, the three-phase current of the generator is shown and in figure d, the extractable power vs the extracted power is plotted. The extracted power closely follows the extractable power. It must be noted, that the extractable power is not same as available power in the wind. Of the available power only a portion of it can be extracted and that is indicated by a number of factors which are approximated in the Cp coefficient calculation. The theoretical maximum is 16/27 as formulated by Betz's law. In reality, this value is much lower.

The first two examples were, simulation and real-time combined and run on three-inverter board in real-time. The next two were, simulation only model. The following two are again simulation and real-time combined model but these are run on the extended DSP board in real-time.



(a) (b)

(c) (d)

Figure 5.20: Wind turbine emulation: (a) overall mode, (b) Cp subsystem, (c) PM generator subsystem, (d) DC motor subsystem, (e) PI loop subsystem

Matrix converters are direct three phase ac-ac converters without need for a DC link capacitor. In this example a direct space vector modulation algorithm for a direct matrix converter is implemented as highlighted in [29]. The final example is an advanced version of the basic direct matrix converter, namely direct three level matrix converter (DTMC) and it is run using direct space vector PWM modulation proposed in [28]. Analysis of functioning of the last example sufficiently proves the function of previous example since it is merely a subset of it and hence its discussion is skipped. The model is shown if Fig. 5.21. This converter requires 24 PWM signals which is more than the 12 available in F28335 DSP and unlike an inverter which required simple

deadtime based commutation, this requires a more complex four step commutation. Both of these post a limitation on not being able to use just a DSP for this purpose and hence the need for FPGA.

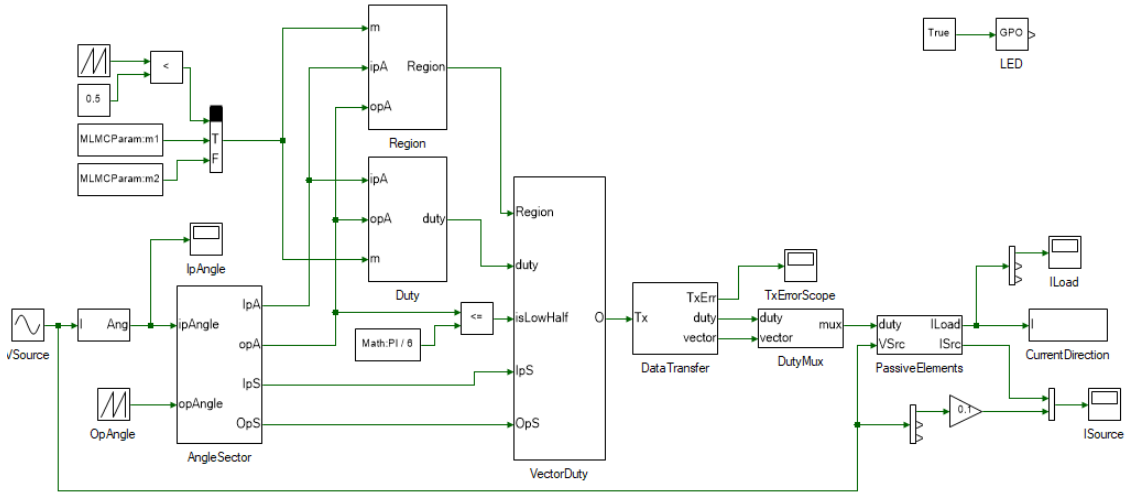


Figure 5.21: DTMC simulation and real-time overall model

The overall model structure is shown in Fig. 5.22. The last file is the configuration bit file for the FPGA which to loaded to the DSP and which in turn loads it to an external SPI flash. Similar to other examples, this project also contains a script file for parameter initialization. The model itself contains multiple components. The one main feature that makes implementing such a huge project in a moderate speed DSP is the conditional evaluation feature explained earlier. Depending on the region and sector of input currents and output voltage within the space vectors different trigonometric formulas are used to compute the duty ratio. If all of these were computed for every possible region and sector, the DSP will fail to run the whole control code within the current step time of 0.2ms. This is made feasible solely because of conditional

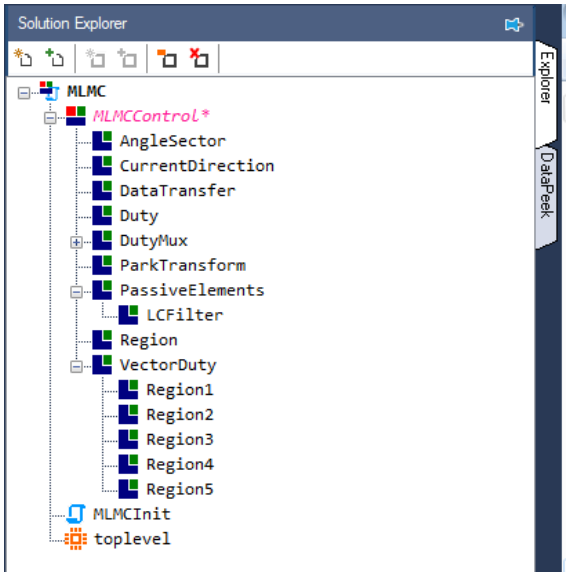


Figure 5.22: DTMC model structure

evaluation that compiler carries out in the backend. The condition tool which selects different code block based on region is shown in Fig. 5.23. and within each of these blocks is even more complex condition selection process as seen in Fig. 5.24.

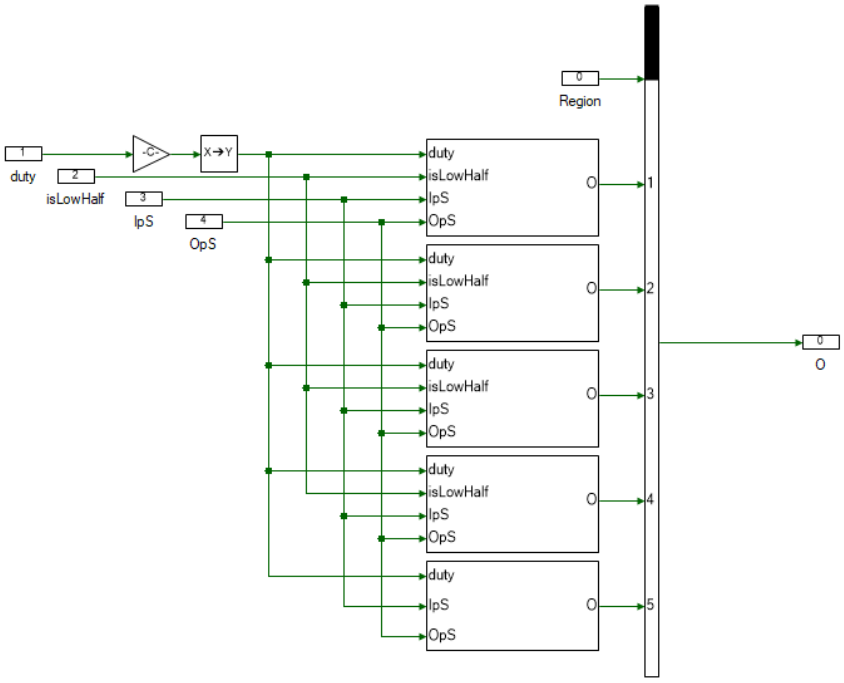


Figure 5.23: Function code selection based on vector region

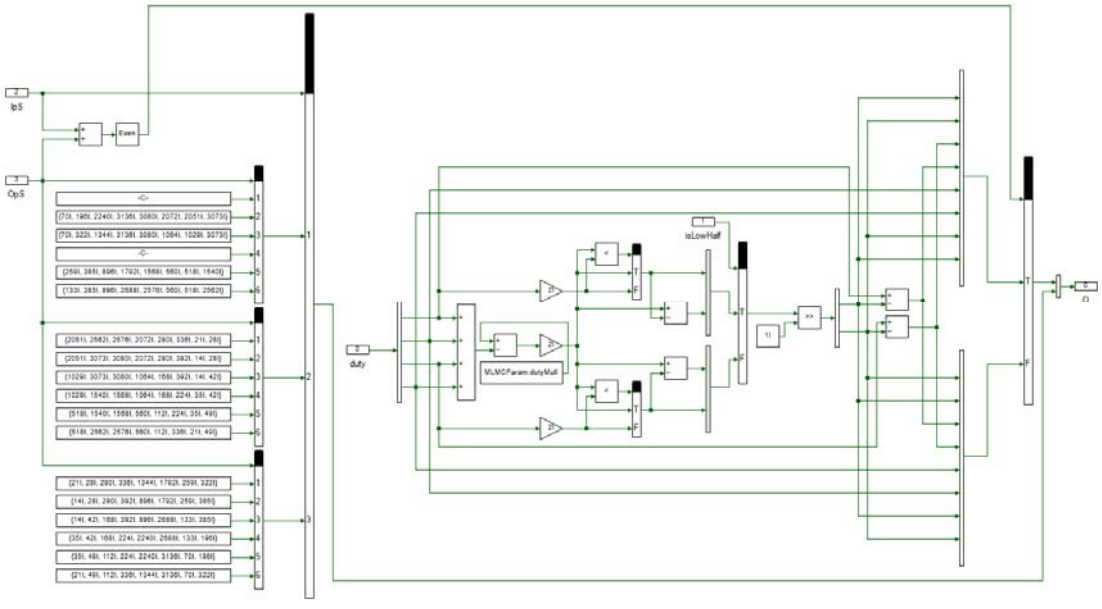
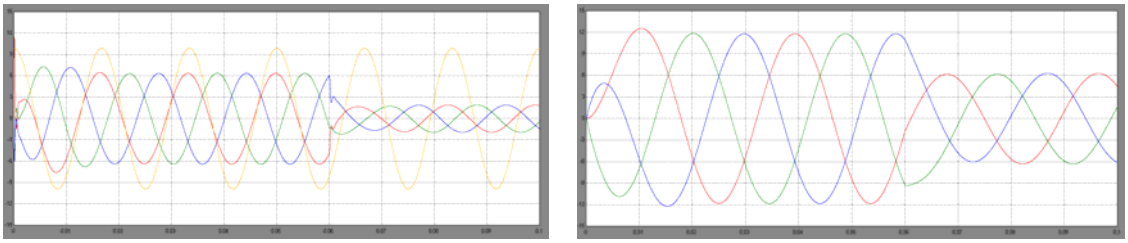


Figure 5.24: Components within one of the five functional code blocks shown in previous figure

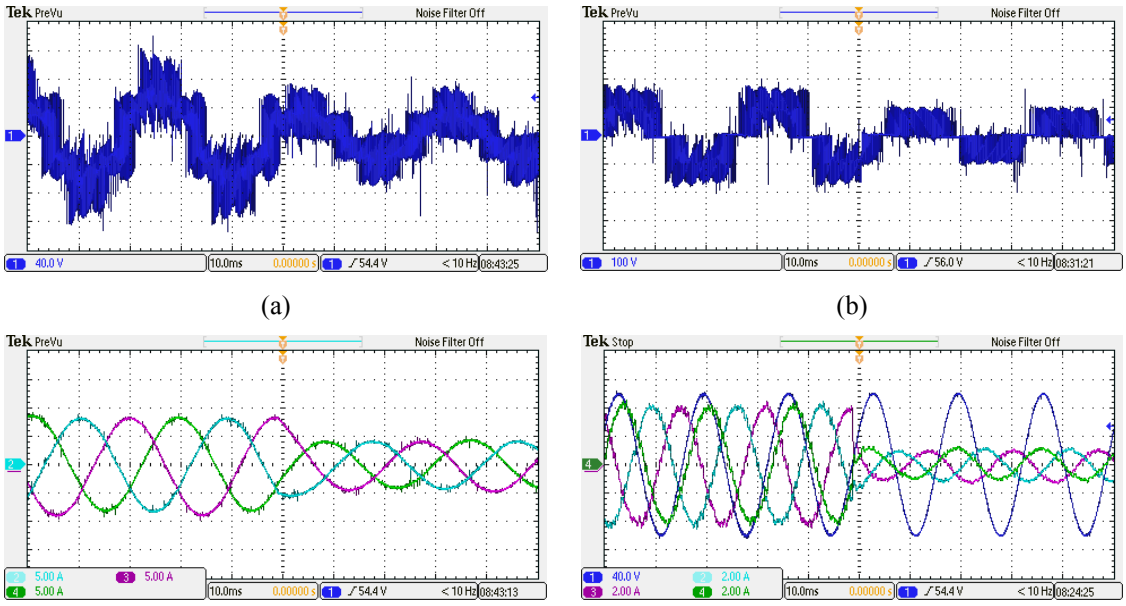
Figure 5.25 shows the sinusoidal three phase input current in phase with input voltage and, the three-phase sinusoidal output load current of a RL load.



(a) (b)

Figure 5.25: DTMC simulation results: (a) input currents and A phase voltage, (b) output RL load currents

The results are real-time mode run is shown in Fig. 5.26 and these are obtained from an external DSO rather than data logging. Since these contain high frequency components these cannot be observed using datalogging with sufficient resolution due to limited bandwidth that makes it only suitable for slow varying signals.



(a) (b)

(c) (d)

Figure 5.26: DTMC simulation results: (a) output phase voltage, (b) output line-line voltage, (c) output load currents, (d) input current and A phase voltage

The results show the converter shifting between multiple levels based on required output voltage leading to lower harmonics compared to tradition matrix converter. This is achieved while maintaining input and output currents sinusoidal and unity power factor. The complete code *modelRun()* for the above model end up being 2887 lines. As mentioned earlier the whole code executes within 0.05ms, much less than the 0.2ms step time solely because of the conditional optimizer which puts most of the code into cascaded 'if-else' blocks leading to far lesser evaluated code at each time step.

5.5 Conclusion

In this chapter, the conversion of model into real-time executable C/assembly code was explored using concepts developed in chapter 4 and libraries developed in chapter 2. This was followed by discussion on the complete structure of the generated code. The two hardware controller prototyping platforms namely the three-inverter and the extended DSP were presented. The former consisted of on board TI's TMS320F28335 DSP and three three-phase inverter modules making it ideal for all low power motor control prototyping applications. The latter was a combination of F28335 DSP and a Xilinx Spartan 3E FPGA linked via a 32-bit and 22-bit independent channels. This is more suitable for general purpose control development and power electronic converter control prototyping.

