Introduction and<br>Reference Manual

$$
u=\int_{0}^{\varphi} \frac{d \theta}{\left(1-m \sin ^{2} \theta\right)^{1 / 2}}
$$



## JCP Software \& Company

Information is this document is subject to change without notice and does not represent a commitment on the part of JCP Software \& Company. The SOFTWARE described in this document is furnished under the Software License Agreement as can be viewed in the back of this manual. The SOFTWARE may be used or copied only under the terms of the license agreement. The purchaser may make one copy of the SOFTWARE for back-up copies. No part of this Manual may be modified in any manner whatsoever without the prior written permission of JCP Software \& Company.

# Copyright © 1994, 2019 by JCP Software \& Company All Rights Reserved 

Manufactured in the United States of America
Published by JCP Software \& Company

The following trade names are referenced throughout this manual.
Slide-Rule ${ }^{\text {TM }}$ is trademark of JCP Software \& Company. Registered trademark submitted for approval.

IBM-PC® is a registered trademark of International Business Machines Corporation.
Windows ${ }^{\top \mathrm{M}}$ is a trademark of Microsoft ${ }^{\circledR}$ Corporation.
$386 \mathrm{SX}^{\text {TM }}$ is a trademark of Intel ${ }^{\circledR}$ Corporation

## Introduction and <br> Reference Manual



## JCP Software \& Company

## Table of Content

Preface ..... 7
Chapter 1 - Introduction ..... 11
Chapter 2 - A Short Tutorial ..... 17
Chapter 3 - The Shell Script Grammar ..... 25
Chapter 4 - The Matrix/Vector Grammar ..... 53
Chapter 5-The Editor ..... 73
Chapter 6 - Plotting Details ..... 79
Chapter 7 - User Dialog's and Menus ..... 105
Chapter 8 - Digital Filter Design ..... 111
Chapter 9 - Spectral Analysis and Convolution/Correlation ..... 143
Chapter 10 - Signal Processing ..... 153
Chapter 11 - Random Signals and their Distributions ..... 173
Chapter 12 - Adaptive Filters ..... 187
Chapter 13-FEC Channel Coding. ..... 207
Chapter 14 - Matrix/Vector Computations ..... 225
Chapter 15 - Special Mathematical Functions ..... 249
Chapter 16 - Differential Equations and Numerical Integration ..... 257
Chapter 17-Control Engineering ..... 273
Chapter 18 - Microwave Transistor Amplifier Design ..... 295
Appendix A-Functions Reference List ..... 323
Appendix B-Equations Reference for the Smith Chart Tool ..... 335
JCP Software \& Company License Agreement ..... 347

## Preface

This manual is about a software tool, as well as about a journey. We begin with a brief history. Beginning in 1956, Texas Instruments fabricated the first transistor circuit, and a few years latter the FFT was discovered in 1965. Signal processing at that time was just in its infancy. Starting in 1974, Intel came out with the 8 -bit 8080 microprocessor with a whooping 2 MHz cycle time, and starting in the eighties, Bell Labs developed the Unix operating system and the C programming language. That's when software development made a giant leap forward. That little 40 pin chip 8080 microprocessor from Intel led to the incredible microprocessors we have today. It's also led companies, such as Microsoft and Apple, to develop operating systems on personal computers beginning in the nineties.

Just as a coin as two sides, namely heads and tails, in a like manner these advances in technology happened both in hardware and software. On the hardware side we have integrated circuits (ICs), the analog-to-digital converter (ADC), the digital-to-analog converter (DAC), the field programmable gate arrays (FPGA), and full systems on a chip (SOC) or application-specific integrated circuits (ASIC). A relationship known as Moore's law, states that the cost of processing a specified processing task on an IC drops be a factor of two every 18 months. In a like manner, the development of the C and $\mathrm{C}++$ programming languages and compliers has lead to a huge increase in the efficiency of software development. Also the development of the YACC/LEX complier technology has led to the efficient development of programming languages for specific tasks such as work stations for ASIC design, etc. All of these technologies as mentioned above made it possible for this author to develop this design tool.

This software tool can be likened to a small wrench in an engineers tool kit, as much more powerful design and analysis systems might be needed. However, as engineers of yesterday carried and used actual sliderules, which lasted a life time, this software tool can last an engineer's life time also, so long as his personal computer and Windows laptop are up and running.

## Organization of Manual

Although some chapters in this manual are highly technical, it is not the intent of this manual to educate the user in these disciplines, as it would make the text unnecessary long and burdensome, and do an injustice to the many technical sources that are widely available. The intent of this manual is develop and describe a software platform that is easy to learn and use, and is capable of solving a variety of engineering design problems. In order to arrive at that goal, we work through different engineering disciplines and design examples to find bugs in the compiler/interpreter, plotting routines, etc., and most of all, to determine what kernel functions are missing to complete these design problems. It turns out that most software problems are functions or features that are left out of the system. Also note that is not the intent of this design system to try a provide a function (or subroutine) to solve as many different situations that might arise, but instead relay on the user to develop there own procedures or modify the provided shell script code provided by this SOFTWARE to solve many problems that come up. I believe at this point, that there are enough kernel functions as listed in Appendix A, for the user to develop algorithms and engineering designs.

Having said that, Chapters 1 through 7 cover introductory material as well as describing the SOFTWARE, while Chapters 8 through 18 cover same basic engineering topics in the area of telecommunications. These chapters have footnotes for topics that reference the list of references provided at the end of each chapter, and some topics and examples show the shell script code so the reader can learn and relate to the information provided in the first seven chapters,

Chapter 1 contains introductory information valuable for everyone. Chapter 2 provides a short installation and a getting started section as well as tutorials on various selected topics dealing with the main menu tile selections. Chapter 3 is a detailed description of the Shell Script language in terms of syntax, and a presentation of the shell's language grammar. Chapter 4 covers Matrix and Vector operations and its associated grammar. Chapter 5 is a description of the built in Custom editor. Chapter 6 covers plotting details with examples of the different plots available in this design tool. Chapter 7 is a
description of the user Dialog's interface with examples, and a description of the user defined Menu option.

Chapter 8 covers Digital Filter design using the GUI interface. Topics covered include: FIR filter design using the Parks-McCellan algorithm; FIR half-band Filters; FIR Differentiators; The Hilbert Transformer; MAXFLAT FIR design; Windowed FIR design; IIR Filter Design; Interpolated FIR filters; Raised Cosine Filter Design; Root raised Cosine Filter Design; Meteor Filter Design; Hogenauer Filter Design; Filter Response Analysis; and Fixed Point Coefficients;

Chapter 9 covers Spectral Analysis including Convolution /Correlation. Topics include Power Density Spectrum using the Window-Overlap Method; PSD for Complex Signals in Noise; and the Zoom FFT using the Chirp Transform Algorithm.

Chapter 10 provides some examples of Signal Processing Applications. Topics include General IIR Filter Design using a user generated dialog procedure; Lattice Filter Structures; The Z transform Inversion; Analog Filter Design using a user generated dialog procedure; The CELP Algorithm used in cell phones; and OFDM presentation of 802.11a (WiFi) for determining carrier frequency offset (CFO), and symbol timing offset (STO).

Chapter 11 covers some examples of Random Signals and their Distributions. Topics covered include the Uniform Random variable; the Gaussian (Normal) Random variable; the Chi-square ( $\mathrm{X}^{2}$ ) Central Random Variable; the Chi-square ( $\mathrm{X}^{2}$ ) Non-Central Random Variable; the Rayleigh Random Variable; and the Rican Random Variable.

Chapter 12 provides some examples in the area of Adaptive Filters. Topics covered include: A Wiener Filter for Noise Canceling; Noise Canceling using the Normalized LMS Algorithm; An example of the Steepest Decent Algorithm; Echo Cancelation using a Fast Block Adaptive Filter; System identification using an Adaptive Filter; A 16-QAM decision Directed Equalizer using NLSM; and QPSK and Blind Equalizers.

In Chapter 13 we show some examples of FEC Channel Coding. Topics include the Golay $(12,24)$ Code; BCH Error Correcting Cyclic Codes; A Reed-Solomon 255,233 with Erasures; The Reed-Solomon CIRC; A Rate K = $1 / 2$ Viterbi Decoder; and a Turbo encoder/decoder example.

Chapter 14 covers examples of Matrix/Vector computations. Topics include The QR Decomposition with Column Pivoting; Cholesky factorization; The Complete Orthogonal Decomposition with Column Pivoting; A Determinate Example; The Hessenburg form of a Matrix; Schur Decomposition; Matrix Inverse Examples; A Lu Factorization Example; Solving Linear Equations; A Least Squares Fit; A Matrix Pseudo inverse Example; A Matrix Norm-2 Least Squares Example; An Over-Determined Set of Equations; Eigenvalue/Eigenvector Examples; A reduced Row Echelon Example; Matrix Vector Norms; Toeplitz Matrices; The Square Root of a Matrix; and a Generalized Eigenvalue Problem.

Chapter 15 provides some examples of Special Functions; Examples include Bessel Functions of the first kind Jn; Bessel Functions Yn, In, and Kn; Incomplete Elliptic Integrals of the $1^{\text {st }}$ and $2^{\text {nd }}$ kind; Error functions $\operatorname{erf}(x)$ and $\operatorname{erfc}(x)$; The Gamma Function $P(a, x)$; The Beta Function $B(z, w)$; The Jacobian Elliptic Function; The Incomplete Gamma Function; The Exponential Integral; Chebyshev Polynomials; Jacobi Polynomials; and Legendre Polynomials.

Chapter 16 provides some examples in the area of solving some ordinary differential equations and Numerical Integration problems. Topics include breaking down a higher order ODE to a cascade of $1^{\text {st }}$ order differential equations, and solving using the Runge Kutta $4^{\text {th }}$ order algorithm, and presenting several examples; A Numerical Integration Example using Simpson's Rule and Romberg Integration with Richardson's improvement; The Minimum of a Function of a Single Variable; and The Minimum of a Function of Several Variables.

Chapter 17 covers selected examples in Control Engineering. Topics include an example of a Unit Step Response; A Unit-Impulse Response; Step Responses of Second-Order Systems; Bode Plots from a Transfer Function; Nyquist Plots from State-Space; Gain Adjustment for \% Overshoot; A PI Compensator; A Lag Compensator; An Ideal Derivative Compensator; A Lead-Lag Compensator; A Regulator Design in State-Space with Pole-Placement; A Lag Compensator; An ideal Derivative Compensator (PD); Controller versus Observable Design in State Space; and a Digital Compensator.

Chapter 18 covers Microwave Transistor Amplifier design examples using a graphical user interface to the Smith Chart with one's mouse and keyboard. Topics include A Simple Impedance Match; Small Transistor Design for MAG; Small Transistor Design for Specified Gain; Small Transistor Design for

Minimum Noise; Small Transistor Design for Stability; Micro-Strip Matching - Different Characteristic Impedances; A Single Stub Tuner; A Double Stub Tuner; and Plotting of Performance Data.

Appendix A is a series of block tables listing the name along with a function statement of the supplied functions that comes with this software, grouped together on a functional basis. Appendix B is a description of the equations used in the Smith Chart Design Tool.

## Acknowledgements

Professor fredric j harris of San Diego for answering my many questions over the years. He's such a master of DSP as well as an excellent teacher. He can quickly come down to one's level, and explain a subject, or helps one solve a difficult problem. A gentleman and scholar, he has one leg in academia, as well as one in industry, and has wide experience consulting on high performance DSP systems. Another gentleman that helped me along the way was Richard Jekel of Cubic communications and Cubic Defense. Dick is one of those engineers who can do system analysis and design, design signal processing systems, hardware digital design, analog design, RF design, etc... He's a fun guy to work with, and I learned a lot from him. Another engineer is Dr Jonathon Cheah, a RF engineer I worked under while at Hughes Network Systems. Besides being an expert RF engineer, he's also is an expert in satellite and mobile communications, and has wide experience in project engineering and engineering management. He got me interested is the Smith Chart tool that I developed for this software.

## About the Front Cover

The graphic displayed is the Jacobian Elliptic function which is also shown on the Front Cover of the Handbook of Mathematical Functions, edited by Milton Abramowitz and Irene A. Stegun. Some of the technology that went into that graphic as shown on the front cover of this manual include: the invention of the transistor in 1956; the CMOS microprocessor and associated memory; the Windows operating system; the C and $\mathrm{C}^{++}$language development and compiler technology; YACC/LEX; color monitor technology; word processor and font technology; color printers and print technology; etc... Another example of how far we come in such a short time.

## About the Author

John C Potter is a graduate of Portland State University in Applied Mathematics and Applied Science. He was elected to Pi Mu Epsilon Nation Honorary Mathematics Society, and made the deans list more than once. He has additional education at UCI Irvine in the areas of computer design, digital logic design, electronics, and digital signal processing. Also multirate signal processing and adaptive filters from Professor fredric j harris at San Diego State University. He is a Co-Inventor of two US patents (one a VLSI device) in the area of Speech Technology. He has experience in 3D plotting, Real-Time Flight Simulators, Marine Navigation, Military Command and Control, Electronic Warfare, SigInt, etc. He is a codesigner of a mult-processing DSP system from 4-bit AMD slices
 (2910 series, high-powered schottky), and designed the 32-bit instruction set to go with it. He has been a project engineer on two occasions, and directed from 8 to 13 engineers on those projects. He developed the software for the first weather display system for the US Weather Service. He also co-developed a state of the art speaker identification system, and installed it at NSA. In 1993 he started JCP Software \& Company, and worked in the San Diego area as a consultant in the tele-communications arena. In this endeavor he worked on CELP vocoders for cell phones and a satellite system, signal processing for digital radios and exciters, modems, TCP/IP, a telemetry system, and the PHY, MAC, and LNK layers in a communication system.

## Chapter 1 - Introduction

Slide-Rule ${ }^{T M}$ has been designed as a basic engineering/scientific toolbox for engineers, scientists, and students. It has been designed for the popular PC platforms and running under Microsoft ${ }^{\circledR}$ Windows. Its design intent is to provide a computational platform to allow one to develop algorithms and carry out complex calculations in a simple and straight forward manner. It comes with over 500 functions that cover Mathematical Functions, Special Mathematical Functions, Random Number and Distributions, Matrix/Vector Operations, Digital Filter Design, Spectral Analysis, Signal Processing Functions, Adaptive Filters, Control Engineering, Polynomial Functions, Differential Equations and Numerical Integration, Non-Linear Methods, Elementary Statistics, File I/O and String Functions, and over 50 Plotting routines.
Slide-Rule's ${ }^{T M}$ universal engine is a simple shell script language that has the look and feel of the ever popular C-language. In addition to the basic constructs of the C-language, are added matrix and vector types, in both real and complex form. These types allow the user to work at a higher level, while at the same time employing the lower level power and familiarity of the C-language type shell script. Since the running of the shell script is done in an interpretive manner, the user is freed from the chore of compiling and linking code using a commercial bought compiler. Also, because of the many built in plotting routines, the user can quickly write and run a shell script that allows one to visualize data in a wide variety of formats. All plotted data to the monitor can also be output to a printer device as well. You can also write all plotted output as an extended metafile that can be imported into popular word processing programs such as Microsoft's Word for Windows.

Besides the supplied internal functions, Slide-Rule ${ }^{T M}$ allows the user to build their own procedures (including subroutines) based on the shell script language. The shell script language is modeled after the universal C-language, and can be considered a subset of it. Only the basic constructs have been implemented in the shell, as well as high level matrix and vector operations, which allow users from all backgrounds to easily write shell script procedures to perform the intended task or analysis that the user desires. Since C is ubiquitous in industry and academia, the practicing engineer can easily switch to this SOFTWARE system without switching gears to a completely different type of language. Included in the internal functions are a extensive set of plotting routines that allows the user to visualize data in either 2D or 3D modes, and with a wide variety of options that allow plots to be customized to the users desire. All plotted output to the display monitor may also be output to a suitable printer for a permanent record. The user can also print variables and format ASCII text strings to a file during the running of the shell script either for a permanent record, or as a verification of computed results for a given procedure. Also included within the internal functions, are file I/O functions that allow the user to open up external disk files for reading and/or writing of data in a wide variety of formats. During the running of a given shell script, the user can open up several plotting windows that are output to one's monitor, and then use the pause function to examine the output. This allows the user the ability to continue the shell or terminate the procedure based on examining the output. Also, there are visual hot keys that provide the user with simplifying shortcuts to examine the printed output file or quickly get back to the shell script file to make a quick modification and then rerun the shell. Slide-Rule has been designed to be user friendly. It even comes with a custom editor that is seamlessly integrated into the system. When a given line of code is in error, you're immediately given an error message, and then put into the custom editor at that line of code. The user can also edit and modify all displayed plot(s) using the Customize dialog procedures as will be explained in detail in Chapter 6. This allows the user to modify the plot(s) by changing colors, line styles, titles, etc., with the click of his mouse and or keyboard input into a dialog box. This not only updates the display immediately, but updates the code as well. The user is freed from the boring tedious task of coding additional lines of code which can be time consuming and frustrating.

## System Design Criteria

The following is the system design criteria that went into the design and motivation for developing this engineering design tool.

- The shell script grammar will be a subset of the C programming language.
- Added to the above requirement, will be added matrix and vector types, both in real and complex forms, to allow programming at a high level.
- The complier will be single pass, and produce 'tree' code, that will be executed in an interpretive manner.
- Because of a single pass compiler, the code will be required to be ordered as follows: Global constants; Global variables; Subroutine code; Global variable and constants (for main code); Main Code.
- The compiler will be developed using a C++ version of YACC/LEX.
- Subroutine code will be able to be loaded and executed with a \#include <"..."> directive.
- A system parameter will exist to specify the sub-directories to look for the subroutine code if not found in the current running sub-directory.
- A file with an extension of *.h will exist to specify multiple subroutine code files which will have the extension of *.txt.
- The variable types will consist of 'int' or 'long' (32-bit signed integers); 'float' or 'double' (IEEE 64-bit double precision); 'complex' (two signed doubles to form a complex number); 'vector' (a collection of 64-bit double precision numbers); 'matrix' (a two dimensional collection of doubles); 'veccmplx' (a complex vector type); 'matcmplx' (a complex matrix type).
- The 'printf' function as found in C, will be implemented, and the output will be written to a file with the shell script file name, but with the extension of *.prn.
- The 'sprintf' function will also be implemented as in C to format ASCII strings to a 'char' type array.
- A function Print(...) will be implemented to print vector, and matrix types both in real and complex form on a row basis with 10 numbers per row for real, and 5 for complex. This function will also do the same for integer, float, and complex arrays, or single variables. This feature is for a permanent record or as a debugging tool.
- The type 'char' will only be used for storing ASCII string data.
- All shell scripts when executed, will have a *.prn file generated, with the run time in seconds with a resolution of milliseconds printed last.
- If no plot(s) are produced as output from a given shell script execution, then the print file will be displayed using the Windows ${ }^{\circledR}$ WordPad program. Otherwise the first plot produced will be displayed to take up most of one's monitor screen.
- The top plot windows bar will display the current plot number and the total number of plot windows produced in the shell script run.
- The tab and shift+tab keys will be used to display the next plot window or the previous plot window. For multiple plots produced in a given shell script, the user will be able to display multiple combinations up to a total of 10. Refer to Multiple Windows Displayed at the end of Chapter 6 - Plotting Details.
- All variables used in the shell script, will have to be specified in a declaration statement before use in an expression, else a compile error will be generated with an appropriate error message, and the custom editor will be executed to display the code, and high lite the offending line of code. This will include any subroutine code that was loaded using the \#include directive.
- All variables used in the shell script, once declared as a certain type, can NEVER be changed to a different type.
- All array type declarations of int, long, float, double, complex, vector, matrix, veccmplx, and matcmplx will have there storage initialized to zero at compile time.
- For types int, long, float, double, and complex, to be an array type, they have to have a least TWO elements. Note if you define with the * operator, the array will have FOUR elements.
- Variables of type int, long, float double, and complex, that are not arrays, are not initialized at compile time, and if not first set to a value with an assignment statement (=), will generate a run type error if detected in an expression.
- All array variables that over run their declared storage during run time will generate a run type error.
- For types of vector, matrix, veccmplx, and matcmplx that are declared with declaration statements, but with no storage specifier, they will have an internal storage of one, and be initialized to zero. Its done this way since if one forgets to allocate storage before use, one gets an exception 11 error, which takes one to no mans land, and there's no way to trace where the error occurred.
- For matrix types, referencing a specific element with double brackets ([] []) will return a float (or complex) type in an expression; referencing with a single bracket $\{[]\}$, will return a row vector (or veccmplx) type in an expression; while referencing with a single bracket ([]) with a preceding colon (:), will return a column vector type. Note the value returned is a temporary, and does NOT violate the principle that a declared variable can never be changed.
- For vector types, referencing with a single bracket ([]), returns a float (or complex) type in an expression.
- In an expression involving matrix types, if the resulting expression results in a $1 \times 1$ matrix dimension, the expression will be turned into a temporary float or complex variable. This also shall hold true for vector expressions.
- All subroutine shell script code, must have type specifers as in C, and a return type of either 'void', 'int' or 'long', float' or 'double', 'complex', 'vector', 'matrix', 'veccmplx', or 'matcmplx'. A run type error will be produced if there is not a perfect match in type specifiers or number of parameters passed in calling a subroutine.
- As in C++, single variables of type int or long, float or double, or complex can be passed by reference to a subroutine that has a type \& reference operator, and have the variable modified at global scope.
- The variable names declared in a subroutine will have local scope, i.e., their private.
- There will be no stack implemented in subroutine code as in C. This means the code in not recursive. The variables declared in the subroutine prototype declaration statement, along with the other internal variable declarations, are all static variables, which mean that they have private storage. This means that if one declares a variable in a declaration statement with a value, that value will be set in the compiler, and not re-done each time the subroutine is called. We make this choice to simplify the compiler and increase execution speed. So if one wants a variable initialized to a certain value each time the subroutine is called, code with a declaration statement along with an additional assignment statement. Note that all arrays passed to a subroutine in the calling sequence, if modified in the subroutine code, will be modified at global scope.
- We don't allow the star operator $\left(^{*}\right)$ to de-reference an array type as in C. One must use brackets ([ ]). We don't need as an example, to de-reference an array of pointers, etc., as this can be quite confusing to engineers and non-professional programmers.
- We don't allow assignment statements to be imbedded in other statements. For example, in an 'if' statement, if one writes 'if(avar = 1)', the user will get a compiler error. However, this statement is allowed in ANSI C code. The user meant to write it as 'if(avar == 1)', but no error is reported in C. This has caused at lot of grief in C coded systems, because the ' $=$ ' key wasn't depressed hard enough, and the programmer can spent hours, or days looking for the error, because the statement as coded, always produces a TRUE result.
- For displayed plots, customize features will be implemented, to allow the user to annotate the current displayed plot with their mouse and keyboard, and at the same time update the shell
script code. A detailed description of these features can be found in Chapter 6-Plotting Details. This is highly important, because we don't want to turn engineers into programmers.
- A custom editor will be written that will have all the features of a cut and paste editor, as well as line editing capabilities such as the Unix vi editor. A series of 'hot keys' (function keys) will be employed to lessen key strokes. The custom editor is described in detail in Chapter 5.
- A series of 'hot keys' (function keys) will be implemented to get to the: custom editor to display the current selected shell script; the print output file; the command prompt (DOS); the windows calculator program; the GUI reference for the Digital Filter design tool; and the function reference documentation of the $500+$ functions as listed in Appendix A. These 'hot keys' will be displayed in red on the bottom windows bar of the Slide-Rule graphic as shown in Figure2.1 of Chapter 2.
- A complete functions reference will be available from the SOFTWARE both in the main program and the custom editor by depressing function key F7. Each function that comes with the software will have a one sentence description of the function, a prototype(s) description, an example if appropriate, and references as necessary. One won't need an internet connection to determine the description of a particular function as listed in Appendix A.
- A menu item will be available for the user to detect the latest version of the SOFTWARE at a WEB site, with the ability to download and install the new version of the SOFTWARE.
- An attempt will be made to detect every possible grammar error in the compiler and in the run time interpreter, and transfer the user to the custom editor with an error message (hopefully at the offending line of code). Note that the grammar is described in detail in Chapters 3 and 4.
- The user is urged not to consult a C programming manual, as it will mislead the user. The shell grammar has certain features like C, but is not an implementation of ANSI C. I could have made up a different shell script, but the idea is to lessen the learning curve for using this tool.
- For certain functions such as the Parks McCellan FIR filter design module, the Matrix/Vector Computations, or the Special Mathematical Functions, as described in Chapters 8, 14, and 15, we use code downloaded from Netlib, and modify as necessary. The bottom line, 'We don't try to re-invent the Wheel".
- The system will use a very small amount of 'wrapper software'. All the plotting functions, dialog functions, etc., will be programmed in the WIN32 API. The reason for this is because the WIN32 API is kernel code, is well documented and NEVER CHANGES from one version of Windows to the next. Wrapper software is software that resides outside of the Windows NT OS, and is subject to constant revision from one software version of Windows to the next, which means a high maintenance situation. Another reason is it means a small foot print in terms of memory, which means faster execution times. When 'wrapper software' is used to build a windows application, one gets a lot of code linked in that is never used.
- All 'tree' code generated will be generated with C++ constructors with operator 'new', and once the code has been executed, will be deleted with C++ operator 'delete. This code deletion or memory releasing is only done for executable code in the main section. Subroutine code and global variables have to stick around until the shell script is finished executing, at which point, all the code and variables are released to the OS with C++ operator 'delete'.
- A given subroutine function can have the same name but have difference prototypes, depending on return type, as long as each prototype has the same number of input parameters. These prototypes will be found in the Function Reference.
- The Smith Chart design tool will be driven by mouse and keyboard input along with dialog box presentations for parameter input. The tool will generate shell script code which is complied and executed, which means that the design secession can be re-run and modified at a later date. For this tool, we require system parameters such as line widths, line color, etc., to be set at the system level.


## Yace/Lex Technology

The compiler was written from scratch using a C++ version of YACC and LEX. YACC is a special tool to write a compiler while LEX is a lexical analyzer, and returns tokens to the YACC system. After completing both files (*.y and *.l), the files are compiled with a YACC/LEX C++ compiler to generate C++ code, which is the compiler C++ code to be compiled and linked to the other compiled C++ files in this system. If the YACC/LEX compiler detects a shift/reduce error, then no output is generated, and one has to fix the YACC code. A shift /reduce error means there's a hole in the logic, i.e., the compiler detects a branch in the specification that allows for two of more paths to be taken, and each one is correct. This means the intended code cannot be generated correctly. Needless to say, for a compiler to generate the intended code correctly for the specifications as specified above, there's many illegal combinations when mixing the different variable types as well as the other specifications. In essence, the compiler is a complex piece of logic, and the interpreter to execute the 'tree' code is even more complex.

## System Requirements

Slide-Rule ${ }^{\text {TM }}$ for Microsoft ${ }^{\circledR}$ Windows is a 32 -bit NT program and requires a PC-compatible system with a processor capable of running Microsoft Windows in 386 -enhanced mode. You should have 20 megabytes of additional hard disk storage to install and run this software. We also suggest that you run your monitor under Windows in its native resolution to get the full detail of plotted output. This SOFTWARE runs on Windows Xp, Windows Vista, Windows 7 (64-bit version), Windows 8, and Windows 10.

## References

[1] Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Second Edition, PrenticeHall, 1988, 1978.
[2] Tony Mason and Doug Brown. lex \& yacc, O'Reilly \& Associates, Inc., 1990.

## Chapter 2 - A Short Tutorial

## Introduction

This Chapter provides a quick start, and step-by-step tutorials that takes you through several examples. This is a fast run through to give a brief overview of the Software. Later chapters will describe in detail or by examples the capabilities of this Software.

## To Install the Software

Double click on file Slide-Rule Setup on the CD disk. Follow the directions as presented. At this point, we recommend that you jump ahead to Chapter 3 and read the first two topics to get an overview of the system, before starting the Quick Start section.

## Quick Start

Having installed the software, double click on the Slide-Rule icon. Then click on the $\underline{\text { Shell/Execute }}$ Script File... menu tile and observe a dialog procedure to select a shell script file. Navigate to the plots sub-directory, and then double click on the symbols shell script file. Note that the basic display changes to Slide Rule Shell and that the cursor changes to an hour glass display. In a (very) short time you should notice that your monitor is being drawn with a 2D view of the various values for the Incomplete Beta function. When the drawing is complete, please note that the hour glass cursor changes to the normal arrow cursor. Referring to the block diagram of Figure 3.1, we note that we selected a shell script file (i.e., code) to be input into the Shell Script Compiler/Interpreter, with the resultant plotted output to the display monitor. Now pull down the File menu. This selection allows the user to do the following.

- Exit the shell by selecting the first tile or depressing the Alt+F4 'hot' keys.
- Output the current selected plot window to your favorite printer for a permanent hard copy record.
- Select and/or setup a printer for hard copy output.
- Select the page layout for printer output.
- Generate an extended metafile output for picture import into a word processor such as Word for Windows.
- Display the version number by clicking on the About Tile.

Now, select the Exit procedure with your mouse or depress the Alt +F 4 combo to get back to the main window. Please note that the Alt +F 4 combo is the universal terminate combination to terminate any program within the Slide-Rule program or the Windows operating system.

Having done this, you should now see on the bottom of the screen the Hot Key Bar as shown below,


Figure 2.1
Please note that the hot keys are in red. Note that the F9 hotkey for a Command Prompt. Now depress the function key F10. You should see the symbols.txt shell script come up in a custom windows type editor program. This allows you to quickly edit the shell script. Now, exit the editor (Alt+F4), and depress the F12 function key. This should bring up the symbols.prn file. The F12 hotkey is wired to the Windows program WordPad to display printed output from Slide-Rule. This file is the standard output file as pertains to the printf and Print statement as described in Chapter 3. Exit this program (Alt+F4) and depress the F11 function key. This will immediately begin execution of the symbols.txt file again. Note that at this point while observing the plotted output, that the F10 and F12 hot keys are wired to the Custom Editor (for editing the shell script input file), and the WordPad program for displaying the printed output file. At this point, terminate the shell (Alt+F4), and click on the Shell/Execute Script File... menu tiles on the top menu bar. You should be in the plots directory. Navigate to the digitalf subdirectory and double click on the butter or select and depress the enter key. At this point, the system is back in the shell script execution program, but executing a different shell script. Right away you should see a pop up dialog box. Now, imagining that this is your simulation, and we look at the plotted data and decide that we want to terminate the shell script. Click on the Cancel button and observe that were in the Custom Editor for this shell script. Note the pause("Magnitude dB Display"); statement after the pCRT function call (at line 119). Depress the F12 function key to look at the butter.prn output file. Terminate this display (Alt+F4), and depress the F11 function key to re-execute this shell script. Again notice the pop up menu as before. This time, just hit the Enter key to continue on with the execution of this particular shell script. Note that during execution, we observe different plot windows being displayed by how the Caption Bar changes.

Now, pull down the Windows bar and you should see a menu selection. Note that this particular shell script has seven plot windows which are displayed on the Menu-bar after the Windows Tile. Now, depress the Tab key (or Shift+Tab keys) to cycle through the seven plots. Now, depress the F12 hotkey and note that WordPad comes up and displays the butter.prn output file for this particular shell script. This allows one to examine the standard output while in the shell. When you're done viewing the butter.prn file, terminate with the Alt+F4 combo.

Now, pull down the Windows menu and select the Cascade selection. Note that all of the plotting windows are displayed in a reduced cascade format. Again go the Windows menu and select the Tile selection. Note how all seven on the plots are displayed in a tile format. You can display these seven windows in a number of reduced formats as displayed under the Windows Tile. Now try printing a couple of these windows. Use the Printer Setup under the File menu to select or setup your printer. Try printing one window in portrait mode, and another in landscape mode ( $\mathrm{Ctrl}+\mathrm{P}$ ). Note that under the File Menu, that the printer you're hooked up with is displayed (as INFO).

## Demo Files

Note from Figure 2.1, that after the Help Tile, there exist several tiles, namely Dialogs, Plotting, FilterDesign, Math, DiffEQU, SpecAnalysis, Special, ControlSystems, AdaptiveFilter, Matrix/Vector, RFmicro, and FEC/OFDM These tiles are generated from a user menu file (Demos.mnu) during startup as will be described in Chapter 7. Using your mouse, you can select a given demo file under a menu-tile with your mouse to execute. This demo system ( 269 shell scripts), allows one to quickly determine the capability of this SOFTWARE package. Note: If one depresses the Ctrl+F11 keys after the shell file runs (and plot(s) displayed), there is an info display on some of these files.

## File New...File...

Under the File Menu tile, the New... File... dialog procedure allows the user to start a new shell script file. This dialog procedure is shown below. The user should navigate to the sub-directory where the new shell script is to be stored.


Figure 2.3
The user just fills in the file name in the File name entry dialog box entry. The extension '.txt' is not needed as the SOFTWARE will fill it in for you. Note that new files cannot be created in the root directory ( $\mathrm{c}:$ \SlideRule), else a warning message. Note that when the new file is displayed in the custom editor, that the user can depress the F10 function key, to select a file that he wants to grab some lines of code from. This will be explained in detail in Custom Editor as described in Chapter 5.

## (INFO) Script Directory = ( ... )

Under the File Menu tile, this entry displays the current sub-directory. This entry should always be checked when the user is designing digital filters under the Filter tile on the main menu, since shell script files (*.txt), print file files (*.prn), coefficients files (*.fir, *.iir, *.itp), and coefficient generation files (*.out) will be stored in that sub-directory. This also holds true if the user is using the Smith Chart tool for microwave transistor amplifier analysis and design. The user
should always check this tile to determine the current sub-directory. Note, if not the correct subdirectory, go to the Change Directory... tile.

## Change Directory...

If not in the root directory ( $\mathrm{c}:$ \SlideRule), navigate to that directory. Then navigate to the directory of choice, and select any file in the directory, and note the following message.

```
Shell Script Path Directory is
C:\SideRule\"..."
```


## File New...Folder...

Under the File Menu tile, the New... Folder... dialog procedure allows the user to start a new folder for storing their shell script files. It is recommended that user NOT create sub-directories more that one level deep from the main directory (c:SSlideRule), since the File Find... procedure only searches for a file $\mathbf{1}$ directory down from the root directory. This dialog procedure is shown below.


Figure 2.4

## File Find File...

Under the File Menu tile, the Find File... dialog procedure allows the user to find a file ONE directory down from the root directory (c:\SlideRule). This dialog procedure is shown below. Note that the extension for shell script file is $t x t$.


Figure 2.5

## Config - Print Line Count

Under the Config Menu tile, the Print Line Count... dialog procedure allows the user to set the maximum print output line count, before the shell is terminated. This prevents the user's disk from being overrun should the user get into an infinite loop that contains a print type statement. This dialog procedure is shown below. Note that if one gets in an infinite loop, hit Ctrl+Alt+Del, and terminate program sigshellp.exe under the process tab in the Windows Task Manager.


Figure 2.6

## Config - Print() Threshold

Under the Config Menu tile, the Print() Threshold... dialog procedure allows the user to set the absolute value for elements printed with the internal function Print(...). This function is used to print matrices, vectors, or arrays, and entry elements. Entries with absolute values less then this threshold are printed as a zero. A more through discussion of this function can be found in Chapter 3. This dialog procedure is shown below.


Figure 2.7

## Config - Include Path

Under the Config Menu tile, the Include Path... dialog procedure allows the user to set directory paths where user subroutines are stored. When the \#include directive is used to load user subroutine code, the software first looks in the current directory, then searches the directories that are defined in this specification. A more through discussion of the include directive can be found in Chapter 3. This dialog procedure is shown below. It is recommended that the user store all of their shell scripts including subroutines that are loaded with the include directive in the same sub-directory, in which case this procedure will not be needed.

Enter Include Path(s) (Seperate with ':') erulehsubs;c:'sliderule


Cancel

Figure 2.8

## Config - User Menu Selection...

A through discussion of the User Menus Selection.... dialog procedure can be found in Chapter 3.

## Config - Config Display YES/NO

Because of the many different display resolutions and monitor physical sizes, the Slide-Rule graphic might not appeal to the user. To change, first move the graphic if desired by grabbing the Caption Bar and moving your mouse. Then resize if desired by grabbing a corner. Then check the Display YES/NO tile under the Config tile. After terminating SlideRule and restarting, note that the program graphic comes up in the same position and size as before.

## Config - Interrupt on Overflow Yes/NO

When the Slide-Rule program is executed by double clicking on the Slide-Rule graphic (on one's desktop), the floating point overflow interrupt condition is ignored. However, by clicking on the Interrupt on Overflow Yes/No tile, then this interrupt becomes enabled. Refer to Arithmetic Conditions in Chapter 3.

## Config - Interrupt on Division by Zero Yes/NO

When the Slide-Rule program is executed by double clicking on the Slide-Rule graphic (on one's desktop), the floating point division interrupt condition is enabled. However, by clicking on the Interrupt on Division by zero Yes/No tile, then this interrupt becomes disabled. Refer to Arithmetic Conditions in Chapter 3.

## Config - Print Un-Referenced Variables YES/NO

Checking this tile with your mouse will cause all un-referenced variables in the shell script to be printed out at the end of the run on the print file (<*.prn), except those in code that has been compiled with the \#include directive. This feature allows one to cleanup some un-necessary clutter after a given shell script has been checked out. Un-check this feature if one desires not to use this feature.

## Config - User Name Bold YES/NO

Under the Menu tile, the User name Bold YES/NO tile allows the user to have their name on the SlideRule graphic either in bold or regular displayed output. For eastern languages such as Chinese, Japanese, Korean, etc., the regular output allows more detail in the complex character symbols generated. By selecting this tile, the user can switch back and forth.

## Config - Smith Chart Parameters

The entries under this tile allow the adjusting of parameters in the Smith Chart design tool. The entries are as follows:

- Smith Chart Line or Symbol Size...
- VSWR Circle...
- Circuit Q Contour...
- Constant Gain Circle...
- Stability Circles...
- Radial Line...
- Noise Circle...
- Impedance Point...
- Admittance Point...
- Smith Chart Line or Symbol Color...
- VSWR Circle...
- Circuit Q Contour...
- Constant Gain Circle...
- Stability Circles...
- Conj Match Circle...
- Radial Line...
- Noise Circle...
- Impedance Point...
- Admittance Point...
- Smith Chart Curve Fit Tolerance...
- Impedance Match Data ON Impedance Plots NO/YES
- Delta Imag Data ON Impedance Plots NO/YES


## Filter

The entries under this tile allow the user to design digital filters as explained in Chapter 8.

## Shell

The entries under this tile are as follows:

- Execute Script File...

This tile entry allows the user to navigate to any directory (under the root directory), select any shell script file (*.txt), and execute it by double clicking on it or selecting it and entering the Enter key.

- Open Script File...

This tile entry allows the user to navigate to any directory, select any shell script file (*.txt), and by double clicking on it or selecting it and entering the Enter key, launch the file into the custom editor as described in Chapter 5.

- Open Script File(HOT BAR) F11

If a given shell script file is on the Hot Bar with F11 in red, then by depressing function key F11, the file will be launched into to shell script compiler/interpreter for execution.

- Edit Script File(HOT BAR) f10

If a given shell script file is on the Hot Bar with F10 in red, then by depressing function key F10, the file will be launched into the custom editor for editing.

## Help

The entries under this tile are as follows:

- Function Reference F7

This entry take one to the Functions Reference menu (similar to that show in Appendix A), whereby the user can look up a given function in terms of prototype specification; a description of input and output parameters; and some documentation and references as necessary.

- GUI Reference F8

This tile is a description of the menu items on the task bar.

- README...

Displays any pertinent info on the current release of the software.

- T Order...

Transfers to WEB Site

- About Slide-Rule

Displays the current version number and a copyright notice.

## Chapter 3 - The Shell Script Grammar

## Introduction

As stated previously, the Shell Script language has the look and feel of the ever popular C programming language, but in a reduced form, i.e., we only use a subset of the ANSI C standard to keep things simple and straight forward, and also to add efficiency to the Compiler/Interpreter. In addition, we add the types of vector and matrix (both in real and complex form), as objects implemented in the C++ programming language style. The result is that we have a shell script language with a low level type C grammar, as well as a higher level system using vector and matrix objects. In addition, since the shell script grammar is similar to the C programming language, it makes it must easier for the engineer to learn this software system since C is ubiquitous throughout much of industry and academia.

Referring to the block diagram of the system in Figure 3.1, we note that every shell script is a file stored on disk before it is executed, and that the execution of this file is first compiled into tree code, and then the tree code is executed in an interpretive manner. This allows the user to quickly change a given statement or statements and rerun the shell script without having to compile and link the code as in a normal computing situation. If the user calls many of the supplied internal functions, the user can get quite acceptable performance since the time consuming procedures are not running in an interpretive fashion.

Since the shell script language is similar in nature to the C and $\mathrm{C}++$ programming languages, this chapter is organized on a per topic basis assuming the reader has prior experience with C and/or $\mathrm{C}++$. If that is the case, you can quickly scan this chapter to learn the similarities, the differences, and the do's and don'ts of programming shell scripts in Slide-Rule. For programming at a higher level using the matrix/vector types, the user is referred to Chapter 4, Matrix/Vector Grammar, and Chapter 14, Matrix/Vector Computations. This chapter follows with the basic description of the shell script language.

## The Shell Script Language and Grammar

The Shell Script language is heavily typed, i.e., all variables must be defined before using in statements or expressions, and every variable must have an assigned type as will be described latter under Assignment Types. Once a given variable is assigned a given type, it cannot be dynamical changed to a different type, nor can it be re-defined as a different type. User written subroutines must have a prototype specification both in the input parameters and the return type. This also applies to the internal functions that come with this software. We note that several of the mathematical functions as well as other functions are overloaded, i.e., they have more than one prototype specification. The correct module to compute the intended result is determined at compile time, and the user, by matching the prototype specification exactly (both in input parameter type and number of parameters), will be guaranteed to call the correct module. As in C and C++, the Shell Script is statically typed.

## Code Layout

The Compiler/Interpreter is a single pass system. This necessitates that we order the code as shown in Figure 3.2. The first block of code (1) should consist of defined symbolic constants. The next block (2) should include global data definition statements if these variables are to be referenced by any of the user's subroutines. Normally these statements wouldn't be placed here since in general it's not a good idea to have subroutines referencing global data. Next (3) comes the user subroutines either coded in the shell script file (to be run) or included with the \#include directive. Block (4) includes the global variables that the main program references. We note that in C and $\mathrm{C}++$, that all variables referenced by the software must first be defined, else an error condition. Since the Compiler/Interpreter is single pass, we note that these variables in block (4) cannot be referenced by the user supplied subroutines in block (3). Finally, block (5) is the main code, and is the starting point of program execution. Refer to Figure 3.3 for an example code segment.