Programming of both these platform via USB HID to SCI was explored with added code feature to transfer FPGA configuration code to external SPI Flash via the DSP eliminating need for any external programmers for programming both the DSP and FPGA. The same channel is used to send data back to the computer for real-time data logging. This has limited bandwidth of 480kbps and suitable for slow and moderately time varying signals. Finally, different motor control and power electronic control application in both simulation and real-time mode was explored to prove the proper functioning of the developed platform.

Chapter 6

Conclusion and further research

In recent times, the age-old process of hand coding real-time control firmware in C has been replaced by tools that auto-translate model based development directly to real-time controller specific code. But these solutions are out of reach for individual developers and small companies due to exorbitant licensing and prototyping platform cost.

In this thesis, a development of new numerical simulation platform was presented with the hope of being a cheaper and better alternative to other solution to enable further adoption of model based development. This platform supports both code based design as well as model based drag and drop design. To this end, a new programming language was developed from scratch and all the concepts of developing an extremely powerful, but simple compiler was discussed. The language was designed with inbuilt matrix support.

In the latter part of this thesis, the model based platform design was presented with details of various components involved. Even though this form of development is extremely easy to use with very little learning curve from the user end, its back-end programming is much more complex due to various aspects that are obfuscated away. These design challenges were addressed.

The final major component was the capability to transition from a model based simulation to real-time control. To do this various optimization was introduced in the code generation starting with assembly optimized math libraries and ending with model level auto-determination of conditional code evaluation. This development eliminates the user to have any programming knowledge or and knowledge of the device in user, it architecture of its registers etc. An extremely user-friendly UI completely hides away all this complexity without limiting functionality. Two controller prototyping platforms were developed on specifically targeted towards motor controls prototyping and another general platform. The proper functioning of this platform was verified by comparing with other commercially available platforms. A slew of motor controls and power electronics application was tested using the simulation and real-time prototyping platform and the results were in many cases much better in terms of performance, readability, and ease of use than many expensive platforms.

The final solution cuts down the cost of real-time prototyping from ten to hundred thousand dollars to less than a thousand dollars. Future research will concentrate of further cutting down this cost to increase reach. The programming language and the model toolboxes is to be extended. Finally, more real-time controllers are to be added into the list of offerings and faster communication link between computer and controller is to be developed as future research and development.

Bibliography

[1] C. Shi and R. W. Brodersen, "A perturbation theory on statistical quantization effects in fixed-point DSP with non-stationary inputs," in *International Symposium on Circuits and Systems*, Vancouver, Canada, 2004.

[2] Y. A. Chapuis, C. Girerd, F. Aubepart, J. P. Blonde and F. Braun, "Quantization problem analysis on ASIC-based direct torque control of an induction machine," in *IECON, 24th Annual Conference*, Aachen, Germany, 1998.

[3] G. Frantz, "Where will floating point take us?," Texas Instruments, 2010.

[4] A. Friedmann, "TI's new TMS320C66x fixed and floating-point DSP core conquers the 'Need for Speed'," Texas Instruments, 2010.

[5] Rys, "Origin of Quake3's Fast InvSqrt()," March 2007. [Online]. Available: https://www.beyond3d.com/content/articles/8/.

[6] R. Green, "Faster Math Functions," May 2016. [Online]. Available: https://basesandframes.wordpress.com/2016/05/17/faster-math-functions/.

[7] B. W. Kernighan and D. M. Ritchie, "The C Programming Language," Prentice Hall, 1988, pp. 250-251.

[8] G. J. Hekstra and E. F. Deprettere, "Floating point Cordic," in *11th Symposium on Computer Arithmetic*, Ontario, Canada, 1993.

[9] X. Hu, R. G. Harber and S. C. Bass, "Expanding the range of convergence of the CORDIC algorithm," *IEEE Transactions on Computers,* vol. 40, no. 1, pp. 13-21, 1991.

[10] M. D. Ercegovac and T. Lang, "Digital Arithmetic," Morgan Kaufmann Publishers, 2003, p. Chaper 11.

[11] W. J. Cody and W. Waite, Software Manual for the Elementary Functions, Prentice Hall, 1980.

[12] A. V. Aho, R. Sethi and J. D. Ullman, Compilers, Principles, Techniques and Tools, Addison Wesley, 1988.

[13] "Using the CodeDOM," Microsoft, 30 03 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/using-the-codedom.

[14] "How to: Run Partially Trusted Code in a Sandbox," Microsoft, 30 03 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/misc/how-to-run-partially-trusted-code-in-a-sandbox.

[15] D. Delling, P. Sanders, D. Schultes and D. Wagner, "Engineering Route Planning Algorithms," in *Algorithmics of Large and Complex Networks*, Springer, pp. 117-139.

[16] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," in *25th Annual Symposium on Foundations of Computer Science. IEEE*, 1984.

[17] "Data Contract Serializer," Microsoft, 30 03 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/data-contract-serializer.

[18] I. J. Nagrath and M. Gopal, "State variable analysis and design," in *Control Systems Engineering*, New Age International, 2007, p. 588.

[19] R. L. Burden and D. J. Faires, Numerical Analysis, Brooks/Cole, 2001.

[20] J. Axelson, USB Complete: The Developer's Guide, 2005.

[21] Future Technology Devices International Limited, "FT260 User Guide," [Online]. Available: http://www.ftdichip.com/Support/Documents/ProgramGuides/AN_394_User_Guide_for_FT260.pdf.

[22] Texas Instruments, "TMS320x2833x, 2823x Boot ROM," 08 2008. [Online]. Available: http://www.ti.com/lit/ug/spru963a/spru963a.pdf.

[23] Texas Instruments, "TMS320F28335 Flash APIs," 14 08 2008. [Online]. Available: http://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/171/0755.controlSUITE_5F00_Flash2833x_5F00_API_5F00_Readme.pdf.

[24] N. Mohan, Electric Drives, MNPERE, 2003.

[25] N. Mohan, Advanced Electric Drives, MNPERE, 2001.

[26] N. Mohan, T. M. Undeland and W. P. Robbins, Power Electronics, Converters, Application and Design, Wiley, 2009.

[27] S. Tewari, "The potential of sodium sulfur battery energy storage to enable further integration of wind," 2012.

[28] S. Raju, "Multi-level matrix converters," PhD dissertation, University of Minnesota, 2017.

[29] D. Casadei, G. Grandi, G. Serra and A. Tani, "Space vector control of matrix converters with unity input power factor and sinusoidal input/output waveforms," in *Power Electronics and Applications*, Brighton, UK, 1993.

Appendix A

A.1 Language Keywords

Keywords	*Usage* *(text with '<>' are comments for the reader and not part of the language)*
True, *False*	Result of a conditional operation Ex. 1: a = True Ex. 2: a = (b > c) *<a is True if b > c, else a is False>*
If, *ElseIf,* *Else,* *Then,* *End If*	Evaluates a block of code if condition is True. Format: If <*some condition*> Then <*Some code – optional*> <*Optional*> ElseIf <some condition> Then <*Some code – optional*> <*Optional*> Else <*Some code – optional*> End If Ex: If *a < 0* Then *b = -a* Else *b = a* End If
Public, *Private,* *Local*	Access modifiers of a declaration. If marked as Public, then the definition can be accessed from anywhere. If marked Private, it can be accessed from only within the enclosing entity. A variable within a function can only be marked as Local. It can be accessed only within the function and is destroyed when the function reaches end of its code block. Ex. 1: Public Module *moduleName* Ex. 2: Private Function *functionName()* Ex. 3: Public *variableName* As Integer = 500 Ex. 4: Private *variableName1, variableName2* As Double Ex. 5: Local *variableName* As Single

Boolean, Short, Integer, Long, Single, Double

Data types. The language is strongly typed. Each operation returns a datatype and error or warning is thrown if there is an incompatibility.

Type	Size	Default Value
Boolean	Varied	False
Short	16-bit signed Integer	0S
Integer	32-bit signed Integer	0I
Long	64-bit signed Integer	0L
Single	32-bit IEEE floating-point	0.0F
Double	64-bit IEEE floating-point	0.0

Ex. 1: Public *variableName* As Integer
Ex. 2: Public *functionName()* Returns Single
Ex. 3: Public *variableName* As Boolean = True

As

Preceded by variable name, followed by the data type of variable.

Format: Public <*or Private or Local*> *variableName* As Double
Format: Public <*or Private*> *functionName(argumentName* As Double)

Native

By default, all variables are marked as matrices unless otherwise declared with Native keyword. A variable can be declared as Native if it has just a single element throughout its lifetime. If at any point in the code, the native variable is assigned a matrix it will throw a runtime error unless the matrix itself has just a single element. If a matrix has just a single element it is helpful to declare as Native for more clarity and also it allows of optimization since a matrix operation takes at least 10 times more time to execute than a native operation.

Format: Public <*or Private or Local*> *variableName* As Native Double

Ex. 1: Public *variableName* As Native Double
Ex. 2: Public *variableName* As Native Double = 3.45
Ex. 3: Public *variableName* As Native Double = [3.45]

In the last case it was assigned to a matrix, but since the matrix has just a single element it will be casted to native element. If it was instead assigned to a row matrix [12.3, 3.45] then it would throw a runtime error.

Function, End Function

A function block is collection of code that can be called from other function block or from within the function itself (recursive function). A function may or may not return a value and this is indicated in the declaration and it may or may not take some value (arguments).

Format (if function does not return a value):
Public <*or Private*> Function *functionName()*

Format (if function returns a value):
Public <*or Private*> Function *functionName()* Returns Integer

Format (if function has arguments):
Public <*or Private*> Function *functionName(argName1* As Integer*, argName2* As Double*)* Returns Integer

Ex. 1: Private Function *functionName()* Returns Double
    <*Some code – optional*>
    Return *doubleValue*
End Function

Ex. 2: Public Function SomeFunction()
    <*Some code – optional*>
End Function

Ex. 3: Public Function AddLong*(*a As Long, b As Long*)* Returns Long
    Return (a + b)
End Function

A function can be called from other functions by calling the function name and supplying the function parameters. The language supports function overloading. Consider below function which calls two of the functions declared above.

Ex. 4: Public Function f1()
    Local a, b As Long = 10
    Local c As Long
    someFunction() <*calls this function which executes it code*>
    c = AddLong(a, b)

	End Function
Constant	Any function or 'variable' can be declared as constant. A 'variable' declared as constant cannot be changed after its initialization. Attempting to do so will throw compiler time exception. A function declared as constant does not allow changing any variable that is declared outside the function. Ex. 1: Public Module testModule Public Constant PI As Double = 3.147 Public tempVar As Double Public Constant Function GetArea(radius As Double) Returns Double Local localVar As Double = PI*radius*radius Return localVar End Function Public Function GetCircumfrence(radius As Double) Returns Double tempVar = 2*PI*radius Return tempVar End Function End Module In above example, PI is declared as a constant which means it cannot be changed anywhere else in the code. All constants must be initialized during declaration. In constant function GetArea, the area computed is stored in a local variable since constant functions do not allow modifying variables outside the function it cannot use tempVar declared in the module to store the result temporarily as was the case in the function GetCircumfrence which is not declared as constant.
Pure	Pure functions are functions which do not write to or read from any variable outside the function declared as Pure. This is one level stricter than Constant function. In constant function the function was not allowed to modify external variables, in this it not even allowed to read external variable. These functions do not depend on the state of the Module, they only depend on the inputs and for a given input always returns the same output. This allows for paralleling the function call. Ex. 1: Public Pure Function factorial(num As Integer) Returns Integer Local result As Integer = 1 While num > 1

	result = result * num num = num - 1 End While Return result End Function Public Function ComputeFactorial() Local a As Integer = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] For i As Integer = 1 To 10 a(i) = factorial(i) Next End Function In the above example, factorial of 1 through 10 is computed and the result is stored in a row matrix of size 10. Since factorial function was declared as Pure, the compiler in many cases when possible auto resolve to run factorial of 1 through 10 on as many processors as possible in parallel to speed up execution. This is solely possible because the function does not depend on any outside values or modify them. The compiler might auto identify a function as Pure even if it isn't marked as such, but marking it manually is recommended for code clarity.
Return, Returns	If a function returns a value, then the end of the function is marked by the keyword Returns followed by the datatype of the returned value. When a function returns a value, it is marked by the keyword Return followed by the return value. The keywords can be used interchangeably, the compiler auto fixes based on context. Ex. 1: Public Function *functionName*() Returns Double Return 3.147 End Function
Module, End Module	A module is an enclosure for functions and variables. It allows for data encapsulation. Any declaration within the module that is marked as Private can only be accessed from within the Module and anything marked as Public can be accessed from within and outside the module. Format: Public <*or Private*> Module *moduleName* <*Some variables – optional*> <*Some functions – optional*> End Module

	Ex. 1: Public Module mathModule Private prevValue as Double = 0.0 Public Function AccumulateValue(x As Double) Returns Double prevValue = Add(prevValue, x) Return prevValue End Function Private Function Add(a As Double, b As Double) Returns Double Return a + b End Function End Module In the above function function 'AccumulateValue' calls function 'Add' which is Private but within the same module so it is fine. To call AccumulateValue from outside this module, scope resolution operator ':' is used as shown: Ex. 2: Public Module filterModule Public Function ProcessValues(x As Double) Local result As Double <*Some code*> result = mathModule:AccumulateValue(x) <*Some code*> End Function End Module In the above example, if instead of calling the function AccumulateValue, the function Add within mathModule was called it would have thrown a compile time error since it was marked as Private.
While, *End While*	Similar to '*if*' block in the sense that it runs the code within it if the condition is true. It is different form '*if*' block in the sense it keeps running the same code block again and again till the condition becomes false. Format: While *condition* <*Some code – optional*> End While

	Ex. 1: Public Function factorial(num As Integer) Returns Integer Local result As Integer = 1 While num > 1 result = result * num num = num - 1 End While End Function
To, Step	Iterates through all values from the start value till the end value in steps mentioned by step value. If step value is not mentioned then uses default step of 1. Format: *startValue* To *endValue* Step *stepValue* Format: *startValue* To *endValue* <*uses default step of 1*> Ex. 1: For i = 1 To 10 Step 2 Ex. 1: a = b(1 To 10, 2 To 8 Step 2) In Ex. 1, the '*for*' block code is executed as long as i lesser than or equals 10 starting with i = 1 and incrementing i by 2 at end of every loop completion. In Ex. 2, consider b to be a 20x10, i.e. 20 rows and 10 columns matrix. The code selects rows 1 through 10 and columns 2, 4, 6 and 8 of matrix b and assigns the result, i.e. 10x4 matrix to a.
All	Selects all the elements in that dimension. With '*To*' and '*Step*' selective elements of a dimension were extracted. Using '*All*' allows to extract all the elements of that dimension. Ex. 1 = a = b(All, 1 To 5, All) In above example, consider b to be 3D matrix of size 15x10x5, i.e. 15 pages, 10 rows and 5 columns, the above code extracts all the pages of b, and within those pages extracts rows 1 through 5 and within those rows extracts all the columns and returns a 15x5x5 matrix.
For, *End For*	'*For*' loops execute a code block a fixed number of times mentioned in its condition. Format: For *variableName* = *startValue* To *endValue* Step *stepValue* <*Some code – optional*> End For Format: For *variableName* As Integer = *startValue* To *endValue* Step *stepValue*

```
        <Some code – optional>
    End For

Ex. 1: For i = 1 To 10 Step 5
        <Some code – optional>
    End For
Ex. 2: For i As Integer = 1 To 10 Step 5
        <Some code – optional>
    End For
Ex. 3: For i As Double = 0.1 To b Step c
        <Some code – optional>
    End For
```

In Ex. 1, the code block is executed twice, during first run, i = 1 and when 'End For' is reached code jumps back to beginning of 'For', where i is incremented by 5. The 'For' loop is run again since i =6 which is less than 10. At the end of this run code jumps back to 'For' where i is incremented by 5 and becomes 11. At this point code exits the 'For' block. Example 2 does the same as Ex. 1. But here, the increment variable is declared locally. In previous case it was assumed increment variable i was declared somewhere else in the function and was being used here. In Ex. 2, it is assumed i has not be declared before and declared here. This i can only be used within the 'For' block and not outside it In Ex. 3, the end value and step values are values contained in some declared variable. These values are used only during the first time 'For' loop runs. It can change at any point while the loop is running and those values will not be updated. If dynamically changing condition is required consider using a 'While' loop.

A.2 Language Operators

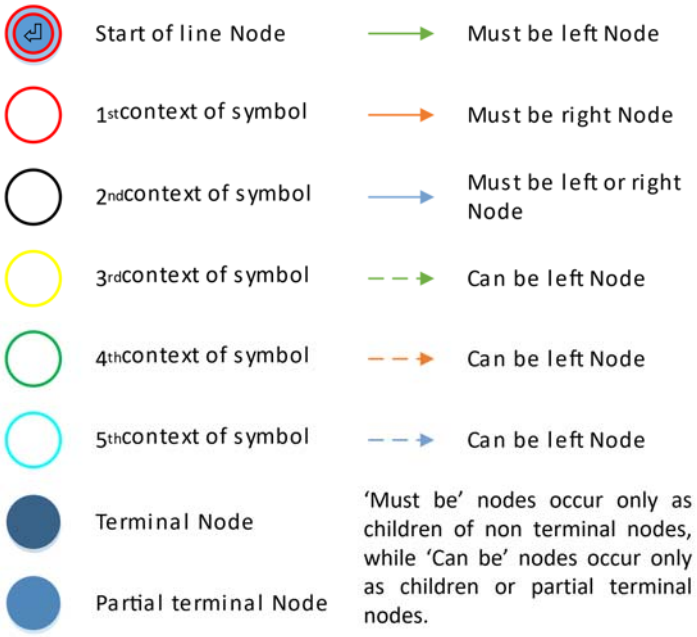
List of all language operators in their order of precedence, highest to lowest.

Operators	*Usage* *(text with '<>' are comments for the reader and not part of the language)*
Paranthesis	'(' and ')'. Are used in multiple context. 1. For controlling order of expression evaluation: (a + b) * c 2. For calling and decalring function: *functionName(a, b)* 3. Mentioning index of matrix: Let A be 2x3. A(1, 3) selects 1st row 2nd column. Priority of contents within this enclosure is reduce by the priority of the enclosure itself.
Brackets	'[' and ']'. Are used for declaring a matrix. Ex: [1, 2, 3] = 3 element row matrix. [[1, 2, 3]] = 3 element column matrix. [[1, 2, 3], [4, 5, 6]] = 2 row, 3 column (2x3) matrix. [[[1, 2], [4, 5]], [[6, 7], [8, 9]], [[10, 11], [12, 13]]] = 3 page, 2 row, 2 column (3x2x2) matrix. The matrix dimension can be simply expanded (limited solely by memory) by adding '[ ]'. Any idex can be accessed using '( )' mentioned above by explicating mentioning the index. Else a range of indices can be selected as well across any combination of dimension using 'To', 'Step' and 'All' keywords mentioned above in A.1. Matrix has a single datatype, i.e. all elements within it must be of same type. If in above case if any of the element was double instead of all being integer then the whole matrix will be double. Also, matrix must be of uniform size, i.e. in a 2 row matrix, 1st row cannot have 3 elements and 2nd have 2 elements. They need to be uniform. Priority of contents within this enclosure is reduce by the priority of the enclosure itself.
Braces	'{' and '}'. Matrices have limitations that different datatypes cannot be clubbed together and different dimensions cannot be clubbed together. To overcome this issue, Mux is introduced in the language. This is formed by enclosing any collection of data of any type and dimension within '{ }'. You can mux a 2x3 double matrix A, 4x5x6x7 boolean matrix B, and a single element integer C all into a single mux D as: D = {A, B, C}. A mux is not a matrix i.e. {A, {B, C}} is same as {A, B, C}, it is flatted out into single array. To access B in both cases it is D(2) and to access C it is D(3) and so on. Priority of contents within this enclosure is reduce by the priority of the enclosure itself.
ID	This refers to reference and definition names. Each declared module, function and variable has a name and they are referred to in the code via these defined names. Depending on what the ID refers to, its functionality changes.

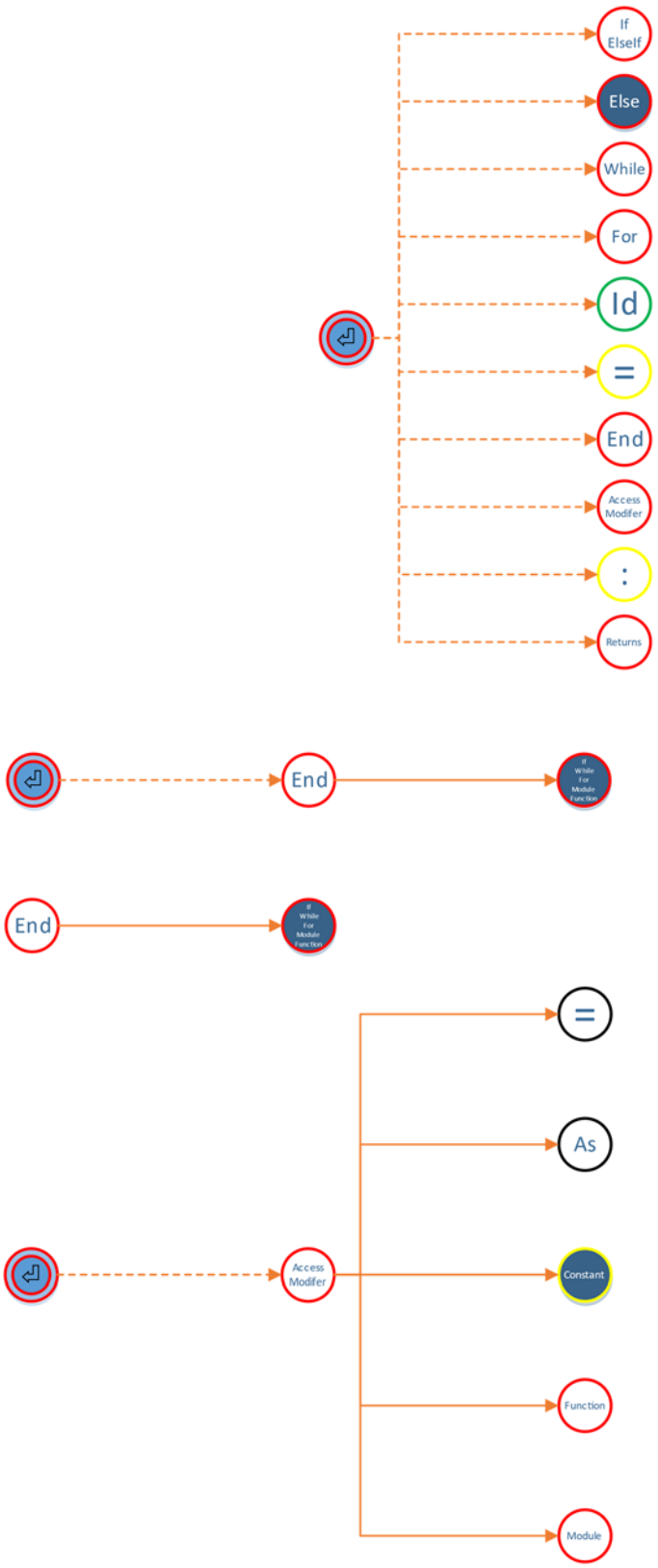
Scope Resolution	':'. The cope resolution operator is used to resolve the exact variable or function that is being referred to. For instance, two different modules may have variables and functions of name same as those within another module. To differentiate between the two, the scope resolution operator is used as: *moduleName:variableName.*
Unary Operator	In the order of precedence (lowest to highest): +, -.
Binary Operators	These include in the order of lowest to highest: << (left shift), >> (right shift), +, -, *, /, % (Modulo), ^ (Power). Contrary to other languages these do not have any specific directional associativity. These operations are matrix capable. In the current version '*' performs cross matrix multiplication. An operator is yet to be introduced for dot multiplication, dot division etc., even though these capabilities are available in the backend math library. Shift operators only work on integral types, i.e. Short, Integer and Long.
Comparison Operators	These include in the order of lowest to highest: >, >=, <, <=, =, <> (Not equal to). Except for <> and = operators the remaining 4 only accept numerical datatypes on their left and right while the former two accept both numerical and Boolean type.
Bitwise Operators	These include in the order of lowest to highest: Xor, And, Or. These perform bitwise operation. Inputs can only be integral types, i.e. Short, Integer or Long, or else of Boolean type. Bitwise operation has no significance on real-type, i.e. Single and Double and are not supported. To extract exponent and mantissa information use appropriate functions in inbuilt library rather than resorting to these operators.
Logical Operators	These include in the order of lowest to highest: Xor, OrElse, Or, AndAlso, And, Not. AndAlso evaluates the expression on its left first and if it is True only then evaluates the expression on its right. This mean if expression on its left turns out to be false and there is a function call on its right, the call is never made since output of 'And' is False even if one input is false. For case of OrElse, it does not evaluate expression on right if Left is True. These are called short circuiting operators. If both left and right expression need to be evaluated irrespective of the outcome of one of them use 'And' and 'Or' respectively. The result of this is always Boolean type.
Separator	Comma: ','. These are used in multiple context within the language. 1. Function parameter separator: Used to separate parameter declaration within a function definition and while sending parameters during a function call. 2. Matrix entry separation: A = [[1, 2, 3], [4, 5, 6]] where A is a 2x3 matrix. 3. Mux entry separation: B = {A, C, E} where A, C and E can be of any datatype and of any dimensions. 4. Dimension separation: If a particular index within matrix A is accessed then ',' is used for separating those indices: A(2, 3) selects 2nd row 3rd column.