Figure 3.1 - Block Diagram
(1) Constant Statements
(2) Global Statements
(3) Include Files
(4) Global Statements
(5) Main Code

Figure 3.2 - Code layout

| Bloc | Code |
| :---: | :---: |
| 1 | const int GSIZE = 61; |
| 2 | float zarray[GSIZE][GSIZE]; |
| 3 | $\begin{aligned} & \text { \#include "hillsub.txt" } \\ & \text { \#include "hillsubs.txt" } \end{aligned}$ |
| 4 | ```float xmax,xmin,ymax,ymin,zmax,zmin; int i,j,k; int type; int fd;``` |
| 5 | ```hills(GSIZE,GSIZE,xmin,xmax,ymin,ymax,zmin,zmax); printf("xmin = %f\n", xmin); printf("xmax = %f\n", xmax); printf("ymin = %f\n", ymin); print("ymax = %f\n", ymax); printf("zmin = %f\n", zmin); printf("zmax = %f\n", zmax); openPlot("HILLS.TXT Example"); colorBar(10); surwir3D(1); conlab3D(8,1,1); colorB(207,207,207); xLabel("RANGE"); yLabel("CROSS RANGE",1); zLabel("RESPONSE"); camloc3D(266.7,140.0,25.0); confil3D(45); surfil3D(45); opts3D(0,0,0); consurf3D(3,GSIZE,GSIZE,zarray,xmin,xmax,ymin,ymax); pCRT();``` |

Figure 3.3 Example Code Layout

## White Space

White space may be used as necessary to make the shell script more readable. Please note that tabs ('\t'), spaces (' '), and new line ('\n') characters count as white space when the shell script is parsed into lexical elements. As an example,

```
float xy123, z1,z2;
```

and

```
float xy123,z1, z2;
```

produce the same results;

## Comments

Comments come in two flavors. The first begins with the "/*" characters and ends with the "*/" characters. Please note that this type of comment may not be nested. The second type of comment begins with the characters "//" and extends to the end of the line, i.e., a string of ASCII characters terminated by a new line character (' $\backslash n$ '). Examples of comments follow:

```
/* This is a Comment */
float xy123, z1,z2; // This is another comment
```


## Pseudo Ops \#define, \#ifdef, \#else, \#ifndef, and \#endif

The pseudo ops as listed above are implemented for conditional code inclusion as standard in C-type code. The \#if expression, and \#elif expression are not implemented because the \#define variable, \#ifdef varaible, \#ifndef variable, \#else, and \#endif pseudo ops meet the requirement for conditional code inclusion and a related goal of keeping the shell script language as simple and straight forward as possible. An example of the \#ifdef pseudo op for code inclusion is as follows:

```
#define DEBUG
#ifdef DEBUG
printf("xmin = %f\n", xmin);
printf("xmax = %f\n", xmax);
#endif
```

If the \#define $D E B U G$ was not included as shown above, then the two printf statements would not be complied and executed. However, if the \#define DEBUG was not included, and if the user instead used the \#ifndef DEBUG pseudo op, then the two printf statements would be complied and executed. Please note that the \#define pseudo op is not a macro for text replacement as in a standard C-compiler that uses a pre-processor, but can only be used as described above for code inclusion. Also, the \#endif pseudo op must be included for each \#ifdef or \#ifndef, else a compiler error is generated. Note that \#ifdef/\#endif statements may be nested to a level of 2 .

## Keywords

The keywords include all of the internal function names as listed in Appendix A, and the list as given below. These keywords are reserved, and may be only used for their intended purpose. Please note that all keywords and variable names are case sensitive. Keyword identifiers follow:

| \#define | \#ifdef | \#ifndef | \#endif | \#else |  |
| :--- | :--- | :--- | :--- | :--- | :--- |
| int | long | float | double | complex | char |
| matrix | vector | matcmplx | veccmplx | void |  |
| const | pi | pi2 | EPS | MAX | MIN |
| Print | echof | printf | Sizeof | rowSize | colSize |
| if | else | for | continue | while | do |
| break | switch | case | default | return | sprintf |
| open | close | read | write | skip |  |
| Print | PrintX | Printo | PrintB | new | delete |

Please note that if you use a keyword unintentionally, you will normally receive the error message Syntax error, and be transferred to the custom editor at the offending line of code. If your scratching your head on this, just enter the keyboard combo Ctrl+F12 with the cursor on the offending line of code to get a pop-up dialog box listing all the reserved words on that line of code.

## Variable Names

All variables used throughout the shell script must first have been defined before they are referenced. As in the C-programming language, every variable must start with one of the 26 letters of the alphabet (either upper or lower case) or the underscore ( $\quad$ ). The rest of the name may include the alphabet characters (upper or lower case), the ten digits 0 through 9 , and the under score character. Variable names may be up to 32 characters in length.

## Uniqueness and Scope

Although variable identifier names are arbitrary as defined within the rules above, duplicate names are illegal if they share the same scope. For all user written subroutines, the variables used in the calling sequence and those locally defined share only local scope, i.e., all other subroutines and the main code and its variables (as located in blocks (4) \& (5) as shown in Figures 3.2 and 3.3) have no access to these variables. Also, as defined above, all user written subroutines have no access to global variables which are located in block (4) as shown in Figures 3.2 and 3.3. Users should note that if global variables are defined in block (2), that user written subroutines cannot use those variable names as local variables, since those subroutines have access to those global variables. However, as described previously, it's not a good programming practice to have subroutines accessing global variables. If using the \#include directive to include a subroutine code, and the file contains global variables, then those global variables are automatically converted to static variables, i.e., only the subroutine(s) within the include file have access to these variables. This methodology makes for efficient and faster execution.

## Integer Constants

Integer constants can be decimal, octal, or hexadecimal, and follow the normal naming conventions. Please note that all integer constants are signed 32-bit quantities within the shell. Unsigned quantities don't exist. Hint! For decimal constants, don't proceed with a zero, else the compiler will convert to an
octal constant. To declare a symbolic integer constant which follows the naming conventions as described under Variable names, enter as in the following examples.

```
const int GRID_SIZE = 61;
const long ARRAY_SIZE = GRID_SIZE * GRID_SIZE;
```

For assignment statements, we have the following examples.

```
isize = 1024; // set isize to 1024 decimal
jsize = 0x400; // set jsize to 400 hexidecimal
zsize = 01000; // set zsize to 1000 octal
```

Note that you can use the keyword long in place of int as shown above. This allows users who have a piece of C-code that they want to import into Slide-Rule to omit the task of converting all 16 -bit short declarations to longs. Long integers work just as good as short integers. Note! If you try to modify a variable as defined with the const directive, you will receive a run-time error.

## Floating Point Constants

Floating point constants include the decimal point and an optional + or - sign. You may also include scientific notation such as:

$$
\text { 12.345E-09 or } 456.234 \mathrm{e}+23
$$

Please note that all constants are stored internally as 64-bit IEEE 754 floating point numbers. To declare a symbolic floating constant which follows the naming conventions as described under Variable names, enter as in the following example.

```
const float PI = 3.14159;
const double PI2 = 2. * PI;
```

For assignment statements, we have the following examples.

```
xvar = 3.14159/2.;
yvar = 2.63456E-9;
Zvar = 1;
```

Note that you can use the keyword double in place of float as shown above for the same reasons as described above under Integer Constants.

## Complex Constants

Complex constants consist of two floating point values, i.e., a real part and an imaginary part. They are stored internally as two 64-bit IEEE 754 floating point numbers. The letter j is used to define the imaginary part of the complex number. Examples of use are as follows:

```
Z1 = 3.5 + 3j;
Zvar = -1j;
Zvar = 1 + 1j;
```

Note in the second and third examples that we need to include the number 1, else the compiler thinks the second part is the variable $j$, and not the imaginary part of a complex number. So for the imaginary part
of a complex number, the letter $\mathbf{j}$ must always be proceeded by an integer or float point literal. As an example, in defining a complex vector with the statement,

```
veccmplx J[] = { -2+2*sqrt(3)j, -2-2*sqrt(3)j, -6 };
```

the user will receive a run-time error. The correct way to write this statement is,

```
veccmplx J[] = { -2 + 2j*sqrt(3), -2 -2j*sqrt(3), -6 };
```

This can also be written as:

```
veccmplx J;
J = { -2 + 2j*sqrt(3), -2 -2j*sqrt(3), -6 }
```


## Character Literals

A Character literal may be defined by using single quotation marks around a single character. The permissible characters are a-zA-Z0-9. An example might be,

```
char type;
type = 'F';
```

This type of variable is handy to use in switch statements, as they allow the user to remember a given case by the letter. This variable type is compiled as an integer, but may be printed in the printf function by using the $\%$ c format specification. An example might be,

```
matrix M1 [][4] = { 1, 2, 3, 4, 5, 6, 7, 8,
                                    10, 11, 12, 13, 14, 15, 16 };
float ftemp;
char type;
switch (type) {
        case 'F':
        ftemp = normM(M1, 'F'); printf("frobenius norm = %g\n", ftemp);
        break;
        case 'I':
        ftemp = normM(M1, 'I'); printf("Infinity norm = %g\n", ftemp);
        break;
        case '1':
        ftemp = normM(M1C, '1'); printf("norm-1 = %g\n", ftemp);
        break;
}
```


## Character Strings

Character strings may be pre-defined in a statement to be used in a printf statement or as an argument to a function. An example of a character string definition might be,

```
char *label[] = {"Magnitude in dB\n" };
char *labels[] = {"Magnitude in dB\n", "Magnitude\n" };
```

You may not use variables defined with the keyword char in arithmetic operations. The keyword char is only used for character string definition. To use the first string (label) definition in a printf statement, the code looks like,

```
printf("%s\n", label[0]);
printf("%s\n", labels[0]);
printf("%s\n", labels[1]);
```

Note that the literals 0 and 1 could have been variables such as $\mathrm{i}, \mathrm{j}$. We can print these strings in a user defined subroutine as follows:

```
void abc(char * label)
{
    printf(label[0]);
}
abc(labels[0]);
abc(labels[1]);
```

The indexing within the subroutine is necessary since de-referencing a pointer is not allowed within the shell. We note that in ANSI C that the following statements are legal, however, we don't allow them in the Shell because it becomes confusing to the novice user.

```
char pattern[] = "bold";
char pattern[] = { 'b', 'o', 'l', 'd' };
```


## Assignment Types

Besides the char type as described above, there exist additional types as defined below:

| float | signed 64-bit double precision value <br> int |
| :--- | :--- |
| signed 32 bit integer |  |
| double | signed 64-bit double precision value |
| long | signed 32 bit integer |
| complex | two signed doubles to form a complex number |
| vector | a collection of floats (64-bit) to form a vector type |
| matrix | a two dimensional collection of floats to form a matrix type |
| veccmplx | a complex vector type |
| matcmplx | a complex matrix type |

Please note that variables of type int, float, complex, matrix, vector, matcmplx, and veccmplx can also be defined as single or double dimensioned arrays. Some examples might be:

```
float df[144];
int idif[64];
complex Z0[10], Z1[10];
double zarray[NGRID][NGRID];
vector V[10];
matrix A[10][10];
int i, j, k;
float range, cross_range;
```

For the types, float (double), int (long), and complex, you can mix these in arithmetic operations that make since. Normal conversion of one type to another type follows the normal conventions. For
example, a literal or variable of type float that is an R -value will be converted to an integer in an assignment statement before storing into a L-value of type int. To convert the real portion of a complex to float, use the function real(...), else you will receive a run time error. Refer to Chapter 4 for mixing matrix/vector variable types.

## Dynamic Allocation using operators new and delete

For the types of int (long), float (double), and complex, once a variable of these types has been defined, the user can dynamically reallocate the size of the array by using operator new. An example follows:

```
int len, rows, cols, idif[2];
float *dif;
len = 100; rows = 10; cols = 5;
idif = new int[len]; dif = new float[rows][cols];
```

Note that the variable array dif is initialized to a length of 2 to be considered an array. Also, the expression in brackets must evaluate to a type of int. You can also dynamically delete storage for these types by using operator delete. An example follows:

```
delete [] idif;
```

Operators new and delete operate on variable types of int (long), float (double), complex, matrix, vector, matcmplx, and veccmplx. For types of int, float, and long using operator delete, the storage size returned equals two, while for matrix, vector, matcmplx, and veccmplx, the returned storage size is one.

## The Assignment Operator

The $=$ (equal sign) separates a variable assignment ( L -value) from a R -value. It can also be used to initialize arrays of type int, float, and complex, as well as the types of vector, matrix, veccmplx, and matcmplx. Examples of array initialization follow:

```
int jtable[] = { 1, 2, 6, 8, 3, 4 };
float xy[] = {0, 23.3, 9, 45.6, 67.3, 90. };
complex C0[] = { 0, 1 + 2.3j, 3.3j, 2 + 2j, 6 };
vector vec1[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
veccmplx c0[] = { 1+2j, 2-4j, 2.2- 3j };
matrix mat1[][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
matcmplx mat2[][2] = { 1+2j, 2-1.1j, 3-4j, 5+2.1j };
int xyz = 56;
float Y13 = 23.435, Z12 = 124.876;
```

The above initializations should be self explanatory, but note that the initialization is done at compile time, and not at run time. Please note there's no need to state the size of the array as the shell will do this for you. In the case of a two dimensional matrix or a double dimensioned array of type int (long), float (double), or complex, the number given in the above example is the column dimension, and the array/matrix is initialized on a row basis, i.e., the sequential data in brackets ( $\{\ldots\}$ ) fills the rows in a sequential manner. Note that for double dimensioned arrays, that if a value is put into the first square brackets, then a run time error will be generated. Also, for double dimensioned array types, if the number of elements in the $\{\ldots\}$ isn't equal to an integral number of columns, then a run-time error will be generated. You can also initialize an array with other pre-defined arrays, variables, and function statements. Some examples of this might be,

```
float K = 5;
float a = 1.5;
vector num[] = { 0, 0, 1.2*K, 2.4*K*a, 1.2*K*a*a };
vector den[] = { .36, 1.86, 2.5+1.2*K, 1+2.4*K*a, 1.2*K*a*a };
int L, M = 40;
float alpha;
vector z29[29]; // vector z29 consists of 29 zero's
vector Hrs[] = { 1, 1, 1, 1, 1, .5, z29, .5, 1, 1, 1, 1 };
vector HrsOpt[] = { 1, 1, 1, 1, 1, .39, z29, .39, 1, 1, 1, 1 };
alpha = (M-1)/2.;
k1 = vecGen(0, 1, (M-1)/2); k2 = vecGen((M-1)/2+1, 1, M-1);
vector angH[] = { -alpha*(2*pi)/M*k1, alpha*(2*pi)/M*(M-k2) };
```

Note that the above initializations can be written only once. Writing a given variable a second time with the same syntax as shown above, will generate a complier error. Note that in Chapter 4, it will be shown how to redefine a vector/matrix or complex vector/matrix sequence.

## Arithmetic operator's

The following arithmetic operators can be used on type's int, float, and complex in expression statements. These operators can also be used on matrix/vectors types, and will be described in Chapter 4, The Matrix Vector Grammar. Operations can be freely mixed among these types that make sense. The list is as follows:

```
+ addition
- subtraction
* multiplication
/ division
^ power operator (for types float/double and complex only!!)
+= plus equals (assignment statement)
-= minus equals (assignment statement)
*= times equals (assignment statement)
/= divided by equals (assignment statement)
\wedge= power operator equals (for types float/double and complex only!!)
```

Note the following statements:

```
float a;
a = 1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1))))));
printf("a = %.12f\n", a);
```

produces the following result:

$$
a=1.000000000000
$$

The reason for this is that an integer divided by a larger integer always produces a zero. Now if we change the statement to:

```
a = 1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1.))))));
printf("a = %.12f\n", a);
```

we get the following result:

$$
a=1.615384615385
$$

Here we note that the expression is evaluated from right to left, and an integer value of 2 divided by a floating point value of 1 always produces a floating point value. In the end we wind up adding an integer value of 1 to a floating point value $\sim=7 / 13$. Note if we change the statements to:

$$
\begin{aligned}
& \mathrm{a}=1+1 . /(1+1 /(1+1 /(1+1 /(1+1 /(1+1 /(1)))))) ; \\
& \text { printf("a }=\% .12 \mathrm{f} \backslash \mathrm{n} ", \mathrm{a}) ;
\end{aligned}
$$

we get the following result:

$$
a=1.500000000000
$$

This little demonstration points out that one needs to be careful when using literal integers when in reality we want a floating result which can always be obtained by adding the decimal point, thus specifying a floating point literal. As a general rule, if one of the operands is a floating point type, then you don't need to add a decimal point to the literal. An example might be,

```
int N = 4;
float J;
J = -(2*pi)/N;
```

Since pi is a floating point constant, then we don't need to add a decimal point to the integer literal 2 since it is automatically converted to floating point as well as the integer variable N .

Note!! The power operator ${ }^{\wedge}$ is a short cut version of the pow function.

```
Var = alpha^2;
Var = pow(alpha, 2);
```

The example above shows equivalent statements. For matrix/vector operations, we use the.$^{\wedge}$ operator. Refer to topic in Chapter 4, The Matrix/Vector Grammar.

## Logical operator's

The following logical operators can be used on types int, float, and complex in an if statement. Operations can be freely mixed among these types. The list is as follows:

```
== equal to
>= greater than or equal
<= less than or equal
!= not equal
> greater than
< less than
&& logical AND catenation of two expressions
| logical OR catenation of two expressions
! logical NOT of an expression
```


## Bitwise operator's

The following bitwise operators can be used ONLY on variables of type int/long. The list is as follows: An exception for operators ${ }^{\wedge}$ and $\wedge=$ on types float/double and complex (refer to Arithmetic operator’s topic).

```
& bitwise and of two expressions
| bitwise or of two expressions
^ bitwise xor of two expressions
% modulo of two expressions
>> shift right
<< shift left
~ one's complement of an expression (unary)
&= bitwise and (assignment statement)
|= bitwise or (assignment statement)
^= bitwise xor (assignment statement)
%= modulo (assignment statement)
>>= shift right (assignment statement)
<<= shift left (assignment statement)
```

Note that for the shift operators, that these are arithmetic shifts. To implement as unsigned shifts, just use the lower bits of the integer variables you define.

## Increment/Decrement operator

The operator's ++ and -- can be used in the update statement of a for loop, and also to increment or decrement a variable by 1 . As an example,

```
int i, j;
for(i = 0; i < 25; i++) { /* this use is AOK */
    j++; /* this use is AOK */
}
```

Note that the Shell doesn't have pointer variables as in the C-programming language, so that when you use these operators, you are always either incrementing or decrementing a variable by 1 . You may use these operators on floating point variables, but the preferred method is as follows:

```
fcount += 1;
```


## The Sizeof, rowSize, colSize, rowLen, $\&$ colLen functions

The Sizeof function is used to get the size of an array of type's int (long), float (double), or complex. And example might be,

```
int j;
float buffer[300];
for(j = 0; j < Sizeof(buffer); j++)
{
```

Note that this construct differs from the C-programming language sizeof operator in that it returns the number of elements in the array and not the number of bytes. This function may also be used on type's matrix, vector, matcmplx, and veccmplx. Note that for type's matrix and matcmplx, that the return value is the number of elements in the matrix. You can also use the rowSize and colSize functions on doubled dimensioned arrays of type's int (long), float (double), and complex, as well as matrix and matcmplx types, to get the row or column dimension. Refer to the Help Menu/Functions Reference/Grammar and Control Logic. An example follows.

```
matrix M1[10][300];
int M,N;
M = rowSize(M1); N = colSize(M1);
Print(M); Print(N);
M = 10
N = 300
```

To get the row or column length, use the rowLen and colLen functions. Refer to the Help Menu/Functions Reference/Grammar and Control Logic.

## The include directive

The include directive is used to load symbolic constants or subroutine code into a shell script. Because the shell is a single pass parser, these statements should be in block (3) of the shell script file as shown in Figures 3.2 and 3.3. An example might be,

```
#include "hillsubs.txt"
#include "hillssub.txt"
```

When first developing user written subroutines, include them in the main shell file (block 3) until no compile errors are generated. If you do get a compile error in a subroutine that was included by this directive, then the shell is terminated, and the subroutine is displayed in the Custom Editor with the cursor highlighted at the offending line of code. Note that the first two statements of hillsubs.txt (which resides in sub-directory subs) are:

```
#ifndef _HILLSUBS
#define _HILLSUBS
```

And the last is:

```
#endif
```

This precludes the code being included twice with a given shell script, and is a standard C-type programming methodology. The include directive can consist of including a single file containing one of more subroutines, or one can include a file which consists of a collection of include directives. An example might be:

```
#include "control.h"
```

This file control.h might look like:

```
#include "residueS.txt"
#include "stepTF.txt"
#include "stepSS.txt"
#include "impulseTF.txt"
#include "impulseSS.txt"
#include "bodeTF.txt"
#include "TFtoPZ.txt"
#include "SStoTF.txt"
#include "TFtoSS.txt"
#include "PZtoTF.txt"
#include "PZtoSS.txt"
#include "bodeSS.txt"
#include "SStoPZ.txt"
#include "rlocusTF.txt"
#include "rlocusSS.txt"
#include "parallelTF.txt"
#include "seriesTF.txt"
#include "feedbackTF.txt"
#include "nyquistTF.txt"
#include "nyquistSS.txt"
#include "secondOrdSys.txt"
#include "LTIsysTF.txt"
#include "LTIsysSS.txt"
#include "margin.txt"
#include "nicholschart.txt"
#include "nicholspts.txt"
#include "groupDA.txt"
```

Please note that the complier compiles shell script code at about $50,000+$ lines per second!!

## The if statement

The if statement is used to control the flow of execution. Shown below is an example of the use of an if statement,

```
    if(xval <= 30.)
{
}
```

Although not necessary, one might want to get in the habit of using the $\}$ brackets even if you have only one statement between them. Note that the following assignment statement embedded in the if statement, while legal in ANSI C, generates an error in the shell.

```
if(var23 = 24) {
}
```

The user meant to write $\operatorname{if}(\operatorname{var} 23=24)$. Even the most experienced programmer makes this mistake. This can be a very hard error to find, so we don't allow it in the shell. Another example that generates a
run-time error is an assignment statements within an expression, as assignment statements embedded within an expression are not allowed within the shell.

```
if(Gp == (tmp = ellipticC(pi/2, k1)));
```


## The else statement

The else statement is used in conjunction with the if statement. An example might be,

```
float xy;
xy = 24.0;
if(xy < 22.)
    printf("xy < 22.0\n");
else if(xy <23.)
    printf("xy < 23.0\n");
else if(xy < 24.)
    printf("xy < 24.0\n");
else
    printf("xy >= 24.0\n");
```

Will produce the result to the print file (*.prn) of

```
xy >= 24.0
```


## The compact if then else expression

The if then else conditional such as

$$
\begin{aligned}
& \text { if(a > b) } \\
& \text { z = a; } \\
& \text { else } \\
& \text { z = b; }
\end{aligned}
$$

can be written in a compact way as:

```
expr1 ? expr2 : expr3
```

where expression expr1 is evaluated first. If it is non-zero (true), then expression expr2 is evaluated. Otherwise expr3 is evaluated. Thus to set $\mathbf{z}$ to the maximum of $\mathbf{a}$ or $\mathbf{b}$ (as above), or to print out the maximum of $\mathbf{a}$ or $\mathbf{b}$, we have:

```
z = (a > b) ? a : b; printf("Maximum of a or b is %d\n", z);
printf("Maximum of a or b is %d\n", (a > b) ? a : b);
```


## The for statement

The for statement is used to execute program loops, and is a generalization of the while statement. Within the parentheses, there are three parts, separated by semicolons. The first part, the initialization statement, is executed before the loop proper is entered, and allows multiple assignment statements separated by commas. The second part is the test that controls whether the loop proper will be executed.

If the condition is false, then the loop is terminated, else it is executed. The third part is executed after the loop proper is executed and before the second part is executed to re-evaluate the condition of the loop. The third part also allows for multiple assignment statements separated by commas. Note that the first and/or third parts in the parenthesis are optional (not the two semi-colons however). Note that the curly brackets (optional) are used to enclose more than one statement within the loop proper. These loops may be compounded as in the following example.

```
int i,j,k;
float array[25][50];
float x, y, del, theta;
k = 0;
theta = 0;
del = 2*pi/2500.;
for(i = 0; i < 50; i++)
{
    for(j = 0; j < 50; j++)
    {
        array[i][j] = sqrt(x*x + y*y);
        theta += del;
        x = cos(theta);
        y = sin(theta);
    }
    k += 1;
}
```

Please note that in the above example that a run time error will be generated when index i exceeds its array storage $(\mathrm{i}=25)$. Also note that the for instruction statement could have been written as:
for (i = 0, k = 0; i < 50; i++, k += 1)
thus eliminating two statement lines.

## The while statement

The while statement can also be used to execute loops. The condition statement between parentheses is evaluated first, and if true, the loop is executed. This condition is then re-evaluated, and the loop continues to execute until the condition statement becomes false. Note that the curly brackets (optional) are used to enclose more than one statement within the loop proper. The following is a typical example.

```
int i,j,k;
k = 0;
while ( k < 50)
{
    k += 1; // expression between parens cannot be assignment
}
```


## The do while statement

The do while statement can also be used to execute loops. This loop will always execute at least once before the condition statement is evaluated to determine whether the loop should be re-executed. This loop continues until the condition statement becomes false or the loop is terminated with the break statement. Note that the curly brackets are used to enclose more than one statement within the loop proper. The following is a typical example.

```
int i, j, k;
k = 0;
do
{
    k += 1;
} while(k < 50);
```


## The break statement

The break statement allows one to exit a for loop, a while loop, a do while, or a switch statement before the main loop terminates. An example might be as follows,

```
while(1)
{
    if(x >= 100.0) break;
}
```


## The continue statement

The continue statement allows one to continue in a for loop, a while loop, or a do while loop, and not execute the additional statements in the rest of the loop. An example could be as follows,

```
for(j = 0; j < 100; j++) {
    if(x >= 100.0) continue;
        // additional statements
}
```


## The switch statement

The switch statement allows one to implement a multi-path if/then else conditional in a compact manner. The switch conditional expression must be of integer type, and the values in the case statements must be integer literals. The default statement in the switch can be in any place and not necessarily at the bottom of the switch. An example of a switch statement follows:

```
ival = 5;
```

```
switch(ival)
{
case 0:
case 1:
case 2:
case 3:
case 4:
    printf("Got here for cases 0 -> 4\n");
    break;
case 5:
    printf("Got here for case 5\n");
    break;
case 6:
    printf("Got here for case 6\n");
    break;
default:
    printf("Got here for all other value of ival\n");
    break;
}
```

Note that the break statement causes an immediate exit from the switch, else we just fall through to the next case. Although the break statement is logically unnecessary for the default case as shown above, add it in case some day another case gets added at the end.

## The printf statement

The printf statement follows the normal C-standards as described in a C-programming manual. An example might be,

```
int i;
float Ax[100];
printf("Ax[%02ld] = %.4f\n", i, Ax[i]);
```

Note that this internally supplied function is the same as the standard printf found in the C-programming language. The format specifiers that this printf function supports include the following format specifications along with modifiers,
\%d, \%f, \%g, \%G, \%e, \%E, \%x, \%X, \%s, \%c

To print a complex number, we have,

```
printf("Z0 = %f %+fj\n", real(z0), imag(z0) );
```

To print an integer with a field width of 5 with blank fill, we have,

```
printf("I55 = %5d\n", I55);
```

To print an integer with a field width of 5 with zero fill, we have,

```
printf("I55 = %05d\n", I55);
```

To print a floating point number with a fractional precession of 12 , we have,
printf("Range = \%.12f\n", Range);

To print a floating point number with a fractional precession of 12 in scientific notation, we have,

```
printf("Range = %.12e\n", Range);
```

Please note that the 1 (el) modifier is optional in the printf function for integer arguments because the type int and long are one and the same in the shell, and there's no need to generate an error if a user omits the 1 (el) modifier. Each conversion specification begins with the character $\%$ and ends with the conversion character ( $\mathrm{d}, \mathrm{f}, \mathrm{g}, \mathrm{G}, \mathrm{e}, \mathrm{E}, \mathrm{x}, \mathrm{X}, \mathrm{or} \mathrm{s}$ ). Between the $\%$ and the conversion character there may be, in order:

Flags (in any order) which modify the specification:
-, which specifies left adjustment of the converted argument in its filed

+ , which specifies that the number will always be printed with a sign
space, if the first character is not a sign, a space will be prefixed
$\mathbf{0}$, for numeric conversions, specifies padding with leading zeros.
A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally a space, but 0 if the zero padding flag is present.

A period which separates the field width from the precision.
A number, the precision that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for $\mathrm{e}, \mathrm{E}$, or f conversions, or the number of significant digits for g or G conversions, or the number of digits to be printed for an integer

## Conversions for printf

## Character Argument Type Converted To:

d int/long; signed decimal.
o int/long; unsigned octal (without a leading zero).
x , X int/long; hexadecimal (without a leading 0x or 0 X ) using abcdef or ABCDEF
s char; characters form a character array of type char.
c char, a single variable of type char (output equals ASCII character). the precision. The default precision is 6 . A precision of 0 suppresses the decimal point.
e, E float/double; decimal of the form [-]m.dddddde +xx or [-]m.ddddddE +xx , where the d's is specified by the precision. The default precision is 6 . A precision of 0 suppresses the decimal point.
g , G float/double; \%e or \%E is used if the exponent is less than -4 or greater than or equal to the precision, otherwise $\% \mathrm{f}$ is used. Trailing zeros and a trailing decimal point are not printed.

## The sprintf function

The sprintf function has the same functionally of the printf function as defined above, except that formatted output is stored in a used defined character buffer. An example might be:

```
char buf[61];
sprintf(buf,"var23 = %d\n", var23);
```


## The Print , PrintX, PrintO, and PrintB Functions

The internal function Print is provided to allow the user to easily print matrix or vector types. You can also print single or double dimensioned arrays or variables of type int, float, and complex as well as single variables. The following is an example piece of code and the corresponding printed output result. Note that the PrintX, PrintO, and PrintB functions allow for printing variables or arrays of type int (long) in hex, octal, or binary format.

```
matcmplx mat1[][2] = { -3+2j, -7+9j, 1-8j, 5-4j };
matrix mat2[][3] = { 1, 4, 7, 2, 5, 8, 3, 6, 9 };
veccmplx vec1[] = {-1+1j, -1-1j };
vector vec2[]={ 4, 5, 6, 7, 8 };
float abc[][2] = { 1, 2, 3, 4 };
float F12 = 55.746358923;
Print(mat1);
Print(mat2);
Print(vec1);
Print(vec2);
Print(abc);
Print(F12);
```

produces the output:

```
matrix(cmplx) -> mat1(2,2)
[row
0000 -3 +2j
0001 1 -8j
]
matrix -> mat2(3,3)
[row
0000
0001
0002
]
vector(cmplx) -> vec1(2)
[col
0000 -1 +1j -1 
]
vector -> vec2(5)
[col
0000 4
]
array(real) -> abc(5)
[
6
7
]
F12 = 55.746358923
```

Note that under the Menu tile Config/Print() Threshold..., that the absolute value of elements less than the value under this dialog procedure, are printed as zero. The default value is $2.204 \mathrm{e}-14$, which is roughly 100 times the EPS value of $\sim 2.204 \mathrm{e}-16$. Note! For single variables of type float, the format is $\% .16 \mathrm{~g}$, while for complex it's $\% .16 \mathrm{~g} \%+.16 \mathrm{gj}$. Also, when printing the $*$.prn file using WordPad, if the matrix or vector doesn't fit in portrait mode, switch to landscape followed by cntl +a , and then change the font size to 9 . It turns out that Wordpad cannot be initialized as such.

## User written Subroutines

The user supplied subroutines allow the user to write procedure's that can be called at various places within the shell script. As described earlier, subroutines must precede the main body of the shell script statements since the shell script is a single pass parser. Each subroutine must begin with a type of void, int, float, complex, matrix, vector, matcmplx, or veccmplx, and the subroutine can only return one of these types. The type specifier is then followed by the name of the subroutine. A left and right parenthesis must then follow with an optional parameter list which must include type specifiers. The main body of statements then follow, which must be enclosed in angle brackets. A return statement should be the last statement of the procedure. If you have a void procedure, declare the function of type void and omit the return statement (unless you have an early exit). An example could be as follows:

```
int Cal_Range(float x, int ix, float *buffer, float& yscale)
{
    int i,j,k;
    float ftemp;
    for(j = 0; j < ix, j++)
    {
    yscale = ...;
    }
    k = 0;
    if(ftemp > 50.) k = 1;
    return(k);
}
```

Based upon the example, it should be fairly obvious on how to proceed in the writing of subroutines. Please note that for variable yscale, that what ever we pass as an argument (4th parameter), that that variable will be modified at global scope since that variable is being passed by reference. For variable x in the above example ( $1^{\text {st }}$ parameter), what ever we pass will not be modified at global scope, but only locally within the scope of the function since we are passing the variable by value. Please note that local variables can have the same names as global variables (since they are in a different scope), and do not modify the variable (with the same name) at the global level. Also, we can pass literals as arguments to parameter variables x and ix in the above example. An example of this might be,

```
int sFlag;
float yBuffer[1000], xScale;
sFlag = cal_Range(25, 100, yBuffer, xScale);
```

Note that for passing double dimensioned arrays of type int/long, float/double or complex, the prototype can have the following different formats as shown in the following examples.

```
int putValues(float * array){ ... }
```

```
int putValues(float array[][10]){ ... }
```

For single dimensioned arrays, we can have.

```
int putValues(float * array){ ... }
int putValues(float array[]){ ... }
```

Note that in the above example, where we specify that array has 10 columns, that the value is irrelevant, since in Slide-Rule the dimensions of the array are automatically carried into the subroutine. This is also true for types of matrix, vector, matemplx, and vecemplx. Note that you can use functions Sizeof (to return the size in type elements of an array), or rowSize/colSize (to return the row/column dimension of a double dimensioned array or matrix). Refer to the Help Menu.

Note! Local arrays within the subroutine are initially initialized to zero when compiled. However, on subsequent calls to the subroutine, these local variables won't be initialized. So if you expect these variables to be zero upon entry, you must initialize these local variables each and every time you call the subroutine. Note that single variables are not initialized, and if used in an expression without first being set to some value, then a run time error will be generated. If set to a value (as shown below for variable Nx ), then this variable is set only at compile time, and won't be reset on subsequent calls to the subroutine.

An example might be:

```
vector ABC(matrix A, matrix B)
{
    matrix y[10, 10];
    int i, N, Nx = 12;;
    y = zerosM(10, 10); // matrix y must initially be ALL zeros
    return(y[:0]); // return first column of matrix y
}
```

Note! For arrays and matrix/vector types, if these variable names, as passed to the subroutine as parameters, are modified within the subroutine, then they are MODIFIED at global scope.

Note that if matrices A and B are modified, then they are modified at global scope. If this is a problem, then you might change to the following:

```
vector ABC(matrix A, vector B,)
{
        matrix y;
        matrix a, b, c, d;
        int i, N;
        y = zerosM(10, 10); // matrix y must initially be ALL zeros
        a = A; b = B; // create local copies
    return(y[:0]);
}
```

Note that if you passing a vector to a subroutine which is a row or column of a matrix, then the row or column of the matrix is not modified at global scope, since the vector is passed as a temporary vector. An example might be:

```
matrix matB;
vector vecR;
vecR = xyz(matB[j]); // pass row j of matB to subroutine xyz
```

Note!! Locally defined variables within a subroutine are stored as static variables, and not stored on a volatile stack as in most C environments. This allows the user to define constants within the subroutine proper which are local to that particular routine.

Note!! Since the subroutine code in the shell script grammar doesn't store input parameters on a local stack, then by definition, subroutines in Slide-Rule are not recursive. It was done this way to gain speed. Subroutines or functions are written to modularize the code and make the given analysis procedure more readable.

For the novice user. Although a given subroutine can only return one value, note that all of the arguments passed to the subroutine can be modified at global scope.

- Use the $\boldsymbol{\&}$ operator on non-array variables to modify that variable at global scope, else it won't be modified at global scope. All arrays of type int, float, and complex, if modified in a subroutine, will be modified at global scope.
- All objects of type matrix, vector, matcmplx, and veccmplx, if modified within the subroutine, will be modified at global scope. To preclude this, copy to different variables as defined within the subroutine. An example was shown above in subroutine ABC.
Note!! If you have a statement at the beginning of a subroutine such as:
float $\operatorname{var2}=2 ;$
then don't expect var2 to be equal $(=)$ to 2 , if you modify it on a previous call. The code segment above sets var2 to 2 at compile time, and not when entering the subroutine from a function call. If you want it to be equal to 2 , on each and every call, the code should look like this:

```
float var2;
var2 = 2;
```

Note!! At this point its best to point out that handling arrays as vectors or matrix types is the preferred method of programming, since we can reference elements within the vector or matrix in the same manner as conventional arrays, but can handle these types at a much higher level as will be described in Chapter 4, The Matrix/Vector Grammar. Note that he shell doesn't allow the return of array types of int, float, or complex. The shell was designed this way, to encourage the user to use the matrix/vector types.

## Argument Overloading for Functions Calls

For internal and shell script functions, the following is a list of the legal argument types for the parameter types as specified in the prototype statement.

- int literal int, literal float, integer variable, floating variable
- float literal int, literal float, integer variable, floating variable
- complex literal int, literal float, integer variable, floating variable, complex literal, complex variable
- float * single or double dimensioned floating array
- complex * single or double dimensioned complex array
- matrix matrix
- vector vector
- matcmplx matcmplx
- veccmplx veccmplx

Note!! Expressions by definition are temporary, and all temporary expressions that are passed in as arguments, will not be modified at global scope. As an example, if you create a vector (or complex vector) by referencing a row or column of a matrix (or complex matrix) in a function call, then the row or column of the matrix (or complex matrix), will not be modified at global scope.

## The return statement

The return statement allows the user to return a type of int, float, complex, matrix, vector, matcmplx, or veccmplx from a user written function. For a void procedure, a return statement without a value allows an early exit from the procedure.

## Initialization of variables

All global array variables of all types have there storage initialized to zero when declared. For array types, you can access elements values in the array without first setting these array elements to some value. The value you access of course will be zero. For non-array types, you must first set the variable to some value in an assignment statement, else you will receive a run time error, and be transferred to the Custom Editor (as described in Chapter 5), with the Editor positioned on the offending line of code. An example follows:

```
float x,y,z;
y = 23.45;
z = sqrt(x*x + y*y);
```

Produces the following error message:
(line 4) Variable(x) not previously set

## Arithmetic Conditions

The arithmetic conditions that produce a run time error include the following:

- Integer division by zero.
- Invalid floating point operation ( $0 / 0,0$ INF)
- Floating point division by zero ( $\mathrm{x} / 0.0$ )
- Floating point total loss of precision.
- Float point stack overflow.

Any arithmetic operation involving a NaN .

Note that a floating point overflow condition goes to +INF or -INF, and that a non-zero numerator divided by zero goes to either +INF or -INF depending on the sign of the numerator term. Also note that integers always wrap on an overflow condition. Underflow conditions are ignored.

Other conditions that that produce a run time error include argument DOMAIN errors such as,

```
sqrt(-5)
log10(-1)
log(-20)
```

If one has a vector or matrix variable, and one or more terms are zero, just add EPS (2.2204460492503131e-16) to the vector variable.

Note!! When one double clicks on the Slide-Rule graphic, the following conditions as stated above are in effect. However, one can enable an interrupt (under the Config tile) on a floating point overflow condition while the Slide-Rule graphic program is in effect (running). The same holds true for division by zero. Note however, that operation of $0 / 0$ or $0 *$ INF will always give an invalid floating point message with an interrupt.

## The File open/close and read/write Functions

Included in the internal functions are the I/O functions open/close and read/write which allows a user to open up to 20 (disk) files for reading and/or writing. Since input/output is normally covered in a Cmanual (such as The C Programming Language by Kernighan/Ritchie), we include these function specifications here for completeness. The descriptions can also be found under the Help Menu.

| Module | open |
| :---: | :---: |
| Function | Open a file for reading and or writing. |
| Syntax | ```int open(char *filename, char *descriptor); where, filename \(=\) ASCII file including path if necessary descriptor \(=\) ASCII file descriptor = "r", Open for reading only = "w", Create for writing. If file already exists, it will be overwritten. \(=\) " b ", binary mode(note! this flag used in conjunction with " r " or "w" such as "rb" or "wb"). \(=\) " \(\mathrm{r}+\) ", Open an existing file for updating(reading and writing) \(=\) " \(w+\) ", Create a new file for updating (reading and writing).``` |

Returns a file descriptor used for a file specification in routines read, write, and close.

## Module close

Function Closes an opened file.
Syntax void close( int fileID);
where,
fileID $=$ integer file descriptor returned from file open call.

## Module read, readM, readV

Function Read in data from an opened file.
Syntax Real Version:
int read(float *buffer, int count, int type, int fileID);
Integer Version:
int read(int *buffer, int count, int type, int fileID);
Complex Version:
int read(complex *buffer, int count, int type, int fileID);
where,

> buffer = array to read into
> count = maximum number of elements in buffer to read into
> type = type of input data in file
> $=1$, signed character data
> $=2$, signed short(16-bits) integer data
> $=3$, signed long(32-bit) integer data
> $=4$, floating point data ( 32 bits, float type)
> $=5$, floating point data ( 64 bits, double type)
> $=6$, ASCII data in streams format
> fileID = integer file descriptor returned from file open call.

## Matrix Version:

int readM(matrix A, int rows, int cols, int type, int fileID);
int readM(matcmplx B, int rows, int cols, int type, int fileID);
where,

```
\(\mathrm{A}=\) real matrix(can be a NULL matrix)
B = complex matrix(input data is real complex, real complex , ...)
rows = row dimension of matrix
cols \(=\) column dimension of matrix
```


## Vector Version:

int readV(vector A, int length, int type, int fileID);
int readV(veccmplx B, int length, int type, int fileID);
where,
$A=$ real vector(can be a NULL matrix $)$
$B=$ complex vector(input data is real, real $j, \ldots)$
length = dimension of vector

Note! Each version returns number of type elements read, and ALWAYS store the return value to a variable, else a run type error!!

## Module write

Function Writes data to an opened file.
Syntax Real Versions:
int write(float *buffer, int count, int type, int fileID);
int write(vector vec, int count, int type, int fileID);
int write(matrix mat, int count, int type, int fileID);

## Complex Versions:

int write(complex *buffer, int count, int type, int fileID);
int write(veccmplx vec, int count, int type, int fileID);
int write(matcmplx mat, int count, int type, int fileID);
where,
buffer = array to write from
vec $=$ vector array to write from
mat $=$ matrix array to write from
$=1$, signed character data
$=2$, signed short(16-bits) integer data
$=3$, signed long(32-bit) integer data
$=4$, floating point data ( 32 bits, float type)
$=5$, floating point data ( 64 bits, double type)
$=6$, ASCII data in streams format
fileID $=$ integer file descriptor returned from file open call.
Returns:
number of elements written.

## Chapter 4 - The Matrix/Vector Grammar

## Introduction

This chapter presents the basic Matrix/Vectors Grammar similar to the Shell Script Grammar as presented in Chapter 3. Following that, examples of basic Matrix/Vector operations will be presented to acquaint the user with these functions. We note that in Chapter 3, the basic grammar is similar to the ANSI C language, but in a reduced form, i.e., we only use a subset of the ANSI C standard to keep things simple and straight forward. We add the types of vector and matrix (both in real and complex form), as objects, with the result is that we have a shell script language with a low level type C grammar, as well as a higher level system using vector and matrix objects. In Chapter 14, Matrix/Vector Computations, we will discuss and show examples of such topics as solving for the eigenvalues and eigenvectors of a matrix system; solving a system of linear equations; etc...