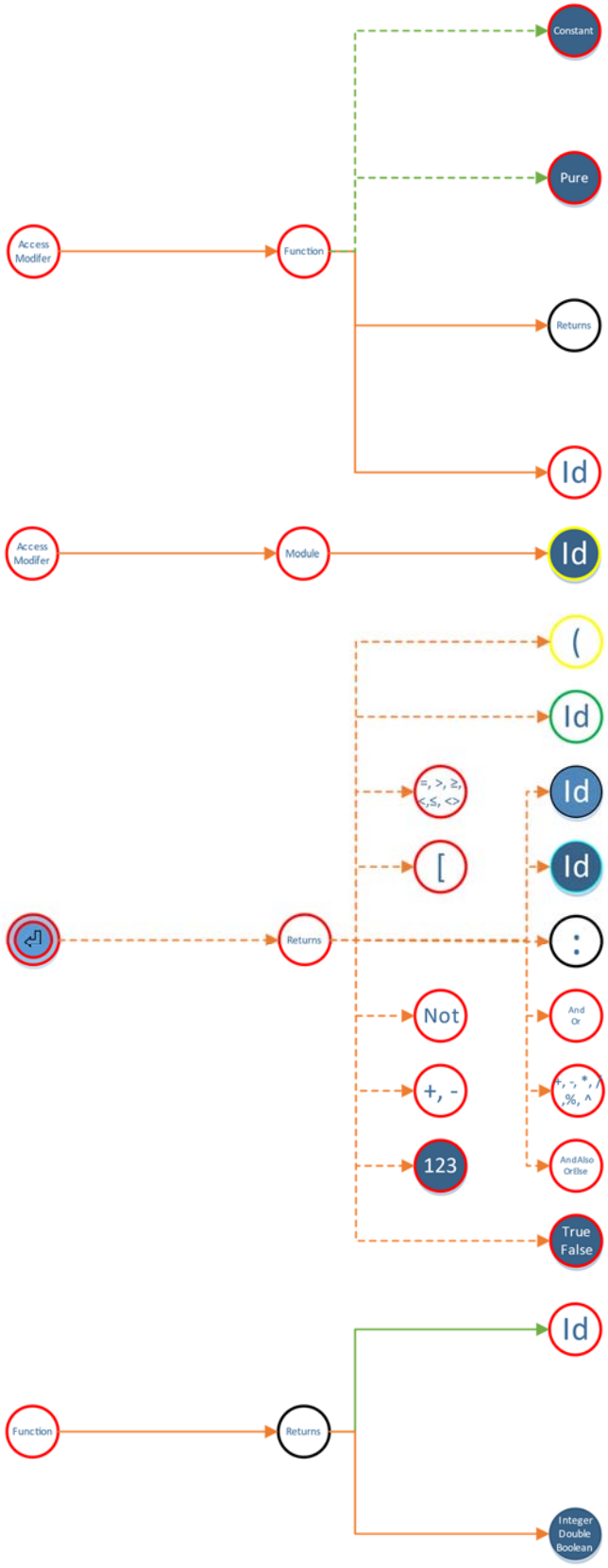
A.3 Language Rules

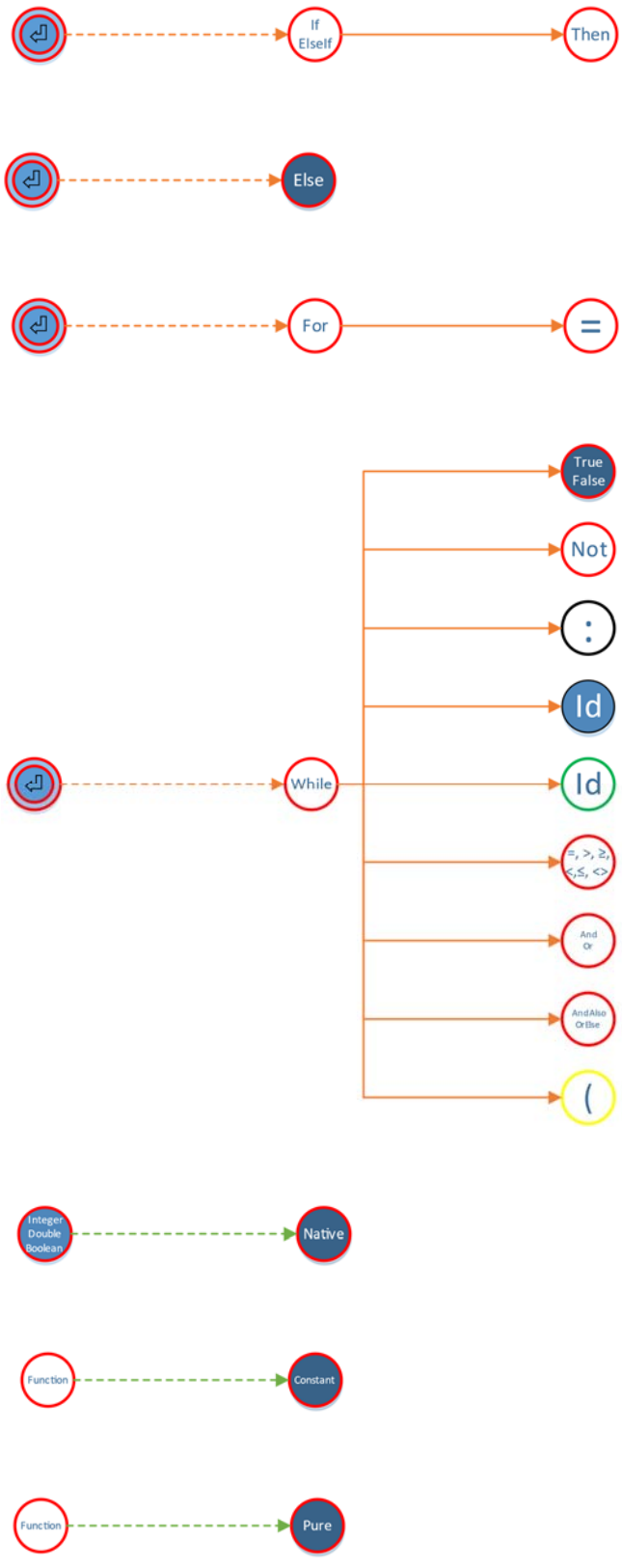
This section contains the list of nodes that make up the language that and the localized possible tree structures for each of these nodes. The whole language can be built starting with entry node '⏎' and traversing till terminal node is reached. Below are the legends for the local AST and all the nodes.

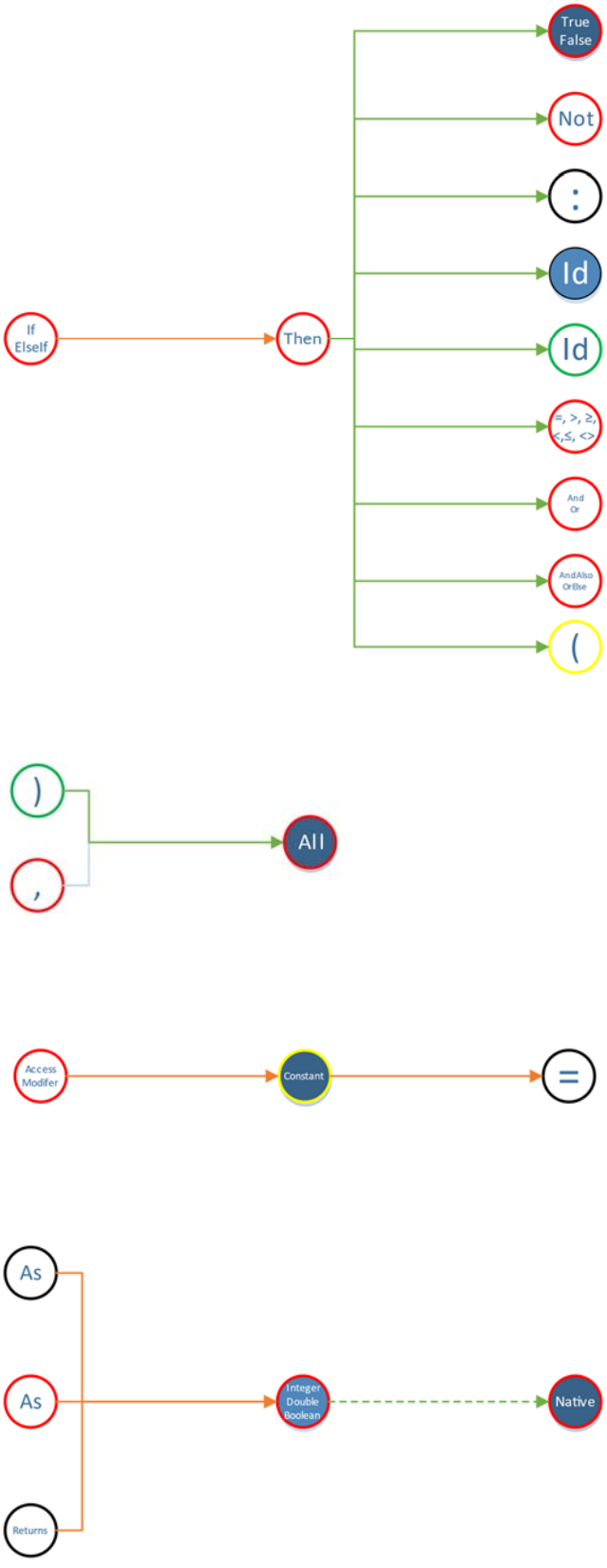


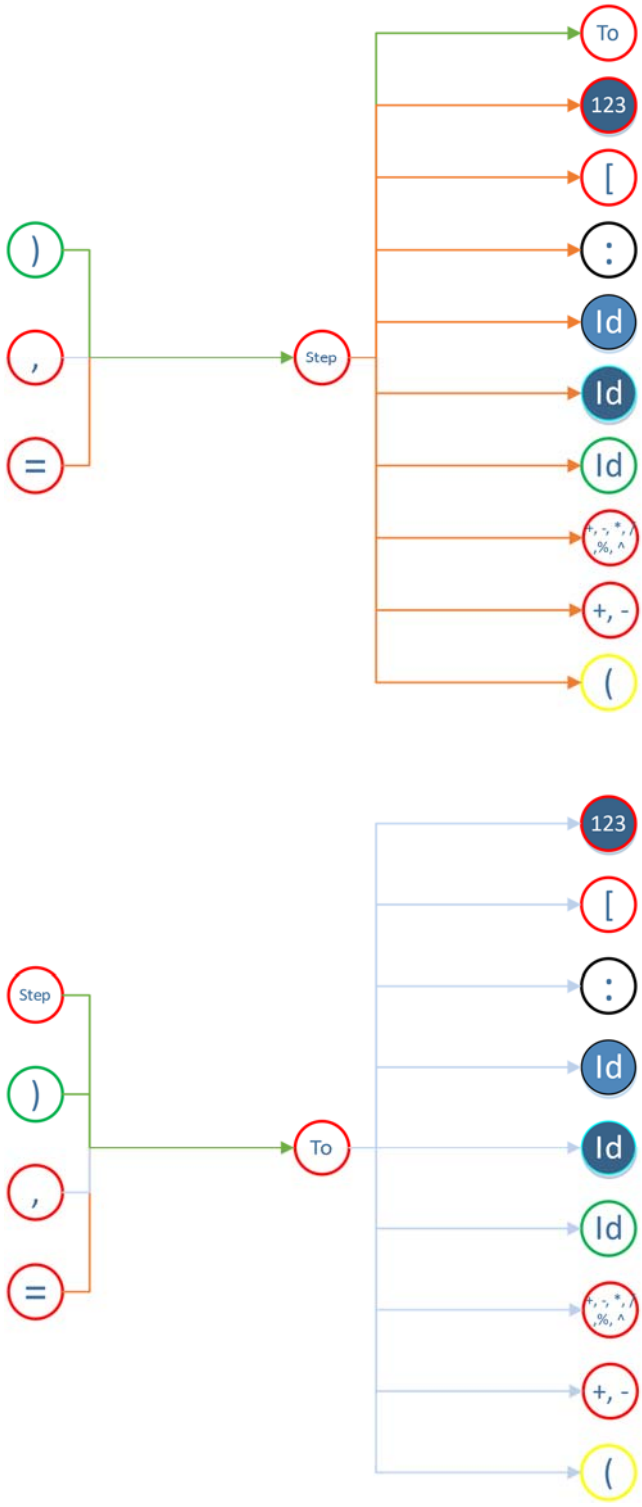

GeneralLine Entry
General DataType
EndCondition
GeneralLogical Operator
IfFamily
KeywordAll
TrueOrFalse
ShortCircuit Operator
Conditional Then
General Number
DimensionSeparatorComma
NotLogicalOperator
ElseFamily
FunctionDefId
FncDefParamSeparatorComma
KeywordNative
Conditional While
DataRefId
DataSeparatorComma
Function Constant
End Statement
ModuleDef DataDefId
ParameterSeparatorComma
DataConstant
Conditional Increment
FunctionRefId
InternalScope Resolution
KeywordPure
KeywordTo
ForRefId
FirstScope Resolution
GeneralOpen Bracket
Keyword Step
FunctionDefOpenParenthesis
LineScope Resolution
GeneralClose Bracket
KeywordForAs
ParameterOpenParenthesis
ForEqualTo
Keyword InitializationAs
GeneralOpen Parenthesis
Initialization EqualTo
GeneralAccess Modifiers
DimensionOpenParenthesis
Assignment EqualTo
Keyword Module
FunctionDefCloseParenthesis
GeneralBinary Operator
Keyword Function
ParameterCloseParenthesis
GeneralUnary Operator
DataReturns
GeneralCloseParenthesis
InnerUnary Operator
Keyword Returns
DimensionCloseParenthesis
GeneralComparisonOperator

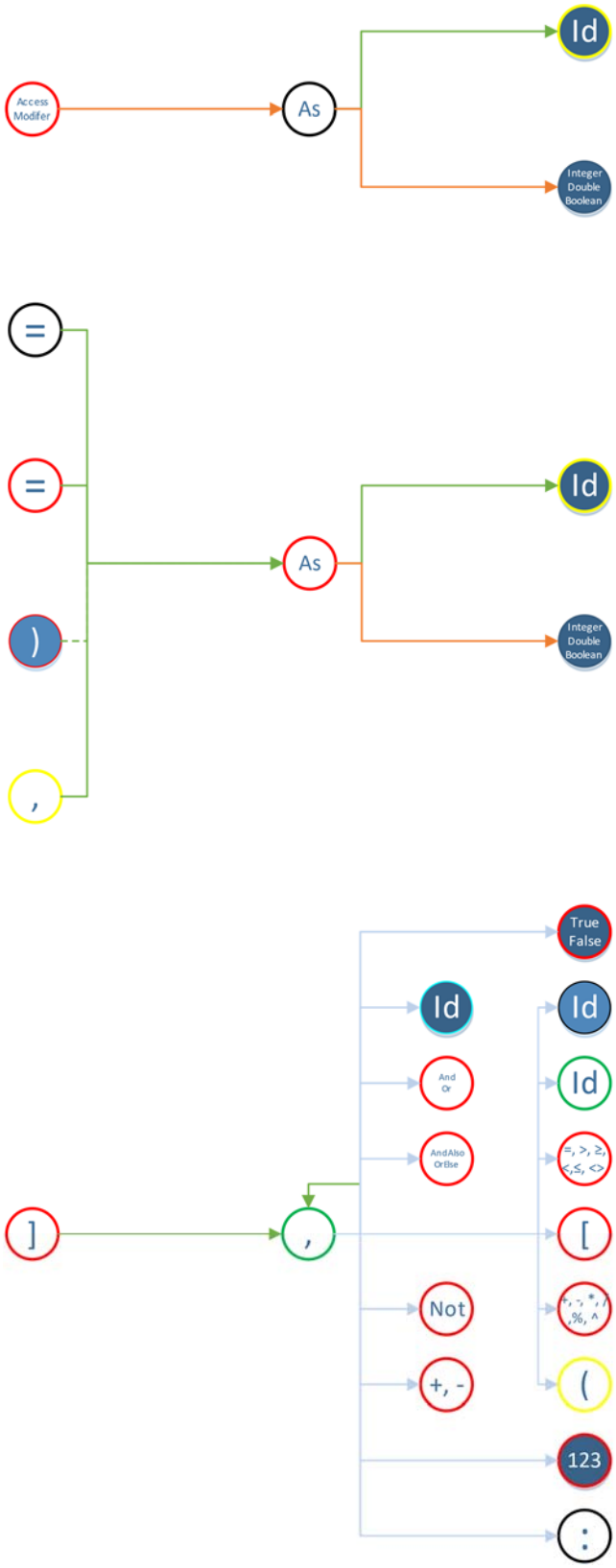

If
ElseIf
Else
While
For
Id
=
End
Access
Modifer
:
Returns
End
If
While
For
Module
Function
End
If
While
For
Module
Function
=
As
Access
Modifer
Constant
Function
Module