## Matrix/Vector operations with scalars

For expressions we can add (+), subtract (-), multiply ( $\left(^{*}\right)$, or divide(/) a real or complex matrix or vector by a real or complex scalar (either a variable or literal). The operation is done on an element by element basis, and the resulting matrix or vector is real if both operands are real, else complex if the left operand is complex. Note when using scalars, that for the multiply operation, we use the * operator, and not .*, which is reserved for matrix/vector operations on an element by element basis. Note that for assignment operations ( = ), that the left and right matrix or vectors can be of different sizes, since the left operand is replaced. Note that the left matrix or vector can be complex, while the right expression can be either real or complex.
Assume the following variables; Amat, a real matrix; Bmat, a real matrix (with row/column dimensions the same as Amat); Avec, a real vector; Bvec; a real vector (with length the same as Avec); Cmat, a complex matrix; ; Dmat, a complex matrix (with row/column dimensions the same as Cmat); Cvec, a complex vector; Dvec, a complex vector (with length the same as Cvec); Aconst, a real scalar (either a variable or a real literal); Cconst, a complex scalar (either a variable or a complex literal). The following are permissible scalar operations on these variables. Note that these operations are done on an element by element basis.

```
Amat = Bmat + Aconst;
Amat = Bmat - Aconst;
Amat = Bmat * Aconst;
Amat = Bmat / Aconst;
Amat += Aconst;
Amat -= Aconst;
Amat *= Aconst;
Amat /= Aconst;
Amat = Aconst;
Cmat += Aconst;
Cmat -= Aconst;
Cmat *= Aconst;
Cmat /= Aconst;
Cmat = Aconst;
Cmat = Dmat + Cconst;
Cmat = Dmat - Cconst;
Cmat = Dmat * Cconst;
Cmat = Dmat / Cconst;
Cmat = Amat + Cconst;
```

```
Avec = Bvec + Aconst;
Avec = Bvec - Aconst;
Avec = Bvec * Aconst;
Avec = Bvec / Aconst;
Avec += Aconst;
Avec -= Aconst;
Avec *= Aconst;
Avec /= Aconst;
Avec = Aconst;
Cvec += Aconst;
Cvec -= Aconst;
Cvec *= Aconst;
Cvec /= Aconst;
Cvec = Aconst;
Cvec = Dvec + Cconst;
Cvec = Dvec - Cconst;
Cvec = Dvec * Cconst;
Cvec = Dvec / Cconst;
Cvec = Avec + Cconst;
```

```
Cmat = Amat - Cconst; Cvec = Avec - Cconst;
Cmat = Amat * Cconst; Cvec = Avec * Cconst;
Cmat = Amat / Cconst; Cvec = Avec / Cconst;
Cmat += Cconst;
Cvec += Cconst;
Cmat -= Cconst;
Cvec -= Cconst;
Cmat *= Cconst;
Cmat /= Cconst;
Cvec *= Cconst;
Cvec /= Cconst;.
Cmat = Cconst; Cvec = Cconst;
matrix Amat[2][3]; // definition of Amat as a 2x4 real matrix
// and it's initialized to zero at compile time
// Xmat, a 1x1 real matrix set to zero
matrix Xmat;
matrix Bmat[2][3]; // Bmat initialized to zero at compile time
matcmplx Cmat[2][2]; // Cmat initialized to zero at compile time
Amat += 1; // AOK as defined above, matrix Amat(2\times3) all 1's
Amat = Bmat +5; // AOK as defined above, matrix Amat(2\times3) all 5's
Cmat += 2 -3j; Print(Cmat); Print(Amat);
Cmat = Amat -2; Print(Cmat);
Bmat = Amat * 2;
*****************************************************************************
matrix(cmplx) -> Cmat(2,2)
[row
[rlll
]
matrix -> Amat(2,3)
[row
0000 5 5 5
0001 5 5 5
]
matrix(cmplx) -> Cmat(2,3)
[row
[llllll
]
matrix -> Bmat(2,3)
[row
0000 10 10 10
0001 10 10
]
*************************************************************************
```


## Matrix/Vector copy operations

Matrix/Vector copy operations are performed with simple assignment statements. The receiving matrix (or vector) can be a NULL matrix (or vector), or it can have been previously defined with a different size then the new size. If not a NULL matrix (or vector), the internal storage is deleted, and then created with the new size. The variable (name) must have been previously defined, and the receiving matrix/vector must be of the same type ( matrix/vector) as the R operand. The L operand can be complex while the R operand can be real). The following are the possible combinations of this operation.

```
Avec = Bvec; // Avec a real vector, Bvec a real vector
Cvec = Bvec; // Cvec a complex vector, Bvec a real vector
Cvec = Dvec // Cvec a complex vector, Dvec a complex vector
Amat = Bmat; // Amat a real matrix, Bmat a real matrix
Cmat = Bmat; // Cmat a complex matrix, Bmat a real matrix
Cmat = Dmat; // Cmat a complex matrix, Dmat a complex matrix
```

Assuming the definitions as previously defined, we can store a real matrix/vector to a complex matrix/vector as the imaginary part by using the cmplx function. For a real matrix/vector to a complex matrix vector as the imaginary part, we have,

$$
\text { Cmat }=\operatorname{cmplx}(0, \text { Amat }) ; \quad \text { Cvec }=\operatorname{cmplx}(0, \text { Avec }) \text {; }
$$

This causes the real portion or the complex portion of Cmat and Cvec to be all zeros. Note that the zero literal can be any integer, or real number.

Or for both real and complex portions, where Amat, Bmat, Avec, and Bvec are real.

```
Cmat = cmplx(Amat, Bmat); Cvec = cmplx(Avec, Bvec);
```

Note for these last statements, that matrix and vector arguments must be of the same size and be real. Also note that arithmetic operations can take place on the operands in parenthesis. An example might be,

$$
\begin{aligned}
& x b b R=2^{*} \exp \left(c m p l x\left(0,-2^{*} i^{*}(f c+D f c) * t-p h i c\right)\right) \cdot{ }^{*} x R ; \\
& x b b R=2^{*} \exp \left(-2 j^{*} \text { pi* }^{*}(f c+D f c) * t-\text { phic }\right) \cdot{ }^{*} x R ;
\end{aligned}
$$

or,

To store the real or imaginary portion of a complex matrix/vector to a real matrix/vector, we use the real and imag functions. The following are the combinations of these operations.

```
Amat = real(Cmat); Avec = real(Cvec);
Bmat = imag(Cmat); Bvec = imag(Cvec);
Amat = real(Bmat); // generates run type error
Amat = imag(Bmat); // generates run type error
Avec = real(Bvec); // generates run type error
Avec = imag(Bvec); // generates run type error
```


## Matrix/Vector operations, element by element

Assume the variables as defined above. Note that the $\mathbf{R}$ expression for 2 or more variables for the $=$ operand, they must be of the same size (row/column/length dimensions) and type (matrix or vectors). For the $+=,-=, . *=$, and $/=$ operands, the R and L operands must have the same dimensions and be of the same type if the L operand is real. For mixed operands, i.e., real and complex for the R expression, the receiving matrix/vector must be of type complex (matcmplx/veccmplx). Note also, that the R expression can be real and the L -variable can be complex. We note that for element by element matrix/vector multiplication, we use the .* (dot star) operator, and not the * operator which is reserved for matrix product operations. Note that for the other operators ( + , - , and $/$ ), that element by element operations are implied (since theirs no other option in matrix operations except on an element by element basis). The following are permissible operations on these variables on an element by element basis.

```
Amat = Bmat + Bmat2;
Amat = Bmat - Bmat2;
Amat = Bmat .* Bmat2;
Amat = Bmat / Bmat2;
Amat += Bmat;
Amat -= Bmat;
Amat .*= Bmat;
Amat /= Bmat;
Cmat = Bmat + Dmat2;
Cmat = Bmat - Dmat2;
Cmat = Bmat .* Dmat2;
```

```
Avec = Bvec + Bvec2;
Avec = Bvec - Bvec2;
Avec = Bvec .* Bvec2;
Avec = Bvec / Bvec2;
Avec += Bvec;
Avec -= Bvec;
Avec .*= Bvec;
Avec /= Bvec;
Cvec = Bvec + Dvec2;
Cvec = Bvec - Dvec2;
Cvec = Bvec .* Dvec2;
```

```
Cmat = Bmat / Dmat2; Cvec = Bvec / Dvec2;
Cmat += Bmat; Cvec += Bvec;
Cmat -= Bmat;
Cmat .*= Bmat;
Cmat /= Bmat;
Cmat = Dmat + Dmat2;
Cmat = Dmat - Dmat2;
Cmat = Dmat .* Dmat2;
Cmat = Dmat / Dmat2;
Cmat += Dmat2;
Cmat -= Dmat2;
Cmat .*= Dmat2;
Cmat /= Dmat2;
```

```
Cvec -= Bvec;
```

Cvec -= Bvec;
Cvec .*= Bvec;
Cvec .*= Bvec;
Cvec /= Bvec;
Cvec /= Bvec;
Cvec = Dvec + Dvec2;
Cvec = Dvec + Dvec2;
Cvec = Dvec - Dvec2;
Cvec = Dvec - Dvec2;
Cvec = Dvec .* Dvec2;
Cvec = Dvec .* Dvec2;
Cvec = Dvec / Dvec2;
Cvec = Dvec / Dvec2;
Cvec += Dvec2;
Cvec += Dvec2;
Cvec -= Dvec2;
Cvec -= Dvec2;
Cvec .*= Dvec2;
Cvec .*= Dvec2;
Cvec /= Dvec2;

```
Cvec /= Dvec2;
```


## The Matrix/Vector power operator .^

The operator ${ }^{\wedge}$ in ANSI C is reserved for a bitwise exclusive or between two integer types and this is true is Slide-Rule.. However, we define the operator.$^{\wedge}$ to be an element by element power operator between matrix and vector types for the following expressions. Assume the variables as defined above. Note that matrix variables must be of the same size (row/column dimensions) as well as vector variables (of equal length). Note that when the operator is used between two matrices or two vectors, that the second matrix/vector becomes the exponent value on an element by element basis. This also applies when a variable or literal is employed, i.e., the right operand always assumes the exponent value, while the left operand assumes the base operand. Note! (float Avar; complex Dvar; )

| Amat $=$ Bmat | . $\wedge$ Bmat2; | Avec $=$ Bvec | $\wedge$ Bvec2; |
| :---: | :---: | :---: | :---: |
| Cmat = Bmat | . $\wedge$ Bmat2; | Cvec = Bvec | $\wedge$ Bvec2; |
| Cmat = Dmat | .^ Dmat2; | Cvec = Dvec | $\wedge$ Dvec2; |
| Amat = Avar | .^ Bmat; | Avec = Bvar | $\wedge$ Bvec; |
| Amat = Bmat | .^ Avar; | Avec $=$ Bvec | $\wedge$ Bvar; |
| Cmat = Avar | .^ Bmat; | Cvec = Bvar | $\wedge$ Bvec; |
| Cmat = Bmat | .^ Avar; | Cvec = Bvec | $\wedge$ Bvar; |
| Cmat = Dvar | .^ Dmat; | Cvec = Dvar | $\wedge$ Dvec; |
| Cmat = Dmat | .^ Dvar; | Cvec = Dvec | $\wedge$ Dvar; |

Note!! Assume the right operand is the literal 2, and the left operand is a vector.

```
vector x, n;
n = vecGen(0, 1, 4); Print(n);
x = 2.^n;
```

The code segment above will generate a run time error, because the 2 . is returned as a token from the lexical analyzer as a floating point literal, so that the operator.$^{\wedge}$ becomes just $\wedge$. The correct method should be,

```
x = (2).^n; Print(x);
**************************************************************************
vector -> n(5)
[index
0000 - 
]
vector -> x(5)
[index
[1000 1 % 2 % 4 %
]
]**
```


## The Matrix Product Operator *

We use the * operator to perform the matrix product operation between two operands. The resulting output of the expression is either a real or complex matrix. We can mix real and complex matrix types, as long as the column size of the L-operand matches the row size of the R -operand. The resulting matrix output is the row size of the L-operand and the column size of the R -operand. We can also use a real or complex vector as one (and only one) of the operands. If used as an L-operand, the vector is converted (as a temp) into a row matrix, while if used as a R-operand, it is converted (as a temp) into a column matrix. If both $L$ and $R$ operand are real, then the resulting expression is real, else complex.

```
Amat * Bmat // Amat a real matrix, Bmat a real matrix
Cmat * Bmat // Cmat a complex matrix, Bmat a real matrix
Bmat * Cmat // Bmat a real matrix Cmat a complex matrix,
Amat * Avec // Amat a real matrix, Avec a real vector
Avec * Amat // Amat a real matrix, Avec a real vector
Cmat * Avec // Cmat a complex matrix, Avec a real vector
Avec * Cmat // Avec a real vector, Cmat a complex matrix
Amat * Cvec // Amat a real matrix, Cvec a complex vector
Cvec * Amat // Cvec a complex vector, Amat a real matrix
Cmat * Cvec // Cmat a complex matrix, Cvec a complex vector
Cvec * Cmat // Cvec a complex vector, Cmat a complex matrix
Dmat * Cmat // Dmat a complex matrix, Cmat a complex matrix
```


## Generating a Vector Sequence

We can generate a vector sequence as shown in the following examples.

```
vector Avec;
int npts;
Avec = vecGen(0, 1, 4); // start at 0, increment by 1, last element = 4
Print(Avec);
Avec = vecLin(-4, 0, 5); // start at -4, last value = 0, 5 elements
Print(Avec);
***********************************************************************
vector -> Avec(5)
[index
0000
]
vector -> Avec(5)
[index
00000 -4 0-3 0
]
```

For a log sequence, we have function vecLog as described below.

```
vector vecLog(float first, float last);
        where,
            first = 1st data value.
            last = last data value
```

For each decade, 75 data points are generated, 50 for the first half, and 25 points for the last. If the starting value is not on a decade, the first data point is forced on a decade value. For example if you specify 0.03 , the first data point will be 0.01 . To get the number of data points in the vector returned, use the Sizeof function (npts $=\operatorname{Sizeof}($ Avec $)$;

```
Avec = vecLog(0.03, 10); Print(Avec);
npts = Sizeof(Avec);
```

```
printf("Number of data points = %d\n", npts);
printf("First data point = %.2f\n", Avec[0]);
printf("Lasr data point = %.2f\n", Avec[npts-1]);
Number of data points = 391
First data point = 0.01
Lasr data point = 10.00
```


## Defining a Vector

In Chapter 3 we showed how to initialize a vector with the following type of statements,

```
vector vec1[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
veccmplx C0[]= { 1+2j, 2-4j, 2.2- 3j };
vector }\operatorname{Hrs[]={1, 1, 1, 1, 1, .5, zerosV(29), .5, 1, 1, 1, 1 };
vector HrsOpt[] = { 1, 1, 1, 1, 1, .39, z29, .39, 1, 1, 1, 1 };
```

The limitation on this type is that it is only allowed once and it's done at compile time. However, we can repeatable re-define a vector sequence with the following syntax; Var $=\{\ldots\}$; where Var is either of type vector or veccmplx, and previously defined. An example might be,

```
vector V1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
vector V4[] = { 7, 6, 5, 4 };
float y = 156.78, x = 123.4;
V1 = { V1, zerosV(29), y, flip(V4), x, V1, V4 };
```

From the above, we have,

| vector $->$ | V1(59) |  |  |  |  |  |  |  |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| [col |  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0000 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0010 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0020 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 156.78 |
| 0030 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |  |  |
| 0040 | 4 | 5 | 6 | 7 | 123.4 | 1 | 2 | 3 | 4 | 5 |
| 0050 | 6 | 7 | 8 | 9 | 10 | 7 | 6 | 5 | 4 |  |
| ] |  |  |  |  |  |  |  |  |  |  |

The above code segment could have been coded as,

```
vector V1; // Note that V1 has length 1 and is initialized to ZERO!!
vector V4[] = { 7, 6, 5, 4 }; // done at compile time
float y = 156.78, x = 123.4; // done at compile time
V1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // done at run time
V1 = { V1, zerosV(29), y, flip(V4), x, V1, V4 };
```

You can also define a temporary vector of an expression, by casting the expression. The syntax is,

$$
\begin{array}{r}
\text { (vector) }\{\text { expression }\} \\
\text { (veccmplx) }\left\{\begin{array}{l}
\text { expression }\}
\end{array}, ~\right.
\end{array}
$$

An example might be,

```
P0 = toeplitz2((vector){ subSV(pR, 0, 2), zerosV(N) },(vector){ pR[0], zerosV(N) });
```

where were passing two real vectors to the subroutine toeplitz2.

The following table lists the action for types within the curly-brackets. Under Action, the type refers to the receiving vector. Note that arithmetic operations of $+,-, *, /$, and.$*$ are permissible.

| Type | Action |
| :--- | :--- |
| Integer literal | Allowed. |
| Real literal | Allowed. |
| Complex literal | Allowed for type veccmplx, generates run type error for type vector. |
| Integer variable | Allowed. |
| Real variable | Allowed. |
| Complex variable | Allowed for type veccmplx, generates run type error for type vector. |
| Integer array | Generates run type error. |
| Real array | Generates run type error. |
| Complex array | Generates run type error. |
| Vector | Allowed. |
| Complex Vector | Allowed for type veccmplx, generates run type error for type vector. |
| Matrix | If a row or column = 1, else generates run type error |
| Complex Matrix | Allowed for type veccmplx if row or column = 1, else run type error |
| Functions | Allowed if return type is allowed as listed above. |

If you need to start out with a vector that is defined, but with no elements, proceed as follows. First define the vector, then set the vector to a null vector, and finally, add some elements. An example follows.

```
vector V1; // Note that V1 has length 1 and is initialized to ZERO!!
V1 = { }; // sets vector V1 to a NULL vector
V1 = { V1, a, b, c, d };
```

Note that in the above example, that vector V1 must be set to a null vector, else the first element of V1 will be a zero since the definition of V1 has length of 1 and is initialed to zero.

## Defining a Matrix

Referring to the previous topic, we can repeatable re-define a matrix sequence with the following syntax; $\operatorname{Var}=\{\ldots\} ; \quad$ where Var is either of type matrix or matcmplx, and previously defined. An example might be,

```
matrix X[][4] = { 0, 0, 1, 0, 0, 0, 0, 1,
    -36, 36, -.6, .6, 18, -18, .3, -. 3 };
matrix Bx[][1] = { 0, 0, 1, 0 };
matrix K[][4] = { 130.4444, -41.5556, 23.1, 15.4185 };
matrix Ka[][2] = {14.4, 0.6, 0.3, 15.7 };
matrix L[][2] = { 0, 0, 0, 0, 1, 0, 0, 1 };
matrix Yxy[][2] = { 1, 0, 0, 1 };
matrix Yzz[][2] = { -.6, .6, .3, -. 3 };
// Previous statements executed during compile time
matrix XX, YY;
Print(X-Bx*K); Print(Bx*K*L); Print(zerosM(2,4)); Print(Yzz-Ka*Yxy);
```

```
XX = { X-Bx*K, Bx*K*L; zerosM(2,4), Yzz-Ka*Yxy }; Print(XX);
YY = { X-Bx*K, Bx*K*L; zerosM(2,4), Yzz-Ka*Yxy }'; Print(YY);
```

From the above, we have,

| X-Bx*K = matrix -> tmpMat(4,4) [row |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0000 | 0 | 0 | 1 | 0 |  |  |
| 0001 | 0 | 0 | 0 | 1 |  |  |
| 0002 | -166.44 | 77.556 | -23.7 | -14.819 |  |  |
| 0003 | 18 | -18 | 0.3 | -0.3 |  |  |
| $\mathrm{Bx}^{*} \mathrm{~K}^{*} \mathrm{~L}=$ matrix $->\operatorname{tmpMat}(4,2)$ |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| [row |  |  |  |  |  |  |
| 0000 | 0 | 0 |  |  |  |  |
| 0001 | 0 | 0 |  |  |  |  |
| 0002 | 23.1 | 15.418 |  |  |  |  |
| 0003 | 0 | 0 |  |  |  |  |
| ] |  |  |  |  |  |  |
| zerosM(2,4) = matrix -> tmpSym(2,4) |  |  |  |  |  |  |
| [row |  |  |  |  |  |  |
| 0000 | 0 | 0 | 0 | 0 |  |  |
| 0001 | 0 | 0 | 0 | 0 |  |  |
| ] |  |  |  |  |  |  |
| Yzz-Ka*Yxy $=$ matrix -> tmpMat $(2,2)$ |  |  |  |  |  |  |
| [row |  |  |  |  |  |  |
| 0000 | -15 | 0 |  |  |  |  |
| 0001 | 0 | -16 |  |  |  |  |
| ] |  |  |  |  |  |  |
| matrix -> XX $(6,6)$ |  |  |  |  |  |  |
| [row |  |  |  |  |  |  |
| 0000 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0001 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0002 | -166.44 | 77.556 | -23.7 | -14.819 | 23.1 | 15.418 |
| 0003 | 18 | -18 | 0.3 | -0.3 | 0 | 0 |
| 0004 | 0 | 0 | 0 | 0 | -15 | 0 |
| 0005 | 0 | 0 | 0 | 0 | 0 | -16 |
| ] |  |  |  |  |  |  |
| matrix -> YY(6,6) |  |  |  |  |  |  |
| [row |  |  |  |  |  |  |
| 0000 | 0 | 0 | -166.44 | 18 | $\bigcirc$ | 0 |
| 0001 | 0 | 0 | 77.556 | -18 | 0 | 0 |
| 0002 | 1 | 0 | -23.7 | 0.3 | $\bigcirc$ | 0 |
| 0003 | 0 | 1 | -14.819 | -0.3 | 0 | 0 |
| 0004 | 0 | 0 | 23.1 | 0 | -15 | 0 |
| 0005 | 0 | 0 | 15.418 | 0 | $\bigcirc$ | -16 |
| ] |  |  |  |  |  |  |

From the code segment above, we note use the syntax Var $=(\ldots)$ '; to take the transpose for matrix YY. Use syntax Var = (...).' for a complex matrix to take the transpose with out taking its conjugate. You can also define a temporary matrix of an expression, by casting the expression. The syntax is,

```
    (matrix) { expression }
(matcmplx) { expression }
```

Note that successive matrix variables that are separated by commas, expand the new matrix to the right, i.e., more columns are added. When separated by the semi-colon operator, the new matrix is expanded by adding more row(s), and the column length of the matrix becomes fixed. When new rows are added with the semi-colon operator, then the number of columns for the new row(s) must match the number of columns in the previous row(s), else a run time error is generated. Also, when adding successive matrix elements with the comma operator, the number of rows of succeeding matrix variables must match the
preceding matrix. The following table lists the action for types within the curly-brackets. Under Action, the type refers to the receiving matrix. Note that arithmetic operations of $+,-,{ }^{*}, /$, and.$*$ are permissible.

| Type | Action |
| :--- | :--- |
| Integer literal | Allowed. |
| Real literal | Allowed. |
| Complex literal | Allowed for type matcmplx, generates run type error for type matrix. |
| Integer variable | Allowed. |
| Real variable | Allowed. |
| Complex variable | Allowed for type matcmplx, generates run type error for type matrix. |
| Integer array | Generates run type error. |
| Real array | Generates run type error. |
| Complex array | Generates run type error. |
| Vector | Allowed. |
| Complex Vector | Allowed for type matcmplx, generates run type error for type matrix. |
| Matrix | Allowed |
| Complex Matrix | Allowed for type matcmplx, generates run type error for type matrix. |
| Functions | Allowed if return type is allowed as list above. |

If you need to start out with a matrix that is defined, but with no elements, proceed as follows. First define the matrix, then set the matrix to a null matrix, and finally, add some elements. An example of building a table of 50 rows and 4 columns of computed values $a, b, c$, and $d$ follows.

```
matrix table; // Note that table has dimensions of 1x1 and length 1,
    // and is initialized to ZERO!!
table = { }; // sets matrix table to a NULL matrix
for(i = 0; i < 50; i++) {
    table = { table; a, b, c, d };
}
```

Note that in the above example, that the matrix table must be the left most argument, else you will get an error message, and be transferred to the custom editor at the offending line of code.

## Multi-dimensional Matrix Arrays

Multi-dimensional matrix arrays can be created by calling routine zeros3D with more than two arguments. For example,

```
#include "matrices.h"
matrix M1, M2;
matcmplx M1c;
vector V1;
float ftemp;
M1 = zeros3D(4, 4, 5);
M1c = zeros3D(2, 2, 10);
```

To access the $\mathrm{k}^{\text {th }}$ matrix of M 1 , the code is as follows

$$
\text { M2 = M1[::k]; // } 0<=k<=4
$$

To store to the $\mathrm{j}^{\text {th }}$ matrix of M 1 , the code is as follows

$$
\text { M1 }[:: j]=\operatorname{magic}(4) ; / / 0<=j<=4
$$

Note! To access elements ( $\mathrm{i}, \mathrm{j}$ ) of the $\mathrm{k}^{\text {th }}$ matrix, first create temporary matrix. For example,

```
M2 = M1[::k]; // 0 <= k <= 4
V1 = M2[3]; // get 4 th row in plane k of M1
ftemp = M2[1][2]; // get element from row 2 column 3 in plane k of M1
```


## Casting Operations

When using both matrix and vector variables, it's often the case that we need to transform a vector into a row matrix, or to transform a matrix into a vector. We can perform these operations by performing casting operations. The syntax for these operations are as follows:

```
(vector) expression;
(veccmplx) expression;
(matrix) expression;
(matcmplx) expression;
```

The return type is specified in parenthesis. The expression type must follow the following rules, else a compile error.

- Return type of vector, expression must equal matrix type or floating array variable.
- Return type of matrix, expression must equal vector type or floating array variable.
- Return type of veccmplx, expression must equal matemplx type or complex array variable.
- Return type of matcmplx, expression must equal veccmplx type or complex array variable.

Example

```
    matcmplx Ryy, Q, pyd;
    veccmplx w, d;
    Ryy = Q'*Q; Ryy += 1e-12*eye(Ryy);
    pyd = Q'*d;
    w = linequ(lu(Ryy),(veccmplx)pyd);
```

On the second line, the matrix product (*) of Q transpose times vector d produces a complex column matrix pyd. One the third line of code, subroutine linequ requires a complex vector type (veccmplx), so we just cast it to that type.

Note!! Its best to enclose the expression in parenthesis since the casting operation has precedence left to right, i.e., it has the same level of precedence as the product operator *. As an example, if were casting a matrix product expression to a vector, and the left operand is a vector, then you would get a run time error. So always cast the expression in paren's.

Example

```
matrix Ryy; // A N x M matrix
    vector w, x; // x has N elements
```

```
w = (vector)(x*Ryy); // w will have a length of M elements
```


## Two Special Conversion Operations

A special case arises when a matrix expression results in a matrix of dimension $1 \times 1$, and were dividing it into a matrix or vector expression with different dimensions. An example might be

```
matrix rls(vector x, vector d, int p, float lambda)
{ // recurive least squares routine
    // x = input data
    // d = desired output
    // p = order of filter
    // lambda = exponental weighting factor
    // e = output vector of error term
    // returns tap weights over time
    int i, N, M;
    matrix X, W, z, g, P;
    float alpha, lambda, delta = .001;
    vector vTmp;
    N = Sizeof(x);
    W = zerosM(N, p); // Initialize weight matrix over time
    X = convMat(x, p); // generates convolution matrix X
    P = eye(zerosM(p,p))/delta;
    M = N-p+1;
    for(i = 1; i <= M; i++) {
            z = P*((matrix)X[i])';
            g = z/(lambda + X[i]*z);
            alpha = d[i] - X[i]*W[i-1];
            W[i] = W[i-1] + (vector)(alpha*g');
            P=(P - g*z')/lambda;
    }
    return(W);
}
```

For line $g=z /\left(\right.$ lambda $\left.+X[i]^{*} z\right)$; as shown above, the product operation $x[i]^{*} z$ produces a 1-by-1 matrix. We than add a float point variable (lambda) to this matrix, with the result still being a 1 x 1 matrix. Matrix $z$ is a p-by- 1 matrix, so when we divide it by the expression (lambda $+x[i] * z$ ), we should get a run time error. However, Slide-Rule automatically turns the expression $\mathrm{X}[\mathrm{i}]^{\star} \mathrm{z}$ into a temporary float before adding the variable lambda to it.

So Note!! When ever a product operation turns into a $1 \times 1$ matrix, Slide-Rule converts the expression to a temporary float or complex expression, depending on whether the resulting matrix is real or complex.

## The Matrix transpose operator ${ }^{\prime}$

The matrix transpose operator ' can be used in an assignment statement or in an expression. The following are examples:

```
Amat = Bmat';
Amat = Bmat'* Cmat;
```

Note that for complex matrices, the transpose operation' takes the complex conjugate. To take the transpose of a complex matrix without taking its conjugate, we use the .' operator combination as follows:
Cmat = Dmat.';

## Matrix row/column operations

The following are examples of the permissible row/column operations that can be performed. We use [i] to specify a row, and [:i] to specify a column, where $\mathbf{i}$ always starts counting from zero.

```
Amat[2] = Bmat[1]*2; // replace row 3 with 2*row 2 of Bmat
aVec = Bmat[3]; // interchange
Bmat[3] = Bmat[2]; // rows
Bmat[2] = Avec; // 3 and 4
Bmat[0] += Bmat[1]*3; // add 3*row2 to row 1
Bmat[2] *= 2; // row 3 times 2
Amat[:2] += Amat[:0] // column 3 = col 3 + col 1
```


## Retrieving Matrix/Vector Elements

We can retrieve Matrix/Vector elements by using double subscripts for matrix variables ( [i][j] ), and a single subscript variable for a vector variable ( $[\mathrm{k}]$ ). For real matrix and vector types, the type returned is a float/double, while for complex matrix and vector types, the operation returns a complex type. The following are examples as such.

```
ftemp = Bmat[1][1]; // retrieve row 2 col 2 of Bmat
ftemp = Avec[3]; // retrieve 4th element of Avec
```


## Setting Matrix/Vector Elements

We can set Matrix/Vector elements in a similar manner as describe above. The following is an example as such.

```
matrix Bmat[5][5];
for(i = 0; i < 5; i++) { // initialize diag
    for(j = 0; j < 5; j++) {// to
        if(j == i)
            Bmat[i][j] = 1; // 1
        else Bmat[i][j] = 0; // non-diag elements to 0
    }
}
Print(Bmat);
matrix -> Bmat(5,5)
[row
0000 1 0 0 0 0
0001
0002
0003
0004
]
```


## Retrieving Matrix Rows and Columns

We can retrieve Matrix rows or columns by using a single subscript. To retrieve row k (counting from zero), from real matrix M1, we have;
V1 = M1[k]; // V1 is real vector type

To retrieve column j (counting from zero), from real matrix M1, we have;

```
V1 = M1[:j]; // V1 is real vector type
```


## Setting Matrix Rows and Columns

We can set Matrix rows or columns by using a single subscript. Suppose we have two real vectors of equal length, and we want to store in a real matrix. The code for this might look like,

```
matrix sys;
vector num[] = { 1, 2, 3, 4 };
vector den[] = { 4, 3, 2, 1 };
sys = zerosM(2, Sizeof(num));
sys[0] = num; // store vector num to row 0 of matrix sys
sys[1] = den; // store vector den to row 1 of matrix sys
Print(sys);
sys = zerosM(Sizeof(num), 2);
sys[:0] = num; // store vector num to column 0 of matrix sys
sys[:1] = den; // store vector den to column 1 of matrix sys
Print(sys);
*************************************************************************
matrix -> sys(2,4)
[row
0000 1 2 0 3
0001 4 3 3 1
]
matrix -> sys(4,2)
[row
0000 1 4
0001 2 3
0002 3 2
0003 4 1
]
```

The caveat here is that vectors num and den must be of equal length, else a run-time error is generated. Note that to store a vector to a matrix as a column, we use the colon operator : within the brackets (sys[:j] $=\mathrm{V} 1$; // store vector V 1 to column j of matrix sys). Another method with less lines of code and more efficient is as follows,

```
Sys = ( num; den }; // store vector num to row 0 of matrix sys
    // store vector den to row 1 of matrix sys
```

The previous code statement can be written as

```
Sys = ( num; den }'; // store vector num to column 0 of matrix sys
    // store vector den to column 1 of matrix sys
```


## Generating a Zero Matrix or Vector

We can generate a zero matrix/vector variable with the following examples. Note that the matrix/vector variables must have been previously defined. Refer to the Functions reference (F7) under the Help Menu (matrix/Vector auxiliary functions).

```
V1 = zerosV(2000); // real vector V1 to 2000 zero elements
V1c = zerosV(2000); // complex vector V1c to 2000 zero elements
M1 = zerosM(50, 50); // real matrix vector M1 to 50 x 50 zero elements
M1c = zerosM(50, 50); // complex matrix vector M1c to 50 x 50 zero elements
```


## Resizing Matrix/Vector Variables

We can resize matrix/vector variables with the following examples. Note that the matrix/vector variables must have been previously defined. Refer to the Functions reference (F7) under the Help menu (matrix/Vector auxiliary functions).

```
vector V1[] = { 1, 2, 3 };
matrix M1[][2] = ( 1, 2, 3, 4 };
vector V2;
matrix M2;
Print(V1);
V2 = resizeV(V1, 5, 0); // vector V1 to 5 elements, 2 0's after element 3
Print(V2);
V2 = resizeV(V1, 5, -1); // vector V1 to 5 elements, 2 0's before element 1
Print(V2); Print(M1);
M2 = resizeM(M1, 3, 3, 0); // matrix M1 to 3x3, 0's on row and column 3
Print(M2);
resizeM(M1, 3, 3, -1); // matrix M1 to 3x3, 0's on row and column 1
Print(M2);
************************* Results *******************************************
vector -> V1(3)
[col
0000 1 2
]
vector -> V2(5)
[col
0000 - 1 2
]
vector -> V2(5)
[col
0000 0 0 0 0
]
matrix -> M1(2,2)
[row
0000 
]
matrix -> M2(3,3)
[row
0000 1 
0001 000 
]
matrix -> M2(3,3)
[row
0000 0 0 0
0001 0 1 2
0002 0 3
]
```


## Stripping a Vector of leading zeros

We can strip a vector of leading zero's as shown in the following example. Note that the vector must have been previously defined. Refer to the Functions reference (F7) under the Help menu (Matrix/Vector Auxiliary Functions).

```
while(V1[0] == 0) reduce(V1, 0, 1);
```


## Getting the size of Vectors/Matrices

We can determine the number of elements in a vector or matrix by using the Sizeof function. An example is shown below.

```
matrix M1[5][5];
vector V1[25];
int M, N;
M = Sizeof(M1); N = Sizeof(V1);
```

Note that in both cases above, that M and N will be equal to 25 . To get row and column dimensions of a matrix, we use the rowSize and colSize functions as shown below.

```
M = rowSize(M1); N = colSize(M1);
```


## Vector/Matrix Sub-Copy Operations

A sub-vector of a vector or a sub-matrix of a row or column matrix can be created with the following expression syntax's.

```
vector V[start, cnt)
veccmplx Vc[start, cnt)
matrix M[start, cnt)
matcmplx Mc[start, cnt)
where,
\(\mathrm{V}=\) input vector (real);
\(\mathrm{Vc}=\) input vector (complex).
\(\mathrm{M}=\) input matrix (real);
Mc = input matrix (complex).
start \(=\) index of starting element (counting from zero)
cnt \(=\) number of element to copy
Return value:
The new (copied) sub-vector or sub-matrix.
```

Note! The start argument must have a range from 0 to the number of elements -1 . An implied increment value of 1 is added to the start value until cnt elements are copied. Note when we refer to a row or column vector, were referring to a matrix where either the row or column dimension is 1 . Any other matrix dimension will cause a run time error. If pulling a row or column from a matrix, and then you want to copy a portion of it to a vector, use the following internal function copy $V$ with the following prototypes.

```
vector copyV(vector V1, int start, int cnt);
veccmplx copyV(veccmplx Vc, int start, int cnt);
```

Examples:

```
#include "matrices.h"
vector V1[] = { 1, 2, 3, 4, 5 };
vector V2, V3;
matrix M1[][1] = { 1, 2, 3, 4, 5 };
matrix M2, M3;
int N;
N = Sizeof(V1);
Print(V1);
V2 = V1[0,N-1]; Print(V2);
Print(M1);
M2 = M1[0,N-1]; Print(M2);
M3 = magic(5); Print(M3);
V3 = copyV(M3[:2],0,N-1); Print(V3);
V3 = copyV(M3[2],0,N-1); Print(V3);
```



## Creating a Sub-Vector

A portion or sub-vector of a vector can be created with the following expression syntax's.

```
vector V[start:inc:last)
veccmplx Vc[start:inc:last)
vector V[start:last)
veccmplx Vc[start:last)
```

where,
V = input vector (real).
Vc = input vector (complex).
start $=$ starting index (a literal integer or integer variable).
inc $=$ increment from start index, continuing to last index.
last $=$ index of last element (a literal integer or integer variable).
Return value:
The new created sub-vector.
Note!! The indices (start and last) must range from 0 to the number of elements -1 . In the example below, the indices start and last must lie in the range of 0 to 4 . Also, for the reduced syntax's without the increment index (inc), the increment value will assume the value of 1 . Note also, that we can use the transpose operator as a shortened version of taking the conjugate of a complex vector. This operator has no effect on a real vector.

An example.

```
vector V1[] = { 1, 2, 3, 4, 5 };
veccmplx V1c[] = { 1+1j, 2-2j, 3+3j, 4-4j, 5+5j };
vector V2;
veccmplx V2c;
int N;
N = Sizeof(V1);
Print(V1);
V2 = V1[N-1:-2:0];
Print(V2);
V2 = V1[0:N-2];
Print(V2);
Print(V1c);
V2c = V1c[0:N-2];
Print(V2c);
V2c = V1c[0:N-2]';
Print(V2c);
***********************************************************************
vector -> V1(5)
[col
0000 
]
vector -> V2(4)
[col
]
vector -> V2(4)
[col
0000 1
]
vector(cmplx) -> V1c(5)
[col
0000 1+1j [ 2-2j 3+3j 4
]
vector(cmplx) -> V2c(4)
[col
0000 1+1j 2-2j 3+3j 4-4j
```

```
]
vector(cmplx) -> V2c(4)
[col
0000 1-1j 2+2j 3-3j 4+4j
]
```


## Creating a Circulant matrix

A circulant matrix is one in which each row is a shifted version of the row above it. An example which uses the Sub-Vector technique as discussed in the previous topic follows.

```
matrix Circulant(vector V1)
{
    int i, N;
    matrix C;
    N = Sizeof(V1);
    C = zerosM(N,N);
    C[0] = V1;
    for(i = 1; i < N; i++) {
        V1 = { V1[N-1], V1[0:N-2] };
        C[i] = V1;
    }
    return(C);
}
vector Vx;
matrix Mx;
    Vx = vecLin(1, 5, 5);
    Mx = Circulant(Vx);
    Print(Mx);
**********************************************************************
matrix -> Mx(5,5)
[row
0000 1 2 0 % 3 %
0001 5
0002 4 4 5 1 0
[lllll
00004 2 < 3 4 4
```


## Creating a Sub-matrix

A portion or sub-matrix of a matrix can be created with the following expression syntax's.

```
matrix M[start1:inc1:last1, start2:inc2:last2);
matcmplx Mc[start1:inc1:last1, start2:inc2:last2);
matrix M[start1:last1, start2:last2);
matcmplx Mc[start1:last1, start2:last2);
matrix M[row, start2:inc2:last2);
matcmplx Mc[row, start2:inc2:last2);
matrix M[row, start2:last2);
matcmplx Mc[row, start2:last2);
matrix M[start1:inc1:last1, col);
matcmplx Mc[start1:inc1:last1, col);
matrix M[start1:last1, col);
matcmplx Mc[start1:last1, col);
```

where,

$$
\mathrm{M}=\text { input matrix (real) }
$$

Mc = input matrix (complex)
startl = starting row index (a literal integer or integer variable).
inc1 = increment from start1 index, continuing to last1 index.
last1 = index of last row element (a literal integer or integer variable).
start2 $=$ starting column index (a literal integer or integer variable).
inc2 $=$ increment from start2 index, continuing to last2 index.
last2 $=$ index of last column element (a literal integer or integer variable).
row $=$ single row specification
$\mathrm{col}=$ single column specification
Return value:
$=$ The new sub-matrix.
Note!! The indices (start and last) must range from 0 to the number of elements -1 . In the example below, the indices start and last must lie in the range of 0 to 4 . Also, for the reduced syntax's without the increment index (inc), the increment value will assume the value of 1 . These expression syntax's require that from the start index to the last index, that the progression must be of increasing order, i.e., the increment value (inc) must be great than zero, and the start index cannot be greater that the last index.

## Examples

```
#include "matrices.h"
matrix M1, M2;
M1 = magic(5); Print(M1);
M2 = M1[1:3,0:4]; Print(M2);
M2 = M1[1,0:4]; Print(M2);
M2 = M1[1:3,3]; Print(M2);
***********************************************
matrix -> M1(5,5)
[row
\begin{tabular}{rrrrrr}
0000 & 17 & 24 & 1 & 8 & 15 \\
0001 & 23 & 5 & 7 & 14 & 16 \\
0002 & 4 & 6 & 13 & 20 & 22 \\
0003 & 10 & 12 & 19 & 21 & 3 \\
0004 & 11 & 18 & 25 & 2 & 9
\end{tabular}
]
matrix -> M2(3,5)
[row
\begin{tabular}{lrrrrr}
0000 & 23 & 5 & 7 & 14 & 16 \\
0001 & 4 & 6 & 13 & 20 & 22 \\
0002 & 10 & 12 & 19 & 21 & 3
\end{tabular}
]
matrix -> M2(1,5)
[row
0000 23
]
matrix -> M2(3,1)
[row
0000 14
0001 20
0002 21
]
```


## The Print function

Routine Print is provided to print out Matrices and Vectors (both real and complex) in an efficient manner. The internal print specification is g12.4 (refer to printf in Chapter 3). If you need more resolution, you can print out individual elements using the printf function. The use of this routine can be found in the examples of this chapter.

## In Summary

This chapter doesn't cover all the Auxiliary Functions for matrix and vector operations, so please consult Appendix A for a list of functions, and The Functions Reference (function key F7) in the software for function prototypes and description as well.

## Chapter 5-The Editor

## Introduction

A custom editor has been developed and is provided as an adjunct to the software to allow the user to quickly edit a script file in an integrated environment. This editor has been designed and developed both as a mouse driven editor and as a line editor using hotkeys. The mouse driven portion allows the user to select and highlight selected text by positioning the caret and selecting text for deletion, copying, or inserting in the standard windows manner. By the use of certain hotkeys, the user can select line(s) of 'code' for deletion, copying, and or other functions as will be described below.

As described in the tutorial section in the previous section, once a given shell script file has been selected, as displayed on the Hot key Bar, the depression of the function key F10 will evoke the Custom Editor and the selected shell script file. If you run a given shell script file by depression of the Function key F11, and an error is detected, the Custom Editor will automatically be invoked with the selected shell script file, and with the cursor positioned at the faulted line of code in question. The user can also select the Custom Editor once in the Shell script viewer (if plots were forthcoming), by depression of function key F10. If no plots were output, the printer output file (<*.prn>) will be displayed in WordPad. To bring up the shell script in the Custom Editor, terminate the printed output display (Alt+F4), select SlideRule as the input focus, and then depress the F10 function key.

## Error Detection

As stated above, once you run or execute a given shell script file and an error is detected, the Custom Editor is automatically invoked with the selected shell script file as input, and the cursor positioned on the faulted line of code. The user can quickly edit the file, then depress function key F11 to save the file (to disk) and then automatically re-run the given shell script file. If the Shell Viewer program along with several plotting windows are open, these will all be automatically closed with this one save and execute step (F11).

## General Description

When the editor is invoked with a given shell script file, the user will be shown an edit window similar to that shown in figure 5.1. The Caption Bar displays the shell script file name along with its file path location. The Menu Bar displays the menu functions that the user can invoke, by either using the mouse, or a combination of the underlined character and the Alt key. As an example, to pull down the File menu, the user depresses the Alt +F keys. The remaining display shows the line number along with the 'code', both in separate windows. When the user invokes the editor with the F10 hotkey, the caret will be left justified on line 1. Since the caret is a vertical flashing line that can be hard to detect, the line number with the caret will be displayed in red while the other line numbers will be displayed in black. The caret position always will be displayed and will always indicate where the user can insert text, erase text with the backspace key, etc. Please note that the line numbers displayed are not part of the shell script file, but are provided to give the user ease of edit operations.


Figure 5.1

## Cut and Paste Operations

Under the Edit Menu are listed the standard visual cut and paste operations, namely Cut, Copy, Paste, and Undo. These edit operations all require that the caret be positioned (except for Undo), and for Cut and Copy operations, that the desired text to be deleted or copied, be highlighted by dragging the mouse (left mouse button depressed). These are standard edit operations in window type editors. These operations can be speeded up by using hotkeys as depicted under the Edit menu and are as follows:


## Deleting Lines

A fast way to delete line(s) of 'code' is as follows. First position the caret on the first line of code (any column will do) either with your mouse or with the up and down arrow keys. Note that the line of code with the caret on it will have its line number displayed in red. Next, hit the F1 function key to 'mark' this line of code. Note the line number is now half green and half red. If deleting more than one line, move the caret to the last line. Direction can be either up or down. Note that when you do this that the original marked line is now all green and the line number with the caret on it is displayed in red. To delete the line(s), simply depress the F2 function key. Please note that this clears the mark and that the marked code is gone. To reverse this operation and restore the deleted code, simply do the Undo operation, i.e., Ctrl +Z .

## Copying Lines

A fast way to copy line(s) of 'code' for latter insertion is as follows. First position the caret on the first line of code (any column will do) either with your mouse or with the up and down arrow keys. Note that the line of code with the caret on it will have its line number displayed in red. Next, hit the F1 function key to 'mark' this line of code. Note the line number is now half green and half red. If copying more than one line, move the caret to the last line. Direction can be either up or down. Note that when you do this that the original marked line is now all green and the line number with the caret on it is displayed in red. To copy the line(s), simply depress the F3 function key. Please note that this clears the mark.

## Shifting Lines of Code

Referring to Figure 5.1, we note that lines 8 through 12 are left justified. Suppose that you want to shift these lines of 'code' right. Please note that in this editor that the ' $\backslash t$ ' tab character positions the code right 4 character spaces. For C-type code, we note that for 'for' loops or 'while' loops, that the code needs to be shifted either right of left depending on previous edit operations. Referring to figure 4.1, we first position the caret on the first line of code (any column will do) with the mouse or with the up/down left/right keyboard arrow keys. Note that the line of code with the caret on it will have its line number displayed in red. Next, hit the F1 function to 'mark' this line of code. Note the line number is half green and half red. If shifting more than one line, move the caret to the last line. Direction can be either up or down. Note that when you do this that the original marked line is now all green and the line number with the caret on it is displayed in red. To shift the marked lines right, simply hit the F5 function key the number of times you want the marked code shifted an equivalent 4 spaces right (done with tab keys). To shift the code left, depress the F6 function key. When done, depress the Escape key to unmark the 'mark' and note that the green 'marked' line is unmarked (either black if multiple lines or red if a single line of code has been shifted).

## Appending or Inserting Copied or Deleted Lines of Code

Previously we talked about copying lines of code with the F1/F3 function keys, and deleting lines of code with the F1/F2 function keys. To append or insert this code, we proceed as follows. If appending, first position the caret on the line of code that we want to append the copied line(s) too. Next, depress the combination of $\mathrm{Ctrl}+\mathrm{P}$, and observe that the code has been inserted after the line (marked by red) that the
caret resides on. If to insert the copied code before the line, just depress the combination $\mathrm{Ctrl}+\mathrm{Shft}+\mathrm{P}$, and note that the copied code is inserted before the line (marked by red) that the caret resides on.

## Coding new Lines of Code

To code statements after the current line with the caret (as marked in red), simply depress the combination $\mathrm{Ctrl}+\mathrm{O}$. If the code is to be written before the 'current line', depress the $\mathrm{Ctrl}+\mathrm{Shft}+\mathrm{O}$ combination. Please note that the caret of the new line (to be coded) is positioned right and aligned with the previously 'current line' in terms of white space.

## Navigating Around

To find a particular piece of code, select the Find... Tile under the Search Tile on the main menu (or Ctrl+F), and fill in the dialog box, then depress the enter key. To repeat, hit the Enter key. Make sure the caret is at the beginning of the file, i.e., line one. For Find and Replace, select the Replace... Tile under the Search Tile on the main menu (or $\mathbf{C t r l}+\mathbf{H}$ ), and fill in the dialog box, then click the Replace tile in the dialog box to find and replace a single entry, or the Replace All tile to find and replace all entries. Make sure the caret is at the beginning of the file, i.e., line one. Don't select the Find Next tile, as that will skip an entry.

A fast way to get to line 1 is to depress the Ctrl+Home key. To get to the bottom of the file, depress the Ctrl+End key. Another fast way to get to a known line of code is to hit the Ctrl+G function key and enter the line number, then depress the enter key. Alternately, one can use the scroll buttons, the scroll bar, the up and down keyboard arrow keys, and the 'PgUp' and 'PgDn' keys. If the caret is at the beginning of a line, hit the End key to move the caret to the end of the line. To move it (the caret) back to the beginning of the line, depress the Home key. The mouse in combination with depressing the left mouse button is another favorite method for positioning the caret.

## Bracket Search

Many times while programming in C or $\mathrm{C}++$, one forgets a bracket ( '\{', '[', '(‘, '\}', ']', ')' ). As an example, in an if statement, we use the characters ' $\{$ ' and ' $\}$ ' to enclose more than one statement. If you position the caret at the beginning of one of these matching pair of characters, then depress the Ctrl+F6 function (or $\mathrm{Ctrl}+\mathrm{Shft}+\%$ ) key, the caret will move to the other matching bracket character if the code is correct. If it doesn't move, then you narrowed down where the offending syntax error occurs. This function works for the three types ' $\{$ ', '[', and ' $(‘$ ', as described above.

## More than one File

Many times while developing an algorithm or a procedure, one wants to paste some code from another file. To do this, first depress the F7 function key to save the current file. Then depress the F10 function key to open the File Open Dialog box. Select the file of interest, then depress the 'Enter' key. Use the navigation aids as described above to go to the code of interest. To 'yank' the code of interest, first mark the first line with the F1 function key. Then move the caret to the last line of code, then depress the F4 function key. This will yank the desired lines of code, then switch you back to your original file where you can paste the lines of code in at the proper place with either the following combinations: Cntl +P ,
after the line with the caret on it; $\mathrm{Ctrl}+\mathrm{Shft}+\mathrm{P}$, before the line with the caret on it. To get back to the alternate or second file, first save the current file to disk (F7), then depress the F8 function key. Note that you can switch back and forth between the two files by continuously depressing the F8 function key.

## Shell script error condition

When executing a given shell script file, and a Message Box appears with a red icon, then this means that an error was detected in the complier, or a run time error was detected in the shell script interpreter. For a complier error, the message could be something like "(line 12/begin line $=12$ )(sigyacc.y, 1370) call to unrecognized function print!!". After depressing the enter key, the custom editor will be executed with the cursor at the offending line of code, in this case, line 12. For a run time error, the message box error
 somewhere starting a line x and ending at line y . After depressing the enter key, the editor program will attempted to find an error at line $y$, and if not, then line $y-1$. If an error is detected, a message indicating the problem will be output, Otherwise an error message will be output listing possible things to check between line x and line y .

## Special Code Array Generation from Clipboard

The user can generate data sequences as described under The Assignment Operator topic in Chapter 3, from highlighted column data in a Microsoft Excel Spreadsheet. This special procedure allows one to generate data into a shell script file by highlighting a column or multiple columns (of data) in a Microsoft Excel spreadsheet. First position the cursor at the selected point that you want data to be inserted into the shell script file, then switch to the Excel window and highlight the column(s) of data, then switch back to the custom editor, then depress function key combination Cntl+F11 to bring up a dialog procedure. Fill in the variable name field, and select whether data type is float, char, vector, or matrix. Types of char and vector assume a single column of data (in Excel). For types of float or matrix, fill in the number of columns data field. Then depress the enter key to generate the data.

## Shell Script Print File

To view the shell script print file, the user just needs to press the F12 function key, as in the Shell Script Viewer when displaying plots. Note that in displaying the (*.prn) print file, that we use the Windows WordPad program. Note that while displaying the print file that the function keys such as F10, etc., no longer are enabled, since WordPad is a Windows system utility. Either terminate (Alt+F4) or select the SlideRule graphic with your mouse.

## Help menu

Under the Help Menu Tile, the user can select a Reference Help file of the more than 500 plus internal functions, or a Help file on this Custom Editor. Depressing of the Help Icon (or Ctrl+R) brings up the Help Reference Functions.

## Chapter 6 - Plotting Details

## Introduction

This chapter takes you through a quick tutorial on how to generate plots quickly, and will present examples as such. Also, there will be shown simplifying shortcuts to make the task for the user less tedious.

## 2D Plots

The 2 D plots (function of two variables in Cartesian coordinates) come in six flavors, and their prototypes are as follows:

```
1) ploty( float xMin, float xIncr, float *ypts, int n);
    ploty( float xMin, float xIncr, vector Vy, int n);
2) plotx( float yMin, float yIncr, float *xpts, int n);
    plotx( float yMin, float yIncr, vector Vx, int n);
3) plotxy( float *xpts, float *ypts, int n);
    plotxy( vector Vx, vector Vy, int n);
4) pxlogy(float *xpts, float *ypts, int n);
    pxlogy(vector Vx, vector Vy, int n);
5) pylogx( float *xpts, float *ypts, int n);
    pylogx( vector Vx, vector Vy, int n);
6) ploglog( float *xpts, float *ypts, int n);
    ploglog( vector Vx, vector Vy, int n);
where,
xpts = array of x data points to plot.
ypts = array of y data points to plot.
Vx = vector of x data points to plot.
Vy = vector of y data points to plot.
yMin = y-value of first data point
yIncr = y increment between y data points
xMin = x-value of first data point
xIncr = x increment between x data points
n = number of data points to plot
```

From the above it should obvious what each of the internal plotting functions do by their naming. For example, routine ploty plots a 2D plot of y-data points (an array of types float (double), vector, or a matrix row or column), where the starting $y$-value starts at $x M i n$, and the $y$-values are evenly spaced by xIncr, and the number of data points to plot is specified by $n$. Now, in order to generate one of these plots, we need only to write three lines of code and of course have an array of data points to plot. The following is an example to plot an array of 100 y -data points where the starting y -value begins at x -value equals zero, and the data points are evenly spaced by 1 . Will name the window of this plot as Plot of $Y$ values Example.

```
openPlot("Plot of Y-values Example");
ploty(0, 1, yData, 100);
pCRT();
```

That's it! You're done. This little piece of code will open up a plotting window and plot the array with automatic scaling and numbering (with tick marks) of the two axis. Routine openPlot opens up the plot, and routine $p C R T$ closes out the plot. You must always have this pair to generate a plot. Also, please note that when we talk about the x -axis, were talking about the horizontal (positive to the right), and for the $y$-axis, were referring to the vertical (positive being up). Now, what if you want to make another plot, say 500 data points from vector gValues. The code is as follows:

```
openPlot("Plot of Y-values Example");
ploty(0, 1, yData, 100);
pCRT();
openPlot("Plot of G-values Example");
ploty(0, 1, gValues, Sizeof(gValues));
pCRT();
```

Now you have two plots displayed in two windows. Now, what about labeling these plots, changing colors, lines styles, etc.? We can do all this and more in two ways. The first way, and the hard way is to code these desires with the 2D plotting help routines as listed in the Functions Reference in Appendix A, and the prototype specifications using function key F7 in the software. The easy way to do this is to stick with the three lines of code as described above, and then when execution of your script file is complete, and your plots are displayed, just select the Customize Tile on the Menu Bar. You will be shown additional menu selections as follows:

Title...<br>X-Axis Label...<br>Y-Axis Label...<br>Grid on/off<br>Horizontal grid on/off<br>Background Color...<br>Axis Color...<br>Text and Arrow Color...<br>Axis Thickness...<br>Y-Label Orientation<br>Axis and Tic Labels suppress<br>Curve style...<br>MinMax Values...<br>Auto Scale<br>Forced Tic Values...<br>Aspect Ratio enable/disable

## Left Mouse-Rubber Band Functions ENABLED Right Mouse-Annotation Functions ENABLED

Select one of these options, fill in the Dialog Box, and hit the enter key. Notice the given plot is updated immediately. Better yet, your shell script file has been updated. There's one caveat to this method, i.e., for multiple plot windows, make sure that the window titles are distinct in the openPlot function calls, since that ASCII title is what is used to search for in updating your shell script file.

## MinMax Customize Feature

Say your plotting a function such as $f(x)=1 / x$. As $x$ is close to zero, then the value of $y$ becomes huge, and your plot becomes distorted because of this. This function allows one to handle this situation. The plot on the left below shows the distorted plot. So we select the MinMax Values tile under the Customize
tile, and enter the values $-20,20,-4$, and 4 into the dialog box as shown. The plot gets updated, and the pminmax function call is inserted just after the Openplot function statement.


## Aspect Ratio enabled/disable

When plotting round objects, we need to make the aspect ratio square, since normally for $\mathrm{x} / \mathrm{y}$ plots it's not. The following plots below show that be just clicking on the Aspect Ratio enable/disable tile under the Customize tile, one can switch back and forth between the two views.



## Curve Pen Style

On any of the $\mathrm{x} / \mathrm{y}$ plots, one can plot multiple curves on a given plot, and the software will automatically adjust the range of x and y axis including the tic marks. If one wants to change the color, curve style, and/or the line thickness, one first selects the curve number, then one or more of the options as shown in the dialog box.


The line style comes in 38 styles, namely, solid line, Dashedline, dottedline, dashdotline, dashdotdotline, square symbol, cross hair, sqr rot 90 deg, triangle - inverted, triangle, solid square, solid triangle - inv, solid triangle, solid sqr - rot 90 deg, line from x-axis (xline), single pixle points, Poles - x marks, Zeros small circles, line + sqr symbol, line + cross hair", line + sqr rot 90 deg, line + triangle - inverted, line + triangle, line + solid square, line + solid triangle - inv, line + solid triangle,line + solid sqr - rot 90 deg , xline + sqr symbol, xline + cross hair, xline + sqr rot 90 deg, xline + triangle - inverted, xline + triangle, xline + solid square, xline + solid triangle - inv, xline + solid triangle, xline + solid sqr - rot 90 deg, xline + staircase, staircase. Find below a plot having used this customize feature.


The other functions as previously listed, either present a dialog box or switch from an on to off condition or vice-versa. In addition to the functions under the Customize tile, we can depress the right mouse button after positioning the mouse curser to a selected point on the displayed plot, to perform the following functions which are listed below.

Text Insert...<br>Draw Arrow...<br>Draw Line<br>Draw Box (text wrap)<br>Draw Box (NO test wrap)<br>Move Text within box<br>Edit Test (text wrap) within Box...<br>Edit Test (NO text wrap) within Box...<br>Delete Box<br>Delete Box and Text plus Arrow

## Annotation with Text Insert, Draw Arrow, and Draw Line

On x/y plot types produced from function calls of plotxy, ploty, plotx, ploglog, plogxy, pxlogy, we can we annotate these drawings with text and arrows interactively with our mouse once the plot has been displayed on the monitor. In the above example, choose one of the plots, pick any point (by cursor location) to annotate the drawing with text, then depress the right most key on your mouse and note the popup menu. While still holding down the right most key, pull the mouse down and over the Text Insert menu selection, then release the right key on your mouse. Note the dialog box in the upper left corner of the monitor. Enter some text in this dialog box, then depress the Enter key (twice) or click the OK button (twice) with your mouse. This procedure allows the user to enter multiple lines of annotation by using a single Enter key after each entry, then terminating with another Enter key. Note that the drawing is refreshed, and the text is displayed to the right of the point that the cursor was pointing to when the right most button on your mouse was depressed. Also note that the cursor point marks the text in the lower-left corner of the displayed text. Now, depress the F10 hotkey, and observe the ptext function call right before the pCRT function call for the given display plot. In effect, the program modifies the shell script for you, such that the next time you run that particular shell file, you don't have re-annotate the whole mess again. If you need to erase a given annotation, select the Edit menu, and click the Undo or Undo All tile. For the Undo function, you can alternately use the $\mathrm{Crtl}+\mathrm{Z}$ combo. These functions will erase either the last interactive function or all of the interactive functions as well as updating the shell script file for you. You can also edit the script file (F10), then depress the F11 hotkey to save the script file and rerun the script again. Please note that once the Shell is terminated, that the next time you run the script file, the functions that were added interactively can no longer be erased with the Undo or Undo All Functions.

Now, repeat the above procedure, but this time, instead of selecting text annotation, select the arrow tile. When you do this, note that a line is drawn interactively from the selection point to the current position of the mouse. Move the mouse cursor point to where you want the tail of the arrow, then depress and release the right mouse button. Note the text dialog box as before. Enter the appropriate text and depress the Enter key (twice) on your keyboard as before. Note the drawing is being re-drawn and that an arrow is being drawn as specified with the appropriate text. Again, depress the F10 hotkey, drag the scroll bar down until we observe the ptext and arrowP function calls right before the pCRT function call.

Note in the above procedure, that the program figures out the correct point to insert the code from the text in the openPlot function call. So for multiple plotting windows in a given script file, make sure the ASCII text in the openPlot function calls are all different.

## Draw Box

In addition to the functions named previously, i.e., Text Insert..., Draw Arrow..., and Draw Line, we can draw a box (with no text inside) for input of a multiline text, or we can draw a box around text for either ; (1) editing with the Edit text with Box... function; (2) to delete the text within the drawn box with the Delete Box and Text function; (3) to move the text and box with the Move text with Box function. This function comes in two favors, namely Draw Box (text wrap) and Draw Box (No text wrap). Choose Text Wrap if the text is like a narrative, and you don't mine if text is wrapped to a new line. Choose No Text Wrap, if you want the lines within the box to stay the same. To perform any of these functions, position the cursor at the upper left corner of the desired box, then right click your mouse, and with the right button on your mouse depressed, move the cursor down over the Draw Box tile, then release the right button on your mouse. Move the cursor down and to the right until you have the desired box size. Then right click your mouse to complete the operation. Note that the size of the box determines the length of the text within the box, and that new lines within the box break on a space character.

## Move Text with Box

This function allows for moving a box with or without text on the drawn plot. To perform this function, position the cursor any where within the box, then right click your mouse, and with the right button on your mouse depressed, move the cursor down over the Move Text with Box tile, then release the right button on your mouse. Move the cursor to position the box and text to the desired location, and then right click your mouse to complete the operation.

## Edit Text within Box

This function allows for editing text within a box on the drawn plot. In comes in two favors, namely Text Wrap and No Text Wrap. Choose Text Wrap if the text is like a narrative, and you don't mine if text is wrapped to a new line. Choose No Text Wrap, if you want the lines within the box to stay the same. To perform this function, position the cursor any where within the box, then right click your mouse, and with the right button on your mouse depressed, move the cursor down over the Edit Text (text wrap) within Box, or Edit text (No text wrap) within box tile, then release the right button on your mouse. Note the Edit Multiline Text in Box dialog that appears. Edit and/or enter the desired text, then depress the enter key to complete the edit operation. If no drawn box, first do the Draw Box function as described above, then return to this function. Note that the size of the box determines the length of the text with in the box, and that new lines within the box break on a space (for the text wrap case). For the text wrap case, the new line character (' $\backslash n$ '), is replaced by a space, while for the No text wrap case, the new line character is replaced by the $\$$ character.

## Delete Box

This function allows for deleting a box on the drawn plot. To perform this function, position the cursor any where within the box, then right click your mouse, and with the right button on your mouse depressed, move the cursor down over the Delete Box tile, then release the right button on your mouse.

Note the drawn box is gone. If there is text within the deleted box, your can draw another box around this text to either lengthen or shorten the length of the text lines. Note that the vertical left, top and bottom lines of the resulting box are adjusted for the text. The right vertical line is fixed (in the code); however it is either stretched or lengthened when the box is drawn.

## Delete Box and Text

This function allows for deleting a box and text on the drawn plot. To perform this function, position the cursor any where within the box, then right click your mouse, and with the right button on your mouse depressed, move the cursor down over the Delete Box and Text tile, then release the right button on your mouse. Note the drawn box and text is gone.

## Annotation with Symbols

In addition to entering ASCII text, as described previously, the user can enter up to 48 Greek symbols, the degree symbol as a superscript, and all of the ASCII text including the 48 Greek symbols as a superscript or as a subscript. The complete list is shown below. As an example, to enter the capital symbol Gamma with a subscript of 'IN', the entry would look like '\Gamma_I_N'. The under-score character is a marker saying in effect that the next ASCII charter is a subscript. To enter a superscript, we use the up tilde ASCII character ' $\wedge$ '. As an example, for the lower case beta raised to the fourth power, the entry would look like $\backslash$ beta^ 4 . Note, to enter more than 1 character as a super or sub script, enclose in curly brackets. An example might be, e $\{\wedge j$ lomega $\}$. Note that curly brackets are reserved for this function, and my not be used in the previous described annotations modes.

| Character | Description | Character Entry | Character | Description | Character Entry |
| :---: | :---: | :---: | :---: | :---: | :---: |
| A | capital Alpha | $\backslash$ Alpha | $\alpha$ | lower case alpha | \alpha |
| B | capital Beta | $\backslash$ Beta | $\beta$ | lower case beta | Ibeta |
| $\Gamma$ | capital Gamma | $\backslash \mathrm{Gamma}$ | $\gamma$ | lower case gamma | \gamma |
| $\triangle$ | capital Delta | $\backslash$ Delta | $\delta$ | lower case delta | \delta |
| E | capital Epsilon | \Epsilon | $\varepsilon$ | lower case epsilon | \epsilon |
| $Z$ | capital Zeta | \Zeta | $\zeta$ | lower case zeta | \zeta |
| H | capital Eta | $\backslash$ Eta | $\eta$ | lower case eta | leta |
| $\Theta$ | capital Theta | \Theta | $\theta$ | lower case theta | \theta |
| I | capital Iota | \ota | l | lower case iota | \iota |
| K | capital Kappa | $\backslash$ Kappa | $\kappa$ | lower case kappa | \kappa |
| $\Lambda$ | capital Lambda | \Lambda | $\lambda$ | lower case lambda | $\backslash \mathrm{lambda}$ |
| M | capital Mu | $\backslash \mathrm{Mu}$ | $\mu$ | lower case mu | $\backslash \mathrm{mu}$ |
| N | capital Nu | \Nu | $v$ | lower case nu | Inu |
| $\Xi$ | capital Xi | \Xi | $\xi$ | lower case xi | \xi |
| O | capital Omicron | \Omicron | $o$ | lower case omicron | \omicron |
| П | capital Pi | \Pi | $\pi$ | lower case pi | pi |
| P | capital Rho | $\backslash$ Rho | $\rho$ | lower case rho | \rho |
| $\Sigma$ | capital Sigma | \Sigma | $\sigma$ | lower case sigma | \sigma |
| T | capital Tau | \Tau | $\tau$ | lower case tau | \tau |
| Y | capital Upsilon | \Upsilon | $v$ | lower case upsilon | \upsilon |
| $\Phi$ | capital Phi | $\backslash$ Phi | $\varphi$ | lower case phi | \phi |
| X | capital Chi | $\backslash \mathrm{Chi}$ | $\chi$ | lower case chi | \chi |


| $\Psi$ | capital Psi | $\backslash$ Psi | $\psi$ | lower case psi | $\backslash \mathrm{psi}$ |
| :--- | :--- | :--- | :--- | :--- | :--- |
| $\Omega$ | capital Omega | $\backslash$ Omega | $\omega$ | lower case omega | lomega |
| $\square$ | Degree symbol | $\backslash \operatorname{deg}$ |  |  |  |

## Zoom mode

On $\mathrm{x} / \mathrm{y}$ plot types, we can zoom into these plots by enclosing a portion of the plot in a rubber band window that we form by using the mouse. In the digitalF sub-directory, select file lowPass and execute. On the Magnitude dB plot window, select a portion of the pass band by positioning the mouse cursor at a selected point. Then hold down the left mouse button and drag the mouse to the right and down. Notice the rectangular square window that is formed. Now let go of the left mouse button and observe the blown up plot. You can try this again on the blown up plot for further magnification of the area of interest. To get back to the normal sized plot, just click on the right mouse button. Note that if you form a rectangular rubber band window with no data points enclosed, or if the rubber band window only surrounds only a single data point, then no blown up plot will be forthcoming. The following plots are from this technique.


## Meta File Generation

A good method for inserting graphic plots into documents is to select the Metafile Generate... under the File menu. If you have a word processor such as Microsoft Word, select the Plotting tile on the main menu and run A mesh Grid, then once the drawing is complete, fill in the MetaFile File Input Name... dialog procedure under the File tile, then switch to the word processor, and import the drawing from Insert/Picture/From File... enhanced meta file format (<*>.emf). The graphic, once inserted into your document, will have the same aspect ratio as shown on your monitor, however, it will be larger then you desire. To resize the graphic, select the graphic with your mouse and double click on it, then Format/Picture... and the size tab. Make sure the Lock aspect ratio and the Relative to original picture size check boxes are checked. Then adjust the Width or Height using the input dialog boxes. Finally position the graphic in your document using the ruler slider. Below, is a plot from this procedure, and inserted into this document using Microsoft Word Insert Picture Mode with metafile input.


As seen from the above example, this method offers good resolution for printer output rather than using a bitmap and the clipboard. This example was sized at a width of 6 inches. To position two documents side by side, proceed as follows. Resize the first graphic to $\sim 3.25$ to 3.5 inches ( 8.255 to 8.89 cm ) and position on the left. Insert the second graphic below the first and resize to the same dimensions. First adjust the margins under File/Page Setup..., such that it's 7 to $\sim 7.5$ inches ( 19.05 cm ), and then select the second graphic with your mouse. Then Format/Drop Cap... and None. Drag the second graphic up and to the right of the first graphic. Below is an example as such.



Another technique is to do an overlay of one metafile plot over another. On the example below, the fist metafile graphic was resized to $\sim 6$ inches in width. The second graphic was resized to $\sim 3$ inches in width; then on the Layout tab, we select In front of text and with the Horizontal Alignment set to other; and finally, on the Color and Line tab, set the color to white with Transparency to $0 \%$.


## The Histogram Plot

The histogram plot comes in a single flavor and its prototypes are as follows:

```
void histoP(float *array, int npts, float min, float max, int nbins);
void histoP(vector V1, int npts, float min, float max, int nbins);
where,
array = input data array of real data
V1 = a real vector of data points
min = the minimum value of the first bin
max = the maximum value of the last bin
nbins = the number of divisions or bins for displaying data distribution
    range : 0 -> 100.
```

The $y$-axis will be plotted as a percentage in the range of 0 to 100 . Note that the min and max values on the calling sequence don't have to be the absolute minimum or maximum of the data array. Hint (for nice labeled tick marks on the x -axis, chose nice values for the min and max values passed to histoP).

This supplied function has the same restrictions and programming caveats as the 2-D plots as described previously, i.e., we must open the plot with the function call openPlot(...), and close out the plot with the
function call $p C R T()$. We can also customize these plots interactively with our mouse once the plot is displayed on the monitor by selecting the $\underline{\text { Customize Tile. This displays the following options: }}$

Title...<br>$\underline{X}$-Axis Label...<br>Y-Axis Label...<br>Grid on/off<br>Horizontal Grid on/off<br>Background Color...<br>Axis Color...<br>Text Color...<br>Axis Thickness...<br>Y-Label Orientation<br>Histo specs...<br>Plot color...

## Left Mouse-Rubber Band Functions DISABLED <br> Right Mouse-Annotation Functions ENABLED

The Histo specs allow you to change the starting and ending values of the first and last bins, and also the number of bins. The Plot color... tile allows one to change the color of the displayed bins. All other options are same as previously described under the 2-D Plots description. In addition to the functions listed above under the Customize tile, we can depress the right mouse button after positioning the mouse curser on the viewed plot, to perform the following functions which are listed below.

```
Text Insert...
Draw Arrow...
Draw Line
Draw Box (text wrap)
Draw Box (NO test wrap)
Move Text within box
Edit Test (text wrap) within Box...
Edit Test (NO text wrap) within Box...
Delete Box
Delete Box and Text plus Arrow
```

Below is the code for a simple histo plot of the distribution of 1000 Guassian data points with zero mean and unit variance generated by a call to internal function uniformV. The code for this follows.

```
Vector V1;
int N = 1000;
V1 = uniformV(N);
openPlot("Histogram Example");
histoP(V1, Sizeof(V1), -4, 3, 70);
yLabel("PERCENTAGE",0);
Title("Gaussian Noise wirh Zero Mean and Unit Variance");
xLabel("1000 Values over 70 bins");
colorB(191,191,191);
colorT(0,0,0);
axesS(0,3,0,0,0);
grid();
pCRT();
```



## The Polar Plot

The Polar plot comes in two flavors and its prototypes are as follows:

```
void polarP(float *r, float *theta, int npts);
void polarP(vector Vr, vector Vt, int npts);
void polarPdB(float *r, float *theta, int npts, float cutoff);
void polarPdB(vector Vr, vector Vt, int npts, float cutoff);
where,
r = array of radial data
Vr = vector of radial data
theta = array of angular data in radians
Vt = vector of angular data in radians
npts = number of points in each array(both must be same size)
cutoff = dB cutoff point in inner circle(must be <= -20.)
```

We can customize these plots interactive with our mouse once the plot is displayed on the monitor by selecting the Customize tile. This displays the following options:

$$
\begin{aligned}
& \text { Title... } \\
& \text { X-Axis Label... } \\
& \text { Background Color... } \\
& \text { Axis Color... } \\
& \text { Text Color... } \\
& \text { Curve style... }
\end{aligned}
$$

Below is an example of a small program to generate polar plots.

```
vector Vr[360], Vt[360];
float theta, x;
int i;
for(i = 0; i < 360; i++)
{
    x = (i - 180)*pi/30.;
    if(abs(x) < 0.0001)
        Vr[i] = 1.5;
        else
            Vr[i] = 1.5 * abs(sin(x)/x);
            Vt[i] = (pi * i)/180.;
}
openPlot("Polar Example 1");
Title("Antenna Pattern - abs. field");
xLabel("Antenna Pattern - abs. field");
penS(0,1,255,0,0);
polarP(Vr, Vt, Sizeof(Vr));
pCRT();
openPlot("Polar Example 2");
Title("Antenna Pattern - dB");
xLabel("Antenna Pattern - dB");
penS(0,3,255,0,0);
polarPdb(Vr, Vt, Sizeof(Vr), -40.);
pCRT();
```

The first plot in the above example follows: Note that when you generate a metafile for this type of plot, to keep the horizontal and vertical lengths the same, else the plot will be distorted.


Antenna Pattern - abs. field

## The Vector Plot

The Vector Plot plots a vector field or gradient field on an $x-y$ plot. Its prototype is as follows:

```
void vectorP(float xstart, float xincr, float ystart, float yincr,
    matrix Mx, matrix My, float scale);
where,
xstart = starting value on x axis for grid
xincr = value for incrementing to next grid point on x-axis
ystart = starting value on y axis for grid
yincr = value for incrementing to next grid point on y-axis
Mx = matrix of delta x changes at each data point
My = matrix of delta y changes at each data point
scale = relative scale value applied to each vector for plotting appearance
```

We can customize these plots interactively with our mouse once the plot is displayed on the monitor by selecting the Customize tile on the menu bar. This displays the following options:

```
Title..
X-Axis Label...
Y-Axis Label...
Grid on/off
Background Color...
Axis Color...
Text and Arrow Color...
Axis Thickness...
Y-label orientation
Forced Tic Values...
Aspect Ratio enable/disable
Scale factor...
```

Left Mouse-Rubber Band Functions ENABLED
Right Mouse-Annotation Functions ENABLED
Scale factor...

In addition to the functions listed above under the Customize tile, we can depress the right mouse button after positioning the mouse curser on the viewed plot, to perform the following functions which are listed below.

```
Text Insert...
Draw Arrow...
Draw Line
Draw Box (text wrap)
Draw Box (NO test wrap)
Move Text within box
Edit Test (text wrap) within Box...
Edit Test (NO text wrap) within Box...
Delete Box
Delete Box and Text plus Arrow
```

Below is an example of a small program to generate a vector plot

```
    * Example of vectorP plot
    * Given teo wires with a current of 50 amps and
    * separated by 4cm, plot a relative Inductive Field B.
    * Note! put the wires at (3, 5) and (7, 5)
*/
matrix By, Bx;
int i, j;
float delx, dely;
float dx, dy;
float x, y;
float temp1, temp2;
float ftemp, mu0, K;
float r1, r2;
mu0 = 4.*pi*1.E-7;
K = mu0*50000./pi2; /* pi2 = 2.*pi */
dx = 0.4; dy = 0.4;
Bx = zerosM(25, 25); By = zerosM(25, 25);
for(i = 0, x = 0; i < 25; i++)
{
    y = 0;
    for(j = 0, y = 0; j < 25; j++)
    {
        temp1 = 3. - x; temp2 = 5. - y;
        r1 = temp1*temp1 + temp2*temp2;
        temp1 = 7. - x;
        r2 = temp1*temp1 + temp2*temp2;
        if(r1 < 0.1 || r2 < 0.1) {
            delx = 0.; dely = 0.;
        }
        else {
            delx = (y - 5.) * (1./r2 + 1./r1);
            delx *= K;
            dely = (3.-x)/r1 + (7.-x)/r2;
            dely *= K;
        }
        Bx[i][j] = delx; By[i][j] = dely;
        y += dy;
    }
    x += dx;
}
openPlot("Vector Example");
Title("B Field - 2 Parallel Wires");
xLabel("X - Distance");
yLabel("Y Distance",0);
vectorP(0, 0.4, ., 0.4, Bx, By, 15);
colorB(207,207,207);
axisS(0,2,0,0,0);
pCRT();
```



## The Polezero Plot

The Polezero Plot plots complex poles and zeros. Its prototype is as follows:

```
void polezeroP(complex *parray, int pnpts, complex *zarray, int znpts);
void polezeroP(veccmplx Vp, int pnpts, veccmplx Vz, int znpts);
where,
parray = complex array of poles
Vp = complex vector of poles
pnpts = number of poles
zarray = complex array of zeros
Vz = complex vector of zeros
znpts = number of zeros
```

Multiple plots per open plot window are not possible.
This plot is used in the GUI design procedure for IIR filters as will be described in Chapter 8. Below is an example of the poles and zeros of an IIR Butterworth Filter Low pass filter with All Pass group delay compensation.


## The 3-D Line Plot

The 3-D line plot is a three dimensional plot of $\mathrm{x}, \mathrm{y}$, and z data points. Its prototype is as follows:

```
void plot3C( float *xpts, float *ypts, float *zpts, int n);
void plot3C( vector Vx, vector Vy, vector Vz, int n);
where,
xpts = array of x data points to plot.
ypts = array of y data points to plot.
zpts = array of z data points to plot.
Vx = vector of x data points to plot.
Vy = vector of y data points to plot.
Vz = vector of z data points to plot.
n = number of data points to plot
```


## Note!

This call must be made between the openPlot function call and the pCRT function call. You must also make a call to the camloc3D function to set the viewing angle. This plot also has customized features under the Customize tile as follows:

## X-Axis Label...

Y-Axis Label...
Background Color...
Camera Elevation...
Camera Azimuth...
Line Thickness...
Line Color...
Axis/Ticks marks on/off
Back Face Lines on/off
Left Mouse-Rubber Band Functions DISABLED
Right Mouse-Annotation Functions DISABLED

```
void camloc3D(float dist, float azimuth, float angle);
    where,
    dist = distance of camera from 3D box in z-axis units
    azimuth = angle of camera about z-axis from the x-axis in degrees.
    angle = elevation angle (degrees) of the camera up from the x-y plane.
```


## Remarks

As a general rule, make parameter dist approximately equal to:

$$
\text { dist }=\text { abs( 6.*(zmax - zmin)); }
$$

Below is an example of a small program to generate a 3-D line plot.

```
vector x[1000], y[1000], z[1000];
int i;
float delta, t;
delta = 10.*pi/999.;
t = 0.;
for(i = 0; i < 1000; i++)
{
        x[i] = 0.5*}\operatorname{sin}(t)\mathrm{ ;
        y[i] = 0.5*}\operatorname{cos(t);
        z[i] = t/(10.*pi);
        t += delta;
}
openPlot("Circular Helix");
surwir3D(2);
colorB(255, 255, 255);
xLabel("X-AXIS");
yLabel("Y-AXIS", 1);
zLabel("Z-AXIS");
/* specify camera location */
camloc3D(15.0,200.0,25.0);
plot3C(x, y, z, 1000);
pCRT();
```



## The Contour Plot

The ContourP plots a x-y contour plot from a grid array of data points. Its prototype is as follows:

```
void contourP(matrix Z, int nlevels,
    float xmin, float xmax, float ymin, float ymax);
where,
Z = input matrix of real data in elevation ( Z[x][y] ).
nlevels = number of contour levels
xmin = minimum x-value in grid
xmax = maximum x-value in grid
ymin = minimum y-value in grid
ymax = maximum y-value in grid
```

This plot also has customized features under the Customize tile as follows:

| X-Axis Label... |
| :--- |
| $\mathbf{Y}$-Axis Label... |
| Background Color... |
| Contour Fill on/of... |
| Color Bar... |
| Axis suppress |
| Axis/Ticks Color... |
| Contour Lines on/off... |
| Contour Labels off/on... |

## Left Mouse-Rubber Band Functions DISABLED Right Mouse-Annotation Functions DISABLED

Please find below a shell script to generate an example contour plot along with the contour plot as shown below.

```
const int GSIZE = 61;
matrix Z[GSIZE][GSIZE];
#include "hillsubs.txt"
float xmax,xmin,ymax,ymin,zmax,zmin;
/* test of Surface plots in 3D */
    xmin = xmax = ymin = ymax = 0; zmin = zmax = 0;
    hills(GSIZE, GSIZE, xmin, xmax, ymin, ymax, zmin, zmax);
    openPlot("Hills Contour");
    colorBar(7);
    confil3D(45);
    Title("Hills Contour Surface");
    xLabel("Range");
    yLabel("Cross Range",0);
    contourP(Z, 16, xmin, xmax, ymin, ymax);
    colorB(223,223,223);
    axisS(0,1,0,0,0);
    grid();
    pCRT();
```



A different flavor of this plot is the Line-Contour plot. It's prototype is as follows:

```
void contour(matrix Z, int nlevels, float xmin, float xmax,
    float ymin, float ymax, vector Levels);
where,
Z = input matrix of real data in elevation ( Z[x][y] ).
nlevels = number of contour levels
xmin = minimum x-value in grid
xmax = maximum x-value in grid
ymin = minimum y-value in grid
ymax = maximum y-value in grid
Levels = vector of contour levels
```

This plot also has customize features under the Customize tile as follows:

Title Label...<br>X-Axis Label...<br>Y-Axis Label...<br>Grid on/off<br>Horizontal Grid on/off<br>Background Color...<br>Axis Color...<br>Text Color...<br>Axis Thickness...<br>Y-Label Orientation<br>Color Bar...<br>Axis suppress<br>Axis and Tick Labels suppress

| Left Mouse-Rubber Band Functions DISABLED |
| :--- |
| Right Mouse-Annotation Functions ENABLED |

In addition to the functions listed above under the Customize tile, we can depress the right mouse button after positioning the mouse curser on the viewed plot, to perform the following functions which are listed below.

Text Insert...<br>Draw Arrow...<br>Draw Line<br>Draw Box (text wrap)<br>Draw Box (NO test wrap)<br>Move Text within box<br>Edit Test (text wrap) within Box...<br>Edit Test (NO text wrap) within Box...<br>Delete Box<br>Delete Box and Text plus Arrow



## The 3D-Mesh Plot

The routine consurf3D plots a $x-y / z$ wire mesh plot from a grid array of data points. Its prototype is as follows:

where,

```
style = an or'ed value of output options
    0 = draw all surface lines
    1 = suppress lines hidden by surface
    2 = suppress hidden lines and bottom of surface
    4 = no axis or tick labels
Z = the matrix[x][y] of elevation points to be plotted
xmin = the minimum x value
xmax = the maximum x value
ymin = the minimum y value
ymax = the maximum y value
```

In order to get colored grated contour, besides inserting a call to routine consurf3D between the openPlot(...) and the plot closeout call $p C R T()$, the user needs to make two additional calls; (1) The colorBar routine to specify which color bar to use, and a call to routine confil3D to specify the number of levels in the bar ( 45 works good). These prototypes have been previously listed under X/Y Contour Plot.

We can also customize these plots interactive with our mouse once the plot is displayed on the monitor by selecting the Customize Tile. This displays the following options:

X-Axis Label...<br>Y-Axis Label...<br>Z-Axis Label...<br>Background Color...<br>Camera Elevation...<br>Camera Azimuth...<br>Surface Wire on/off<br>Contour Fill on/off<br>Back Face on/off<br>Color Bar...<br>Hidden Line on/off<br>Bottom Line on/off<br>Contour Surface top<br>Axis/Tick marks off/on<br>Contour Lines on/off<br>Contour Labels off/on

Selection of one of these functions will either present a dialog procedure or turn off that option. In any case, the display will be updated along with the shell script file. As an example, if the surface wire in on, and you click on that Tile, then the display will immediately be updated with the surface wire's turned off. If they were off, you will be offered a dialog procedure to select one of eight colors to turn this option back on. You can go to the plots sub-directory, and select different plots, and play with these options. Below is an example of a code segment to plot a complex Jacobian. This plot can be viewed below.

```
/* Jacobian Plot.
*/
matrix Z;
float u1max,u1min,u2max,u2min;
int i,j,k;
float u1, u2, q, r;
complex y, snu, ctemp;
float K, Kprime, ftemp, du1, du2;
/* set up data limits */
K = 2.; Kprime = 1.75;
u1min = -K; u1max = 4.375*K;
u2min = 0.; u2max = 5.*Kprime; u1 = u1min;
du1 = .20; du2 = .2;
q = exp(-pi*Kprime/K);
k = 0; K = 2.;
Z = zerosM(61, 51);
for(i = 0; i <= 60; i++) { // generate some data
    for(j = 0, u2 = u2min; j <= 50; j++)
    {
            y = pi*cmplx(u1,u2)/(2.*K);
            ctemp = 1. -4.*q* (1.-2.*q)*}\operatorname{cos}(y)*\operatorname{cos}(y)
            ftemp = abs(sin(y)*(1. -4.*q* ** cos(y)*}\operatorname{cos}(y))/ctemp)
            if(ftemp > 4.0) Z[i][j] = 4.0;
            else Z[i][j] = ftemp;
            u2 += du2;
            if(u2 > 2*Kprime) u2 -= 2*Kprime;
    }
    u1 += du1;
}
openPlot("JACOBIAN Example"); // open up plot window
```

```
opts3D(1,0,1);
conlab3D(16,1,1);
colorBar(8);
surfil3D(45);
confil3D(45);
surwir3D(1);
xLabel("X-AXIS");
yLabel("Y-AXIS", 1);
zLabel("Z-AXIS");
camloc3D(133.2,210.0,20.0);
consurf3D(1, Z,u1min,u1max,u2min,u2max);
colorB(223,223,223);
pCRT(); // Output to the display monitor
```



Note!! To generate one's own plot, start off with the code starting with the openPlot(...) line as shown below, and including the lines up to and including CRT(). For the matrix array, compute as $\mathrm{Z}[\mathrm{i}][\mathrm{j}]$, where the $\mathbf{j}$ index goes first, then the i index. Note that the Z array is just the $\mathbf{z}$-axis values looking down on the $\mathbf{x} / \mathrm{y}$ plane. After displaying the plot, modify by using the customize features.

```
openPlot("JACOBIAN Example"); // open up plot window
conlab3D(16,1,1);
colorBar(8);
surfil3D(45);
confil3D(45);
surwir3D(1);
camloc3D(133.2,210.0,20.0);
consurf3D(1, Z,u1min,u1max,u2min,u2max);
pCRT(); // Output to the display monitor
```


## The Movie Plot

The Movie Plot (movieP) used between an openPlot function call and the pCRT function call for multiple $\mathrm{x}-\mathrm{y}$ plots, where the user desires to see each frame at a set rate. So instead of plotting a family of curves, each data set is plotted to the monitor at the set rate, then erased before the next data set is plotted. Because the data is plotted in reverse video, colors don't work well except on the final curve, where the data is plotted in normal mode. Please note that this mode is for the monitor only. The user should only plot continuous curves or points only (refer to function penS). Its prototype is as follows:

```
int movieP(int msec, float delta);
where,
msec = display update time (on the display monitor) in msec. for each slide.
delta = time that each slide represents in seconds (in the
    real world)for reporting purposes.
```

Note!! The current number of movie plot windows in a given shell script is $\mathbf{1 0}$, and you can't mix movie plot windows with non-movie plotting windows. Between 100 and 500 slides per window should handle most simulations, although the program can handle more. On the Demo Files under the Plotting tile, run Movie Plot, Polyphase Channelizer (movie), and Yet Annother Movie Plot.