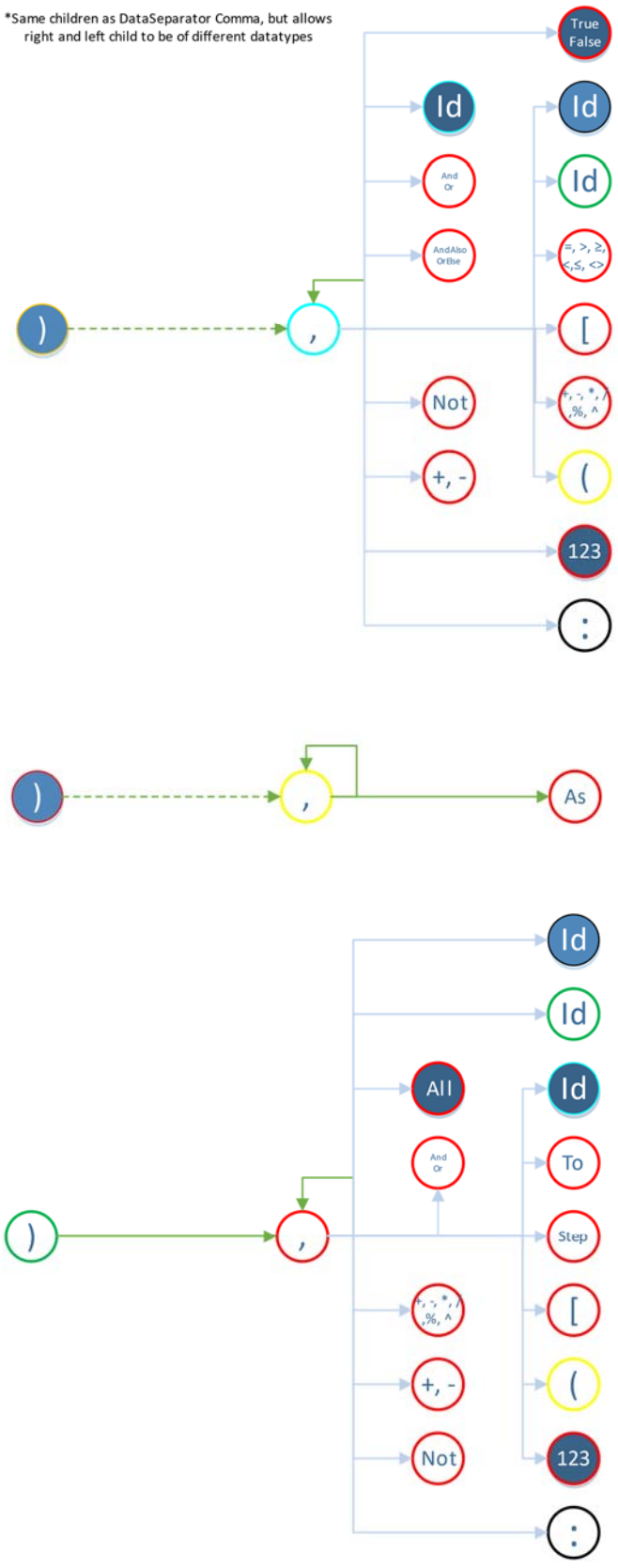

Constant
Pure
Access Modifer
Function
Returns
Id
Access Modifer
Module
Id
(
Id
=, >, ≥, <, ≤, <>
Id
[
Id
Returns
:
Not
And Or
+, -
*, /, %, ^
123
AndAlso OrElse
True False
Id
Function
Returns
Integer Double Boolean

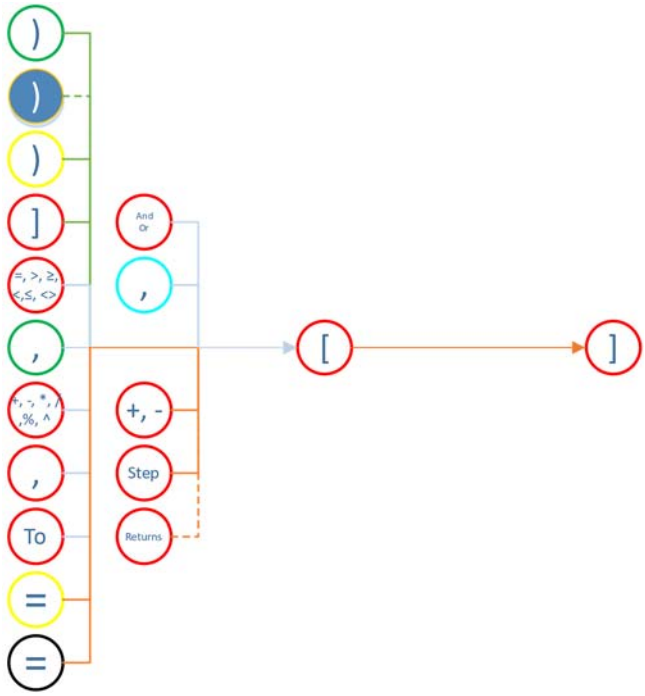

If
ElseIf
Then
Else
For
=
True
False
Not
:
Id
While
Id
=, >, ≥,
<,≤, <>
And
Or
AndAlso
OrElse
(
Integer
Double
Boolean
Native
Function
Constant
Function
Pure

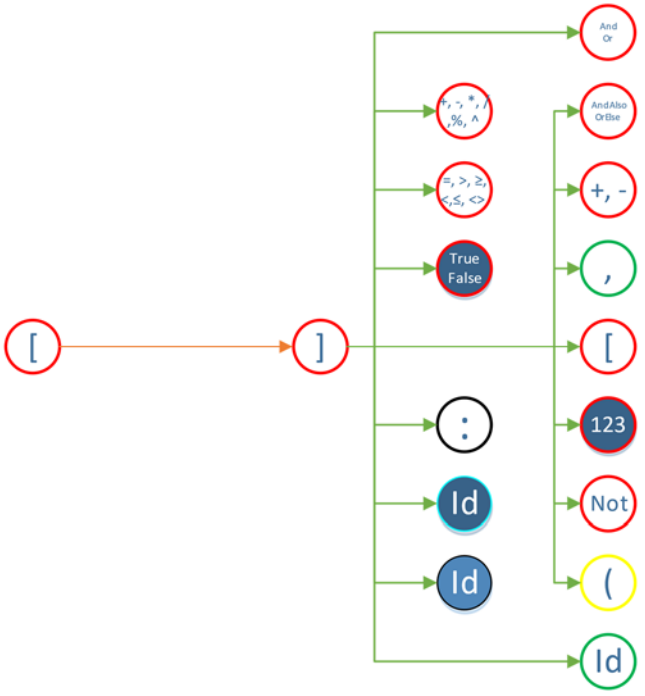

True
False
Not
:
Id
If
ElseIf
Then
Id
=, >, ≥,
<,≤, <>
And
Or
AndAlso
OrElse
(
)
,
All
Access
Modifer
Constant
=
As
As
Returns
Integer
Double
Boolean
Native

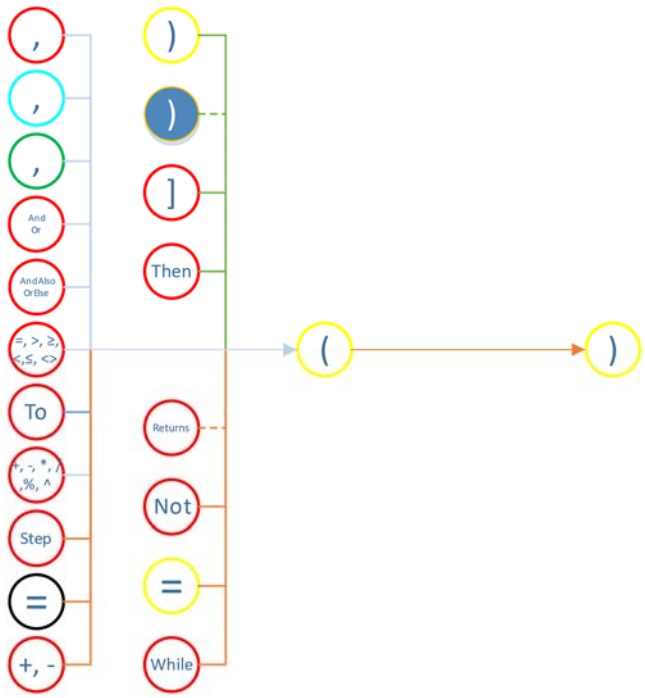

)
,
=
Step
To
123
[
:
Id
Id
Id
+, -
(
Step
)
,
=
To
123
[
:
Id
Id
Id
+, -
(

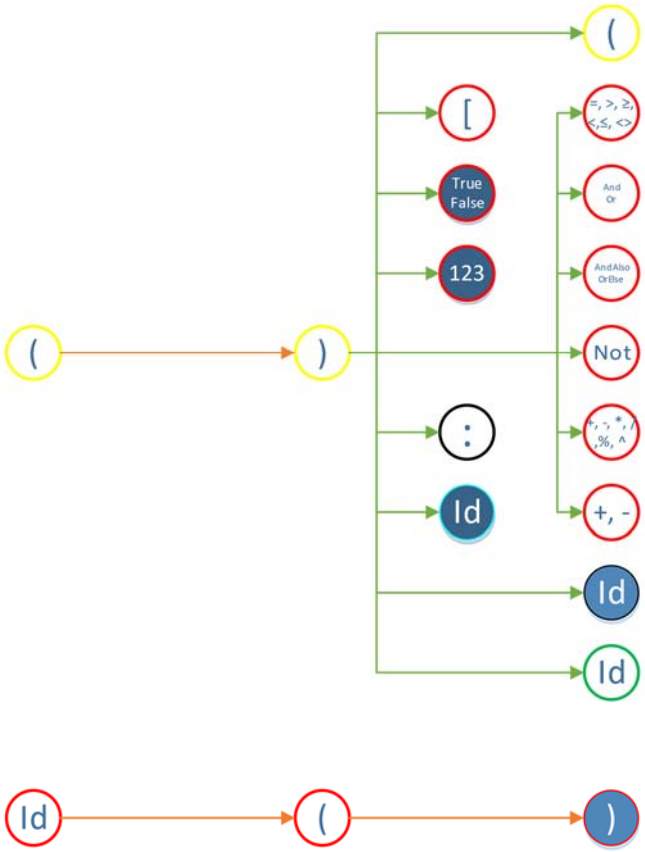

Access Modifer
As
Id
Integer Double Boolean
=
=
)
,
As
Id
Integer Double Boolean
]
,
And Or
AndAlso OrElse
Not
+, -
True False
Id
Id
Id
=, >, ≥, <,≤, <>
[
*, -, *, /, %, ^
(
123
:

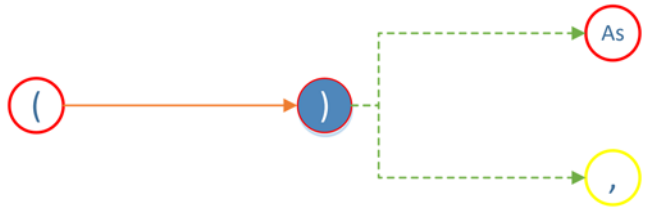

*Same children as DataSeparator Comma, but allows right and left child to be of different datatypes
True False
Id
Id
And Or
Id
AndAlso OrBse
=, >, ≥, <,≤, <>
)
,
[
Not
+, -
(
123
:
)
,
As
Id
Id
All
Id
And Or
To
Step
[
(
+, -
Not
123
:


)
)
)
]
=, >, ≥, <,≤, <>
,
+, -, *, /, \, %, ^
,
To
=
=
And Or
,
+, -
Step
Returns
[
]

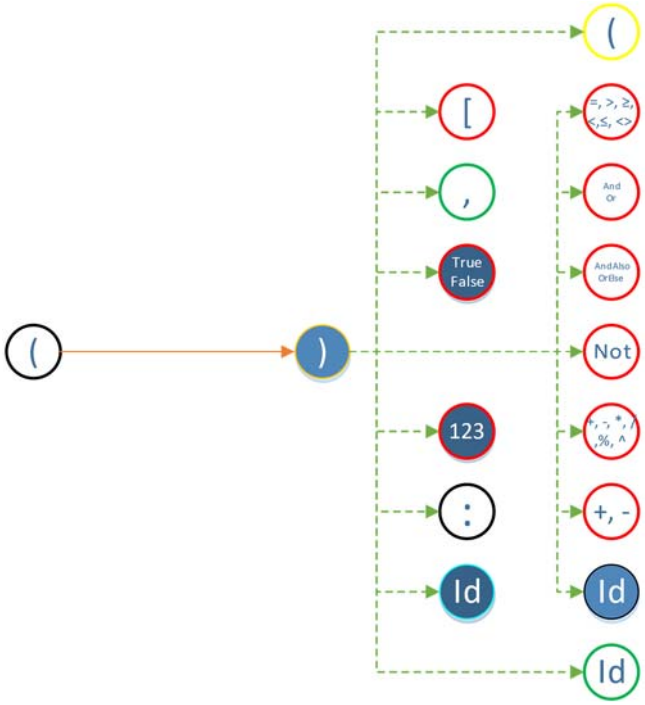

[
]
And Or
+, -, *, /, \, %, ^
=, >, ≥, <,≤, <>
True False
:
Id
Id
AndAlso OrElse
+, -
,
[
123
Not
(
Id


,
,
,
And
Or
AndAlso
OrElse
=, >, ≥,
<,≤, <>
To
+, -, *, /
,%, ^
Step
=
+, -
)
)
]
Then
(
)
Returns
Not
=
While