## Multiple Windows Displayed

When a given shell script executes, there's two options, namely, window(s) plots and a print file, or just a print file. If just a print file, then that file is displayed in the Window's WordPad program. If plotting window(s), then the first plot displayed takes up the entire monitor's screen. On the menu bar, after the Window tile on the main menu, the number of plots are displayed. Under the Window tile, the user can view additional plots with the following commands (depending on the number of plots), namely:

- Close Window Ctrl+F4
- Switch Window Next Tab
- Switch Window Prev Tab +Shift
- Cascade All Windows
- Tile vertical All Windows
- Restore All Windows Ctrl+R
- Display 2 Windows Tiled Ctrl+2
- Display 2 Windows Horizontal Ctrl+Shift+2
- Display 3 Windows Tiled Ctrl+3
- Display 3 Windows Horizontal Ctrl+Shift+3
- Display 4 Windows Tiled Ctrl+4
- Display 4 Windows Horizontal Ctrl+Shift+4
- Display 5 Windows Tiled Ctrl+5
- Display 5 Windows Horizontal Ctrl+Shift+5
- Display 6 Windows Tiled Ctrl+6
- Display 6 Windows Horizontal Ctrl+Shift+6
- Display 7 Windows Tiled Ctrl+7
- Display 7 Windows Horizontal Ctrl+Shift+7
- Display 8 Windows Tiled Ctrl+8
- Display 8 Windows Horizontal Ctrl+Shift+8
- Display 9 Windows Tiled Ctrl+9
- Display 9 Windows Horizontal CtrI+Shift+9
- Display 10 Windows Tiled Ctrl+A
- Display 10 Windows Horizontal Ctrl+Shift+A


## Chapter 7 - User Dialog's and Menus

## The Generic Dialog Procedure

The user, programming at the Shell level, can call a dialog procedure to allow the input of numeric quantities, user selectable flags, and a file name (for input). In particular, up to 14 floating point values can be specified, 3 groups of up to 6 exclusive flags ( 1 only from each group), an additional 10 flags in a separate group (bit or'ed), along with a file input/output name which includes a path specification. The user can also supply a help file which can be called up from the dialog procedure. The prototype specification for this internal callable module is as follows:

Syntax vector dialog(int nd, int n1, int n2, int n3, int n4, int nf, char *Titles, int * buttons, char *FileName)
where,
$\mathrm{nd}=$ number of data entries fields (boxes), 14 max
n1 = number of Group 1 buttons, 6 max
n2 $=$ number of Group 2 buttons, 6 max
n3 $=$ number of Group 3 buttons, 6 max
n4 $=$ number of Group 4 check boxes, 10 max
nf = 1 for a file input selection, $=0$ for none
Titles = character titles for the Dialog entry, the data entry
fields, the Group 1 buttons, the Group 2 buttons, the Group 3
buttons, and the Group 4 check boxes.
buttons = an integer array of size 4 that holds the Group 1->4
option selections (an output).
FileName = character array of size 81 to hold file name and path if
nf = 1 (an output).

## Returns:

A real vector where each vector element is a value from a data entry field.

## Remarks:

Note that labels is a sequential array of ASCII text titles that cover the Dialog box title, then up to the 14 data fields, followed by the Group1, Group2, Group3 buttons group, and finally by the Group 4 check boxes. The value returned from the group 1->3 buttons group will range from 1--> 6, while the value returned from the Group 4 check box group will be a bit or'ed value, with the first check box having a value of 1 , thr second check box having a value of 2 , etc..

An example from the shell script file AllPassRec is as follows:

```
char *Labels[] = { "Recursive All-pass Filter Design",
"Prototype Order (Odd)", "Stopband Edge (.25->.5)",
"Passband Edge","Center Frequency","Order (2 or 4)",
"Low-pass", "Band-Pass","High-Pass", "Stop Band" };
int button[4]; // storage for group button entry values
vector R; // storage for data entry values
char FileName[81];// storage for file path and name
R = dialog(5, 4, 0, 0, 0, 0, Labels, button, FileName);
N = R[0]; // value from 1st data entry field
ws = R[1]; // value from 2nd data entry field
wb = R[2]; // value from 3nd data entry field
```

```
wc = R[3]; // value from 4th data entry field
kk = R[4]; // value from 5th data entry field
type = button[0]; // value from Group 1 button group
// button[1] would be value from Group 2 button group
// button[2] would be value from Group 3 button group
// button[3] would be value from Group 4 button group
// Filename[81] would be file name if nf = 1
```

Note that the appearance of the dialog will depend on the amount of input parameters and flags requested. Up to 48 different dialog templates are available, and are selected at run time. The dialog for the above shell script AllPassRec.txt will appear as follows:


Figure 7.1
Note that the Help button has been grayed out. If the user had prepared a help file (in the same subdirectory) with the name of AllPassRec.doc, then the Help button would be active, and the user would be able to call up and display help information by depressing that button. Also note that this procedure keeps track of input values from a given design session to the next, by storing these values in file AllPassRec.ini.

Another dialog procedure at the user level is the IIR Digital Filter Design module. This module can be launched from Slide-Rule's Main Menu, MyDialogs/IIR Digital Filter Design..., or by executing the shell script file iirDFD.txt in sub-directory digitalF. The dialog for this procedure appears as follows:


Figure 7.2

## User Menus

Referring to Figure 2.1, we note that on the Menu Bar, there are tile entries Dialogs, Plotting, FilterDesign, Math, etc... These are user generated menus that allow the user to attach shell script files on a popup menu that can be either launched into the Compiler/Interpreter for execution, or into the Custom editor for editing and subsequent execution. Up to $\mathbf{1 2}$ main Menu Title entries are allowed, and a total of up to $\mathbf{3 0 0}$ user shell scripts files may be attached to any of the user defined Main Menu Tiles. This allows the user an alternate and quick method to access shell script procedures as apposed to launching these scripts by going through a file dialog procedure from the main Shell Menu procedure as covered in Chapter 2. These entries are attached to the main menu bar, when Slide-Rule is first executed. The user may create a new Menu file or select an existing menu file by bringing up the dialog procedure User Menus Selection... under the Main Menu Tile ConFig. This dialog procedure is shown in Figure 7.3 below.


Figure 7.3
Note in Figure 7.3, that the current User Menu selected is Demos.mnu, a file stored in the root directory ( $\mathrm{c}: \backslash$ SlideRule). In order to select a different set of menu tiles, depress the Click for File Menu Selection push button. Then make a file selection under the entries of *.mnu. After making a file selection, the dialog procedure is terminated, and the new user selected menu will take effect.

To create a new menu specification file, fill in the dialog entry below the Current User Menu Selected, then click the New Menu File push button. Then click or depress the Quit pushbutton and add new entries as described below.

To add or modify a menu specification file (*.mnu), use the Open File editor as found under the File tile, and select Files of type Menu File (*.mnu). An example specification file follows:

```
"MyDialogs" "Cubic Spline Interpolation..." "C:\SlideRule\other\splineDialog.txt"
"MyDemos" "Slide-Rule - Front Cover" "C:\SlideRule\plots\hills.txt"
"MyDemos" "Ford Circles Demo" "C:\SlideRule\plots\FORDCIRC.TXT"
"MyDemos" "Vector Plot" "C:\SlideRule\plots\vector.txt"
"MyDemos" "Polar Plots" "C:\SlideRule\plots\apolar.txt"
"MyDemos" "Hogenauer Filter Plots" "C:\SlideRule\plots\cubicX.txt"
"MyDemos" "More Polar Plots" "C:\SlideRule\plots\polar1.txt"
"MyDemos" "Symbols Plot" "C:\SlideRule\plots\SYMBOLS.TXT"
"MyDemos" "Fractal Plot" "C:\SlideRule\plots\fern.txt"
"MyDemos" "Jacobian Plot" "C:\SlideRule\plots\jacob2.txt"
"MyDemos" "Bessel Functions" "C:\SlideRule\plots\besseljn.txt"
"MyScripts" "Sample 2nd Order ODE" "C:\SlideRule\diffeq\sysode.txt"
"MyScripts" "Runge-Kutta-Fehlberg ODE" "C:\SlideRule\diffeq\rkf45Test.txt"
"MyScripts" "Solution to Van der Pol's Equation" "C:\SlideRule\diffeq\VanderPol.txt"
"MyDemos" "Histogram Plot" "C:\SlideRule\plots\histo.txt"
"MyDemos" "Interpolated FIR (IFIR)" "C:\SlideRule\digitalf\INTERP.TXT"
"MyDemos" "FFT Window Functions" "C:\SlideRule\specanal\FFTWIN.TXT"
"MyDemos" "Demo of Circular Functions" "C:\SlideRule\mathfunc\cirfuncs.txt"
"MyDemos" "Demo of Hyperbolic Functions" "C:\SlideRule\mathfunc\hyperbolicfuncs.txt"
"MyDemos" "Demo of log10 function" "C:\SlideRule\mathfunc\log10.txt"
"MyDemos" "Log & Exp functions" "C:\SlideRule\mathfunc\logexpfunc.txt"
"MyDemos" "Some Sin Waves" "C:\SlideRule\mathfunc\SCURVES.TXT"
"MyDemos" "Power curves" "C:\SlideRule\mathfunc\powfunc.txt"
"MyDemos" "ODE error analysis" "C:\SlideRule\diffeq\ODEsTest.txt"
"MyDemos2" "Meteor FIR Filter Design" "C:\SlideRule\digitalf\MD003.TXT"
"MyDemos2" "CIC Comb Impulse Response" "C:\SlideRule\digitalf\trunc.txt"
"MyDemos2" "Movie Plot" "C:\SlideRule\plots\moviep.txt"
"MyDemos2" "3D Line plots" "C:\SlideRule\plots\plot3c1.txt"
"MyDemos2" "Plot using pminmax" "C:\SlideRule\plots\PMINMAX.TXT"
"MyDemos2" "Polynomial fit to data" "C:\SlideRule\plots\POLY.TXT"
"MyDemos2" "MATLAB cover" "C:\SlideRule\plots\membrane.txt"
"MyDemos2" "3D - demo" "C:\SlideRule\plots\orbits.txt"
"MyDemos2" "3D - demo plot" "C:\SlideRule\plots\tampani.txt"
"MyDemos2" "Some Polar plots" "C:\SlideRule\plots\polar5.txt"
"MyDemos2" "Bessel functions I0,I1,K0,K1" "C:\SlideRule\specmath\BESSELIN.TXT"
"MyDemos2" "Bessel function Jn" "C:\SlideRule\specmath\BESSELYN.TXT"
"MyDemos2" "Beta function" "C:\SlideRule\specmath\BETA.TXT"
"MyDemos2" "Incomplete Beta function" "C:\SlideRule\specmath\BETAI.TXT"
"MyDemos2" "Chebyshev polynomials" "C:\SlideRule\specmath\chebyshevpoly.txt"
```

```
1 "MyDemos2" "Elliptic Integral 1st kind" "C:\SlideRule\specmath\ELLIPTC.TXT"
1 "MyDemos2" "Jocobian Elliptic function" "C:\SlideRule\specmath\ELLIPTJN.TXT"
1 "MyDemos2" "Elliptic Integral 2nd kind" "C:\SlideRule\specmath\ELLIPTS.TXT"
1 "MyDemos2" "Plots of error function - erf" "C:\SlideRule\specmath\ERF.TXT"
1 "MyDemos2" "plot of Gamma function" "C:\SlideRule\specmath\GAMMA.TXT"
1 "MyDemos2" "Plot of Incimplete Gamma function" "C:\SlideRule\specmath\GAMMAP.TXT"
1 "MyDemos2" "Jacobi Orthogonal Polynomials" "C:\SlideRule\specmath\jacobipoly.txt"
1 "MyDemos2" "Legendre Orthogonal Polynomials" "C:\SlideRule\specmath\legendrepoly.txt"
1 "MyDemos2" "Another Movie Plot" "C:\SlideRule\digitalf\HARRIS3.TXT"
```

Note that column one specifies the execution option, i.e., " 1 " to execute, and " 0 " to launch into the Custom Editor. The second entry is the Menu Tile label, while the third entry is the Popup Menu Tile label. The last entry is the full path and file name of the shell script file. Note that the entries can be in any order, except the menu tile names are ordered from left to right as "first come, first selected". After editing is complete, go to the User Menu Selection dialog procedure under the Config tile, and select the specification file.

## Chapter 8 - Digital Filter Design

## Introduction

Digital filters can be considered the backbone of Signal Processing, and as such this chapter will go through several design examples for the different type of digital filters that can be can implemented in this software.

## FIR Filter design using the Parks-McCellan Design Procedure*

The Parks-McCellan algorithm was developed by Parks and McCellan at Rice University, and is based on minimizing the error of the alternation theorem in the Chebyshev sense. A large part of the algorithm was based on a paper (in 1956) by Russian mathematician Remez. Accordingly, the algorithm converges to produces equal ripple in the pass band and stop band. For this software implementation, the Fortran software was downloaded from Netlib, and converted to C. Modifications were made to handle up to 2000 taps; an input parameter to taper the stop band in db/octave; and a parameter to up tilt the pass band with $(\sin (\mathrm{x}) / \mathrm{x})^{N}$, such that the filter can be convolved with a Cascaded Integrated Comb (CIC) filter. Accordingly, we continue with a few specific design examples.

Design a low pass FIR filter with a pass band edge at 1000 Hz , a stop band edge at 1500 Hz , a pass band ripple of 0.25 dB , a stop band rejection of 60 dB with a -4.5 dB rolloff, and a sampling frequency of 10000 Hz . Before we start, we need to check if were in the correct sub-directory by checking the info tile under the File tile. Having done that, first go to the Filter tile, then select Parks-McCellan... and Estimate..., and observe the following dialog box.


[^0]Upon selecting the Okay button, were presented with a second dialog box as show below.


After filing in the stop band taper at -4.5 dB , were presented with four plots, namely

1) A magnitude plot
2) A magnitude $d B$ plot
3) An Impulse response plot
4) A Step response plot


From the Magnitude dB plot, we notice by doing a zoom on the first side lobe, that it's about 0.7 dB short of the -60 dB stop band spec. So we can go straight to the Parks-McCellan... Design... dialog and add say a couple of taps ( $51->53$ ) to redo the design. We can also modify the stop band taper if desired. Note that the design parameters are remembered from the previous iteration. Note also that the shell script file name can also be changed. If we rubber band the pass band a couple of times, we note that the pass band ripple spec is met, and that at 1000 Hz , the reading is down to less than -0.125 dB . Note under the Filter tile, that the entry FIR Nominal gain of 1 is checked. This specifies that the ripple in the pass band is equally above and below zero dB . This can be changed to FIR Maximum gain of 1, i.e., zero dB , such that the ripple doesn't poke its head above that level. You can change to this setting by checking the tile and re-run the shell script to verify. Now if we depress the function F12 key, will get a print out as shown below.
FINITE IMPULSE RESPONSE (FIR)
LINEAR PHASE DIGITAL FILTER DESIGN
REMEZ EXCHANGE ALGORITHM
PASSBAND/STOPBAND
FILTER LENGTH $=53$
$* * * * * *$ IMPULSE RESPONSE $* * * * * *$
$* * * * * *$
H(z) $=\mathrm{B}(z)$
$\mathrm{B}(00 *)=-3.87110317 \mathrm{e}-04=\mathrm{B}(052)$
$\mathrm{B}(001)=-1.77492760 \mathrm{e}-03=\mathrm{B}(051)$
$\mathrm{B}(002)=-3.09713718 \mathrm{e}-03=\mathrm{B}(050)$
$\mathrm{B}(003)=-3.91905708 \mathrm{e}-03=\mathrm{B}(049)$
$\mathrm{B}(004)=-3.15436826 \mathrm{e}-03=\mathrm{B}(048)$
$\mathrm{B}(005)=-4.42114878 \mathrm{e}-04=\mathrm{B}(047)$
$\mathrm{B}(006)=$
$\mathrm{B}(007)=3.42777324 \mathrm{e}-03=\mathrm{B}(046)$
$\mathrm{B}(008)=$
$\mathrm{B}(009)=$

|  | e-03 $=B(042)$ |
| :---: | :---: |
| B (011) | $-1.14302503 \mathrm{e}-02=B(041)$ |
| $B(012)$ | $-1.27814534 \mathrm{e}-02=B(040)$ |
| B(013) | $-6.33405934 \mathrm{e}-03=B(039)$ |
| B (014) | $6.36904278 \mathrm{e}-03=\mathrm{B}(038)$ |
| B (015) | $1.92257202 e-02=B(037)$ |
| $B(016)$ | $2.39819763 \mathrm{e}-02=B(036)$ |
| $B(017)$ | $1.47471893 \mathrm{e}-02=B(035)$ |
| $B(018)$ | $-7.57982579 \mathrm{e}-03=B(034)$ |
| B(019) | $-3.35928232 \mathrm{e}-02=\mathrm{B}(033)$ |
| $B(020)$ | $-4.80220045 \mathrm{e}-02=B(032)$ |
| B (021) | $-3.61987984 \mathrm{e}-02=B(031)$ |
| $B(022)$ | $8.38558745 \mathrm{e}-03=B(030)$ |
| B (023) | $7.92145124 \mathrm{e}-02=B(029)$ |
| B (024) | $1.57422083 \mathrm{e}-01=B(028)$ |
| B (025) | $2.18347721 \mathrm{e}-01=B(027)$ |
| B (026) | $2.41331305 e-01=B(026)$ |


|  | Freq. F1 | Freq. F2 | Desired | Weight |
| :--- | :--- | :--- | :--- | :---: |
| BAND 01 | 0.000000 | 0.100000 | 1.000000 | 1.000000 |
| BAND 02 | 0.150000 | 0.500000 | 0.000000 | 14.400000 |

Now if we want to get the filter coefficients out for assembly for one's signal processor or VLSI chip, we select Coefficient File Gen... -> FIR... under the Filter tile, and are presented with the following dialog box.


Assuming we want floating point, we first load the FIR coefficients (which will have the same file name as the shell script file, but with a .fir extension), then we enter the header and trailer. Say we want each line to have 6 spaces followed by .float, and the trailer to be blank. This produces the following in Notepad, where the user can paste this code in their assembly code.

```
.float -3.87110317e-04,-1.77492760e-03,-3.09713718e-03,-3.91905708e-03, -3.15436826e-03
.float -4.42114878e-04, 3.42777324e-03, 6.49455149e-03, 6.48010009e-03, 2.26297850e-03
.float -4.92352995e-03,-1.14302503e-02,-1.27814534e-02,-6.33405934e-03, 6.36904278e-03
.float 1.92257202e-02, 2.39819763e-02, 1.47471893e-02, -7.57982579e-03, -3.35928232e-02
```

```
.float -4.80220045e-02, -3.61987984e-02, 8.38558745e-03, 7.92145124e-02, 1.57422083e-01
.float 2.18347721e-01, 2.41331305e-01, 2.18347721e-01, 1.57422083e-01, 7.92145124e-02
.float 8.38558745e-03, -3.61987984e-02, -4.80220045e-02, -3.35928232e-02, -7.57982579e-03
.float 1.47471893e-02, 2.39819763e-02, 1.92257202e-02, 6.36904278e-03, -6.33405934e-03
.float -1.27814534e-02, -1.14302503e-02, -4.92352995e-03, 2.26297850e-03, 6.48010009e-03
.float 6.49455149e-03, 3.42777324e-03, -4.42114878e-04, -3.15436826e-03, -3.91905708e-03
.float -3.09713718e-03, -1.77492760e-03, -3.87110317e-04
```

For fixed binary with a 16 -bit two's complement word, then the most significant bit (15) is -1 , bit $14=.5$, etc... Since the largest tap is 0.241331395 , if we multiply by 32768 and convert to hex with rounding, we get $0 x 1$ ee 4 as shown below.

| 0xfff3, | 0xffc6, | 0xff9b, | 0xff80, | 0xff99, | 0xfff2, | 0x0070, | 0x00d5 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| , word 0x00d4, | 0x004a, | 0xff5f, | 0xfe89, | 0xfe5d, | 0xff30, | 0x00d1, | 0x0276 |
| , word 0x0312, | 0x01e3, | 0xff08, | 0xfbb3, | 0xf9da, | 0xfb5e, | 0x0113, | 0x0a24 |
| , word 0x1426, | 0x1bf3, | 0x1ee4, | 0x1bf3, | 0x1426, | 0x0a24, | 0x0113, | 0xfb5e |
| , word 0xf9da, | 0xfbb3, | 0xff08, | 0x01e3, | 0x0312, | 0x0276, | 0x00d1, | 0xff30 |
| ,word 0xfe5d, | 0xfe89, | 0xff5f, | 0x004a, | 0x00d4, | 0x00d5, | 0x0070, | 0xfff2 |
| d 0xff99, | 0xff80, | 0xff9b, | 0xffc6 | 0xfff3 |  |  |  |

To get more precession for a fixed point FIR filter, we select Fixed Full. Since the center tap is $=$ 0.241331395 , we divide into 0.5 to get 2.07184061 . We multiply all the coefficients by this factor to get the center tap $=0 \times 7$ fffe which is $\sim 0.5$. We see the unity gain scale factor that needs to be applied after the FIR filter is run, is 1 ee 3 or 0.241331
// Scale Up Factor $=0.241331$
.data 0xffcb, 0xff0f, 0xfe5b, 0xfdec, 0xfe54, 0xffc4, 0x01d1, 0x0372
.data 0x0370, 0x0133, 0xfd64, 0xf9f0, 0xf939, 0xfca4, 0x0361, 0x0a32
.data 0x0cb8, 0x07d2, 0xfbfb, 0xee2f, 0xe688, 0xeccd, 0x0473, 0x2a03
.data $0 x 537 d, 0 x 73 c d, 0 x 7 f f e, ~ 0 x 73 c d, ~ 0 x 537 d, ~ 0 x 2 a 03, ~ 0 x 0473, ~ 0 x e c c d$
.data 0xe688, 0xee2f, 0xfbfb, 0x07d2, 0x0cb8, 0x0a32, 0x0361, 0xfca4
.data 0xf939, 0xf9f0, 0xfd64, 0x0133, 0x0370, 0x0372, 0x01d1, 0xffc4
.data 0xfe54, 0xfdec, 0xfe5b, 0xff0f, 0xffcb
.data 0x00001ee3 // Unity Gain Scale Factor at B15

## FIR Half-band Filters*

Half-band filters are restricted to having their impulse response to have all but one of the even indexed samples equal to zero. This design procedure employs the technique as stated in the footnote reference to force the odd taps to zero. So we will design a half-band filter with a cutoff at 2000 Hz , a stop band rejection of 60 dB , and a sampling rate of 10000 Hz . We should see an odd number of taps; a magnitude dB value of -6 dB at the quarter sampling rate $(2500 \mathrm{~Hz})$; the center tap impulse response of 0.5 ; and all the odd impulse response coefficients (or taps) equal to zero. The advantage of Half-band filters is that we can design filter structures that are fast, since every other tap is zero except one. Below are some plots for this design example.

[^1]

## The Hilbert Transformer*

[^2]A Hilbert transformer can be considered as a filter with unity gain and a phase shift of approximately 90 degrees at all frequencies. These filters are useful in processing narrow band signals where a complex quadrature component of a real signal shifted 90 degrees from the real input signal is required. Routine hilbert.txt as found in the specanal sub-directory illustrates this technique. Below is a block diagram of the system.


We note that for a Hilbert Transformer of N Taps, that the delay is ( $\mathrm{N}-1$ )/2 taps. Therefore the tap delay line in the top block is set at 26 taps. We then program a shell script to generate a sin wave with shell script signal.txt and output as signal.dat. This signal has the following properties: Frequency $=1000 \mathrm{~Hz}$; Sample Freq. $=24000 \mathrm{~Hz}$; DC offset $=0$; Peak Amp. $=1$; Phase offset $=0$ degrees. We then design a Hilbert transformer under Parks-McCellan with a sample frequency of 24000 Hz , a stop band rejection of 70 dB , and a pass band cutoff of 1000 Hz . We generate another signal (signald.txt) as above except we want the signal delayed 26 samples which works out to a phase shift of 345 degrees. This will emulate the tap delay line in the upper block. Go execute the shell script hilbert.txt in sub-directory specanal and observe the plots as shown below.



## FIR Differentiators*

The FM signal we receive on our car radio can be characterized by the following equation:
$F M_{\text {SIG }}=A_{C} \cos \left(w_{c} t+\beta w(t)\right)$
where,
$A_{c}=$ Amplitude Factor
$\mathrm{w}_{\mathrm{c}}=$ carrier frequency $(\mathrm{rad} / \mathrm{s})$
$\beta=$ beta factor
$\mathrm{w}(\mathrm{t})=$ original signal
We want to develop an example that illustrates the use of the Differentiator in the Parks McCellan FIR filter design procedure. In a typical FM receiver, we heterodyne the signal down to an intermediate frequency or IF frequency. At this point, let's insert a digital demodulator which will consist of an A/D sample and hold circuit, a 16-bit imbedded signal processor, and a DAC circuit. A typical block diagram of the system is as follows:

[^3]

Now, lets insert an anti-aliasing filter (band pass) after the 456 kHz signal and before the $\mathrm{A} / \mathrm{D}$ sample and hold. Now if we sample the signal at 96 KHz , then we can say that the signal is at 24 KHz ( $5^{*} 96-456$ ). In this operation, will employ at trick, i.e., we will sample at 96 KHz , and run a quadrature down sampling filter, such that we produce I and Q samples at base band (at 24 KHz samples/sec.). The original 456 KHz signal will be centered at DC. In a real world implementation, this would be followed by a Low Pass filter to remove noise and spurious signals. At this point we need to demodulate the signal. Recall that,

$$
\begin{aligned}
& \mathrm{FM}_{\mathrm{SIG}}=d / d t(\Omega) \\
& \text { where, } \Omega=\tan ^{-1}(\mathrm{Q} / \mathrm{I}) \\
& \text { therefore } \mathrm{FM}_{\mathrm{SIG}}=\frac{\mathrm{IdQ} / \mathrm{dt}-\mathrm{QdI} / \mathrm{dt}}{\mathrm{I}^{2}+\mathrm{Q}^{2}}
\end{aligned}
$$

The $\mathrm{dQ} / \mathrm{dt}$ and $\mathrm{dI} / \mathrm{dt}$ terms in the above equation can be implemented by passing the I and Q data streams through a differentiator filter as designed under the Parks McCellan FIR filter design method. The basis for these (identical) filters is: type is differentiator; cutoff frequency is 8 KHz ; sample frequency is 24 $\mathrm{KHz}, \%$ peak relative error of 0.1 . The characteristics of this filter are shown below.


Since the delay of the differentiator filter is $15 / 2$, or 7 taps, then the center Tap of each filter is time wise in sync with the output of the filter. The block diagram of the FM demodulator is as follows:


Below is a plot of the demodulated test signal along with a magnitude plot of this signal in the frequency domain.


## MaxFlat FIR Design

The MaxFlat FIR filter design procedure can be found under the Filter Tile under the main menu. The reference for this technique can be found in [5]. The design technique gives coefficients for a maximallyflat pass and stop band symmetric FIR low pass filter with an odd number of terms. Find below plots as an example of the output of this design procedure.


## Windowed FIR Design

The FIR windowed design procedure is the classic design technique that is covered in every introductory course in signal processing. It comes with an estimate dialog procedure. This procedure includes Low pass, High pass, Band pass, and Band stop designs, along with window types of Rectangular, Tapered Rect., Triangular, Hanning, Hamming, Blackman, and Kaiser. Find below plots as an example of the output of this design procedure. Note, that the design parameters are: Passband ripple $=0.2 \mathrm{~dB}$; Stopband Rej. $=50 \mathrm{~dB}$; Passband $($ Ideal $)=1000 \mathrm{~Hz}$; Stopband $($ Ideal $)=2000 \mathrm{~Hz}$; and a sample frequency of 10000 Hz . Note that the Filter is down -6 dB at 1500 Hz , which is the midpoint between the ideal passband specification and the stopband specification. Consult the Function reference (F7->Spectral Analysis-> kaiserWin) for documentation.



## IIR Filter Design*

The IIR design procedure is a classic IIR design procedure than employs the bilinear transform to transform analog prototypes in the s-domain to the z-domain. This procedure covers Low pass, High pass, Band pass, and Band stop designs of the Chebyshev I, Chebyshev II, Butterworth, and Elliptic types. The procedure covers only cascaded bi-quad sections. Note that on any given design, the gain in each bi-quad is adjusted such that the maximum gain at any frequency is 1 . The bi-quad sections are also ordered form least Q to increasing Q per filter stage. Also, in any stage, the zeros of the filter are selected to be closest to the poles. The effect of this ordering is to provide the maximum gain and stability to fixed point designs. Included in this procedure is a check box for adding an all-pass equalizer such that the group delay in the pass band (low pass designs), is flattened out. This technique was implemented from A.G. Deczky's paper from reference [5]. Note that there is no estimator dialog procedure. To increase or decrease the gain of the filter, simply raise or lower the number of bi-quad sections. Find below plots of a butterworth low pass design ( 3 bi-quads) with 3 all-pass bi-quad sections added to smooth out the group delay. Also shown is the group delay without the all-pass equalizer. Note that the Fortran code (for the all pass filters) of reference [5], was downloaded from Netlib, converted to C-code, and modified and striped to integrate with this software.


[^4]

## Interpolated FIR (IFIR) Filters*

Under the Filter tile on the main menu, we have a design procedure for designing Cascaded Interpolated FIR filters (IFIR). The rationale for this is to design a first stage transversal filter, where each delay element $\mathrm{z}^{-1}$ is replaced by k delay elements, equivalent to $\mathrm{z}^{-\mathrm{k}}$ as shown in the block diagram below. This is equivalent to inserting $k-1$ zeros between the original filter coefficients of $F(z)$. Since we're now violating Nyquist, we have k-1 unwanted images. We can solve this problem by following the first filter $\mathrm{F}\left(\mathrm{z}^{-k}\right)$ by a second filter $\mathrm{H}(\mathrm{z})$, to filter out the un-wanted images. We do this by incorporating the meteor FIR design module which will be covered in another section. Please note that each filter is run at the Nyquist rate. Note the tap delay line of F is k times the filter length of F , but when we run this filter, we grab every $\mathrm{k}^{\text {th }}$ sample in the tap delay line. Note that the SOFTWARE figures out the inputs to the Meteor FIR design program. Note that if there is ripple is the interpolating filter, that this can cause a problem when we convolve it with the zero inserted model filter.


In order to illustrate this technique, we'll design a band pass filter with the following specifications: Pass band ripple $=0.5 \mathrm{~dB}$; Stop band rejection $=45 \mathrm{~dB}$; Stop band edge $=52.1 \mathrm{KHz}$; Pass band edge $=52.6 \mathrm{~dB}$; Pass band width $=1 \mathrm{KHz}$; Sample frequency $=200 \mathrm{KHz}$; Interpolating factor equals 8 .

[^5]

## Raised Cosine Filter Design

In communication systems one frequently used $\mathrm{H}(\mathrm{n})$ transfer function belonging to the Nyquist class (zero ISI at the sampling times) is called the raised-cosine filter. For a reference, refer to [10], pp. 139-142. Two types of FIR design methods are offered, namely the classic formula method, and the 6 -dB harris method. The harris technique will be covered in the Root Raised Cosine Filter Design section. In order to design this classic pulse shaping filter, we need the symbol rate, the sample rate, the roll-off factor, and the number of taps. The sample rate is generally four (4) or eight (8) times the symbol rate, the roll-off factor varies from $0<=1$, and the number of taps is always odd and a multiple of the taps per pulse width -1 (the user picks this value). Below are a couple of plots showing the output of this design technique. Note that the impulse response of this filter has unity gain. When used as shaping and up sampling filter, as implemented in a transmitter, we re-scale these coefficients to have unity peak gain (i.e., we re-scale the coefficients such that the largest coefficient has a value of 1 . Refer to [6], p 89.


## Root Raised Cosine Filter Design

The SQRT-Nyquist filter is used in communication systems, usually one at the transmitter and one at the receiver. If we convolve two of these filters (one a copy of the other), then we have the Raised Cosine filter as discussed in the previous section. Two types of FIR design methods are offered, namely the classic formula method, and the $3-\mathrm{dB}$ harris method (refer to [6], pp. 91-97). The harris method, as covered in [6], basically iterates on the Remez (Parks McCellan) algorithm to transform an initial low pass filter to the SQRT-Nyquist spectrum with the specified roll-off while preserving the specified pass band and stop band ripple. Below is an example of this ingenious design method.


## Meteor Filter Design

The Meteor FIR design program is based on finding a filter which satisfies upper and lower limit specifications, and then finding the shortest filter length which allows the constraints to be met, and then finding the filter of that order which is farthest from the upper and lower constraint boundaries in a minmax sense. The simplex algorithm for linear programming is used to find the best linear-phase FIR filter of minimum length, as well as to find the minimum feasible length itself. Refer to reference [9].

The first specification for the user upon being presented the Meteor dialog box procedure is to fill in the shell script name. The next item is the filter sample rate entered in KHz. The user then selects the even radio button for an even number of coefficients, else the odd radio button for an odd number of Taps. Meteor will search from between 256 TAPS (even) or 255 TAPS (odd), down to 2 or 1 TAPS for the best fit to the specifications. This is followed by selecting either the Symmetric radio button (symmetric coefficients, cosine model), or Non-Symmetric (odd symmetric coefficients, sine model). The symmetric option should be used for most designs, while the non-symmetric option should be used for differentiator designs. The next specification can be of two types, namely limit specifications, or concavity specifications. Limit specifications have two parts, an upper(+) spec., and a lower(-) spec. For now, let's design a simple low pass filter with the following specs:

Shell Script Name = MD001; Taps to be even; symmetric design; 8 KHz sample rate; 2 limit specs; Pass band from 0 to 2 KHz ; stop band from 2.5 KHz to 4 KHz ; ripple in pass band from 1.05 to 0.95 on linear scale; a 60 dB stop band attenuation; Note that when we say not-hugged, we mean to get as close as possible, but not necessary touching that spec, while hugged means right on. Notice also, that in the second specification (starting in the 3rd row, that we specify the upper bound as a $\log$ value ( -60 dB ), while the lower bound is linear (0)).

| limit <br> spec | concavity <br> spec | log | linear | not- <br> hugged | hugged | Freq. <br> Lower | Freq. <br> Upper | Attn. <br> Lower | Attn. <br> Upper |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| x |  |  | x | x |  | 0 | 2 | 1.05 | 1.05 |
| x |  |  | x | x |  | 0 | 2 | .95 | .95 |
| x |  | x | x | x |  | 2.5 | 4 | -60 | -60 |
| x |  |  | x | x |  | 2.5 | 4 | 0 | 0 |

We now open up the Meteor Dialog procedure, and enter the Shell Script Name, the sample rate of 8 KHz , symmetric design, even number of coefficients, and limit spec. We then enter the first limit spec which consists of the first two rows of the table above. At this point we click the Add Spec Tile, and note that the Total Specs window is 1, the Spec Number is 1, and the Specification File title "Click for File Input" has changed to "MD001.dat". At this point we have a specification file on disk minus the last limit spec. We now enter the last two rows in the table above, and again click the Add Spec Tile. We now have a complete specification file and are ready to proceed with the design by hitting the Okay button, except for checking the specification file for accuracy. Depress the down scroll button next to the Spec Number to display the first limit spec. Check against the table including log, linear, not-hugged, and hugged. If an error, correct, then click the Modify Tile. Then repeat procedure on limit spec number 2 by depressing the up scroll bar (next to Spec number). When every thing looks AOK, depress the Okay button to start the design. Below are some plots showing the some of the output plots produced. Please note that the Shell Script file name is MD001.txt, the Specification file is MD001.dat, the print output file is MD001.prn, and the file containing the FIR coefficients is MD001.fir. Please note that the FIR coefficients file MD001.fir can be formatted to an ASCII file suitable for inclusion into a file for the target CPU or VLSI chip design by using the Coefficient File Gen... dialog procedure.


We now want to repeat this design, except that in the pass band we want a concave downward smooth slope and passing through -3 dB at the corner frequency. To do this, we open up the Meteor Dialog procedure, click on the Specification File Tile, and select MD001.dat, and note that everything is filled in as per the above design. However, we don't want to modify that design. So change the Shell Script name to MD002, and depress the Modify Spec Tile. This will create a new specification file, namely MD002.dat that we can modify and add to. The specs for this new design are listed in the table below.

| limit <br> spec | concavity <br> spec | log | linear | not- <br> hugged | hugged | Freq. <br> Lower | Freq. <br> Upper | Attn. <br> Lower | Attn. <br> Upper |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| x |  |  | x | x |  | 0 | 2 | 1.0 | 1.0 |
| x |  | x |  | x |  | 0 | 2 | -3 | -3 |
| x |  | x |  | x |  | 2.5 | 4 | -60 | -60 |
| x |  |  | x | x |  | 2.5 | 4 | 0 | 0 |
|  | x | $\mathrm{N} . A$. | N.A. | N.A. | N.A. | N.A. | N.A. | N.A. | N.A. |
|  | x | N.A. | N.A. | N.A. | N.A. | 0 | 2 | N.A. | N.A. |

We note that the first spec has changed from MD001.dat, so we modify and click on the Modify Spec Tile to change the specification file. Now, while still on spec \#1, click on the Concavity radio button. Since the Freq. Lower and the Freq. Upper are already set, just click on the Add Spec Tile. The other parameters are ignored as listed in the specification table above. We now have our required 3 specifications. Check by moving up and down with the scroll buttons as before. If ok, click on the Okay Tile to start execution of the design.

For a third design, we modify MD002 to MD003 as above, but add a fourth specification to add a zero at a frequency of 3 KHz . The specification in shown in the table below along with plots showing the output from these last two design examples.

| limit spec | concavity spec | $\log$ | linear | not- <br> hugged | hugged | Freq. Lower | Freq. Upper | Attn. Lower | Attn. Upper |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| x |  |  | X | X |  | 0 | 2 | 1.0 | 1.0 |
| X |  | X |  | X |  | 0 | 2 | -3 | -3 |
| x |  | X |  | X |  | 2.5 | 4 | -60 | -60 |
| X |  |  | X | X |  | 2.5 | 4 | 0 | 0 |
|  | X | N.A. | N.A. | N.A. | N.A. | N.A. | N.A. | N.A. | N.A. |
|  | X | N.A. | N.A. | N.A. | N.A. | 0 | 2 | N.A. | N.A. |
| x |  |  | X | X |  | 3 | 3 | 0 | 0 |
| X |  |  | X | X |  | 3 | 3 | 0 | 0 |




## Hogenauer Filter Design*

## Introduction

The Hogenauer Filter is used for large up-sampling or down sampling tasks. Its advantage is it has no multiplies, however it has to be implemented in fixed point arithmetic (as explained in the references). Because of the large up-sampling or down sampling ratios, it's normally implemented in digital hardware. Since the logic design of the hardware implementation is well known, the purpose of this design function is to determine; (1) the size of each of the integrator and comb filter registers; (2) and the number of bits that can safely be pruned out from a given register. These two outputs are determined from the following inputs, namely,

1) Whether an up-sampling filter or a down sampling filter.
2) The up-sampling or down sampling ratio.
3) The number of cascaded stages
4) The bit width of the input to the filter (to the LSB side of the filter).
5) The bit width of the output of the filter (from the MSB side).

Based on the above, the outputs are determined in the dialog box without ever pushing the Okay button. A section of inputs ( 5 x ) has to do with a transversal FIR filter, that would be designed under the Filter Menu tile using the Parks-McCellan FIR filter design module. This filter is a compensating filter, in that it allows one modify the pass band with a $(\sin (\mathrm{x}) / \mathrm{x})^{\mathrm{K}}$ uptilt, where K is the number of stages in the CIC or Hogenauer Filter. This compensation allows us to generate a relative flat pass band. In addition, this filter can also be an up-sampler or down-sampling filter. The purpose of these inputs is to allow an overlay plot of the CIC filter with the FIR filter. Included in the plots when the Okay button is depressed are,
(1) Overlay of FIR and CIC in magnitude dB.
(2) A pole-zero plot.
(3) A magnitude plot of each of the integrator outputs.
(4) A magnitude plot of each of the comb register outputs
(5) A magnitude plot of the filtered output.

[^6]Note that the outputs from the integrator and comb resisters and the filtered output plot at the designated bit width are done in a fixed point simulation, such that if a given register is too small, then it will show up in the output plots.

The design module also allows one to select one of seven inputs to the simulated filter, namely.
(1) A sin wave at the level of the input register to the filter
(2) An impulse.
(3) A plus step
(4) A plus minus step.
(5) A plus minus plus step.
(6) A plus minus plus minus step.
(7) A plus minus plus minus plus step

Below is the dialog box that the user is presented with when selecting the Hogenuer Filter... tile under the Filter tile on the main menu.


The dialog procedure for this design requires the following entries:
Shell Script File. Enter a file name for this design procedure. Note! Important!! This value is remembered between design sessions so a user can easily change individual parameters within an overall design without retyping in this parameter. So if starting a new design, be sure to modify this parameter so as not to wipe out an old design!!
Up Sample/Down Sample check buttons. Click one of these boxes depending if an up-sample filter or a down-sample filter.

Register bit width In. Use the up/down vertical scroll buttons to set the bit-size of the data path into the filter. The range is from 6 to 18 -bits.
Up/Down Count. Use the vertical scroll buttons to set the up or down sample ratio. The range is from 2 to 1000 .
Num. of Sections. Use the vertical scroll buttons to set the number of cascaded sections. The range is from 1 to 5 .
Register bit width out. Set this valve using the vertical scroll buttons. The range is from 8 to 16 bits. This parameter affects the filter output plot.
Comb/Integrator bit widths. These values are calculated when ever a specification is changed in the Specs control group. However, you can change individual values. These valves are remembered between different design sessions. Note! The values displayed are the correct values for a given design. You can change a given value to see what the effect would be, but use the initial values in a real world design!!
Comb/Integrator pruning lengths. These values are set to zero when ever a specification is changed in the Specs control group. If an up sampling CIC, they are set to zero (since there's no pruning on an up sample filter). For a down sampling CIC, they are set to the maximum bit growth of the filter, minus the bit growth of each integrator or comb filter, when the Calculate Pruning Bits button is depressed. These values can be set individually, by entering a value in the appropriate box. These valves are remembered between different design sessions. The bit growth of the filter is given by, $B_{\max }=\operatorname{ceil}(\mathrm{N} \log 2(R M))$, where $\mathrm{N}=$ num. of stages, $\mathrm{R}=$ down sample ratio, $\mathrm{M}=$ diff. comb value.

Signal Input Specs. Check one the four boxes to simulate an input signal to simulate the design.

Passband KHz. The pass band in KHz of the FIR shaping filter. This value is used in the Magnitude dB plot to determine aliasing levels. Note! The FIR shaping filter is designed using the Parks McCellan design module with the pass band up-titled by $\left(\frac{\sin (x)}{x}\right)^{N}$, where N equals the number of cascaded sections of the CIC filter.
Stopband KHz. The stop band in KHz of the FIR shaping filter.
Passband ripple (dB). The pass band ripple of the FIR shaping filter.
Stopband Rejection (dB). The stop band rejection of the FIR shaping filter.
Sample freq. KHz. The sample frequency of the FIR shaping filter.
Calculate Pruning Bits. This button will calculate the pruning lengths for each of the individual integrator and comb filter register lengths as described above.

## Accumulator Bit Widths for a Down-Sampling Filter

The gain through a CIC or Hogenauer filter is given by eq. 11-18 of reference [6],

$$
\begin{aligned}
& \text { Gain }=M^{K} \\
& M=\text { length of comb filters, } \mathrm{K}=\text { number of stages }
\end{aligned}
$$

For a Down-sampling Hogenauer Filter, we can prune out certain register bit lengths from the LSB side of the maximum length register. For a test case, we choose a fourth-stage ( $K=4$ ) Hogenauer Filter with a comb filter length of $20(\mathrm{M}=20)$ From the Noble identify, we can transform a Forth-stage DownSampling Hogenauer filter to a Forth-Stage CIC (Cascaded Integrated Comb) filter as shown in the block diagram below.

## Integrators



From the CIC filter above, we set up a digital filter with the coefficients at each stage given by,

```
vector b, a, B, A;
vector b[] = { .05, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -0.05 };
vector a[] = { 1, -1 };
B = { b, b, b, b }; // four stages, length = 84
A = { a, a, a, a }; // four stages, length = 8
```

Given the gain of the filter as defined above, we first compute the impulse response at each stage of the filter. In going to the next stage, we make the just computed stage a NOP, i.e., we set the last coefficient at that filter stage to zero. For the integrator filters, -1 --> 0 , and for the comb filters, -0.05 --> 0 . The impulse response at each stage is multiplied by the total gain of the filter as stated above. From the Gain equation above, the total gain of the filter is 160000 . Computing the impulse responses, we have plots of the impulse responses at each stage as shown below.






From the above and equation 11.23 of reference [6], we compute the Gain from each stage to the output, and have,
$\begin{array}{llllllll}24785 & 1462.4 & 146.29 & 20 & 8.3066 & 4.4721 & 2.4495 & 1.4142\end{array}$
The equivalent bit growth is calculated as,
$\begin{array}{llllllll}14.6 & 10.5 & 7.2 & 4.3 & 3.1 & 2.2 & 1.3 & 0.5\end{array}$
Adding in the noise contributions from each noise source, from equation 11.26 of reference [6], i.e.,
AdditionalBits $=\log _{2}(2 K)=3$
We have rounding up to the nearest integer,
$\begin{array}{llllllll}18 & 14 & 11 & 8 & 7 & 6 & 5 & 4\end{array}$
For a 16 bit input source, the required registers lengths (in bits) for Integrator-1, Integrator-2, Integrator-3, Integrator-4, Comb-1, Comb-2, Comb-3, AND Comb-4 is,
$\begin{array}{llllllll}34 & 30 & 27 & 24 & 23 & 22 & 21 & 20\end{array}$

## A Typical Schematic for a Down Sampler



For $\mathrm{M}=20 ; \mathrm{K}=4$; and 16 -bit input and output registers, we have integrator 1 at 34 bits, with output register at 16 bits, then we have a growth of $34-16$ bits which is equal to $2^{18}=262144$. Since the maximum value of the sine signal is 32767 , and the total gain is $\mathrm{M}^{\mathrm{K}}=20^{4}$, then the maximum signal at the output register should be $32767 * 160000 / 262144 \sim=20000$. A plot of the sine signal at the output register for this case follows.


Find below a couple of plots of the $4^{\text {th }}$ stage integrator and comb registers.



## Design Exercise

You're given a signal at 100 MHz , and we want to down sample it for modem processing in a high speed signal processor. A CIC filter will be employed in an ASIC, with a 12 -bit A/D input and a 12 -bit digital output. The digital channel in the signal processor to the modem will have a bandwidth of 25 KHz , with a stop band rejection of $\sim 95 \mathrm{~dB}$. From the above, we choose a CIC filter with $\mathrm{K}=4$, and $\mathrm{M}=500$. This means the digital output signal from the CIC filter will be at 200 KHz . So the signal processor needs to further down sample the signal. Note for a bandwidth of 25 KHz , the pass-band edge is $\sim 12.5 \mathrm{KHz}$. So we specify a down sampling ratio of 8 , and employ a FIR polyphase filter.* From the above, we input the parameters into the Hogneauer digital filter dialog as shown below.


[^7]From the above, the bits required from Integ 1, Integ 2, ..Comb1, ...Comb4 are 46,38,30,23,19,18,17,16, Since will be using 4-bit high speed look ahead adders, will clock the A/D input at 100 MHz , and each adder at two 100 MHz clock cycles. The system clock will be a square signal ( $50 \%$ duty cycle), where we clock the input register and each 4-bit adder on the rising edge, and each of the adders on the falling edge (which adds in the carry bit generated from the previous rising clock cycle). We add bits to each register to make them modulo 4 , and the registers lengths are then $48,40,32,24,20,20,20$, and 16 bits. This means a total of 554 -bit fast look adders with a through put rate greater than 100 MHz . The registers, $\mathrm{A} / \mathrm{D}$ and output register are 2-complement. Running the Hogneauer Filter we have some plots as displayed below.



## Filter Response Analysis

Under the previous discussion of IIR Filter Design, we gave an example of a Butterworth design with a pass-band edge of $1000 \mathrm{~Hz} ; 3$ bi-quads; 3 all pass bi-quad to flatten the group delay in the pass band; and a sample frequency of 10000 Hz . Assuming we named the shell script file IIRfilter, the 3 files would be generated in the current sub-directory, namely, IIRfilter.txt (shell file), IIRfilter.prn (print file), and IIRfilter.iir (coefficient output file). Now the IIRfilter.out output file would look like the following:

```
6.14379964e-01 -1.55450492e+00 9.99999956e-01 6.31527333e-01 -1.46960394e+00 9.99999601e-01
6.24159804e-01 -1.156597833+00 9.99999988e-01 6.09096301e-02 1.21819260e-01 6.090963013e-02
6.74552873e-02 1.34910566e-01 6.74552832e-02 8.28825780e-02 1.65765156e-01 8.2882578027e-02
1.00000000e+00 -1.55450499e+00 6.143800020-01 1.00000000e+00 -1.46960461e+00 6.31527602e-01
1.00000000e+00 -1.15659785e+00 6.24159812e-01 1.00000000e+00 -1.03206944e+00 2.75707930e-01
1.00000000e+00 -1.14298045e+00 4.12801593e-01 1.00000000e+00 -1.40438485e+00 7.35915184e-01
```

Where the first 3 rows are all the b coefficients, and the next 3 rows are all the a coefficients. The b coefficients sequence from left to right are $b_{10}, b_{11}, b_{12}, b_{20}, b_{21}, b_{22}, b_{30}, \ldots, b_{50}, b_{51}, b_{52}, b_{60}, b_{61}, b_{62}$. The a coefficients similarly would be $a_{10}, a_{11}, a_{12}, a_{20}, a_{21}, a_{22}, a_{30}, \ldots, a_{50}, a_{51}, a_{52}, a_{60}, a_{61}, a_{62}$. Now if we go to the Coefficient File Gen... dialog procedure for IIR, we get the following dialog procedure.


The file produced from this would be as follows:

```
// Scale Factor at B13
// -A2, B2, -A1, B1, B0, NF
    .data -5033, 8192, 12735, -12735, 5033, 8191
    .data -5173, 8192, 12039, -12039, 5173, }819
    .data -5113, 8192, 9475, -9475, 5113, 8191
    .data -2259, 8192, 8455, 16384, 8192, 498
    .data -3382, 8192, 9363, 16384, 8192, 552
    .data -6029, 8192, 11505, 16384, 8192, 678
```

Note the full scaling and that the normalizing factor ( $\mathrm{NF}=\mathrm{A} 0$ ) to effect unity gain in each bi-quad, and also the ordering of the coefficients. This ordering corresponds with the Regular Direct Form II bi-quad as shown in block diagram below:


We first need to remove the .data portion in the file to make it into an ASCII streams format file (where // at the beginning of a line is skipped). Under the File tile on the main menu and Open File for Editing..., select file IIRfilter.out under file of type *.out and edit out each .data. Then select under the Filter tile, Filter Response Analysis... and Fixed Pt. IIR..., fill in the dialog box, then note the 7 plots and a print out of the poles and zeros of the digital filter, which corresponds to the previous example. This is quite useful when one is trying to determine the filter characteristics of a filter with unknown documentation.

## Fixed Point Coefficients

From the previous discussion, it is recommended that the user check the magnitude dB plot when generating filter coefficients in fixed point. For the first FIR filter designed (FIR Filter design using the Parks-McCellan Design Procedure), lets generate the fixed point coefficients for at 16-bit signal processor or ASIC chip, and lets assume that the accumulator is 32-bits, which gives us plenty of head room against an overflow condition. The FIR coefficients File generation dialog box is shown below.


This dialog procedure produces file LowPassFilter.out with the fixed point coefficients scaled at B15 as shown below.

```
// Scale Up Factor = 0.241331
    -53,-241, -421, -532, -428, -60, 465, 882
    880, 307, -668, -1552, -1735, -860, 865, 2610
    3256, 2002, -1029, -4561, -6520, -4915, 1139, 10755
    21373, 29645, 32766, 29645, 21373, 10755, 1139, -4915
    -6520, -4561, -1029, 2002, 3256, 2610, 865, -860
    -1735, -1552, -668, 307, 880, 882, 465, -60
    -428, -532, -421, -241, -53
    7907 // Unity Gain Scale Factor at B15
```

If we now go to the Filter response Analysis... -> fixed Pt. Response... under the Filter tile, and fill in the dialog box as shown.


We get the same filter plots as shown previously in the Parks-McCellan FIR Design Procedure.


## Summary

The documentation on the software under the GUI reference (F6) should answer any remaining questions as pertains to this subject. Note that filter design is so important in signal processing, that the software under the Filter tile presents all of these custom dialog procedures, and internally generates the shell script code, since the internal engine of this software compiles a shell script file into 'tree code', and executes that code. Note that in Appendix A, and under the Digital Filters Block, that only the functions listed with vectors should be used in writing shell scripts for design analysis, as the other functions are no longer supported.

## References

[1] L. R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing, Prentice-Hall, 1975.
[2] A. V. Oppenheim and R. W. Schafer, Discrete-Time Signal Processing, Prentice-Hall, 1989.
[3] A. V. Oppenheim, and A. S. Willsky, with I. T. Young, Signals and Systems, Prentice-Hall, 1983.
[4] D. J. DeFatta, J. G. Lucas, and W. S. Hodgkiss, Digital Signal Processing, John Wiley \& Sons, 1988.
[5] Digital Signal Processing Committee, Programs for Digital Signal Processing, IEEE PRESS, 1979.
[6] fredric j. harris, Multirate Signal Processing for Communication Systems, Prentice-Hall, 2004.
[7] H. J. Fliege, Multirate Digital Signal Processing, John Wiley \& Sons, 1994.
[8] John G. Proakis and Dimitris G. Manolakis, Digital Signal Processing, Principles, Algorithms, and Applications, Fourth Edition , Prentice-Hall of India, 2008.
[9] K. Steiglitz, T. W. Parks and J. F. Kaiser, METEOR: A Constraint-Based FIR Filter Design Program, IEEE Trans. Signal Processing, vol. 40, no. 8, pp. 1902-1909, August 1992.
[10] Bernard Sklar, Digital Communications , Fundamentals and Applications, Second Edition Prentice-Hall, 2004.
[11] Hogenauer, Eugene B., An Economical Class of Digital Filters for Decimation and Interpolation, IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-29, No. 2, April 1981.
[12] Lyons, Richard, Understanding Digital Signal Processing, 2nd edition, 2004.

## Chapter 9 - Spectral Analysis and Convolution/Correlation

## Introduction

Spectral Analysis is an especially important topic in signal processing, since a large majority of applications deal with stochastic processes. Many signals have unwanted signals buried in the noise, and it's important for the engineer to identify these signals. Once unwanted signals are identified, many techniques can be employed to block these signals, including digital filtering, matched filters using correlation techniques, windows to trim side lobes, ..., etc. The task of detecting signals in a given signal space or frequency band is a difficult task because of different noise environments, dead zones, signals with on and off times, etc. For detection, we want to know if these is an unwanted or unknown signal present in a given signal space or frequency band. If a signal is detected, we want to obtain the signal parameters such as frequency, power, signal bandwidth, etc. Since many signals are harmonic, we can use the Discrete Fourier Transform or FFT, in combination with windows to accomplish the task. Accordingly, this SOFTWARE has implemented the following window functions widely used in signal processing tasks, namely: the 4-term Blackman harris Window; the Blackman Window; the DolphChebyshev Window; the Gaussian Window; the Hamming Window; the Hanning Window; the Kaiser Window; theTappered Rectangular Window; and the Triangular Window. The user can run the shell script program FFT Window Functions under the SpecAnalysis tile on the main menu to display these functions. The documentation of these functions can be found in the functions Reference (F7) under the Spectral Analysis tile. The user can also consult the reference material.* Below we show some plots of these functions.





[^8]

## Power Density Spectrum using The Window-Overlap Method

We note that the Rectangular Window has a 13 dB side lobe below the main lobe, while the 4-term Blackmanharris Window has a side lobe of 92 dB below the main lobe. However, the main lobe is 4 times the width of the Rectangular Window. The Rectangular window cannot be used in a robust spectrum analysis task, so we need to reduce the main lobe width of the Blackmanharris Window, so has
not to miss very narrow band signals. A paper by harris comes to the rescue ${ }^{\$}$. We introduce a dialog procedure to allow the user to simulate this powerful spectral analysis procedure, or alternately, to read in external data and process multiple records with resulting output. This procedure is a modification of the standard Welch method of Periodogram averaging, in that the data is windowed over four times the segment size, and then folded into the segment size, thus effectively reducing the main lobe down to the main lobe size of a rectangular window. This folding is given by the following equation from the paper by harris.

$$
H_{4 N}(k)=\sum_{n=0}^{N-1}\left[\sum_{r=0}^{3} w_{4 N}(n+r N) h(n+r N)\right] e^{-j \frac{2 \pi}{N} n k}
$$

The dialog for simulated input allows for three simulated sinusoidal entries along with either additive Gaussian or White noise. For external input, click the File Data Input button in Options Group 2, and specify a data file by depressing the Click for File Input button. Under the Dialogs tile in the main menu, we select the Power Density Spectrum tile and are given the following dialog entry.


We note the input parameters and run the test as entered with the following results.


We note that we detected the three signals at 1000,1100 , and 2000 Hz . Now for the simulated file input case, we go to the Data File Generation... dialog procedure under the Dialogs menu, and enter the following parameters as shown in the dialog box below.


Make sure you navigate to the specanal sub-directory. This generates a plot as follows.


So we go back to the Power Density Spectrum dialog procedure and enter the following parameters as shown below. Make sure the File Data Input button is depressed.


Ignore the error massage, and note the 5 plots produced. Note that one can display all 5 plots by ctrl +5 or ctrl + shft +5 as described in Chapter 6 on Plotting Details. Record 1 and Record 5 plots are shown below.



## PSD for Complex Signals in Noise

For estimating the frequencies of a sum of complex exponentials in white noise, several algorithms exist, namely the MUSIC (MUtiple Signal Classification) Method, the Eigenvector Method, and the Minimum Norm Algorithm.* The methods are closely related, and use eigenvalues and eigenvectors in their associated algorithms. For the MUSIC algorithm, the equation for the Power Density Spectrum is defined as,

$$
P_{x}=\frac{1}{\left|\sum_{k=0}^{M-1} v(k) e^{-j k w}\right|}
$$

For the autocorrelation matrix $R_{x}$ of M samples, then $v(k)$ are those eigenvectors matched to the lowest eigenvalues, whose roots lie on the unit circle. Note that each summation uses a 2048 point FFT with zero padding. This algorithm works best for complex sinusoids or other narrowband spectral components in WHITE NOISE; for COLORED NOISE, there's a degradation in the algorithms performance. Find below an example of this technique.

```
#include "sigsys.h"
int i, k, N = 1024;
veccmplx x[N], n;
vector Px;
float SampF = 10000; // Hz
complex phaseOff;
vector freq[] = { 1250, 1500, 2500, 3000 }; // Hz
vector omega;
n = cmplx(0, vecGen(0, 1, N-1));
omega = 2*pi*freq/SampF;
for(i = 0; i < 4; i++) {
    phaseOff = cmplx(0, 2*pi*randf());
    x += exp(omega[i]*n + phaseOff);
}
X += uniformV(N);
Px = MUSICpds(x, 8, 32);
```

[^9]```
openPlot("MUSICpds");
ploty(0, SampF/Sizeof(Px), Px, Sizeof(Px));
grid();
xLabel("Frequency Hz");
yLabel("Magnitude (dB)",1);
Title("The MUSIC PSD Algorithm for complex sinusoids in noise");
pCRT();
```



## Zoom FFT using the Chirp Transform Algorithm

For a N point FFT algorithm, N complex values are computed at N points distributed equally around the unit circle. To compute in a reduced frequency range on the unit circle, we turn to the CTA*. For a signal at a frequency of 24 KHz , and zooming in at 300 to 900 Hz , we have

$$
\begin{aligned}
& \omega_{k}=\omega_{0}+k \Delta \omega, \quad k=0,1, \ldots M-1 \\
& A=e^{-j \omega_{0}}, \quad W=e^{-j \Delta \omega / 2} \\
& X_{k}=\sum_{n=0}^{N-1} x_{n} A^{n} W^{2 n k}, \quad 2 n k=k^{2}+n^{2}-(n-k)^{2} \\
& g[n]=x[n] e^{-j w 0 n} W^{n^{2}}, \mathrm{n}=0,1, \ldots \mathrm{M}-1 \\
& X_{n}=W^{n^{2}} \sum_{k=0}^{N-1}\left[g[n] W^{-(n-k)^{2}}, \mathrm{n}=0,1, \ldots, \mathrm{M}-1\right.
\end{aligned}
$$

[^10]

veccmplx ChirpZ(vector Vin, float Fh, float Fl)
\{
int tmp, M, N, NFFT = 8192;
complex A, W;
vector k;
veccmplx Xk, Wk, Yk, FFTYk, FFTWk, Gm, Wn, Xm, Vtmp;
M = Sizeof(Vin);
A $=\exp \left(-2 j^{*} \mathrm{pi}^{*} \mathrm{Fl}\right)$;
W = exp(-2j*pi*(Fh-Fl)/(2*(NFFT-M-2)));
Xk = \{ Vin, zerosV(NFFT-M) \};
k = vecGen(0, 1, NFFT-1);
Yk = Xk.*A.^k.*W.^(k.*k);
Wk = W.^(-k.*k);
Wk $=\{$ Wk[0:1:NFFT-M-1], flip(Wk[0:1:M-1]) \};
FFTYk = fft(Yk, NFFT); FFTWk = fft(Wk, NFFT);
Gm = ifft(FFTYk.*FFTWk);
Wn = W.^(k.*k);
Xm = (Gm.*Wn);
Vtmp = copyV(Xm, 0, NFFT-M);
return(Vtmp);
\}
vector Vin;
veccmplx Vout;
float Freq, Fl, Fh;
int fd, npts;
Freq = 24000; Fl = 300; Fh = 900;
fd = open("C:\SlideRule\specanal\SIGFFT3.DAT","r");
npts = readV(Vin, 2048, 6, fd);
close(fd);
Vout = ChirpZ(Vin, Fh/Freq, Fl/Freq);
openPlot("Zoom-FFT");
Title("Zoom FFT using ChirpZ Transform");
xLabel("Frequency - Hz"); yLabel("MAGNITUDE",0);
ploty(Fl,(Fh-Fl)/Sizeof(Vout), abs(Vout), Sizeof(Vout));
grid();
grid();


## References

[1] L. R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing, Prentice-Hall, 1975.
[2] A. V. Oppenheim and R. W. Schafer, Discrete-Time Signal Processing, Prentice-Hall, 1989.
[3] fredric j. harris, On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform, Proceedings of the IEEE, Vol. 66, No 1, January 1978.
[4] fredric j. harris, Multirate Signal Processing for Communication Systems, Prentice-Hall, 2004.
[5] John G. Proakis and Dimitris G. Manolakis, Digital Signal Processing, Principles, Algorithms, and Applications, Fourth Edition, Prentice-Hall of India, 2008.
[6] fredric j harris, On detecting White Space Spectra for Spectral Scavenging in Cognitive Radios, Springer Science, LLC, 7 February 2008.
[7] Monson H. Hayes, Statistical Digital Signal Processing and Modeling, John Wiley \& Sons, 1996.

## Chapter 10 - Signal Processing

## Introduction

This Chapter presents some of the basic functions that would be presented in an introductory course in signal processing. All of the functions presented here reside in the Signal Processing Toolbox. A listing of these functions can be found in Appendix A. Note that all of the functions are written as shell scripts, which allows the user to examine the details of a given functions implementation. In Chapter 8, we covered IIR filter design under the menu tile Filter. These were bi-quad designs which are the preferred implementation in real world situations. So were going to start off covering this topic. The main purpose here is to allow the beginning novice to examine each module, and compare to the appropriate material in a textbook.

## General IIR Filter Design*

Under the Dialogs menu tile, we click the IIR Filter Design tile, and are presented will the following dialog box,


[^11]As noted in the dialog entry box, this procedure allows one the design one of the four filter types as specified in Options group 2, in one the four flavors as selected in options group 1. Note that this is a user generated (programmed) dialog entry as covered in Chapter 7. We note in options group 4, that we can select up to 7 different plot outputs. The two entries of the lower left are used when we select a Chebyshev II design. For the given entry (Elliptic Design), find four of the seven plots below.




Below is a print out by selecting function key F12.

```
Elliptic Digital Filter Design
Filter type is LowPass
Filter order = 6
Sample Frequency = 10000 Hz
Lower Passband Frequency = 1000 Hz
Upper Passband Frequency = 2000 Hz // used in band pass designs
Coefficients of Direct form of filter
vector -> num(7)
[index
0000 0.016048 -0.048171 0.085968 -0.098949 0.085968 -0.048171 0.016048
]
vector -> den(7)
[index
```

```
0000 1 -4.7549 10.043 -11.926 8.3648
]
Coefficients of Cascaded form of filter
Gain = 0.01605
matrix -> B(3,3)
[row
0000 1 -0.076091 1
0001 1 -1.3889 1
0002 1 -1.5368 1
]
matrix -> A(3,3)
[row
lllr
]
Zero's and Poles's of filter
Zeros = vector(cmplx) -> Vz(6)
[index
0000 0.038046-0.99928j 0.038046 +0.99928j 0.69444 +0.71955j
0.69444 -0.71955j 0.76839 +0.63999j 0.76839 -0.63999j
]
Poles = vector(cmplx) -> Vp(6)
[index
0000 0.78791-0.2161j 0.78791 +0.2161j 0.79157 +0.49056j
0.79157 -0.49056j 0.79796 -0.57938j 0.79796 +0.57938j
]
```

Note the shell script code for this design nodule can be found in digitalfliirDFD.txt. You can examine it by selection function key F10. Below is a table of the functions used in this design tool.

| File | Directory | Function | Description |
| :--- | :--- | :--- | :--- |
| bilinear.txt | sigsys | bilinear | Bilinear transform |
| butterap.txt | sigsys | butterAP | Butterworth analog prototype |
| butterdf.txt | sigsys | butterDF | Butterworth digital filter design |
| chebylap.txt | sigsys | cheby1AP | Chebyshev-I analog prototype |
| cheby1df.txt | sigsys | cheby1DF | Chebyshev-I digital filter design |
| cheby2ap.txt | sigsys | cheby2AP | Chebyshev-II analog prototype |
| cheby2df.txt | sigsys | cheby2DF | Chebyshev-II digital filter design |
| ellipticap.txt | sigsys | ellipticAP | Elliptic analog prototype |
| ellipticdf.ttxt | sigsys | ellipticDF | Elliptic digital filter design |
| dir2cas.txt | sigsys | dir2cas | Direct form of filter to 2-order sections |
| freqF..txt | sigsys | freqF | Frequency response of digital filters |
|  | Internal func | stepF | Step response of digital filters |
|  | Internal func. | impulseF | Impulse response of digital filters |
| groupDelay.txt | sigsys | groupDelay | Group delay calculation |
| Smapping.txt | sigsys | Smapping | Analog freq. band transformation |
| Strans.txt | sigsys | STrans | Prototype freq. band transformation |
| dir2cas.txt | sigsys | dir2cas | Direct form of filter to 2 ${ }^{\text {nd }}$-order sections |
|  | Internal func | polyR | Roots of polynomial for vector input |
|  | Internal func | polyM | Polynomial multiplication |
|  | Internal func | poly | Polynomial coefficients from its roots |
|  | Internal func | ellipticJ | Jacobian elliptic function |
|  | Internal func | ellipticC | Elliptic integral of 1 ${ }^{\text {st }}$ kind |
|  |  |  |  |

## Lattice Filter Structures

Lattice filter structures are used extensively in speech processing and certain adaptive filters because they have desirable properties, computation efficiency and robustness to round off errors. The routines for performing these lattice conversions can be found in the software help menu in the Signal Processing Toolbox.* Note that lattice filters have several properties, namely $\$$ :

1) The reflection coefficients that are generated by the Levinson-Durbin recursion, to solve the autocorrelation normal equations, are bounded in magnitude by $\left|\Gamma_{j}\right| \leq 1$.
2) If $a_{p}(k)$, and $\Gamma_{j}$ is the set of corresponding reflection coefficients, then the polynomial in $a_{p}$ will be a minimum phase polynomial if and only if $\left|\Gamma_{j}\right|<1$ for all $j$.
3) If $\boldsymbol{a}_{p}$ is the solution of the Toeplitz normal equations $\boldsymbol{R}_{p} \boldsymbol{a}_{p}=\boldsymbol{e}_{p} \boldsymbol{u}_{1}$, then $A_{p}(z)$ vill be minimum phase, if and only if $\boldsymbol{R}_{p}$ is positive definite, $\boldsymbol{R}_{\mathrm{p}}>0$.
4) The autocorrelation method produces a stable all-pole model.
5) If $\mathrm{a}_{\mathrm{j}}$ is a set of filter coefficients, and $\left|\Gamma_{j}\right|<1$ for $\mathrm{j}=1, \ldots, \mathrm{p}-1$, and $\left|\Gamma_{p}\right|=1$, then the polynomial has all its roots on the unit circle.

## All-Zero Lattice Filters

A FIR filter of length $M$ can be converted to a lattice structure. The FIR coefficients are converted to lattice $(\mathrm{K})$ parameters, and are called the reflection coefficients. The block diagram for an all zero lattice structure is as follows.


[^12]
## All-Pole Lattice Filters

The lattice structure for an all pole IIR filter to lattice structure is as follows.


| A Demo of an all pole IIR filter to a Lattice form, |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: |
| and from the Lattice form back to the all pole dire |  |  |  |  |
| Input IIR coefficient vector -> $\mathrm{a}(4)$ |  |  |  |  |
| [index |  |  |  |  |
| 0000 | 2 | 1.2667 | 0.54167 | 0.066667 |
| ] |  |  |  |  |
| Lattice coefficeint vector -> K(4)[index |  |  |  |  |
|  |  |  |  |  |
| 0000 | 5 | 0.5 | 0.25 | 0.033333 |
| ] |  |  |  |  |
| Conversion back to all pole IIR vector -> den(4) |  |  |  |  |
| [index |  |  |  |  |
| 0000 | 2 | 1.2667 | 0.54167 | 0.066667 |
| ] |  |  |  |  |

## Lattice Ladder Filters

The lattice ladder structure for an IIR filter containing both poles and zeros follows.


A Demo of a pole-zero filter to a Lattice/Ladder form, and from the Lattice/Ladder form back to the pole-zero direct form

Input numerator coefficient vector -> num(4)
[index
$\begin{array}{lllll}0000 & 1 & 2 & 2 & 1\end{array}$ ]
Input denominator coefficient vector -> den(4)
[index
$\begin{array}{lllll}0000 & 2 & 1.2667 & 0.54167 & 0.066667\end{array}$
]
Lattice coefficeint vector -> K(4)
[index

| 0000 | 0.5 | 0.5 | 0.25 | 0.033333 |
| :--- | :--- | :--- | :--- | :--- |

]
Ladder coefficeint vector -> C(4)
[index

| 0000 | 0.1875 | 0.875 | 1.3667 | 1 |
| :--- | :--- | :--- | :--- | :--- |

]

| Numerator vector $->\mathrm{b}(4)$[index |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: |
| 0000 | 1 | 2 | 2 | 1 |
| Denominator vector -> a(4) |  |  |  |  |
|  |  |  |  |  |
| [index |  |  |  |  |
| 0000 | 2 | 1.2667 | 0.54167 | 0.066667 |
| ] |  |  |  |  |

## The z Transform Inversion*

Given a rational function, we want to compute the residue, and direct and/or polynomial terms in $\mathrm{z}^{-1}$, and then convert it back to rational form.

$$
X(z)=\frac{1+.2 \sqrt{2} z^{-1}}{1-.7 \sqrt{2} z^{-1}+0.81 z^{-2}}
$$

Using the function residueZ in the Signal Processing Toolbox, we have

```
#include "sigsys.h"
vector num[] = { 1, .2*sqrt(2) };
vector den[] = { 1, -.7*sqrt(2), .81 };
vector k;
veccmplx p, r, a, b;
Print(num); Print(den);
p = residueZc(num, den, r, k);
printf("Poles = "); Print(p);
printf("Residues = "); Print(r);
printf("Direct term = "); Print(k);
printf("Going in reverse\n");
a = residuezc(r, p, k, b);
Print(b); Print(a);
*****************************************************
vector -> num(2)
[index
0000 1 0.28284
]
vector -> den(3)
[index
0000 1 -0.98995 0.81
]
Poles = vector(cmplx) -> p(2)
[index
0000 0.49497 +0.75166j 0.49497 -0.75166j
]
Residues = vector(cmplx) -> r(2)
[index
0000 0.5 -0.5174j 0.5 +0.5174j
]
Direct term = vector -> k(0)
[]
Going in reverse
vector(cmplx) -> b(2)
[index
0000 1 +0j 0.28284 +0j
]
vector(cmplx) -> a(3)
[index
```

```
0000 1 +0j -0.98995 +0j 0.81 +0j
]
```


## Analog Filter Design

Under the Dialogs menu tile, we click the Chebyshev-II Analog Filter Design tile, and are presented will the following dialog box,


For the given entry (LowPass), find some of the output plots below.


## The CELP Algorithm*

In all cell phones, the CELP (Code Excited Linear Prediction) algorithm is used to compress speech samples. In most cases the algorithm resides in a separate signal processor embedded in an ASIC. The model for the speech analysis and synthesis is based on modeling the vocal tract as a linear all-pole (IIR) filter that has the system function

$$
H(z)=\frac{G}{1+\sum_{k=1}^{p} a_{p}(k) z^{-1}}
$$

The speech is usually sampled at 8 KHz from an 8 -bit A/D Mu-law codec chip is blocks of 160 samples, for 20 msec . frames. The codec chip has a built in anti-aliasing filter, with a pass band of $3.3 \mathrm{KHz}(-3$ $\mathrm{dB})$, and a stop band of $4 \mathrm{KHz}(-25 \mathrm{~dB})$. The first order of processing is to convert the logarithmic 8 -bit mu-law samples into 12 -bit linear samples. Next, a pre-emphasis filter is run to add gain to the higher frequencies in the sample block. In order to solve for the $\boldsymbol{a}_{p}$ coefficients in the above equation, we form an autocorrelation matrix $\boldsymbol{R}_{s s}$ and solve for the autocorrelation vector $\boldsymbol{r}_{a a}$ to generate 10 LPC coefficients $\boldsymbol{a}_{p}$ by solving the following matrix equation $\boldsymbol{a}_{p}=-\boldsymbol{R}_{5 s}{ }^{-1} \boldsymbol{r}_{\mathrm{ss}}$. This equation is solved with the recursive Levinson-Durbin algorithm, because it's fast, and we want to lessen the use of battery power. Now, when we examine the 10 LPC coefficients, we note that there's quite a spread in the range of the coefficients. Since the transmitting channel is limited in bandwidth, were limited on the amount of information we can
put into the digital message to be sent. Also note that the digital message will have to have additional bits of FEC coding using a Viterbi encoder. So we perform a trick. We convert the 10 LPC coefficients into line spectral pairs. The inverse LPC filter is given by,

$$
A(z)=1+a_{1} z^{-1}+\cdots+a_{p} z^{-p}
$$

For a 10th order system, we compute the polynomials $\mathrm{P}(\mathrm{z})$ and $\mathrm{Q}(\mathrm{z})$.

$$
\begin{aligned}
P(z)= & A(z)+z^{-11} A\left(z^{-1}\right) \\
= & 1+\left(a_{1}+a_{10}\right) z^{-1}+\left(a_{2}+a_{9}\right) z^{-2}+\cdots+ \\
& \quad\left(a_{10}+a_{1}\right) z^{-10}+z^{-11} \\
Q(z)= & A(z)-z^{-11} A\left(z^{-1}\right) \\
= & 1+\left(a_{1}-a_{10}\right) z^{-1}+\left(a_{2}-a_{9}\right) z^{-2}+\cdots+ \\
& \quad\left(a_{10}-a_{1}\right) z^{-10}-z^{-11}
\end{aligned}
$$

The LPC coefficients $A(z)$ can be recovered from $P(z)$ and $Q(z)$ by

$$
A(z)=\frac{[P(z)+Q(z)]}{2}
$$

After solving for the roots of $P$ and $Q$, we note that the polynomial $Q(z)$ has a real root at $\mathrm{z}=1$, and the $R(z)$ polynomial has a real root $\mathrm{z}=-1$; with all the other roots complex. All of the roots lie on the unit circle; with the roots of $P$ and $Q$ alternating. So the output vector contains the angles (in radians) from 0 to pi of a line from the origin to the complex roots in the positive half of the unit circle. This is a tremendous advantage, since the range of the coefficients is fixed, which makes quantization much simpler. After we have the 10 Q and P coefficients in the upper half of the unit circle, we quantize them into a range of smaller bit lengths depending on which coefficient, and then convert these quantized coefficients back to the A's, or LPC coefficients. Then a search of two code books is done to determine the best fit to the given block of speech. The code books are designed to model most of the speech sounds generated by the vocal track of human speech. In each iteration of the search, speech samples (160) are generated from the quantized LPC coefficients, and cross correlated with the input speech samples. The highest score of this search then stores the saved code book parameters that will be encoded into the sent message along with the quantized line spectral pairs. On the receiver side, the algorithm converts the line spectral pairs to LPC coefficients, and uses the code book parameters to excite the allpole filter to generate the 160 PCM samples to be output to a 12 -bit DAC that drive's the receiver's speaker. Note that at the receiver, the output samples to the speakers DAC have to be started no later then $\sim 100$ milliseconds after receiving the message. This is a broad overview of the speech algorithm, however, there's more to it, such as full rate, half rate, quarter rate, and eighth rate frames, and other goodies, because although one wants to have the best sounding fit to the actual speech, were limited by the number of CPU instructions in the signal processer, since CPU cycles in CMOS devices use current, and the less CPU cycles per frame, the longer the battery will last. Having presented the above, find a simulation of using line spectral pairs from a real speech sample of 160 samples.

```
#include "sigsys.h"
int fd, cnt, N = 10;
vector Sig, Rss, Ap, K, LSP, P[N+1], Q[N+1], ApChk;;
float er, ftemp;
fd = open("c:\sliderule\specanal\speech.bin","rb");
cnt = skip(7000, 3, fd); //skip 7000 samples to get a speech frame
cnt = readV(Sig, 160, 3, fd); // read in 160 samples;
close(fd);
rss = autocor(Sig, N); // Solve for auto-correlation vector
printf("The auto-correlation vector "); Print(Rss);
er = LevDurbin(Rss, Ap, K); // Solve for the LPC coefficients
printf("The inverse LPC filter "); Print(Ap);
LSP = lineSpec(Ap, P, Q); // Get line spectral pairs
printf("The line spectral pairs "); Print(LSP);
Print(P); Print(Q);
ApChk = (P + Q)/2;
printf("Converting back to LPC cpefficients "); Print(ApChk);
****************************************************************************
The auto-correlation vector vector -> Rss(11)
[index
0000 2.0391e+05 1.9033e+05 1.6952e+05 1.522e+05 1.3696e+05
    1.1715e+05 93870 6 73918 60761 47894
]
The inverse LPC filter vector -> Ap(11)
[index
0000 1, 1.1.229 0.43356 -0.14878 -0.18515 0.12486
]
The line spectral pairs vector -> LSP(10)
[index
0000 0.14265 0.34175 0.54726 1.0126 1.2641
    1.4829 1.781 2.1264 2.4896 2.8092
]
vector -> P(12)
[index
0000 1 -1.2843 0.37375 -0.11307 -0.17373 0.26927
    0.26927 -0.17373 -0.11307 0.37375 -1.2843 1
]
vector -> Q(12)
[index
0000 1 - 1.1736 0.49337 -0.18449 -0.19657 -0.019557
    0.019557 0.19657 0.18449 -0.49337 1.1736 -1
]
Converting back to LPC cpefficients vector -> ApChk(12)
[index
0000 1 -1.229 0.43356 -0.14878 -0.18515 0.12486
]
```



## OFDM - Orthogonal Frequency Division Multiplex*

The principal of OFDM is to take a high rate data channel, and split it into a number of lower rate channels, and transmit them simultaneously over a number of subcarriers frequencies. Because the symbol rate over the subcarriers is slower by a factor of N subcarriers, the amount of dispersion in time caused by multipath delay is decreased significantly. Because of this, an adaptive filter or a rake type receiver is not needed. A guard time inserted in with every OFDM symbol, almost entirely eliminates intercarrier interference. Each subcarrier shows a random phase shift and amplitude change caused by carrier frequency offset, timing offset, and frequency selective fading. Two methods exist to handle these conditions, namely coherent and differential detection. For coherent detection, is idea is to lessen the training data to find the reference values, and will be the topic of this discussion. In particular, will look at IEEE 802.11a, commonly known as WiFi.

## OVERVIEW OF IEEE STANDARD 802.11A

The current 802.11a standard consists of 20 MHz channels in the frequency ranges of; $5.15-5.35 \mathrm{GHz}$; $5.425-5.675 \mathrm{GHz}$; and $5.725-5.875 \mathrm{GHz}$. A block diagram of OFDM channel processing follows.


The following tables show the different modulation schemes and timing parameters used in the IEEE 802.11a standard.

| RATE | Data Rate (Mbits/s) | Modulation | NBCAR | Code Rate | NBSYM | NCSYM |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 1101 | 6 | BPSK | 1 | $1 / 2$ | 24 | 48 |
| 1111 | 9 | BPSK | 1 | $3 / 4$ | 36 | 48 |
| 0101 | 12 | QPSK | 2 | $1 / 2$ | 48 | 96 |
| 1101 | 18 | QPSK | 2 | $3 / 4$ | 72 | 96 |
| 1101 | 24 | 16 QAM | 4 | $1 / 2$ | 96 | 192 |
| 1101 | 36 | 16 QAM | 4 | $3 / 4$ | 144 | 192 |
| 1101 | 48 | 64 QAM | 6 | $2 / 3$ | 192 | 288 |
| 1101 | 54 | 64 QAM | 6 | $3 / 4$ | 216 | 288 |

## NBCAR: Number of coded bits per subcarrier.

NBSYM: Number of data bits per OFDM symbol.
NCSYM: Number if coded bits per OFDM symbol.

| Parameter | Value |
| :--- | :--- |
| Number of data subcarriers | 48 |
| Number of pilot subcarriers | 4 |
| Number of total subcarriers | 52 |


| Number of padded zeros | 11 |
| :--- | :--- |
| Number of samples in guard interval | 16 |
| Subcarrier frequency spacing | 312.5 KHz |
| IFFT/FFT period | 3.2 us |
| IFFT/FFT size | 64 |
| Preamble duration | 16 us |
| Guard duration | 0.8 us |
| Long preamble duration | 8 us |
| Short preamble duration | 8 us |
| OFDM symbol duration | 4 us |
| Guard interval duration for long preamble | 1.6 us |

## The OFDM Modulator Scheme

On the transmit side of the OFDM transceiver, a 64 IFFT is used to transfer the QAM symbols to a complex OFDM signal at baseband. The resulting is then modulated to a RF frequency as shown in the block diagram.
The plots below shows an example of four subcarriers within an OFDM signal, and the spectra of the individual subcarriers. Note that the spectra are a group of Dirac pulses located a subcarrier frequencies with a square passband that has zeros for all frequencies which are integer multiples of the subcarrier bandwidth.



## IEEE 802.11A Message Format

| RATE | Reserved | Length <br> 4 bits | Parity <br> 1 bit | Tail <br> 12 bits | SERVICE | PSDU | Tail | Pad Bits |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 6 bits | 16 bits |  | 6 bits |  |  |  |  |  |

SERVICE: Scrambler initialization + Reserved bits
PSDU: Physical layer Services Data Unit
Tail: To return the encoder to the zero-state Pad Bits: To make the length of DATA a multiple of NCBPS

| PLCB Preamble | SIGNAL | DATA |
| :--- | :--- | :--- |
| 12.5 symbols | 1 OFDM symbol | Variable number of OFDM symbols |

PLCB: Physical Layer Convergence Preamble (10 short training sequences, plus 2.5 long training sequences

## The short training sequence

- Signal detection
- AGC convergence
- Diversity selection
- Timing acquisition
- Course frequency acquisition


## The long training sequence

- Channel estimation
- Fine frequency acquisition


## Carrier Recovery

At this point we want to perform a simulation, and estimate the STO (symbol time offset) and CFO (carrier frequency offset). Plots of the time-domain short preamble and the time-domain long preamble are shown below.



We next join these signals, add noise, and set the CFO to 1.56 .


Note that in the IEEE 802.11a fixed-lag $\left(\mathrm{N}_{\mathrm{g}}\right)$ correlation of the short training sequence is used for the course CFO estimate, and the fixed lag correlation of the long training sequence is used for the fine CFO estimate.

$$
\begin{aligned}
& \varepsilon_{c}=\frac{N}{2 \pi N_{g}} \arg \sum_{n=1}^{N} x_{s}\left[n+N_{g}\right] x_{s}^{*}\left[N[n], N=64, N_{g}=16\right. \\
& \varepsilon_{f}=\frac{1}{2 \pi} \arg \sum_{n=1}^{N} x_{l}[n+N] x_{l}^{*}[N[n], N=64
\end{aligned}
$$



## The Preamble Structure of the IEEE Standard 802.11a

| 8 us Short preamble |  |  |  |  |  |  |  |  |  | 8 us Long preamble |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 16 |  |  |  |  |  |  |  |  |  | 32 | 64 | 64 |
| t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 | t10 | GI2 | T1 | T2 |

From the plot of theTime-domain short preamble, we can see that every $t$ segment ( 16 samples) repeats the same values as the previous segment. From the code segment for the course CFO estimate (courseCFOest), we can see that the correlation starts at t 8 and correlates ( 16 samples) with $\mathrm{t}^{*}$, and then again with t 9 over $\mathrm{t} 10^{+}$. The angle (arg) of the resulting complex number is the estimate of the course CFO. With a first estimate of the CFO, we then call subroutine CompSigCFO to correct the elements in T1 by the estimated CFO value. We then correlate T1 over T2 to get the fine CFO estimate, and then add them together for the updated CFO, all while staying in the preamble when a OFDM message is detected.

## Symbol Synchronization

In a similar manner, one can use correlation techniques to estimate the STO (symbol timing offset value). We note that the Guard interval or cyclic prefix (CP) is 16 samples, while the OFDM data length consists of 64 samples, or the length of the IFFT/FFT. The method is to correlate TWO sliding windows of 16 samples separated by the FFT length of 64 , while looking for a peak value. The correlation value is divided by the product of the norms of the two vectors to defeat false detections due to a noisy signal, and a threshold is employed to determine if a peak has been detected or not. The following plots show simulation results using this algorithm. The shell script code follows.



```
//
// Simulate symbol time offset (STO)
//
tframe = cmplx(uniformV(65),uniformV(65))/sqrt(12);
for(i = 0; i < 3; i++) {
    symbol = cmplx(uniformV(Nfft),uniformV(Nfft));
    symbol /= sqrt(12);
    symbolCP = { symbol[48:63], symbol };
    tframe = { tframe, symbolCP };
}
Lframe = Sizeof(tframe); Print(Lframe);
noise = 0.2*cmplx(uniformV(Lframe),uniformV(Lframe))/sqrt(12);
r = tframe + noise;
sigW = zerosM(2, Ng); corr = zerosV(Lframe); STOs = { };
for(i = 0; i < Lframe; i++) {
    Vc = sigW[0];
    // update signal window 1
    sigW[0] = (veccmplx){ Vc[1:Ng-1], r[i] }; //
    m = i - Nfft;
    if(m > 0) {
        Vc = sigW[1];
        // update signal window 2
        sigW[1] = (veccmplx){ Vc[1:Ng-1], r[m] };
        den = normV(sigW[0],'2')*normV(sigW[1],'2');
        corr[i] = abs(sigW[0]*sigW[1]')/den;
        if(corr[i] > cor_thres) STOs = {STOs, m };
    }
}
TSTOs = zerosV(Sizeof(STOs)); TSTOs[0] = 64+Ng;
for(i = 1; i < Sizeof(TSTOs); i++) TSTOs[i] = TSTOs[i-1]+Nfft+Ng;
```


## References

[1] John G. Proakis and Dimitris G. Manolakis, Digital Signal Processing, Principles, Algorithms, and Applications, Fourth Edition, Prentice-Hall of India, 2008.
[2] John G. Proakis and Vinay K. Ingle, Digital Signal Processing, using MATLAB, Brooks/Cole Publishing Company, 2000.
[3] L. R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing, Prentice-Hall, 1975.
[4[ Leyva-Ramos, J.; Denman, E.D., Matrix partial-fraction expansion method, ELECTRONIC LETTERS 3rd Feb. 1983 Vol. 19 No. 3.
[5] Monson H. Hayes, Statistical Digital Signal Processing and Modeling, John Wiley \& Sons, 1996.
[6] L.R. Rabiner. R.W. Schafer, Digital Processing of Speech Signals, Prentice-Hall, Inc., 1978
[7[ Richard van Nee and Ramjee Prasad, OFDM for Wireless Multimedia Communications, Artech House, 2000.
[8] Won Y. Yang, et al..., MATLAB ${ }^{\circledR}$ SSimulink $^{\circledR}$ for Digital Communictions, A-JIN, Inc., 2009

## Chapter 11 - Random Signals and their Distributions

## Introduction

Signals can be classified into one of two types, namely deterministic and random. Deterministic signals can be reproduced exactly, and don't vary over time. A random signal, or random process, on the other hand, is a signal that is not repeatable in a predictable manner. These later type signals occur repeatedly in real world engineering and scientific systems. Accordingly, we have mathematical models and variables to analyze these types of signals or stochastic processes. The following topics cover the more basic models that occur repeatedly in the field of digital communications. Note that the shell script code listed in this chapter is without the annotation code, as one can generate this code with mouse and keyboard entries once a given plot is displayed, and have the code automatically inserted into the shell script file, as covered in Chapter 6 on Plotting Details.

## The Uniform Random Variable*

A uniform random variable x has a PDF as follows:

$$
p(x)= \begin{cases}\frac{1}{b-a} & a \leq x \leq b \\ 0 & \text { else }\end{cases}
$$

Its mean and variance are defined as

$$
E(x)=\mu_{x}=\frac{b-a}{2} \quad \operatorname{VAR}(x)=\sigma_{x}=\frac{(b-a)^{2}}{12}
$$

In order to model this random variable, Slide-Rule employs functions randf and randV. Function randf when called, returns a (floating point) number in the range of zero to one, while function randV returns a uniform random vector. The algorithm to generate these uniform random numbers uses a linear congruence algorithm and an additional random shuffling process to effectively remove any sequential correlation in the linear congruence equation. The following example shows histogram and autocorrelation plots of a uniform random vector, which shows the PDF is basically flat and that there in no correlation between the uniform generated samples.