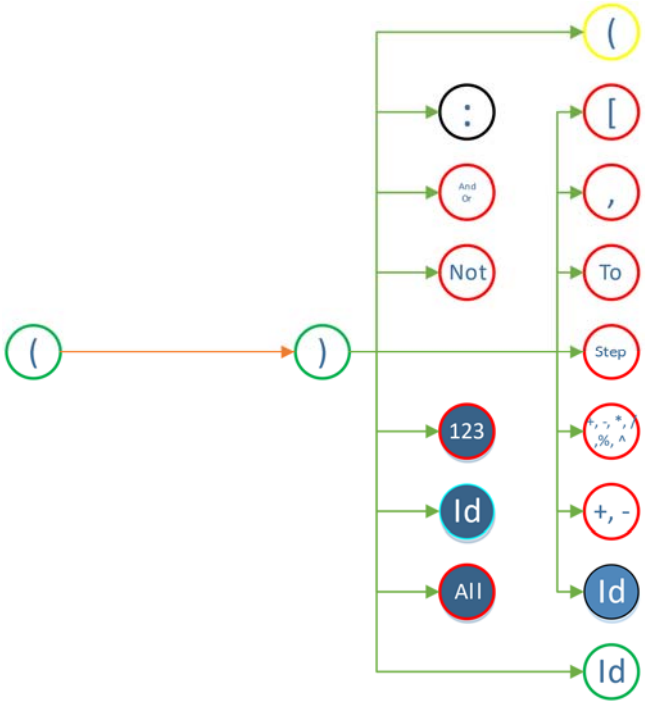

(
[
True
False
123
:
Id
(
)
=, >, ≥,
<,≤, <>
And
Or
AndAlso
OrElse
Not
+, -, *, /
,%, ^
+, -
Id
Id
Id
(
)

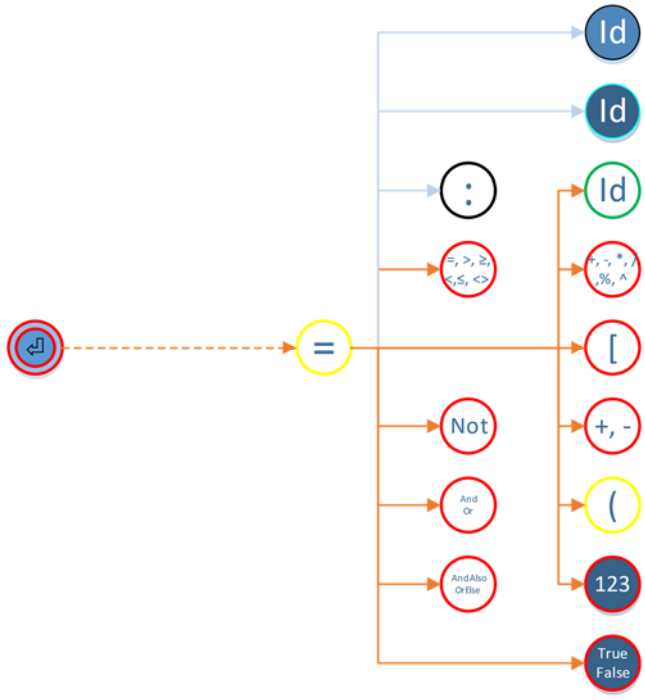

(
)
As
,

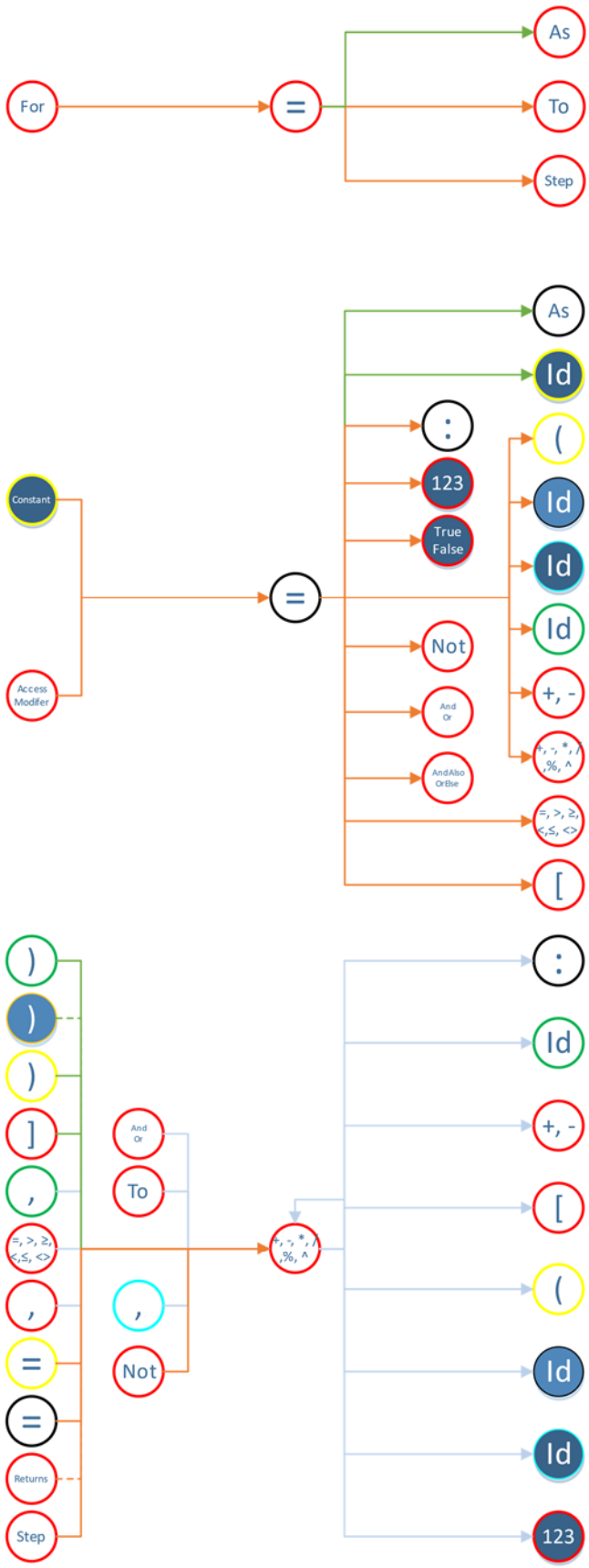

Id
(
)

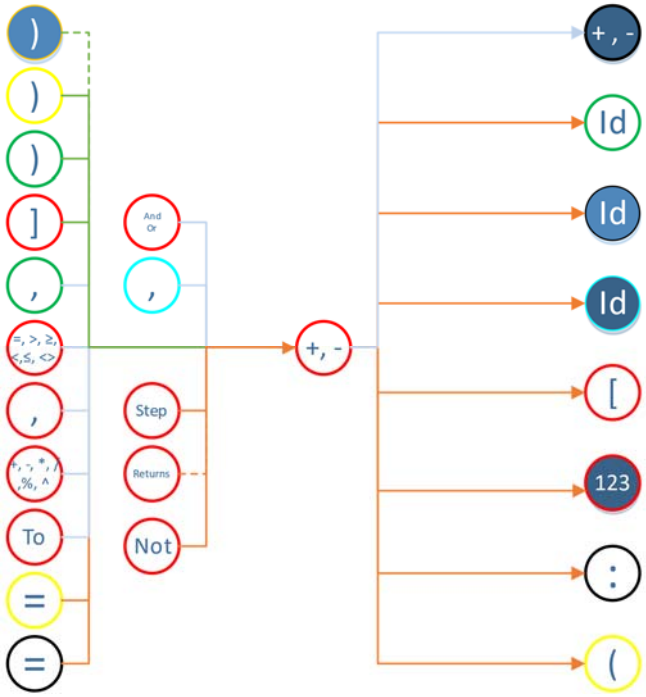

(
)
[
,
True
False
123
:
Id
(
=, >, ≥, <,≤, <>
And
Or
AndAlso
OrElse
Not
+, -, *, /, %, ^
+, -
Id
Id

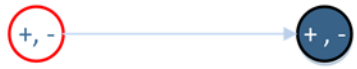

Id
(
)

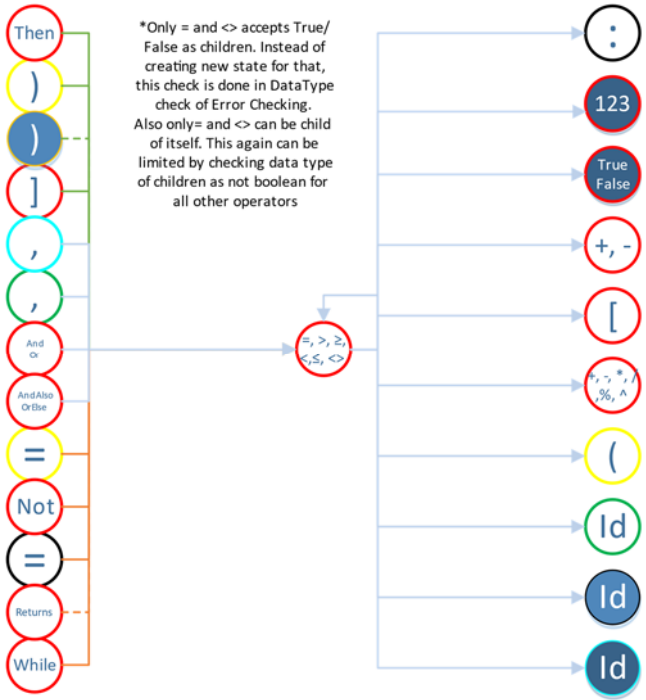

(
)
:
And
Or
Not
123
Id
All
(
[
,
To
Step
+, -
Id
Id

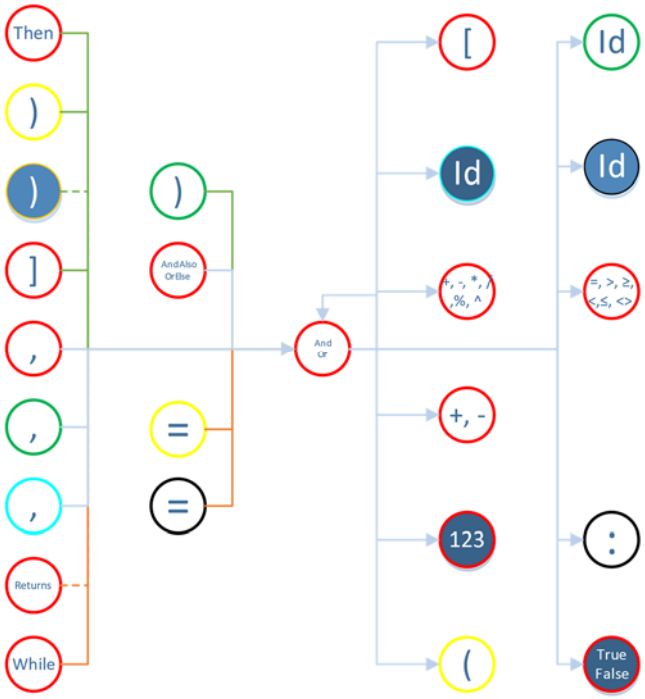

=
:
=, >, ≥,
<,≤, <>
Not
And
Or
AndAlso
OrElse
Id
Id
Id
[
+, -
(
123
True
False

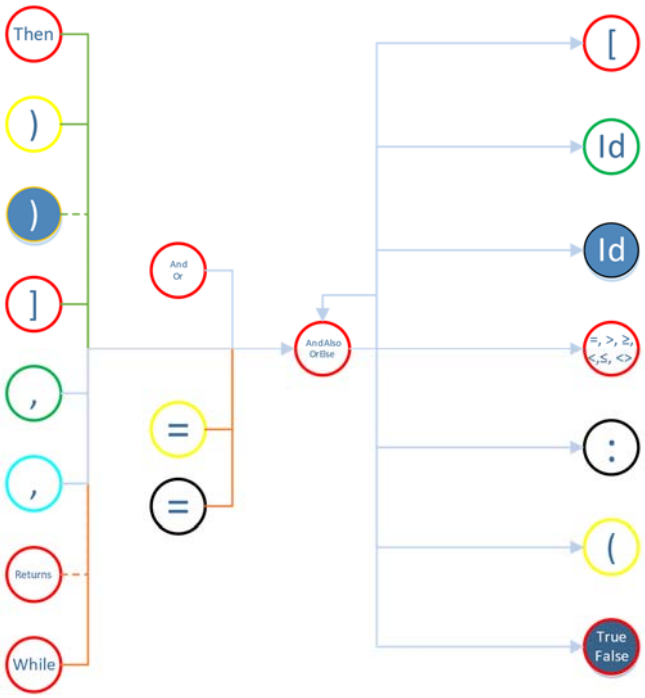

For
=
As
To
Step
Constant
Access Modifer
=
As
Id
:
(
123
Id
True False
Id
Id
Not
+, -
And Or
*, /, ,%, ^
AndAlso OrElse
=, >, ≥, <,≤, <>
[
)
)
)
]
,
=, >, ≥, <,≤, <>
,
=
=
Returns
Step
And Or
To
,
Not
+, -, *, /, ,%, ^
:
Id
+, -
[
(
Id
Id
123

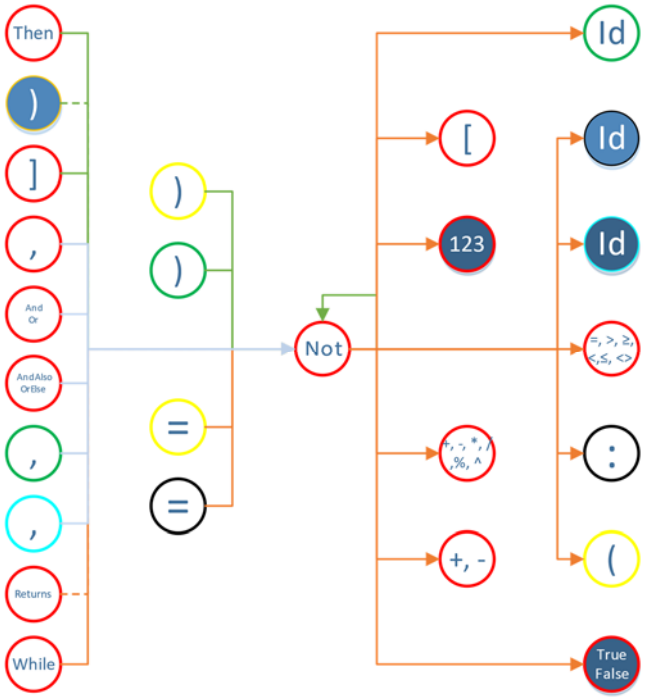

)
)
)
]
,
=, >, ≥, <,≤, <>
,
+, -, *, /, %, ^
To
=
=
And Or
,
Step
Returns
Not
+, -
+, -
Id
Id
Id
[
123
:
(