```
vector Vsig, Vacor;
float Vmean, Vvar;
Vsig = randV(10000)- 0.5;
openPlot("Histogram of Uniform Random variable");
histoP(Vsig, Sizeof(Vsig), -.5, .5, 50);
pCRT();
Vmean = mean(Vsig); Vvar = var(Vsig); Print(Vmean); Print(Vvar);
Vacor = autocor(Vsig, 300); // 300 lags
openPlot("Auto-Correlation of Vsig");
ploty(0, 1, Vacor, Sizeof(Vacor));
```

[^13]

We note that mean of this signal is near zero, and that the variance closely agrees with the equation as stated above. Note that in the code that the annotations (title, x-label, y-label, text, and arrow) were all
entered with mouse and keyboard actions once the two plots were displayed on the monitor, but are not shown in the code segment.

## The Gaussian (Normal) Random Variable*

The Gaussian random variable x has the PDF as follows:

$$
p(x)=\frac{2}{\sqrt{2 \pi \sigma^{2}}} e^{-\frac{(x-\mu)^{2}}{2 \sigma^{2}}} \quad \sigma^{2}=\text { the variance, } \mu=\text { the mean value }
$$

Recall from the central limit theorem, that all random variables, no matter what their distribution, approaches (over time) a Gaussian distribution if the variables are statistically independent. For that reason, the Gaussian random variable is widely used in engineering and scientific simulations. In order to model this random variable, Slide-Rule employs functions randn and nornalV. Function randn when called, returns a (floating point) Gaussian deviate with unit power, while function normalV returns a normal random vector. The algorithm to generate these Gaussian random numbers follows; The Rayleigh probability distribution function with random variable x :

$$
p(x)= \begin{cases}0, & x<0 \\ 1-e^{x^{2} / \sigma^{2}} & x \geq 0\end{cases}
$$

is related to a pair of Gaussian random variables p and q through the transformation

$$
\begin{aligned}
& p=r \cos (\theta), \quad q=r \sin (\theta) \quad 0 \leq \theta \leq 2 \pi \\
& x=1-e^{x^{2} / \sigma^{2}}=r \\
& r=\sigma \sqrt{2 \ln (1 /(1-x))} \\
& p=r \cos (\theta) \text { and } q=r \sin (\theta)
\end{aligned}
$$

The internal code in Slide-Rule is as follows:

```
DOUBLE gaussianN()
{
        static long iset = 0;
        static DOUBLE gset;
        DOUBLE r, v1;
        if(iset == 0) {
            while((v1 = randf()) == 1); // randf(), uniform (0->1)
            r = sqrt(2.0*log(1.0/(1.0-v1)));
            v1 = randf();
            gset= r*sin(2*M_PI*v1);
            iset = 1;
            return r*cos(2*M_PI*v1);
    } else { iset=0; return gset; }
}
```

[^14]Note that this function returns with unit power, and to obtain Gaussian random numbers at a different power level, we just multiply by the square root of the variance, or sigma. The following example shows a random Gaussian signal and a PDF plot along with the calculated probability values.



The code for the above example is a follows:

```
#include "normalPDF.txt"
vector Sig, x, p;
float mu, sigma2, xVal, prob, val = 0;
int minx, maxx;
Sig = sqrt(5)*normalV(10000) + 1.5;
mu = mean(Sig); sigma2 = var(Sig);
minx = mu - 3*sigma2; maxx = mu + 3*sigma2;
x = vecGen(minx, .01, maxx);
p = normalPDF(Sig, x);
printf("Tne signal mean = %f\n", mu);
printf("Tne signal variance = %f\n", sigma2);
prob = normalProb('R', Sig, val);
printf("Probability of exceding x = %f = %f\n", val, prob);
xVal = normalProb('I', Sig, prob);
printf("x value at probality %f = %.3f\n", prob, xVal);
openPlot("Signal");
ploty(0, 1, Sig, Sizeof(Sig));
pCRT();
openPlot("Gaussian PDF");
plotxy(x, p, Sizeof(x));
pCRT();
```


## The Chi-Square ( $X^{2}$ ) Central Random Variable*

For X with n independent and identically distributed zero mean Gaussian random variables with common variance, then X is an $X^{2}$ random variable with v degrees of freedom. The PDF of this random variable is given by,

$$
p(x)=\frac{1}{2^{v / 2} \Gamma(v / 2) \sigma^{2}} x^{v / 2-1} \exp \left(-x / 2 \sigma^{2}\right), \quad x>0
$$

where $\Gamma(u)$ is the Gamma function
The mean and variance are given by,

$$
\begin{aligned}
& E(x)=v \sigma^{2} \\
& \operatorname{VAR}(x)=2 v \sigma^{4}
\end{aligned}
$$

A plot of this function can be found below.

[^15]

The code for this plot follows (with annotation statements left out):

```
vector df[] = { 1, 2, 4, 6, 8 };
int i, j;
vector x, p;
float xVal, Prob;
x = vecGen(EPS, .1, 10);
p = zerosV(Sizeof(x));
openPlot("Chi-Squared Demo");
pminmax(0.1,10,0,0.9);
for(j = 0; j < Sizeof(df); j++) {
        for(i = 0; i < Sizeof(x); i++) {
            p[i] = chisqrPDF(df[j], x[i]);
    }
    plotxy(x, p, Sizeof(x));
}
pCRT();
Prob = chisqrProb('L', 6, 4);
printf("Left tail probability (v = 6) for x = 4 = %.4f\n", Prob);
Prob = chisqrProb('R', 6, 4);
printf("Right tail probability (v = 6) for x = 4 = %.4f\n", Prob);
xVal = chisqrProb('I', 6, Prob);
printf("X value for right tail probability (v = 6) %.4f = %.4f\n",Prob,xVal);
```


## The Chi-Square ( $X^{2}$ ) Non-Central Random Variable*

For X with n independent and identically distributed Gaussian random variables with common variance but with different means $\left(\mathrm{m}_{\mathrm{i}}\right)$, then X is a $X^{2}$ random variable with n degrees of freedom. The PDF of this random variable is given by,

$$
\begin{aligned}
& p(x)=\left\{\frac{1}{2 \sigma^{2}}\left(\frac{x}{x^{2}}\right)^{(n-2) / 4} e^{-\left(x^{2}+x\right) / 2 \sigma^{2}} I_{n / 2-1}\left(\frac{\mathrm{~s}}{\sigma^{2}} \sqrt{x}\right) \quad x>0\right. \\
& p(x)=0 \quad x \leq 0 \\
& s=\sqrt{\sum_{i=1}^{n} m_{i}^{2}} \quad I(u) \text { the modified Bessel function }
\end{aligned}
$$

The mean and variance are given by,

$$
E(x)=n \sigma^{2} \quad \operatorname{VAR}(x)=2 n \sigma^{4}
$$

A plot of this function can be found below.


The Rayleigh Random Variable*

[^16]The Rayleigh random variable is made up of two Gaussian random variables each distributed according to $\mathrm{N}\left(0, \sigma^{2}\right)$, and is defined as,

$$
X=\sqrt{X_{1}^{2}+X_{2}^{2}}
$$

Its PDF is given by

$$
p(x)=\frac{x}{\sigma^{2}} e^{-x^{2} / 2 \sigma^{2}} \quad x>0
$$

From the previous discussion, we can see that a Rayleigh random variable is the square root of a ChiSquare ( $\mathrm{X}^{2}$ ) random variable with two degrees of freedom. The mean and variance are given by:

$$
E(x)=\sigma \sqrt{\frac{\lambda}{2}} \quad \operatorname{VAR}(\mathrm{x})=\left(2-\frac{\pi}{2}\right) \sigma^{2}
$$

This random variable is used quite often is the study and simulation for signals received in urban radio multipath channels. The following example shows plots two random Gaussian signals, both with zero mean but with different $\sigma^{2}$, and a PDF plot along with the calculated probability values.




The code for this plot follows:

```
#include "sigsys.h"
vector Sig1, Sig2, x, p;
float mu, sigma2, prob, val = 0;
float Prob, xVal;
int minx, maxx;
Sig1 = sqrt(5)*normalV(10000);
Sig2 = normalV(10000);
x = vecGen(0, .01, 10);
p = rayleighPDF(Sig1, Sig2, x);
Prob = rayleighProb('R', Sig1, Sig2, x, 4);
printf("Probability for exceding x = 4 is %f\n", Prob);
xVal = rayleighProb('I', Sig1, Sig2, x, Prob);
printf("x value at probability %f = %f\n", Prob, xVal);
openPlot("Rayleigh PDF");
plotxy(x, p, Sizeof(x));
pCRT();
openPlot("Signal 1");
ploty(0, 1, Sig1, Sizeof(Sig1));
pCRT();
openPlot("Signal 2");
ploty(0, 1, Sig2, Sizeof(Sig2));
pCRT();
```


## The Rican Random Variable*

The Rican random variable is made up of two Gaussian random variables each distributed according to $\mathrm{N}\left(m_{1}, \sigma^{2}\right)$ and $\mathrm{N}\left(\mathrm{m}_{2}, \sigma^{2}\right)$, (i.e., the means can be different but the variances are the same) and is defined as,

$$
X=\sqrt{X_{1}^{2}+X_{2}^{2}}
$$

Its PDF is given by

$$
\begin{aligned}
& p(x)=\frac{x^{n / 2}}{\sigma^{2}} I_{0}\left(\frac{s x}{\sigma^{2}}\right) e^{-\left(x^{2}+s^{2}\right) / 2 \sigma^{2}} \quad x>0 \\
& s=\sqrt{m_{1}^{2}+m_{2}^{2}} \quad I_{0}(u) \text { the modified Bessel function }
\end{aligned}
$$

This random variable is used quite often is the study and simulation for signals received in urban radio direct line of site channels. The following example shows plots two random Gaussian signals, both with different mean but with the same $\sigma^{2}$, and a PDF plot along with the calculated probability values.

[^17]

Signal 2



The code for this plot follows:

```
#include "sigsys.h"
vector Sig1, Sig2, x, p;
float mu, sigma2, prob, xVal, val = 4;
int minx, maxx;
Sig1 = sqrt(5)*normalV(10000)+1;
Sig2 = normalV(10000)+1.2;
x = vecGen(0, .01, 10);
p = ricianPDF(Sig1, Sig2, x);
prob = ricianProb('R', Sig1, Sig2, x, val);
printf("Probability of exceeding x = %f = %f\n", val, prob);
xVal = ricianProb('I', Sig1, Sig2, x, prob);
printf("x-value at probability %f = %f\n", prob, xVal);
openPlot("Rican PDF");
plotxy(x, p, Sizeof(x));
pCRT();
openPlot("Signal 1");
ploty(0, 1, Sig1, Sizeof(Sig1));
pCRT();
openPlot("Signal 2");
ploty(0, 1, Sig2, Sizeof(Sig2));
pCRT();
```


## References

[1] Steven M. Kay, Fundamentals of Statistical Signal Processing, Detection Theory, Prentice Hall Signal Processing Series.
[2] John G. Proakis and Masoud Salehi, Digital Communications, Fifth Edition, McGraw Hill, 2008.
[3] Dimitris G. Manolakis, Vinay K. Ingle, Stephen M. Kogon, Statistical and Adaptive Signal Processing, McGraw Hill, 2000.

## Chapter 12 - Adaptive Filters

## Introduction

Adaptive filters are widely used in the field of engineering for applications such as; System identification; Predictive de-convolution; Adaptive equalization; Blind equalization; Linear predictive coding; Adaptive differential pulse-code modulation; Spectrum analysis; Signal detection; Adaptive noise canceling; Echo suppression; and Adaptive beam forming among others. This chapter presents some examples in the areas of System identification, Adaptive equalization, Adaptive noise canceling, and Echo suppression

## A Wiener filter for Noise Canceling*

A Wiener filter is based on finding the solution to the Wiener-Hopf equations. The restrictions for the solution in the digital domain are; a linear transversal filter, or FIR filter, must be used to calculate the desired response $\boldsymbol{d}$; The autocorrelation matrix $\boldsymbol{R}$ must be computed from the autocorrelation function, with the number of lags equal to the number of taps in the FIR filter; A cross-correlation vector must be computed between the tap inputs $\boldsymbol{u}$ and the desired response $\boldsymbol{d}$. From the above statements, we see that a Wiener filter is only applicable for linear systems, i.e., for non-linear systems, other solution algorithms are required. A block diagram of the FIR filter follows


The correlation matrix $\boldsymbol{R}$ is computed by $E\left[u(n) u^{H}(n)\right]$, where $E$ is the statistical expectation operator, and $u$ is the tap input vector. In expanded form we have,

[^18]\[

R_{x}=E\left\{x x^{H}\right\}=\left[$$
\begin{array}{ccccc}
r_{x}(0) & r_{x}^{*}(1) & r_{x}^{*}(2) & \ldots & r_{x}^{*}(p) \\
r_{x}(1) & r_{x}(0) & r_{x}^{*}(1) & \ldots & r_{x}^{*}(p-1) \\
r_{x}(2) & r_{x}(1) & r_{x}(0) & \ldots & r_{x}^{*}(p-2) \\
\vdots & \vdots & \vdots & & \vdots \\
r_{x}(p) & r_{x}(p-1) & r_{x}(p-2) & \ldots & r_{x}(0)
\end{array}
$$\right], \quad p=M-1
\]

The cross-correlation vector is computed by $\boldsymbol{p}=E\left[u(n) d^{*}(n)\right]$. Thus the Wiener-Hopf equation in matrix form is given by $\boldsymbol{R w}=\boldsymbol{p}$, where $\boldsymbol{w}$ is the tap-weight vector to the transversal filter. The block diagram for an adaptive noise canceller is:


For an example of a speech signal corrupted by noise, plots of the signals in a simulation example follow.



The shell script for this example is shown below.

```
#include "sigsys.h"
int i, N = 15000, L = 13;
vector v, a, b, v1, s;
int fd, cnt;
vector err, wo, sig;
fd = open("c:\sliderule\specanal\speech.bin","rb");
cnt = readV(sig, 100000, 3, fd);
close(fd);
openPlot("Speech Data");
penS(14,1,0,0,0);
ploty(0, 1, sig, Sizeof(sig)); // sig = speech signal
grid();
xLabel("Sample Number");
Title("Speech Signal");
yLabel("AMPLITUDE",0);
pCRT();
v = uniformV(N);
b = { 1, zerosV(50), -. 99 };
a = { 1, - 1.87, .96 };
v1 = filter(b, a, 3.3*v);
openPlot("Noise");
ploty(0, 1, v1, Sizeof(v1)); // v1 = noise signal
yLabel("AMPLITUDE",0);
xLabel("Sample Number");
Title("Construction Noise");
pCRT();
s = sig + v1;
openPlot("Speech + Noise"); // s = speech + noise signal
ploty(0, 1, s, Sizeof(s));
yLabel("AMPLITUDE",0);
xLabel("Sample Number");
Title("Speech plus Noise");
pCRT();
// s = signal d in block diagram
// v1 = signal u in block diagram
// L = number of tap weights in FIR adaptive filter
// w0 = calculated weights in adaptive filter
// err = filtered speech signal
wo = wienerF(v1, s, L, err);
openPlot("Filtered Speech");
penS(14,1,0,0,0);
ploty(0, 1, err, Sizeof(err));
yLabel("AMPLITUDE",0);
xLabel("Sample Number");
Title("Filtered Speech");
grid();
pCRT();
```


## Noise Canceling using the Normalized LMS Algorithm*

We can see from the previous example that a Wiener Filter does a wonderful job, however it's computational intensive in that we have to compute the signal statistics in the $1^{\text {st }}$ and $2^{\text {nd }}$ moments. It's not suitable for real time applications, so we have to turn to a more lean computational algorithm. That algorithm is the Least-Mean-Square Algorithm, commonly called the LMS algorithm. The Normalized LMS algorithm used in this simulation for vector input $\mathbf{u}$ and desired output vector $\mathbf{d}$ for an optimal weight vector $\mathbf{w}$ that solves the following equation is given by,

$$
\min _{w}=\mathrm{E}|\mathrm{~d}-\mathrm{uw}|^{2}
$$

and can be approximated iteratively via the recursion

$$
\begin{aligned}
& y_{i}=u_{i} w_{i-1}, \quad e_{i}=d(i)-y_{i} \\
& w_{i}=w_{i-1}+\frac{\mu}{\varepsilon+\operatorname{norm}\left(u_{i}\right)} u_{i}^{*}\left[d(i)-u_{i} w_{i-1}\right], \quad i .>=0, w_{-1}=\text { zero vector }
\end{aligned}
$$

where $\mu$ is a positive step - size (usually small) and $\varepsilon=1 e-6$

Repeating the previous example where we have the same speech signal corrupted by noise, we have,


[^19]Comparing this filtered output to the previous output of the previous example, we see that at the beginning that this LMS algorithm takes some time to achieve at satisfactory result, and that it's not quite as good as the Wiener filter, however the computational cost is much lower.

## Learning Curves for LMS and the NLMS Algorithms*

The LMS algorithm for vector input $\mathbf{u}$ and desired output vector $\mathbf{d}$ for an optimal weight vector $\mathbf{w}$ that solves the following equation is given by,

$$
\min _{w}=E|d-u w|^{2}
$$

can be approximated iteratively via the recursion
$y_{i}=u_{i} w_{i-1}, \quad e_{i}=d(i)-y_{i}$
$w_{i}=w_{i-1}+\mu u_{i}^{*}\left[d(i)-u_{i} w_{i-1}\right], \quad i .>=0, w_{-1}=$ zero vector
where $\mu$ is a positive step - size (usually small)
We see that the computational cost of the standard LMS algorithm is less that the Normalized LMS algorithm, so a simulation is in order to determine leaning curves for a typical application, and how much we gain in terms of convergence time with the extra computational burden.


[^20]We see from the plot above, that the NLMS algorithm convergence rate is almost 3 times faster then the LMS algorithm. From reference [1], the computational cost for complex values signals requires $8 \mathrm{M}+2$ real multiplications and 8 M real additions per iteration. On the other hand, the cost for the NLMS algorithm is $8 \mathrm{M}+6$ real multiplications, and $8 \mathrm{M}+5$ real additions.

## An Example of the Steepest Decent Algorithm*

Normally in an Adaptive Filter text book, the author presents the Steepest Decent Algorithm before launching into the LMS Algorithm and its variants. Having said that, the Steepest Decent Algorithm for vector input $\mathbf{u}$ and desired output vector $\mathbf{d}$ for an optimal weight vector $\mathbf{w}$ that solves the following equation is given by,

$$
\min _{w}=\mathrm{E}|\mathrm{~d}-\mathrm{uw}|^{2}
$$

can be approximated iteratively via the recursion

$$
\begin{aligned}
& w_{i}=w_{i-1}+\mu\left[R_{d u}-R_{u} w_{i-1}\right], \quad i .>=0, w_{-1}=\text { zero vector } \\
& y_{i}=u_{i} w_{i}, e_{i}=d(i)-y_{i} \\
& 0<\mu<2 \lambda_{\max }, \lambda_{\max } \text { is largest eigenvalue of } R_{u} \\
& R_{d u} \text { is the cross - covariance vector of }\{\mathrm{d}, \mathrm{u}\} \\
& R_{u}=E\left(u^{*} u\right), E \text { the statistical expection operator }
\end{aligned}
$$

The plots from a simple sine wave with additive Gaussian noise are shown below. The last plot shows a plot for a quadratic cost function, whereby the algorithm forces a convergence towards the bottom of the bowl along a gradient vector.

[^21]


The first top most plot shown the convergence as the algorithm converges towards the bottom of the bowl (but never gets to an exact solution).

## Echo Cancelation using a Fast Block FFT Adaptive Filter*

Using a Fast Block DFT Block Adaptive Filter, the following simulation example shows a replicated sample of speech, and the output from running this type of adaptive filter.

[^22]


The shell script for this example is shown below.

```
#include "sigsys.h"
int fd, fw, cnt;
vector sig, speech, room, echo;
vector wb, y, e;
vector Va[] = { 1 };
int N, i, j, len;
int B = 32; // block size of data
int M = 1024; // length of filter in full band
float mu = 0.1;
fd = open("c:\sliderule\specanal\speech.bin","rb");
cnt = readV(sig, 100000, 3, fd);
close(fd);
N = Sizeof(sig);
len = (N/B)*B;
speech = { };
for(i = 0; i < 8; i++)
    speech = { speech, sig[0,len] };
len *= 8;
fd = open("c:\SlideRule\digitalF\echo.bin", "rb");
cnt = readV(room, 100000, 5, fd);
close(fd);
echo = filter(room, Va, speech);
wb = fbnlms(speech, echo, mu, B, M, e, y);
openPlot("Speech Signal");
penS(14,1,0,0,0);
ploty(0, 1, speech, len);
grid();
xLabel("Sample Number");
Title("Speech Signal");
yLabel("AMPLITUDE",0);pCRT();
openPlot("Echo Signal");
penS(14,1,0,0,0);
ploty(0, 1, echo, len);
grid();
xLabel("Sample Number");
Title("Echo Signal");
yLabel("AMPLITUDE",0);
pCRT();
openPlot("error signal");
Title("Filtered Echo signal using FFT-block adaptive filter and \epsilon-
NLMS (\mu = 0.1)");
ploty(0, 1, e, Sizeof(e));
yLabel("Amplitude", 1);
grid();
```


## System Identification using an Adaptive Filter*

Mathematical models are fundamental to science and engineering fields. An adaptive filter used for providing a linear model of an unknown Plant, is a powerful tool. Note that if the Plant is dynamic in nature (i.e., time varying), that the model output will be time varying. Below is a block diagram for determining the system identification of an unknown Plant.


The following example shows some plots using this technique.


[^23]


## A 16-QAM Decision Directed Equalizer using NLMS*

The block diagram for this simulation is as follows.


The following are plots from a simulation of the above mentioned title. The shell scripts for this simulation can be found under the AdaptiveFilter tile on the main menu.





## A Blind QPSK Equalizer using Constant Modulus Alg. CMA2-2*

The following are plots using the CMA2-2 Blind Equalizer.


[^24]

## A Learning Curve for an Affine Projection Algorithm*



[^25]
## QPSK and Blind Equalizers*

The following plots show a simulation of learning curves for some blind equalizers, and some plots of results of different equalizers.


[^26]


## References

[1] Ali H. Sayed, Adaptive Filters, IEEE Press, John Wiley \& Sons, 2008.
[2] B. Farhang-Boroujeny, Adaptive Filters, Theory and Applications, John Wiley \& Sons, 2003.
[3] Simon Haykin, Adaptive Filter Theory, Third Edition, Prentice-Hall, 1996.

## Chapter 13 - FEC Channel Coding

## Introduction

In the modern world of digital computers and digital communications, error control has become an integral part in the design of modern communication and digital storage systems. In fact, systems such as satellite communications, large scale digital storage systems, deep space communications, cell phones, etc., would be virtually impossible without this technology. The basic technology is to add extra bits to a digital message or digital memory word that allows the receiver to perform error correction if the original message gets corrupted. These extra bits are called Error-Correcting Codes. This chapter will present examples of some common errorcorrecting codes used in the field of digital communications. These examples will be presented more on the side of implementation details and the how-to, rather then on the mathematical theory. For the mathematical theory and a through presentation of the underlying technology, the reader is referred to the excellent references at the end of this chapter.

## The Golay $(24,12)$ Code*

The Golay $(23,12)$ is a binary perfect code, and has a minimum distance of 7 , and is capable of correcting any combination of 3 or fewer random errors in a block of 23 digits. The $(24,12)$ Golay code is an extension of the $(23,12)$ Golay code with the addition of a parity-check bit to each code word. This code has a minimum distance of 8 , and is capable of correcting any combination of 3 or fewer random errors in a block of 24 digits, and detecting all error patterns of four digits. It has been used widely in many communications systems, including the Voyager space probe (circa 1979). This code is a systematic code, and has a generator matrix as follows,

$$
G=\left[\begin{array}{ll}
P & I_{12}
\end{array}\right]
$$

Its parity matrix is defined as,

[^27]\[

\mathrm{P}=\left[$$
\begin{array}{l}
100011101101 \\
000111011011 \\
001110110101 \\
011101101001 \\
111011010001 \\
110110100011 \\
101101000111 \\
011010001111 \\
110100011101 \\
101000111011 \\
010001110111 \\
111111111110
\end{array}
$$\right]
\]

To encode a 12-bit word in Slide-Rule, the user calls a function encodeGolay(u), which in turn calls subroutine getSynGolay(u). Each bit in the parity code word is generated from a matrix equation (in a Galois Field) given by $\mathbf{p}_{\mathbf{i}}=\mathbf{u}^{(\mathbf{i})} \mathbf{P}$. To decode a 24-bit word that's been $(24,12)$ Golay encoded, we call shell script function decodeGolay( $\boldsymbol{u R}$ ), where uR is a 24-bit encoded word. As stated above, this code is systematic where the upper 12-bits are the information bits, and the lower 12-bits are the generated code word. A test suite (file golayTest.txt found in subdirectory fec) generates all the 4096 bit combinations in a 12 -bit word; then encodes these 12 -bit words by a call to function encodeGolay(u); then injects a 1-bit error in each of the encoded 24bit words (uR); then calls function decodeGolay(uR) to decode each of the 4096 words, then checks for errors. The process repeats for 2-bit errors, 3-bit errors, and finally 4-bit errors. Below is a print out of the results of this simulation.

```
Demo of the (24,12)Golay Code
(1) Generate 4096 12-bit patterns
(1b) Encode these patterns into 24-bit Golay (24,12) code(s)
(1c) A random 1 bit error in each code
(1d) Check for errors in the 4096 patterns
(2) Repeat (1), but with 2 errors for each code
(3) Repeat (1), but with 3 errors for each code
(4) Repeat (1), but with 4 errors for each code
No errors detected for 1 bit error in code
No errors detected for 2 bit errors in code
No errors detected for 3 bit errors in code
4 0 9 6 ~ e r r o r s ~ d e t e c t e d ~ f o r ~ 4 ~ b i t ~ e r r o r s ~ i n ~ c o d e
```

Note that the functions as stated above can be found in shell script file golay.txt which resides in sub-directory sigsys.

## The BCH Error Correcting Cyclic Codes*

The Bose, Chaudhuri, and Hocquenghen (BCH) codes are a class of powerful random errorcorrection cyclic codes.. These codes are an extension of the Hamming codes for multiple binary error correcting capabilities. For a 1-bit error correcting code, the Hamming and BCH codes are equivalent for the same generator polynomial. BCH codes are defined by the following:

| Block length: | $\mathrm{n}=2^{\mathrm{m}}-1$, |
| :--- | :--- |
| Number of parity-check digits | $\mathrm{n}-\mathrm{k}<=\mathrm{mt}$. |
| Minimum distance | $d_{\text {min }}>=2 \mathrm{t}+1$ |

From the above, we see that the code is capable of correcting any combination of $t$ or fewer errors in a block of $2^{\mathrm{m}}-1$ binary digits. BCH codes are commonly referenced as $\mathrm{BCH}(\mathrm{n}, \mathrm{k}, \mathrm{t})$. As an example, the BCH code $(255,187,9)$ has a block length of 255 bits, 187 message bits, and an error correction capability of 9 bits. The $\mathbf{g}(\mathrm{x})$ generator polynomial for a t-error correcting BCH code of length $2^{\mathrm{m}}-1$, is the lowest degree polynomial in $G F(2)$ that has $\alpha, \alpha^{2}, \alpha^{3}, \cdots, \alpha^{2 t}$ as its roots. It turns out that the conjugates of the above sequence (i.e., the even powers of $\alpha$ have the same roots as the odd powers. For the minimum polynomial $\phi_{i}(X)$ of $\alpha^{i}$, the generator polynomial $g(\mathrm{X})$ is defined as the least common multiple (LCM) of $\phi_{1}(X), \phi_{3}(X), \cdots, \phi_{2 t-1}$. One way of getting the generator polynomial is to look up the polynomial in Appendix $C$ of reference [1]. However, the octal digits sequences are rather lengthy, and the chance that one makes an error is large. Beside, it's a tedious process at best. At better way is to call a subroutine. Such a subroutine exists in Slide-Rule, namely function $\boldsymbol{g e n} \mathbf{p o l y B C H}(\mathbf{m}, \boldsymbol{n}, \boldsymbol{k}, \boldsymbol{t})$, where m is $G F\left(2^{\mathrm{m}}\right)$, n is the block length $\left(\mathrm{n}=G F\left(2^{\mathrm{m}}\right)-1\right)$, k is the message length, and $t$ is the number of errors that can be corrected in a block. Before this subroutine is called, the user must call function $\operatorname{initBCH}(\mathbf{n}, \boldsymbol{k}, \boldsymbol{t}$, Ptype), where n is the block length, k is the message length, t is the error capability, and Ptype is either the big indian or little indian sense of $g(X)$. This subroutine calls subroutine generate $\boldsymbol{g f B C H}$ to generate the alpha_to and index_of lookup tables for doing arithmetic in a Galois Field. Subroutine gen_polyBCH finds the all the roots based on $m$ and the value of $t$. As an example, for $m=9$, and for $\mathrm{t}=3$; then there's the outer loop from $0<\mathrm{i}<\mathrm{t}$; and an inner loop from $1<\mathrm{j}<\mathrm{m}$, that generates all the roots by doubling the previous starter root modulo n . For the first iteration in the outer loop for this example, the roots go $1,2,4,8,16,32,64,128$, and 256 . For the next iteration, the roots go $3,6,12,24,48,96,192,284,257$, and the last sequence for this example, the roots go $5,10,20,40,80,160,320,129,258$. Note that after each iteration of the inter loop, a check is made such that for any root generated in the inner loop that matches a root in the stored roots array, then none of the roots generated in that inner loop are stored in the roots array. Otherwise, all the roots for that iteration are stored in the roots array. When this process is complete, we then generate the generator polynomial. The following is a listing of gen_polyBCH which shows how $g(\mathrm{X})$ is generated. Note all of the subroutines as described above are in shell script file bch as found in folder sigsys under the SlideRule directory.

```
void gen_polyBCH(int m, int nn, int k, int t)
{
    int alphaP[256][10], roots[1024];
    int i, j, ll, kk, tmp, numBits;
```

[^28]```
int rootF, ii, jj, reg, cnt;
// generate alpha powers
alphaP[0][0] = 1;
for(i = 0, numBits = 0; i < t; i++) {
    for(j = 1; j < m; j++) {
        tmp = (alphaP[i][j-1]*2)%nn;
        if(tmp != 0 && tmp != alphaP[i][0]) alphaP[i][j] = tmp;
        }
        // check for previous alpha powers in set
        // if found, don't include alpha powers in roots[]
        for(ll = 0, rootF = 0; ll < m; ll++) {
            tmp = alphaP[i][ll];
            if(tmp == 0) break;
            for(kk = 0; kk < i-1; kk++) {
                for(j = 0; j < m; j++)
                    if(tmp == alphaP[kk][j]) { rootF = 1; break; }
                    if(rootF) break;
            }
            if(rootF) break;
        }
        if(!rootF) { // Include alpha powers on root[]
            for(j = 0; j < m; j++) {
                tmp = alphaP[i][j];
                if(tmp != 0) { roots[numBits+1] = tmp; numBits++; }
                else break;
            }
        }
            alphaP[i+1][0] = alphaP[i][0]+2;
}
// Compute the generator polynomial
g[0] = alpha_to[roots[1]];
g[1] = 1; // g(x) = (X + roots[1]) initially
for (ii = 2; ii <= numBits; ii++) {
    g[ii] = 1;
    for (jj = ii - 1; jj > 0; jj--)
        if (g[jj] != 0)
            g[jj] = g[jj - 1] ^ alpha_to[(index_of[g[jj]] + roots[ii]) % nn];
        else
            g[jj] = g[jj - 1];
    g[0] = alpha_to[(index_of[g[0]] + roots[ii]) % nn];
}
```

\}

For the encoding of BCH codes, we call the following subroutine in Slide-Rule, namely encodeBCH( $\boldsymbol{n}$, len, int *data, int ${ }^{*} \boldsymbol{b b}$ ), where $\mathbf{n}=G F\left(2^{m}-1\right)$, len is the message length, data is the integer array of length len to be encoded, and $\mathbf{b b}$ is the parity array for storing the BCH code of length $\mathbf{n}$ - len.
For the decoding of BCH codes, we call the following subroutine in Slide-Rule, namely $\operatorname{decodeBCH}\left(\mathbf{n}\right.$, len, int $\boldsymbol{t}$, int *recd), where $\mathbf{n}=G F\left(2^{\mathrm{m}}-1\right)$, len is the message length before encoding, $\mathbf{t}$ is the error correcting capability of the code, and recd is the BCH encoded message of length $\mathbf{n}$.
The steps for decoding BCH codes are:

1) Calculate the syndromes
2) Determine the error location polynomial by the Berlekamp-Massey Algorithm
3) Finding the roots and their location by the Chien Search
4) Make the error corrections by Forney's Algorithm

Note that the functions as stated above can be found in shell script file bch.txt which resides in sub-directory sigsys. In Slide-Rule, there's a BCH test suite in file bchTest.txt found in subdirectory fec.


This is a (511, 121, 58) binary BCH code BCH code(511,121,58)
Generator polynomial (in octal):
$g(x)=1541271357655772525604106044116552170135$ 4434750270764625323125324327677742325121 2771735535703734317405374311756764357501 74233577257
Random data block, length = 121
1st octal digit = 1 bit(s)
0012541063115447246627031666733272125160
3
Coded data block (parity+data block), length = 511
1st octal digit = 1 bit(s)
0464531112341753440671457202042226167241
0642710120053007736544210243762007002700
4323405263542503322721551210133343031443
3446466435201254106311544724662703166673
32721251603
Recieved data block, length = 511
1st octal digit = 1 bit(s)
0464533113341613440671457400042227167041 2732730321053006736444214243720007202500 4133405063546543024361557200532343232463 3746426404205254126310544724660707166673 22621251602
Transmit data, length = 511
1st octal digit = 1 bit(s)
0464531112341753440671457202042226167241
0642710120053007736544210243762007002700 4323405263542503322721551210133343031443 3446466435201254106311544724662703166673 32721251603
Recovered data, length = 511
1st octal digit = 1 bit(s)
0464531112341753440671457202042226167241
0642710120053007736544210243762007002700 4323405263542503322721551210133343031443 3446466435201254106311544724662703166673 32721251603
Succesful decoding!!, number of errors $=58$
Error positions corrected
510486480457449423410394387384
383373363362353348347341330322

| 318 | 311 | 310 | 298 | 295 | 293 | 292 | 288 | 287 | 280 |
| ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: |
| 274 | 260 | 246 | 242 | 230 | 221 | 209 | 205 | 190 | 177 |
| 165 | 147 | 140 | 134 | 126 | 125 | 124 | 123 | 119 | 110 |
| 99 | 80 | 74 | 73 | 40 | 39 | 27 | 17 |  |  |

## A Reed-Solomon 255, 223 with erasures*

The Reed-Solomon encoder/decoder is used widely in communication systems where burst errors occur. The following test using a 255,223 code corrects 32 byte errors when the erasure positions in the block are known. This test file can be found under the SideRule tile A ReedSolomon 255,223 with erasures tile. Note that the functions as stated below in the code can be found in shell script file reedsol.txt which resides in sub-directory sigsys. Other tests include $A$ Reed-Solomon 255,239 without erasures, A Reed-Solomon 28,24 without erasures, and A ReedSolomon 32,28 with erasures.

```
Int data[] = { // Test block of 223
114,119, 24, 73, 97,112,242, 85,228,125,
187,112,222,153, 71, 65, 59, 91,176, 92,
106, 52,173,204, 65, 34,203,197,126,117,
247, 18,115,192,197, 10,188,130, 60, 94,
207, 54, 1,221,202,171, 92,205, 82,156,
    74, 7,185,161,124,180,138,125, 22,142,
    68,229, 207,240, 13, 75, 35, 33,133,162,
    61,115,132,174,130, 15,219,239, 50,137,
210,113,144,155,183, 53,200,249, 92,157,
194, 99,214, 80,146,192,156, 64,247, 73,
110, 39,225, 19,139, 2,142, 57,161, 8,
114,184,134,155, 49, 81, 17,120,120,228,
210,146,191,176,173, 55,193,102, 67,159,
    70,181,160,143, 36, 37,139, 13,170,239,
    82, 73,145,180,110,254, 59, 95,154,122,
224, 73, 61,191,147, 88,115,121, 62,174,
    73,242, 19,211,164,165,129,143,186,104,
155,176, 20,204, 62,242,136, 8,228,196,
148, 49, 62, 40, 5,193, 94,243, 88,166,
207,191,139,170, 97, 53,137,207,147,119,
154,229, 93,217, 15, 55,202,239,141,199,
    63,217, 22,229, 99, 49,247,122, 10, 13,
    12,186, 209
    };
    int i, j, count, parity[32], recd[255], eras_pos[32];
    int pp[] = { 1, 0, 1, 1, 1, 0, 0, 0, 1 };
initRSF(8, 255, 223, pp);
encodeRSF(255, 223, 223, data, parity);
Print(data); Print(parity);
for(i = 0; i < 223; i++) recd[i] = data[i];
for(j = 0; i < 255; i++, j++) recd[i] = parity[j];
printf("Good Block "); Print(recd);
for(i = 0, j = 5; i < 32; i++, j += 5)
{
    recd[j] = 6; eras_pos[i] = j;
}
printf("Bad Block 32 errors "); Print(recd);
count = decodeRSF(255, 223, 255, recd, eras_pos, 32);
Print(count);
```

[^29]| Good Block array(int) -> recd(255) |  |  |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 000000 | 114 | 119 | 24 | 73 | 97 | 112 | 242 | 85 | 228 | 125 |
| 000010 | 187 | 112 | 222 | 153 | 71 | 65 | 59 | 91 | 176 | 92 |
| 000020 | 106 | 52 | 173 | 204 | 65 | 34 | 203 | 197 | 126 | 117 |
| 000030 | 247 | 18 | 115 | 192 | 197 | 10 | 188 | 130 | 60 | 94 |
| 000040 | 207 | 54 | 1 | 221 | 202 | 171 | 92 | 205 | 82 | 156 |
| 000050 | 74 | 7 | 185 | 161 | 124 | 180 | 138 | 125 | 22 | 142 |
| 000060 | 68 | 229 | 207 | 240 | 13 | 75 | 35 | 33 | 133 | 162 |
| 000070 | 61 | 115 | 132 | 174 | 130 | 15 | 219 | 239 | 50 | 137 |
| 000080 | 210 | 113 | 144 | 155 | 183 | 53 | 200 | 249 | 92 | 157 |
| 000090 | 194 | 99 | 214 | 80 | 146 | 192 | 156 | 64 | 247 | 73 |
| 000100 | 110 | 39 | 225 | 19 | 139 | 2 | 142 | 57 | 161 | 8 |
| 000110 | 114 | 184 | 134 | 155 | 49 | 81 | 17 | 120 | 120 | 228 |
| 000120 | 210 | 146 | 191 | 176 | 173 | 55 | 193 | 102 | 67 | 159 |
| 000130 | 70 | 181 | 160 | 143 | 36 | 37 | 139 | 13 | 170 | 239 |
| 000140 | 82 | 73 | 145 | 180 | 110 | 254 | 59 | 95 | 154 | 122 |
| 000150 | 224 | 73 | 61 | 191 | 147 | 88 | 115 | 121 | 62 | 174 |
| 000160 | 73 | 242 | 19 | 211 | 164 | 165 | 129 | 143 | 186 | 104 |
| 000170 | 155 | 176 | 20 | 204 | 62 | 242 | 136 | 8 | 228 | 196 |
| 000180 | 148 | 49 | 62 | 40 | 5 | 193 | 94 | 243 | 88 | 166 |
| 000190 | 207 | 191 | 139 | 170 | 97 | 53 | 137 | 207 | 147 | 119 |
| 000200 | 154 | 229 | 93 | 217 | 15 | 55 | 202 | 239 | 141 | 199 |
| 000210 | 63 | 217 | 22 | 229 | 99 | 49 | 247 | 122 | 10 | 13 |
| 000220 | 12 | 186 | 209 | 146 | 52 | 115 | 140 | 94 | 85 | 208 |
| 000230 | 186 | 234 | 164 | 44 | 111 | 0 | 247 | 210 | 6 | 28 |
| 000240 | 32 | 157 | 156 | 38 | 54 | 106 | 68 | 167 | 162 | 253 |
| 000250 | 154 | 50 | 184 | 219 | 216 |  |  |  |  |  |



| 000010 | 187 | 112 | 222 | 153 | 71 | 65 | 59 | 91 | 176 | 92 |
| :--- | ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: |
| 000020 | 106 | 52 | 173 | 204 | 65 | 34 | 203 | 197 | 126 | 117 |
| 000030 | 247 | 18 | 115 | 192 | 197 | 10 | 188 | 130 | 60 | 94 |
| 000040 | 207 | 54 | 1 | 221 | 202 | 171 | 92 | 205 | 82 | 156 |
| 000050 | 74 | 7 | 185 | 161 | 124 | 180 | 138 | 125 | 22 | 142 |
| 000060 | 68 | 229 | 207 | 240 | 13 | 75 | 35 | 33 | 133 | 162 |
| 000070 | 61 | 115 | 132 | 174 | 130 | 15 | 219 | 239 | 50 | 137 |
| 000080 | 210 | 113 | 144 | 155 | 183 | 53 | 200 | 249 | 92 | 157 |
| 000090 | 194 | 99 | 214 | 80 | 146 | 192 | 156 | 64 | 247 | 73 |
| 000100 | 110 | 39 | 225 | 19 | 139 | 2 | 142 | 57 | 161 | 8 |
| 000110 | 114 | 184 | 134 | 155 | 49 | 81 | 17 | 120 | 120 | 228 |
| 000120 | 210 | 146 | 191 | 176 | 173 | 55 | 193 | 102 | 67 | 159 |
| 000130 | 70 | 181 | 160 | 143 | 36 | 37 | 139 | 13 | 170 | 239 |
| 000140 | 82 | 73 | 145 | 180 | 110 | 254 | 59 | 95 | 154 | 122 |
| 000150 | 224 | 73 | 61 | 191 | 147 | 88 | 115 | 121 | 62 | 174 |
| 000160 | 73 | 242 | 19 | 211 | 164 | 165 | 129 | 143 | 186 | 104 |
| 000170 | 155 | 176 | 20 | 204 | 62 | 242 | 136 | 8 | 228 | 196 |
| 000180 | 148 | 49 | 62 | 40 | 5 | 193 | 94 | 243 | 88 | 166 |
| 000190 | 207 | 191 | 139 | 170 | 97 | 53 | 137 | 207 | 147 | 119 |
| 000200 | 154 | 229 | 93 | 217 | 15 | 55 | 202 | 239 | 141 | 199 |
| 000210 | 63 | 217 | 22 | 229 | 99 | 49 | 247 | 122 | 10 | 13 |
| 000220 | 12 | 186 | 209 | 146 | 52 | 115 | 140 | 94 | 85 | 208 |
| 000230 | 186 | 234 | 164 | 44 | 111 | 0 | 247 | 210 | 6 | 28 |
| 000240 | 32 | 157 | 156 | 38 | 54 | 106 | 68 | 167 | 162 | 253 |
| 000250 | 154 | 50 | 184 | 219 | 216 |  |  |  |  |  |
| $]$ |  |  |  |  |  |  |  |  |  |  |

## The Reed-Solomon CIRC*

The Cross-Interleave Reed-Solomon Code (CIRC) is part of the standard compact disc (CD) digital audio system developed in 1979 by the Philips Corp. of the Netherlands and Sony Corp. of Japan, for the storage and reproduction of audio signals. The audio signals (right and left channels) are alternately sampled by a 16 -bit A/D at 44,100 samples $/ \mathrm{sec}$., Reed-Solomon encoded, and recorded on an audio CD disc. The encoding scheme uses two Reed-Solomon codes, namely an outer RS $(28,24)$ code, and an inner RS $(32,28)$ code, with interleaving. Both codes are shortened versions of the RS $(255,251)$ code. Each code is capable of correcting two bytes out of a block, or 4 bytes if the error locations (erasures) are known. Below is a block diagram (from [2], pgs 469-475) of the CIRC encoder and decoder.