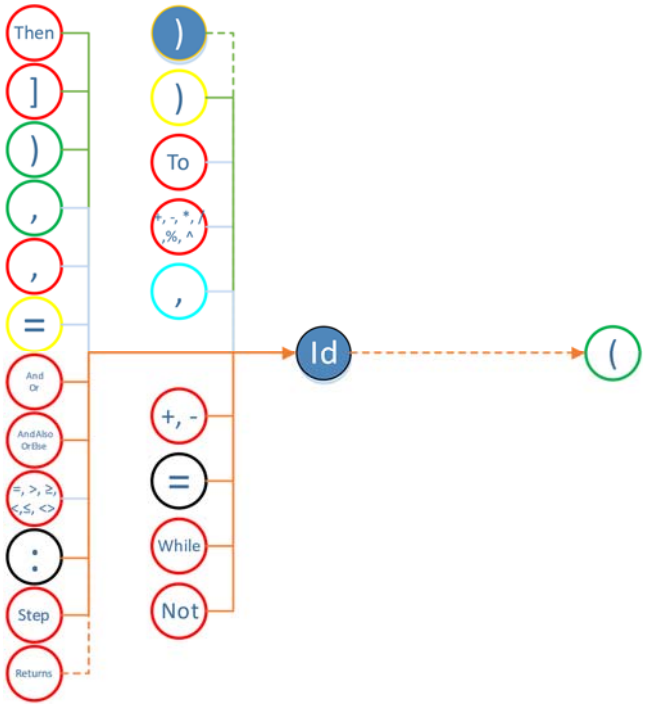

+, -
+, -

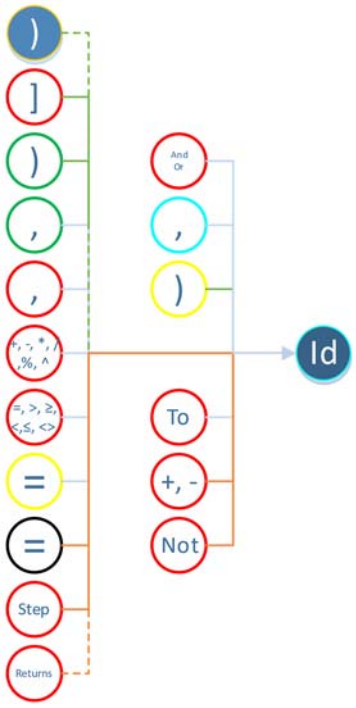

Then
)
)
]
,
,
And Or
And Also Or Else
=
Not
=
Returns
While
*Only = and <> accepts True/ False as children. Instead of creating new state for that, this check is done in DataType check of Error Checking. Also only= and <> can be child of itself. This again can be limited by checking data type of children as not boolean for all other operators
=, >, ≥, <,≤, <>
:
123
True False
+, -
[
+, -, *, /, %, ^
(
Id
Id
Id

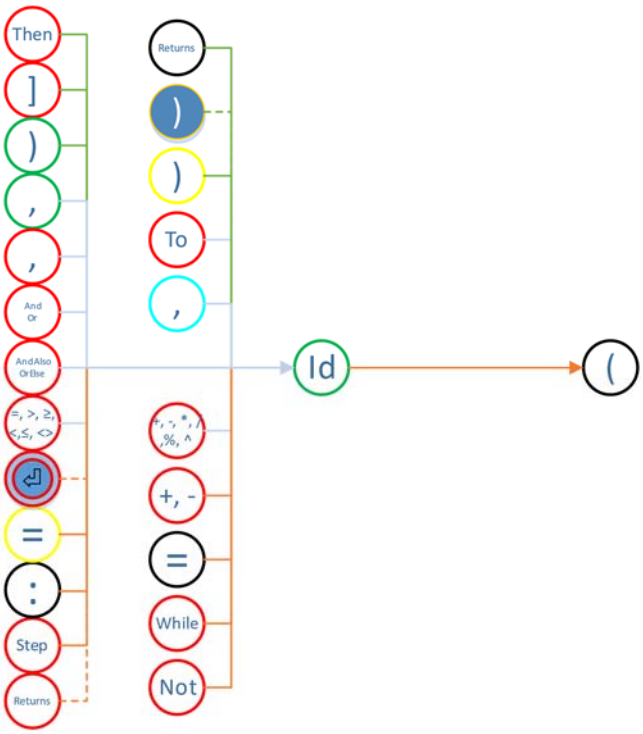

Then
)
)
]
,
,
,
Returns
While
)
AndAlso
OrElse
=
=
And
Or
[
Id
+, -
123
(
Id
Id
=, >, ≥,
<,≤, <>
:
True
False

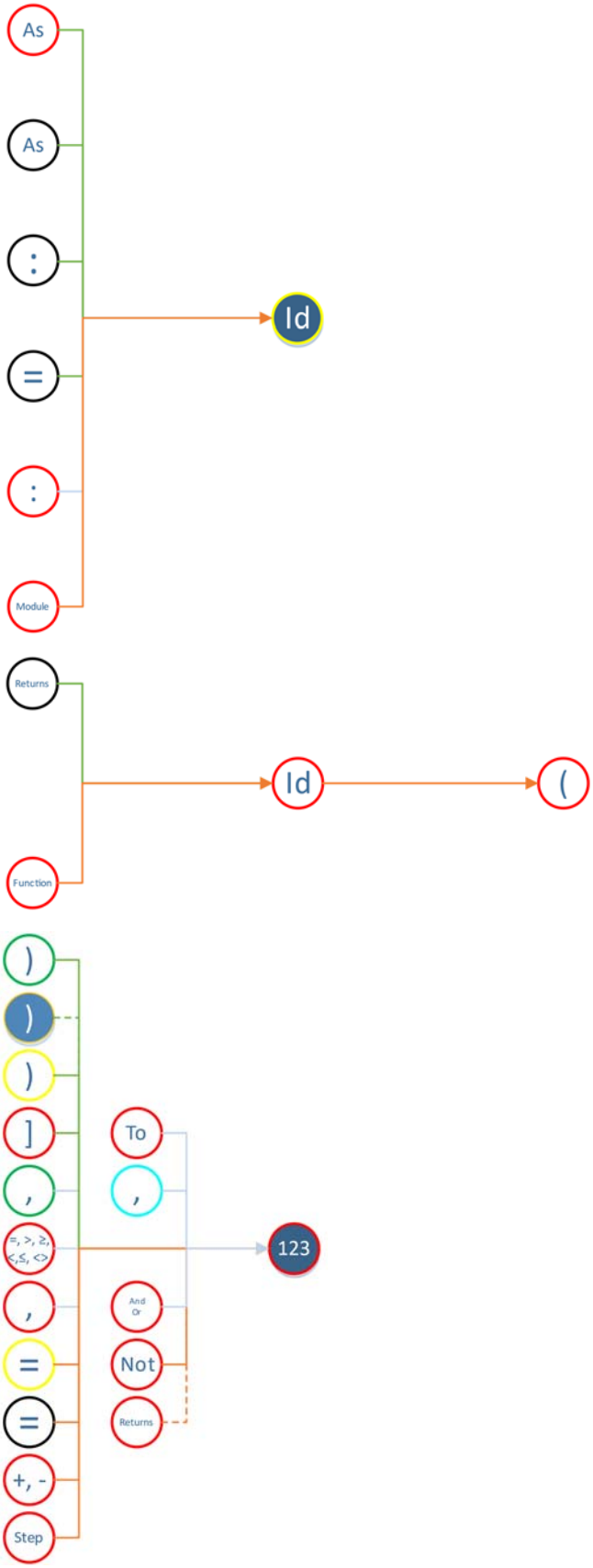

Then
)
)
]
,
,
,
Returns
While
And
Or
=
=
AndAlso
OrElse
[
Id
Id
=, >, ≥,
<,≤, <>
:
(
True
False

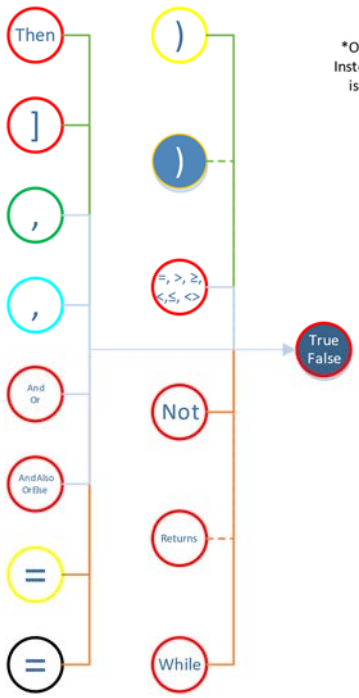

Then
)
]
,
And
Or
AndAlso
OrElse
,
,
Returns
While
)
)
=
=
Not
Id
[
Id
123
Id
=, >, ≥, <,≤, <>
+, -, *, /, %, ^
:
+, -
(
True
False


Then
]
)
,
,
=
And
Or
AndAlso
OrElse
=, >, ≥, <,≤, <>
:
Step
Returns
)
)
To
+, -, *, /, %, ^
,
+, -
=
While
Not
Id
(

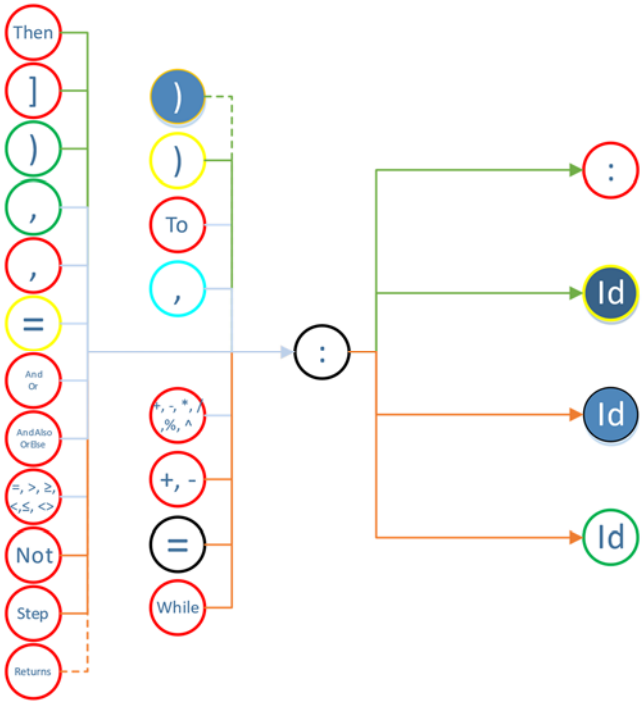

)
]
)
,
,
+, -, *, /, ,%, ^
=, >, ≥, <,≤, <>
=
=
Step
Returns
And
Or
,
)
Id
To
+, -
Not

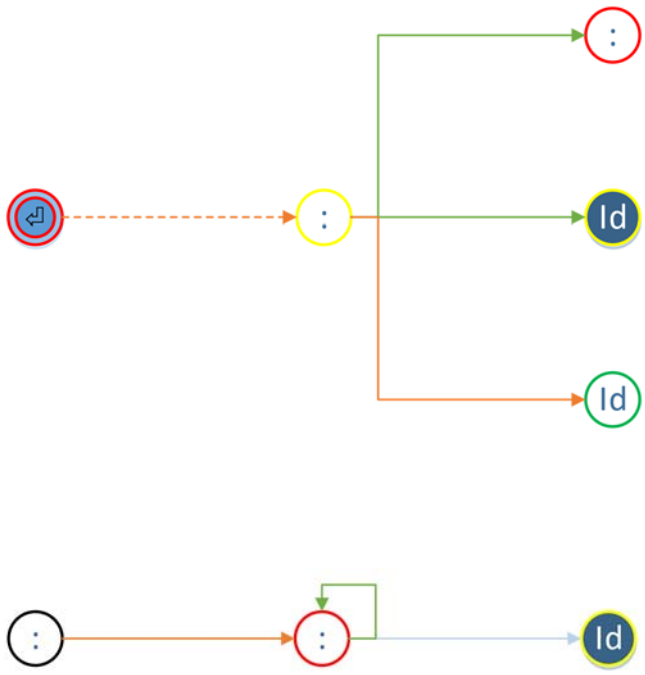

Then
]
)
,
,
And
Or
AndAlso
OrElse
=, >, ≥, <,≤, <>
=
:
Step
Returns
Returns
)
)
To
,
Id
(
+, -, *, /, ,%, ^
+, -
=
While
Not


As
As
:
=
:
Module
Id
Returns
Function
Id
(
)
)
)
]
,
=, >, ≥, <, ≤, <>
,
=
=
+, -
Step
To
,
And
Or
Not
Returns
123



*Only = and <> accepts True/False as children.
Instead of creating new state for that, this check is done in DataType check of Error Checking




:
:
Id
Id
:
:
Id

A.4 Intermediate Language Commands

All dot operations are element by element operation. If one of operand is matrix and other is native, then operation is carried out on each element of matrix and the native element. If both are matrix, then operation is carried out on each element of same index.

List of IL commands:

Compiler *Script only* *Model only* *Script and Model*

Instruction	*Hex*	*Usage*	*Format*
LVD	0x0A	*Load double value on to stack*	*LVD, 32-bit LSB, 32-bit MSB*
LVF	0x0B	*Load single IEEE float on to stack*	*LVF, value*
LVL	0x0C	*Load long value on to stack*	*LVL, 32-bit LSB, 32-bit MSB*
LVI	0x0D	*Load integer value on to stack*	*LVI, value*
LVS	0x0E	*Load short value on to stack*	*LVS, value*
LI1	0x0F	*Load integer 1 on to stack*	*LI1*
LI0	0x10	*Load integer 0 on to stack*	*LI0*
LD0	0x11	*Load doube 0 on to stack*	*LD0*
LF0	0x12	*Load single 0 on to stack*	*LF0*
LL0	0x13	*Load long 0 on to stack*	*LL0*
LS0	0x14	*Load single 0 on to stack*	*LS0*
LOT	0x15	*Load boolean True on to stack*	*LOT*
LOF	0x16	*Load boolean False on to stack*	*LOF*
LOL	0x06	*Load local variable*	*LOF, index of variable (first declaration is index 0)*
LOA	0x07	*Load fucntion argument*	*LOA, index of variable (first function argument is index 0)*
LRF	0x08	*Load reference variable*	*LFR (reference must be loaded on to reference stack)*
LMD	0x09	*Load method*	*LMD (function must be loaded on to reference stack)*
FCAL	0x05	*Function call (different from LMD, used when a function may return a value but the value is not used by calling function)*	*FCAL (function must be loaded on to reference stack)*

ASV	0x1A	*Assign value. Assigns topmost value on stack to the previous element on the stack which must be a refernece loaded on the stack using LRF*	*ASV*
ASR	0x1B	*Assign reference. Makes the last but one element on stack to the last element on stack.*	*ASR*
AIT	0x1C	*Assign if True. Top most element on stack is the value to be assigned, preced by condition for assignment, preceded by reference to be assigned to. If any of the element was a matrix, then does element by element condition check and assignment. If more than one of the stack element is matrix, then all the three must be dot operation compatible.*	*AIT*
SEL	0x1D	*Select elements from a matrix.*	*SEL, number of dimensions explicitly mentioned, start index, to index, step index of each dimension mentioned in that order.*
NEM	0x1E	*New matrix. All the elements of the matrix must have been previously loaded on to the stack. If say matrix was 3D matrix A, then A(1,1,1) is loaded first followed by A(1,1,2) and so on.*	*NEM, number of elements, number of dimension, size of each dimension.*
ADD	0x1F	*Addition. The operand on right of operator must be on top of the stack and on the left must be before it. This is dot operation.*	*ADD*
SUBT	0x20	*Subtraction. Format same as ADD.*	*SUBT*
DMUL	0x21	*Dot multiplication. Format same as ADD.*	*DMUL*
DDIV	0x22	*Dot division. Format same as ADD.*	*DDIV*
CMUL	0x23	*Cross multiplication. The operand on right of operator must be on top of the stack and on the left must be before it. If any of the operand is matrix, dimension of both must be cross operation compatible.*	*CMUL*
CDIV	0x24	*Cross division. The operand on right of operator must be on top of the stack and on the left must be before it. If any of the operand is matrix, dimension of both must be cross operation compatible and second operand must be square matrix or native.*	*CDIV*
DPOW	0x25	*Dot power. The operand on right of operator must be on top of the stack and on the left must be before it. Operand must be square matrix.*	*DPOW*
CPOW	0x26	*Cross power. The operand on right of + must be on top of the stack and on the left must be before it. Operand must be square matrix.*	*CPOW*
DMOD	0x27	*Dot modulo. Format same as ADD.*	*DMOD*
SHL	0x28	*Shift left. The operand on right of operator must be on top of the stack and on the left must be before it. Both operand must be of integral type. This is a dot operation.*	*SHL*
SHR	0x29	*Shift Right. Format same as SHL.*	*SHR*

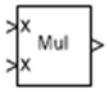
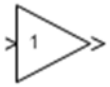
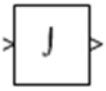
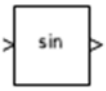
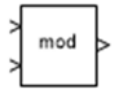
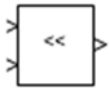
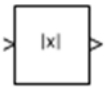
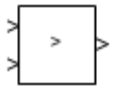
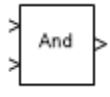
NEG	0x2A	*Negate. Operand must be loaded on the stack prior to this instruction.*	*NEG*
CINV	0x2B	*Cross inverse. Operand must be loaded on the stack prior to this instruction. Operand must be square matrix or native.*	*CINV*
DINV	0x2C	*Dot inverse. Operand must be loaded on the stack prior to this instruction.*	*DINV*
CAT	0x2D	*Concatenate matrix. Prior to this isntruction, all matrices to be concated must be loaded followed by the output variable where the result is stored, followed by the dimension on which concatenation is pivoted, on to the stack. Inputs must be dot operation compatible.*	*CAT, number of input matrices*
DCAT	0x2E	*Split matrix. Prior to this isntruction, references to variable where each split matrix is stored must be loaded followed by the input matrix variable to be split, followed by the dimension on which split is pivoted, on to the stack. The dimension size of the pivot must be of integral multiple of number of outputs..*	*DCAT, number of output matrices*
GT	0x2F	*Greater than. The operand on right of operator must be on top of the stack and on the left must be before it. This is dot operation.*	*GT*
GEQ	0x30	*Greater than or equal to. Format same as GT.*	*GEQ*
LT	0x31	*Lesser than. Format same as GT.*	*LT*
LEQ	0x32	*Lesser than or equal to. Format same as GT.*	*LEQ*
EQ	0x33	*Equal to. Format same as GT.*	*EQ*
NEQ	0x34	*Not equal to. Format same as GT.*	*NEQ*
ORL	0x35	*Logical and bitwise OR. The operand on right of OR must be on top of the stack and on the left must be before it. This is dot operation. If element on stack is boolean then its logical OR, else if it is integeral then bitwise OR else exception.*	*ORL*
ANDL	0x36	*Logical and bitwise AND. Fomat same as ORL.*	*ANDL*
XORL	0x37	*Logical and bitwise XORL. Format same as ORL.*	*XORL*
NOTL	0x38	*Logical and bitwise NOTL. Operand must be loaded on to stack prior to this instruction. If boolean operand then logical NOT, if integeral then bitwise NOT else exception.*	*NOTL*
SIN	0x39	*Sine. Argument must be loaded on to stack prior to this instruction.*	*SIN*
COS	0x3A	*Cosine. Format same as SIN.*	*COS*
TAN	0x3B	*Tangent. Format same as SIN.*	*TAN*
CSC	0x3C	*Cosecant. Format same as SIN.*	*CSC*
SEC	0x3D	*Secant. Format same as SIN.*	*SEC*

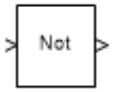
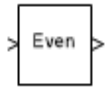
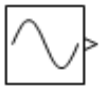
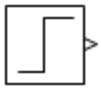
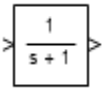
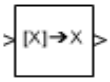
COT	0x3E	*Cotangent. Format same as SIN.*	*COT*
ATAN	0x3F	*Inverse tangent. Format same as SIN.*	*ATAN*
ATAN2	0x40	*Four quadrant inverse tangent. 'y' loaded onto stack first followed by 'x' and then call to this instruction.*	*ATAN2*
EXP	0x41	*Exponent. Format same as SIN.*	*EXP*
LOG	0x42	*Logrithm to base 10. Format same as SIN.*	*LOG*
LN	0x43	*Natural logrithm. Format same as SIN.*	*LN*
FLR	0x44	*Floor. Rounds to least of nearest two integral value. Format same as SIN.*	*FLR*
CEIL	0x45	*Ceiling. Rounds to highest of nearest two integral value. Format same as SIN.*	*CEIL*
RND	0x46	*Round. Rounds (banker's rounding) to nearest integer. Format same as SIN.*	*RND*
IAT	0x47	*Is Any True. Loads True to stack if any element of matrix is True or if native is True. Operand must be loaded on to stack prior to this isntruction.*	*IAT*
IAF	0x48	*Is Any False. Loads False to stack if any element of matrix is False or if native is False. Format same as IAT.*	*IAF*
CTE	0x49	*Convert to native. Converts a matrix to native element. The matrix must have just a single element in it. Operand must be loaded on to stack prior to this isntruction.*	*CTE*
TIF	0x4A	*Ternary IF. Checks a condition and loads a value on to stack if true, else another value if false. This is an element wise operation and operands must be dot compatible. Prior to this instruction, the condition must be loaded on to stack, followed by value if true and then value if false.*	*TIF*
ING	0x4B	*Integration. Prior to instruction, the output variable where result is stored is loaded on to stack, followed by the input. The integration is carried out by the solver selected.*	*ING*
DIF	0x4C	*Differentiation. Prior to instruction, the output variable where result is stored is loaded on to stack, followed by the input. The differentiation is carried out by the solver selected.*	*DIF*
BRU	0x17	*Branch unconditionally*	*BRU, number of instruction to jump, number of references to jump*
BRT	0x18	*Branch on true. Prior to this instruction, the condition must be loaded on to the stack and must be a native boolean.*	*BRT, number of instruction to jump, number of references to jump*
BRF	0x19	*Branch on false. Format same as BRT.*	*BRF, number of instruction to jump, number of references to jump*
EMP	0x00	*Empty field. Internal instruction, used by compiler during code compression.*	*EMP, number of empty fields*
NOP	0x01	*No operation*	*NOP*

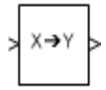
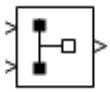
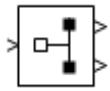
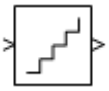
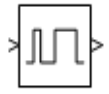
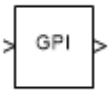
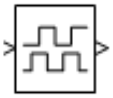
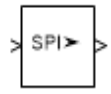
EOP	0x02	*End of program. Exists program execution immediately.*	*EOP*
RET	0x03	*Function Return*	*RET*
EXC	0x04	*Throw Exception*	*EXC, exception code*
DLG	0x4D	*Data log. Prior to this instruction, the log variable must be loaded on to stack followed by the variable to be logged.*	*DLG*
PKG	0x4E	*Pack data. This instruction is used to pack variables of different type and sizes into sequential memory fields that is sent as a serieal bitstream in real-time mode. It must be preceded by loading all the variables to be transmitted on to the stack with first to be sent loaded last followed by the variable where the linearized sequential bitstream is saved.*	*PKG, number of variables to be packed.*
UPK	0x4F	*Unpack data. This instruction is used to unpack a sequentail memory fields that is received as serial bitstream into variables of different type and sizes in real-time mode. It must be preceded by loading all the variables to be unpacked on to the stack with first to bereceivedloaded last followed by the variable where the linearized sequential bitstream is saved.*	*PKG, number of variables to be unpacked.*
BRP	0x50	*Branch point marker. Marks the point where one or more branch to instruction jumps to.*	*BRP*
CEL	0x51	*Conditional evaluvate marker. This is used by tools which do not evaluvate as a single block but rather evaluvates certain inputs based on certain condition. This instruction is used to indicate all branches that a tool could take.*	*CEL, number of branches; port index of branch, instruction stack count, reference stack count for each branch.*
EOL	0x52	*End of line. Marks the end of a single unit, in case of Script, a single code line, in case of model, a single tool. Any runtime exception within a unit points to a single line of code or tool. This is used for building debug info.*	*EOL*

A.5 List of Tools

All tools support matrix and mux array operations.

Tool Name	Tool Default	Description
Adder/Subtractor		Adder/subtractor.
Multipler/Divider		Multiplier/divider.
Gain		Fixed dot multilier.
Integrator		Numerical integerator using selected solver. Currently supports 4 fixed step solvers.
Differentiator		Numerical differentiator using using upto 6^{th} order forward difference.
Transcendental Functions		All trignometric functions, exponent, log, pow etc.,
4 Quadrant Arc Tangent		Inverse tangent with output between 0 to 2π.
Modulo		Returns remainder of division operator.
Left/Right Shift		Left and right shift for integral operators.
Rouding Functions		Flor/ceil and banker's rouding functions.
Absolute		Returns postive of negative/positive value.
Comparison Operators		>, <, ≥, ≤, ≠, = operators. Output of these are of boolean type.
Logical/Bitwise Operator		And/Or/Xor operattors. Logical operators takes boolean inputs and returns boolean output. Bitwise does bitwise operation on only integral types and returns integral value.

Logical/Bitwise Invert		Same as previous tool, except not operation and takes just single input.
IsEven/IsOdd		Returns true if input is even/odd based on what is selected, else false.
Constant Source		Constant numerical value.
Boolean Source		Constant boolean value.
Sine wave		Sine wave source of adjustable magnitude, frequency and phase shift.
Sawtooth/Triangle		Sawtooth and triangle source of adjustable magnitude and frequency.
Step Source		Step source of adjustable initial, final value and step time.
Mag-Time Chart		Value as function of time plot. Plots live in both simulation and real-time mode.
Transfer Function		Transfer function block using canaolical forms.
Sub-system		Subsystem to enclose more tools within it.
In-port		Input port of subsystem.
Out-port		Output port of subsystem.
Mux		Mux signal to convert inputs to a sequential array that contains all the input elements. This is just to make it easier to develop by being able to run multiple signal via single wire. This does not add any code of its own, but rahter links each of its output to its input during compile time.
De-Mux		Break out muxed signal to indiviual components.
Matrix to Element Case		Converts a matrix to native element, only if the matrix has just single element, else will throw runtime exception.

Numeric to Numeric Cast		Casts from one numeric type to another.
Boolean to Numeric Cast		Converts booltan to numeric type with adjustabe True and False equivalent values.
Matrix Concatenate		Concatenates multiple matrices of same dimensions into a new matrix, along any dimension.
Matrix Separate		Splits a mtrix along any dimenion into mutliple evenly dimensioned matrices.
Saturation		Staturates input signal to adjustable postive and negative limits.
Index Selector		Index base conditional evaluvator block. When otpimization is enable generates code such that it only evaluvates tools that are needed and skip evaluvation of tools not needed in the current iteration based on the input selector index.
If/Else Selector		Same as previous tool except, input is limited to a True/False case.
ADC		Real-time controller analog to digital converter peripheral. Auto genrates controller specific code inreal-time. In similation, introduces ceertain aspects of real world non-liearites if enabled, else acts as 1:1 block.
PWM		Pulse width modulation, real-time controller peripheral.
Digital Input		Digital input peripheral.
Digital Output		Digital output peripheral.
A quad B		A quadrature B encoder peripheral. Outputs position, and speed based on which is selected.
SPI/SCI Write		Serial interface write. Can write matrices, natives or any number of combinations of those.
SPI/SCI Read		Serial interface read.