[^30]The details of the encoder and decoder can be found below in the labeled block diagrams. In order to investigate the CIRC properties and inter-workings in terms of the error correcting code capabilities of the CIRC algorithm, we perform a simulation using the Reed-Solomon shortened codes as stated above, and implement the detailed block diagrams as found below for the CIRC encoder and decoder. We first generate two sine waves with the number of appropriate samples, where the right channels maximum peak is approximately half the range of a 16 -bit $\mathrm{A} / \mathrm{D}$, and the left channel is approximately one quarter the range of a 16 -bit $\mathrm{A} / \mathrm{D}$, and with both channels in phase. A plot of the input follows.


For the first part of the simulation we encode the inputs and perform the CIRC encoding, and then follow with the CIRC decoding scheme, but with no erasures. Some plots follow.


We see on the left plot, that if 2064 consecutive bits (or 258 bytes) are corrupted on the CD, that the CIRC algorithm (with erasures turned off) successfully restores the corrupted data, while for 300 bytes (or 2400 bits) corrupted, that the output audio signals are garbled.

We repeat the simulation, but with erasures turned on. Some plots follow.


Now for this case, we see on the left plot, that if 4128 consecutive bits (or 516 bytes) are corrupted on the CD, that the CIRC algorithm (with erasures turned on) successfully restores the corrupted data, while for 550 bytes (or 4400 bits) corrupted, that the output audio signals are garbled. Referring the detailed decoder block diagram, for the Inner decoder $\mathrm{D}_{1}$, when the RS ( 32.28 ) fails (i.e., more than 2 correctable errors detected, that we mark each of the 28 bytes as an erasure before the data bytes are input to the Cross de-interleaver. Notice that each byte arrives as input to the Outer decoder $\mathrm{D}_{2}$ at different times, such that the RS $(28,24)$ decoder can successfully correct up to four erasures. If more than 4 erasures, then he Outer decoder simply passes the marked data bytes on to the Delta de-interleaver.
Now, since each stereo channel is sampled at 22050 16-bit samples per second, or approximately 45 micro-seconds between samples, we can perform linear interpolation to restore a corrected sample, i.e., we just grab the next good sample, add the two/ and divide by two. Some plots follow.



The simulation as presented above was done by calling fast versions (i.e., internal functions) of the Reed-Solomon encoder and decoder. However, a shell script version of the same functions can be found as file reedsol.txt in sub-directory sigsys. The code handles shortened versions of RS codes, and has been tested for the 28,24 and 32,28 RS codes. Because of the shortened codes, the reader should note that if examining the code, that the following area's have been modified because of shortened codes; the syndrome calculation of the received message (or polynomial); the calculation of the erasure location polynomial if erasures, the calculation of the error locations in the Chien Search. Also, in the CIRC simulation, its was found that the RS 32,28 decoder incorrectly reported that it had corrected 2 bytes, but upon closer examination, it was found not to be true. This happened very rarely, and when every other byte in the 32 byte block was corrupted. Further analysis found that the error was data dependent, and that the error locations reported by the Chien Search were found beyond the 32 byte boundary. So after the Chien Search, code was put into the algorithm that if doing a shortened code, and any error location reported in 'no-mans land', that the algorithm reports a decode failure.


Encoder Block Diagram


## Decoder Block Diagram

## A Rate K = 1/2 Viterbi Decoder*

A Rate $=1 / 2$ Viterbi encoder/decoder exits in Slide-Rule as a fast internal version for simulations, and a shell script version with the same functional code and with identical results and prototypes (except name). The shell script version allows users to examine the code implementation, since trying to figure out the how a Viterbi decoder works from a text book for someone new to this algorithm, is not a trivial task. The current version in Slide-Rule allows for constraint lengths of $\mathrm{K}=3,5,7$, and 9 . Also, the decoder outputs after decoding an entire message, and doesn't output a continuous stream after the decoder has reached 5 K symbols. Below is a BER plot of a simulation of a BPSK channel with additive Gaussian noise for difference constraint lengths K and $\mathrm{Eb} / \mathrm{No}(\mathrm{dB})$ levels.

[^31]

## Remarks on the Viterbi Encoder and Decoder

The Rate $=1 / 2$ Viterbi Encoder as implemented in Slide-Rule, encodes an additional K-1 message symbols, with the result that an additional $2^{*}(\mathrm{~K}-1)$ bits are output from the encoding process. As an example, for a 100 -BPSK message, the encoder outputs 208-bits for $\mathrm{K}=5$. The reason for this is to bring the decoder back to the zero-state, since it's started off in the zero-state. Note that for an un-coded channel, $E_{S} / N_{O} d B=E_{B} / N_{O} d B$. However, for BPSK we have $E_{S} / N_{O} d B=E_{B} / N_{O} d B .+10 \log 10(\mathrm{k} / \mathrm{n})$ $=E_{B} / N_{O} d B .+10 \log 10(1 / 2)=E_{B} / N_{O} d B .-3.01 \mathrm{~dB}$. In an AWGN channel, the signal is corrupted by additive Gaussian noise, which has a power spectrum of $\mathrm{N}_{0} / 2$ watts $/ \mathrm{Hz}$. The variance $\sigma^{2}$ of this noise is equal to $N_{0} / 2$. If the energy per symbol $E_{S}$ is set equal to 1 , then $E_{S} / N_{0}=1 / 2 \sigma^{2}$. Then $S N R=10^{(E s / N o d B) / 10}$ such that the Gaussian noise level is set by $\sigma=\sqrt{1 /(2 * S N R)}$ times the output of a Gaussian noise source of power level 1. Note for the data points in the above simulation, that each data point represent a random 100-bit encoded message repeated 20,000 times. However, for the $\mathrm{K}=9$ at $E_{B} N_{O}=4 \mathrm{~dB}$, only 1 decoder error was forth coming for a 100-bit message repeated 200,000 times. Note, besides the fast internal coded Viterbi Encoder/Decode, there exists a shell script version that can be found in sub-directory sigsys with the file name of viterbi.txt. The user can run additional Viterbi simulations running A Viterbi Rate $1 / 2 K=7(133,177)$ Demo and A Viterbi Rate $2 / 3 K=7(133,177)$ (punctured) shell scripts by selecting these tiles under the FEC/OFDM tile under the main menu. We note that the majority of block codes are obtained by algebraic means, while convolution codes are found only by computer simulations. Only codes with no catastrophic error propagation are considered. For the punctured code above, we note that this is not the best Rate $=2 / 3 \mathrm{~K}=7(133,177)$ code. A better punctured code would be $\mathrm{R}=7(163,135), 163$ which has a free distance of 6 .*

[^32]
## Turbo Coding*

Turbo coding is an iterative decoding technology that is widely used in deep space communication applications. These codes have exceptionally good performance for block lengths greater then about 10000 bits, and with iterative decoding, can achieve BER's as low as $10^{-5}$ in SNR's within 1 dB of the Shannon limit. For this example, we turn to the logmap algorithm, which is a modification of the BCJR algorithm devised by Bahl, Cocke, Jelinck, and Raviv in 1974. In particular, the logmap algorithm substitutes additions for multiplications to reduce the computational burden and to solve overflow issues. As shown in the block diagram below, the algorithm uses a recursive encoding technique, to generate a bit stream 3 times the length of the input data stream.


Turbo encoder
The following is the state transition diagram and trellis for a $(1,5,7)$ RSC code.


[^33]The MAP decoding algorithm is based on the Log Likelihood Ratio (LLR), or

$$
L\left(c_{k}^{1}\right) \cong \ln \left[\frac{P\left(c_{k}^{(1)}=+1 \mid \vec{r}\right)}{P\left(c_{k}^{(1)}=-1 \mid \vec{r}\right)}\right] \quad \text { or } \quad c_{k}^{(1)}=\operatorname{sign}\left[L\left(c_{k}^{(1)} \mid \vec{r}\right)\right]
$$

Incorporating the trellis structure as shown above, this can be reformulated as follows:

$$
L\left(c_{k}^{(1)}\right)=\ln \left[\frac{\sum_{S^{+}} P\left(s_{s k-1}=S^{\prime}, s_{k}=S, \vec{r}\right)}{\sum_{S^{-}} P\left(s_{s k-1}=S^{\prime}, s_{k}=S, \vec{r}\right)}\right]=\ln \left[\frac{\sum_{S^{+}} \alpha_{k-1}\left(S^{\prime}\right) \gamma_{k}\left(S^{\prime}, S\right) \beta_{k}(S)}{\sum_{S^{-}} \alpha_{k-1}\left(S^{\prime}\right) \gamma_{k}\left(S^{\prime}, S\right) \beta_{k}(S)}\right]
$$

First $\gamma_{k}$ is computed as follows:

$$
\gamma_{k}\left(S^{\prime}, S\right)=\exp \left(\frac{L_{c}}{2} \sum_{i=1}^{2} c_{k}^{i} r_{k}^{i}\right) \exp \left[\frac{L^{e x t}\left(c_{k}^{1}\right)}{2}\right]
$$

Since the coding rate for BPSK $\mathrm{R}=1 / 2$, the tern $\mathrm{L}_{\mathrm{c}}$ is calculated as follows:

$$
L_{c}=4 R \frac{E_{b}}{N_{0}}=2 \frac{E_{b}}{N_{0}}
$$

Both $\alpha_{k}$ and $\beta_{k}$ are computed recusively as follows:

$$
\begin{aligned}
\alpha_{k}(S)=\sum_{\text {all } '^{\prime}} \gamma_{k}\left(S^{\prime}, S\right) \alpha_{k-1}\left(S^{\prime}\right) & \alpha_{0}(0)=1, \alpha_{0}(S \neq 0)=0 \\
\beta_{k-1}\left(S^{\prime}\right)=\sum_{\text {all } S} \gamma_{k}\left(S^{\prime}, S\right) \beta_{k}(S) & \beta_{N}(0)=1, \beta_{n}(S \neq 0)=0
\end{aligned}
$$

Note from the above equations, that $S$ ' is computed forward (to the next state) through the trellis, while $S$ is computed backward (from the previous state). Note also that $\alpha$ is computed from the beginning of the bit stream, while $\beta$ is computed from the end of the bit stream and backwards to the beginning. The equations as presented above are in essence the BCJR algorithm. We can cut down on the number of multiplies and overflow conditions, by taking the natural logarithm and exponentials of the above equations. So we have as follows:

$$
\begin{gathered}
\Gamma_{k}\left(S^{\prime}, S\right)=\frac{L_{c}}{2} \sum_{i=1}^{2} c_{k}^{i} r_{k}^{i}+\frac{L^{e x t}\left(c_{k}^{1}\right)}{2} \\
A_{k}(S)=\ln \left(\sum_{\text {all } \left.S^{\prime} e^{\Gamma\left(S^{\prime}, S\right) \mathrm{A}_{k-1}\left(S^{\prime}\right)}\right)} \text { and } B_{k-1}\left(S^{\prime}\right)=\ln \left(\sum_{\text {all } S} e^{\Gamma\left(S^{\prime}, S\right) \mathrm{B}_{k}(S)}\right)\right.
\end{gathered}
$$

Using the following approximation and the Jacobian logarithm to avoid summation of exponentials, we have:

$$
\ln \left(\sum_{i} e^{x_{i}}\right) \approx \max \left(x_{i}\right) \quad \text { and } \quad \ln \left(e^{x}+e^{y}\right)=\max (x, y)+\ln \left(1+e^{-|x-y|}\right)
$$

Based on the above equations, we have,

$$
\begin{aligned}
& A_{k}(S)=\max _{S^{\prime}}\left[\Gamma\left(S^{\prime}, S\right)+A_{k-1}\left(S^{\prime}\right)\right] \quad \text { and } \quad B_{k-1}\left(S^{\prime}\right)=\max _{S}\left[\Gamma\left(S^{\prime}, S\right)+B_{k}(S)\right] \\
& L L r_{k}=\max _{S^{+}}\left[A_{k-1}\left(S^{\prime}\right)+B_{k}(S)+\Gamma_{k}(S, S)\right]-\max _{S^{-}}\left[A_{k-1}\left(S^{\prime}\right)+B_{k}(S)+\Gamma_{k}\left(S^{\prime}, S\right)\right], \text { and } \\
& L_{k}^{e x t}=L L r_{k}-L c r_{k}^{1}-L_{k}^{e x t}
\end{aligned}
$$

Based on the above, the user can view shell script file logmap.txt found in sub-directory sigsys to get a clearer understanding of the logmap turbo encoder/decoder algorithm. For an example of this algorithm, a simulation follows. The interleaver used in this example is a helical interleaver of length 24 given by the following:
$1,6,11,16,21,2,7,12,17,22,3,8,13,18,23,4,9,14,19,24,5,10,15,20$
For the simulation, we run two cases as follows:

1) A random serial bit stream of 0 's and 1 's of length 1784 bits is generated. This data stream is then converted into 2238 -bit words. This data generates a 32 byte parity block by calling a Reed Solomon 255,223 encoder (encodeRS). The parity block is then converted into a serial bit stream. A bit stream (C1k) is then generated which consists of 24 leading 0 's; the 1784 random bit stream; the RS serial parity stream; and finally 24 trailing 0 's for a total length of 2088 bits of 0 's and 1's. We then RSC generate C2k as shown is the Turbo encoder block diagram. C3k is then interleaved and RSC generated. We then take the C1k bit stream and interleave it to form C1kI. The data streams are then converted to BPSK by multiplying by -2 and adding +1 . Gaussian noise is then added as previously discussed under Remarks on the Viterbi Encoder and Decoder. These streams* are then input into the logmap iterative turbo decoder a shown is the block diagram below.


Structure of turbo decoder
The simulation is then run, and only turbo decoder errors are counted for the 2040 bits within the zero header and trailer padding. After the turbo errors are counted, the Reed Solomon 255,223 decoder is run, and only errors in the 223 data block are counted.
2) In a similar manner, we run a random bit stream of 65520 , and pad with 24 leading and 24 trailing zeros. For this case, we do just turbo decoding.

## Remarks on the Simulation

It was found through a number of trials, that padding of equal length to the size of the interleaver (24), gave better results. This is probably do to the fact that the Lext array started off at zero. It was also noted that the errors generated during simulation runs occurred in patches, much like a green lawn, where grass just doesn't seem to grow in certain areas. That was the primary reason the Reed Solomon encoder/decoder $(255,223)$ was incorporated to improve the BER level per Eb/N0 noise level. The code for the simulation runs used a fast version of the algorithm, namely logmapF. The simulation for the message length of 2040 can be found as shell script file testLogMapF.txt in sub-directory fec. The code as shown in shell script file logmap.txt is specific to the $\operatorname{RSC}(1,5,7)$ recursive code, which should make it easily for the user to incorporate this example into a high speed signal processor or in an ASIC design. We note that the term $\ln \left(1+e^{-|x-y|}\right)$ could be implemented in a LUT, which would greatly speed up a custom design and implementation. One could also implement the max-logmap algorithm, but suffer a 0.35 dB performance loss, as documented in the literature.

[^34]The results for 7 iterations (which end in Decoder I) are shown in the following plot. Note for case 1 , that averages are taken over 100 messages, while for case 2 , an average is taken over 10 messages.

[1] Shu Lin, Daniel J. Costello Jr, Error Control Coding, , Pearson Education, Inc., 2011
[2] Bernard Sklar, Digital Communications, Fundamentals and Applications, Second Edition Prentice-Hall, 2004.
[3] John G. Proakis and Masoud Salehi, Digital Communications, Fifth Edition, McGraw Hill, 2008.
[4] Chip Fleming, A Tutorial on Convolutional Coding with Viterbi Decoding, Spectrum Applications
[5] Yuan Jiang, A Practical Guide to Error-Control Code Using MATLAB, Artech House, 2010
[6] Shu Lin, An Introduction to Error-Correction Codes, Prentice-Hall, 1970

## Chapter 14 - Matrix/Vector Computations

## Introduction

This chapter presents some basic Matrix/Vectors Computations that arise over and over again in the fields of science and engineering. Functions such as the singular value decomposition (svd), the $\mathbf{q r}$ function, etc., are interfaced to the LAPACK routines (version 3.1.1, 2/26/2007). This library was built from the C code downloaded from Netlib, and compiled under a C++ compiler. It consists of the double precision and complex double precision modules, and consists of $\sim 300,000$ lines of code. The current implementation in Slide-Rule is a small subset of this library, but can easily be expanded for more advanced implementations such as the sparse, banded, etc. matrix methods. The original LAPACK project was funded by the NSF with an initial release in February, 1992. Contributors include: University of Tennessee; Oak Ridge National Laboratory; Argonne National Laboratory; University of California, Berkeley; Cray Research, Inc.; Numerical Algorithms Group Ltd.; Rice University; University of Kentucky; Courant Institute of Mathematical Sciences, New York University; and others. Supersedes LINPACK, EISPACK, and the BLAS.

The intent of this chapter is to show how to use these functions, and not go into the mathematical theory. For that, the reader is referred to the excellent references provided at the end of this chapter. Note, the prototypes to these functions can be found in the software under function key F7 -> Matrix/Vector Operations.

## The QR Decomposition with Column Pivoting

The QR decomposition or factorization of an m-by-n matrix $A$ is given by $A=Q R P^{T}$, where matrix $Q$ is orthogonal, matrix $R$ is upper triangular, and $P$ is a permutation matrix.* An example follows.

```
matrix A[][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
matrix Q, R, P, X;
matcmplx Ac[][3] = { 19+19j,-1-1j,1+1j,-2-2j,1+1j,-20-20j,-1-1j,1+1j,
                    2+2j,1+1j,21+21j,-1-1j };
matcmplx Qc, Rc, Xc;
printf("Input "); Print(A);
printf("Rank of A = %f\n", rank(A));
P = qr(A, Q, R); printf("Orthogonal output "); Print(Q);
printf("Upper Triangular output "); Print(R);
printf("Permutation "); Print(P);
X = Q*R*P'; printf("Q*R*P' = "); Print(X);
P = qr(Ac, Qc, Rc);
printf("Orthogonal output "); Print(Qc);
printf("Upper Triangular output "); Print(Rc);
printf("Permutation "); Print(P);
Xc = Qc*Rc*P'; printf("Qc*Rc*P' = "); Print(Xc);
Input = matrix -> A(4,3)
[row
0000 1 2
0001 4 5 5
0002 7 7 8
0003 10 11 12
]
```

[^35]Rank of $A=2.000000$
Orthogonal output matrix -> $\mathrm{Q}(4,4)$
[row

| 0000 | -0.18257 | -0.8165 | 0 | 0 |
| ---: | ---: | ---: | :--- | :--- |
| 0001 | -0.36515 | -0.40825 | 0 | 0 |
| 0002 | -0.54772 | 0 | 0 | 0 |
| 0003 | -0.7303 | 0.40825 | 0 | 0 |

]
Upper Triangular output matrix -> R(4,3)
[row

| 0000 | -16.432 | -12.78 | -14.606 |
| :--- | ---: | ---: | ---: |
| 0001 | 0 | 1.633 | 0.8165 |
| 0002 | 0 | 0 | 0 |
| 0003 | 0 | 0 | 0 |

0003
Permutation matrix -> $P(3,3)$
[row

| 0000 | 0 | 1 | 0 |
| :--- | :--- | :--- | :--- |
| 0001 | 0 | 0 | 1 |
| 0002 | 1 | 0 | 0 |

0002
1
1
0
]
Q*R*P' = matrix $->$ X $(4,3)$
[row

| 0000 | 1 | 2 | 3 |
| ---: | ---: | ---: | ---: |
| 0001 | 4 | 5 | 6 |
| 0002 | 7 | 8 | 9 |
| 0003 | 10 | 11 | 12 |

]
Input $=$ matrix(cmplx) -> Ac $(4,3)$
[row

| 0000 | 19 | $+19 j$ | -1 |
| :--- | ---: | ---: | ---: |
| 0001 | -2 | $-2 j$ |  |
| 0002 | -1 | $-1 j$ |  |
| 0003 | 1 | $+1 j$ |  |

]
Rank of $A C=3.000000$
Orthogonal output matrix(cmplx) -> Qc(4,4)
[row

| 0000 | -0.033558 | -0.033558j | 0.032074 | +0.032074j | 0.70405 | +0.70405j | $\bigcirc$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0001 | 0.033558 | +0.033558j | -0.70182 | -0.70182j | 0.028674 | +0.028674j | 0 |
| 0002 | 0.033558 | +0.033558j | 0.073675 | +0.073675j | -0.048058 | -0.048058j | 0 |
| 0003 | 0.70471 | +0.70471j | 0.031439 | +0.031439 ${ }^{\text {j }}$ | 0.034449 | +0.034449j | 0 |
| ] |  |  |  |  |  |  |  |
| Upper Triangular output matrix(cmplx) -> Rc(4,3)[row |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| 0000 | 29.799 | +0j | -2.6846 | +0j | -0.067116 | +0j |  |
| 0001 | 0 | +0j | 28.369 | +0j | 3.9416 | +0j |  |
| 0002 | 0 | +0j | 0 | +0j | 26.804 | +0j |  |
| 0003 | 0 | +0j | 0 | +0j | 0 | +0j |  |
| ] |  |  |  |  |  |  |  |
| Permutation matrix -> P(3,3) |  |  |  |  |  |  |  |
| [row |  |  |  |  |  |  |  |
| 0000 | 0 | 0 | 1 |  |  |  |  |
| 0001 | 1 | 0 | 0 |  |  |  |  |
| 0002 | 0 | 1 | 0 |  |  |  |  |
| ] |  |  |  |  |  |  |  |
| Qc*Rc*P' $=$ matrix(cmplx) -> Xc (4,3) |  |  |  |  |  |  |  |
| [row |  |  |  |  |  |  |  |
| 0000 | 19 | +19j | -1 | -1j | 1 | +1j |  |
| 0001 | -2 | -2j | 1 | +1j | -20 | -20j |  |
| 0002 | -1 | -1j | 1 | +1j | 2 | +2j |  |
| 0003 | 1 | +1j | 21 | +21j | -1 | -1j |  |
| ] |  |  | ****** | , | ********** |  |  |

## Cholesky Factorization

The Cholesky factorization for a symmetric positive definite matrix $A$, is given as $A=B B^{T}$, where $B$ is a unique lower triangular matrix with positive diagonals entries. A matrix is positive definite if $x^{T} A x>0$ for all nonzero x. * An example follows.

```
#include "pascal.txt"
matrix A, B, X;
A = pascal(9, 0) printf("Input Matrix = "); Print(A);
printf("Rank of A = %d\n", rank(A));
printf("det(A) = %f\n", det(A));
B = chol(A); Print(B);
X = B*B'; printf("B*B' = "); Print(X);
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline \multicolumn{10}{|l|}{Input
[row} \\
\hline 0000 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline 0001 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline 0002 & 1 & 3 & 6 & 10 & 15 & 21 & 28 & 36 & 45 \\
\hline 0003 & 1 & 4 & 10 & 20 & 35 & 56 & 84 & 120 & 165 \\
\hline 0004 & 1 & 5 & 15 & 35 & 70 & 126 & 210 & 330 & 495 \\
\hline 0005 & 1 & 6 & 21 & 56 & 126 & 252 & 462 & 792 & 1287 \\
\hline 0006 & 1 & 7 & 28 & 84 & 210 & 462 & 924 & 1716 & 3003 \\
\hline 0007 & 1 & 8 & 36 & 120 & 330 & 792 & 1716 & 3432 & 6435 \\
\hline 0008 & 1 & 9 & 45 & 165 & 495 & 1287 & 3003 & 6435 & 12870 \\
\hline
\end{tabular}
]
Rank of A = 9
det(A) = 1.000000
matrix -> B(9,9)
[row
\begin{tabular}{lrlrrrrrrr}
0 \\
0000 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0001 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0002 & 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0003 & 1 & 3 & 3 & 1 & 0 & 0 & 0 & 0 & 0 \\
0004 & 1 & 4 & 6 & 4 & 1 & 0 & 0 & 0 & 0 \\
0005 & 1 & 5 & 10 & 10 & 5 & 1 & 0 & 0 & 0 \\
0006 & 1 & 6 & 15 & 20 & 15 & 6 & 1 & 0 & 0 \\
0007 & 1 & 7 & 21 & 35 & 35 & 21 & 7 & 1 & 0 \\
0008 & 1 & 8 & 28 & 56 & 70 & 56 & 28 & 8 & 1
\end{tabular}
]
B*B' = matrix -> X(9,9)
[row
\begin{tabular}{lllrrrrrrr}
0000 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0001 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
0002 & 1 & 3 & 6 & 10 & 15 & 21 & 28 & 36 & 45 \\
0003 & 1 & 4 & 10 & 20 & 35 & 56 & 84 & 120 & 165 \\
0004 & 1 & 5 & 15 & 35 & 70 & 126 & 210 & 330 & 495 \\
0005 & 1 & 6 & 21 & 56 & 126 & 252 & 462 & 792 & 1287 \\
0006 & 1 & 7 & 28 & 84 & 210 & 462 & 924 & 1716 & 3003 \\
0007 & 1 & 8 & 36 & 120 & 330 & 792 & 1716 & 3432 & 6435 \\
0008 & 1 & 9 & 45 & 165 & 495 & 1287 & 3003 & 6435 & 12870
\end{tabular}
]
```


## The Complete Orthogonal Decomposition with Column Pivoting

The Complete Orthogonal Decomposition of a matrix is given by $A=Q^{*} T^{*} Z^{T} * P^{T}$, where matrix $Q$ is orthogonal, matrix $T$ is upper triangular, and $P$ is a permutation matrix. ${ }^{* *}$ An example follows.

[^36]```
matrix A[][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
matrix A, Q, T, Z, P, X;
matcmplx Qc, Tc, Zc, Xc;
P = cod(A, Q, T, Z); printf("Orthogonal output "); Print(Q);
printf("Upper Triangular output "); Print(T);
printf("Permutation "); Print(P);
printf("Z = "); Print(Z);
X = Q*T*Z'P';
printf("Q*T*Z'P' = "); Print(X)
printf("Input = "); Print(Ac);
P = cod(Ac, Qc, Tc, Zc);
printf("Orthogonal output "); Print(Qc);
printf("Upper Triangular output "); Print(Tc);
printf("Permutation "); Print(P);
Print(Zc);
Xc = Qc*Tc*Zc'*P';
printf("Q*T*Z'*P' = "); Print(Xc);
******************************************************************
Input = matrix -> A(3,4)
[row
lllll
0002 9 10 11 
]
Orthogonal output matrix -> Q(3,3)
[row
0000 -0.26726 0.87287 0
0001 -0.53452 0.21822 0
0002 -0.80178 -0.43644 0
]
Upper Triangular output matrix -> T(3,3)
[row
0000 15.843 16.116 11.546
0001 0 2.4057 0.46084
0002 0 0 0
]
Permutation matrix -> P(4,4)
[row
\begin{tabular}{lllll}
0000 & 0 & 1 & 0 & 0 \\
0001 & 0 & 0 & 0 & 1 \\
0002 & 0 & 0 & 1 & 0 \\
0003 & 1 & 0 & 0 & 0
\end{tabular}
]
Z = matrix -> Z(4,3)
[row
lrorn <r0.94471 
0002 9 10 10
]
Input = matrix(cmplx) -> Ac(3,4)
[row
\begin{tabular}{rrrlrrrrr}
0000 & 1 & \(+1 j\) & 2 & \(+2 j\) & 3 & \(+3 j\) & 4 & \(+4 j\) \\
0001 & 5 & \(+5 j\) & 6 & \(+6 j\) & 7 & \(+7 j\) & 8 & \(+8 j\) \\
0002 & 9 & \(+9 j\) & 10 & \(+10 j\) & 11 & \(+11 j\) & 12 & \(+12 j\)
\end{tabular}
0002 9 +9j 10 +10j 11 +11j 12 +12j
```



## A Determinate Example

For a square matrix $A$, then its determinate is given by $\operatorname{det}(A)=a$. An example follows.

```
matrix A[][6] = { -5, 6, 1, -1, 0, 2,
    1, 3,-1, -1, 3, 1,
    4, 2,-2, 1, 1, 0,
    0, 0, 3, 1,-2, 4,
    1, 0, 0, -2, 1, 7,
    0, 0, 1, 1,-1, 7 };
float Adet;
printf("Input = "); Print(A);
Adet = det(A); printf("Determinant of A = %.5f\n", Adet );
Input = matrix -> B(6,6)
[row
\begin{tabular}{rrrrrrr}
0000 & -5 & 6 & 1 & -1 & 0 & 2 \\
0001 & 1 & 3 & -1 & -1 & 3 & 1 \\
0002 & 4 & 2 & -2 & 1 & 1 & 0 \\
0003 & 0 & 0 & 3 & 1 & -2 & 4 \\
0004 & 1 & 0 & 0 & -2 & 1 & 7 \\
0005 & 0 & 0 & 1 & 1 & -1 & 7
\end{tabular}
]
    Determinant of B = -3144.00000
```


## The Singular Value Decomposition

For a real m-by-n matrix $A$, then there exists orthogonal matrices $U$ and $V$ such that $U A V^{T}=\operatorname{diag}\left(\sigma_{1}, \sigma_{2}, \cdots, \sigma_{p}\right.$, where $\sigma_{1} \geq \sigma_{2} \geq \cdots \geq \sigma_{p} \geq 0$. The $\sigma_{i}$ are the singular values of $A$, and the $u_{i}$ and $v_{i}$ are the ith left singular vector and right ith singular vector respectively. Note that when we talk about the vectors of $U$ and $V$, we're referring to the columns' of $U$ and $V$.* An example follows.

```
matrix A[][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
matrix U, V, S, Achk;
S = svd(A, U, V); printf("S = svd(A, U, V); "); Print(S);
printf("Left singular vectors of A "); Print(U);
printf("Right singular vectors of A "); Print(V);
Achk = U*S*V';
printf("Achk = U*S*V'; "); Print(Achk);
*****************************************************************
Input matrix -> A(4,3)
[row
0000 1 2 3
llll
0003 10 11 12
]
S = svd(A, U, V);
matrix -> S(4,3)
[row
rrrr
0003 0 0 0
Left singular vectors of A matrix -> U(4,4)
[row
0000 -0.14088 -0.82471 0.52548 -0.15452
0001 -0.34395 -0.42626 -0.58546 0.59769
00012 -0.54702 
]
Right singular vectors of A matrix -> V (3,3)
[row
0000 -0.50453 0.76078 0.40825
0001 -0.57452 0.057141 -0.8165
0002 -0.6445 -0.64649 0.40825
]
Achk = U*S*V'; matrix -> Achk(4,3)
[row
llll
0002 7 7 8
0003 10 11 12
]
```


## The Hessenburg form of a Matrix

The Hessenberg decomposition of a square matrix $A$ is given by $A=B^{*} Q_{0} H Q_{o}{ }^{T} *_{i n v}(B)$, where matrix $Q_{0}$ is orthogonal, and matrix $B$ is a balancing matrix. Without the balancing matrix, $H=Q_{o}{ }^{T} A Q_{o . *}$." An example follows.

[^37]```
#include "magic.txt"
matrix A, H, Q, B, Chk;
A = magic(5); printf("Input = "); Print(A);
H = hess(A, Q, B); printf("The Hessenburg form of A "); Print(H);
printf("Orthogonal output "); Print(Q);
printf("Balancing "); Print(B);
Chk = B*Q*H*Q'*inv(B);
printf("Chk = B*Q*H*Q'*inv(B); "); Print(Chk);
************************************************************************
Input = matrix -> A(5,5)
[row
\begin{tabular}{rrrrrr}
0000 & 17 & 24 & 1 & 8 & 15 \\
0001 & 23 & 5 & 7 & 14 & 16 \\
0002 & 4 & 6 & 13 & 20 & 22 \\
0003 & 10 & 12 & 19 & 21 & 3 \\
0004 & 11 & 18 & 25 & 2 & 9
\end{tabular}
]
The Hessenburg form of A matrix -> H(5,5)
[row
\begin{tabular}{rrrrrr}
0000 & 17 & -28.941 & 1.847 & -4.4603 & 2.2572 \\
0001 & -27.677 & 33.94 & 26.187 & -2.228 & 1.2675 \\
0002 & 0 & 25.096 & 20.687 & -6.6055 & -0.1973 \\
0003 & 0 & 0 & -5.963 & -16.816 & -12.445 \\
0004 & 0 & 0 & 0 & -9.0122 & 10.189
\end{tabular}
0004
]
Orthogonal output matrix -> Q(5,5)
[row
\begin{tabular}{rrrrrr}
0000 & 1 & 0 & 0 & 0 & 0 \\
0001 & 0 & -0.83102 & 0.46304 & 0.27429 & 0.14057 \\
0002 & 0 & -0.14453 & -0.71444 & 0.59186 & 0.34408 \\
0003 & 0 & -0.36131 & -0.36799 & -0.034767 & -0.85605 \\
0004 & 0 & -0.39745 & -0.37384 & -0.75714 & 0.3592
\end{tabular}
]
Balancing matrix -> B(5,5)
[row
\begin{tabular}{llllll}
0000 & 1 & 0 & 0 & 0 & 0 \\
0001 & 0 & 1 & 0 & 0 & 0 \\
0002 & 0 & 0 & 1 & 0 & 0 \\
0003 & 0 & 0 & 0 & 1 & 0 \\
0004 & 0 & 0 & 0 & 0 & 1
\end{tabular}
0004
]
Chk = B*Q*H*Q'*inv(B); matrix -> Chk(5,5)
[row
\begin{tabular}{lrrrrr}
0000 & 17 & 24 & 1 & 8 & 15 \\
0001 & 23 & 5 & 7 & 14 & 16 \\
0002 & 4 & 6 & 13 & 20 & 22 \\
0003 & 10 & 12 & 19 & 21 & 3 \\
0004 & 11 & 18 & 25 & 2 & 9
\end{tabular}
```


## Schur Decomposition

The Schur decomposition of a matrix is given by $A=B^{*} Z^{*} S^{*} Z^{T} * \operatorname{inv}(B)$, where $Z$ is orthogonal, and matrix $B$ is a balancing matrix. An example follows.

```
#include "matrices.h"
matrix A, S, Z, B, Achk;
```

| A = gallery(5); printf("Input "); Print(A); |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: |
| S = schur(A, Z, B) ; printf("Schur matrix "); Print(S); |  |  |  |  |  |
| printf("Orthogonal (an output) "); Print(Z); |  |  |  |  |  |
| printf("Balancing transform (an output) "); Print(B); |  |  |  |  |  |
| Achk $=\mathrm{B}^{*} \mathrm{Z}^{*} S^{*} Z^{\prime} * \operatorname{inv}(\mathrm{~B}) ;$; |  |  |  |  |  |
| printf("Achk $\left.=\mathrm{B}^{*} \mathrm{Z}^{*} \mathrm{~S}^{*} \mathrm{Z}{ }^{\prime *} \mathrm{inv}(\mathrm{B})=\mathrm{"}\right)$; Print(Achk);$* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *$ |  |  |  |  |  |
|  |  |  |  |  |  |
| Input matrix -> $\mathrm{A}(5,5)$ |  |  |  |  |  |
| [row |  |  |  |  |  |
| 0000 | -9 | 11 | -21 | 63 | -252 |
| 0001 | 70 | -69 | 141 | -421 | 1684 |
| 0002 | -575 | 575 | -1149 | 3451 | -13801 |
| 0003 | 3891 | -3891 | 7782 | -23345 | 93365 |
| 0004 | 1024 | -1024 | 2048 | -6144 | 24572 |
| ] |  |  |  |  |  |
| Schur matrix matrix -> S $(5,5)$ |  |  |  |  |  |
| [row |  |  |  |  |  |
| 0000 | -0.026601 | 119.06 | -299.24 | -627.97 | 49332 |
| 0001 | $-3.2435 \mathrm{e}-06$ | -0.026601 | 5.7129 | 9.9908 | -826.04 |
| 0002 | 0 | 0 | 0.010461 | -3.7404 | 224.12 |
| 0003 | 0 | 0 | 0.00025449 | 0.010461 | 51.29 |
| 0004 | 0 | 0 | 0 | 0 | 0.03228 |
| ] |  |  |  |  |  |
| Orthogonal (an output) matrix -> $\mathrm{Z}(5,5)$ |  |  |  |  |  |
| [row |  |  |  |  |  |
| 0000 | -0.000147 | 0.65119 | -0.70988 | -0.2683 | 0.0053321 |
| 0001 | 0.058037 | 0.72032 | 0.4688 | 0.50747 | -0.021684 |
| 0002 | -0.19683 | 0.23844 | 0.51814 | -0.7899 | 0.10951 |
| 0003 | 0.67416 | 0.013955 | 0.085078 | -0.20559 | -0.70413 |
| 0004 | 0.7095 | -0.0059003 | 0.024408 | -0.065352 | 0.70122 |
| ] |  |  |  |  |  |
| Balancing transform (an output) matrix -> B(5,5) |  |  |  |  |  |
| [row |  |  |  |  |  |
| 0000 | 0.5 | 0 | 0 | 0 | 0 |
| 0001 | 0 | 1 | 0 | 0 | 0 |
| 0002 | 0 | 0 | 2 | 0 | 0 |
| 0003 | 0 | 0 | 0 | 4 | 0 |
| 0004 | 0 | 0 | 0 | 0 | 1 |
| ] |  |  |  |  |  |
| Achk $=$ B*Z*S*Z'*inv(B) = matrix -> Achk $(5,5)$ |  |  |  |  |  |
| [row |  |  |  |  |  |
| 0000 | -9 | 11 | -21 | 63 | -252 |
| 0001 | 70 | -69 | 141 | -421 | 1684 |
| 0002 | -575 | 575 | -1149 | 3451 | -13801 |
| 0003 | 3891 | -3891 | 7782 | -23345 | 93365 |
| 0004 | 1024 | -1024 | 2048 | -6144 | 24572 |
| ] |  |  |  |  |  |

## Matrix Inverse Examples

The inverse of a square matrix $A$ is given by $\operatorname{inv}(A)=$, and $A * \operatorname{inv}(A)=I$. An example follows.

```
matcmplx A[][2] = {1 + 5j, 2 +6j, 3 + 7j, 4 + 8j };
matrix B[][5] = { 17, 23, 4, 10, 11,
    24, 5, 6, 12, 18,
        1, 7, 13, 19, 25,
        8, 14, 20, 21, 2,
    15, 16, 22, 3, 9 };
matrix Binv, Bchk;
matcmplx Ainv, Achk;
printf("Input = "); Print(B);
Binv = inv(B) printf("inv(B) = "); Print(Binv);
```



## A LU Factorization Example

The LU factorization of an m-by-n matrix is defined by $A=L^{*} U$, where $L$ is a lower triangular matrix with 1's on the diagonal, and $U$ is an upper triangular matrix with zero's below the main diagonal.* The factorization works for $\mathrm{m}>\mathrm{n}, \mathrm{m}$ equals n , and $\mathrm{m}<\mathrm{n}$. An example follows.

```
matrix A[][3] = { 8, 2, 9, 4, 9, 4, 6, 7, 9 };
matrix X, L, U, Chk;
Print(A);
X = lu(A);
U = triU(X); Print(U);
L = triL(X); Print(L);
Chk = L*U; Print(Chk);
```

[^38]

## Solving Linear Equations

We can solve linear equations for $x$. given $M x=y$ or $M Y=B$, for n independent equations in n unknowns, by calling the following functions in Slide-Rule. Their prototypes are as follows.

```
vector linequ(matrix M, vector y);
veccmplx linequ(matcmplx Mc, veccmplx y);
float linequX(matrix M, matrix B, matrix Y);
```

where matrix $\mathbf{M}$ must be a square matrix of full rank , and $\mathbf{y}$ is a vector whose length must equal the rank of M
linequ uses driver's dgesv and zgesv of LAPACK version 3.1.1.
linequX users expert driver dgesvx of LAPACK version 3.1.1, and solves for the multiple columns of matrix $\mathbf{B}$. The solution is found in the columns of matrix $\mathbf{Y}$ (an output).
The reciprocal condition Rcond number (a return value) is also computed on the input matrix $\mathbf{M}$, after it has been equilibrated (if necessary);

An example follows.


```
    2, \(1, \quad 21, \quad-1\),
    -1, 2, \(-2, \quad 22\);
vector y, Chk;
float detA;
vector b[] = \{ 1, 2, 3, 4 \};
printf("Input "); Print(A);
printf("Input "); Print(b);
\(\operatorname{det} A=\operatorname{det}(A) ; \operatorname{printf("Matrix~determinant~}=\% f \backslash n ", \operatorname{det} A)\);
y = linequ(A, b); printf("Solution "); Print(y);
mChk = A*y; printf("Check = A*y "); Print(mChk);
```

```
*************************************************************
Input matrix -> A(4,4)
[row
0000 19 -1 1 -2
0001 1 -20 -1 
0rror
0003 -1 2 - 2 -2 
]
Input vector -> b(4)
[index
0000 1 2 0 % 3
]
Matrix determinant = -173189.000000
Solution vector -> y(4)
[index
0000 0.061494 -0.094146 0.15134 0.20693
]
Check = A*y matrix -> mChk(4,1)
[row
0000 1
0001 2
0002 3
0003 4
]
```


## A Least Squares Fit

We can solve least square fit problems of the type. $M x=y$ or $M Y=B$, by calling the following functions in Slide-Rule. Their prototypes are as follows.

```
vector lsq(matrix M, vector y);
veccmplx lsq(matcmplx M, veccmplx y);
float lsqX(matrix M, matrix B, matrix X);
```

Routine Isq uses LAPACK driver routines dgels and zgels. The matrix M must have full rank and the length of vector $\mathbf{y}$ must equal the rank of $\mathbf{M}$. The return value is the solution vector.

Routine IsqX uses LAPACK driver routine dgelsd to solve a minimum norm-2 least squares solution to a rank deficient matrix $\mathbf{M}$. Multiple solutions of the columns of matrix $\mathbf{B}$ are stored in the columns of the solution matrix $\mathbf{X}$. The row length of the matrix $\mathbf{B}$ must equal the row length of matrix $\mathbf{M}$. This function returns the rank of matrix $\mathbf{M}$.

An example follows.

```
matrix A[][4] = { 51, 25.5, 17.17, 13.01,
    25.5, 17.17, 13.01, 10.51,
    17.17, 13.01, 10.51, 8.84,
    13.01, 10.51, 8.84, 7.65 };
vector b[] = {-98.96, -47.3, -23.3, -17.53 };
vector x, chk;
int Rank;
printf("Input "); Print(A);
printf("Input "); Print(b);
Rank = rank(A); printf("Rank of A = %d\n", Rank);
x = lsq(A, b) printf("lsq(A, b) = "); Print(x);
chk = matToV(A*x); printf("A*x = "); Print(chk);
```

| Input matrix -> A (4,4) |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: |
| [row |  |  |  |  |
| 0000 | 51 | 25.5 | 17.17 | 13.01 |
| 0001 | 25.5 | 17.17 | 13.01 | 10.51 |
| 0002 | 17.17 | 13.01 | 10.51 | 8.84 |
| 0003 | 13.01 | 10.51 | 8.84 | 7.65 |
| ] |  |  |  |  |
| Input vector -> b(4) |  |  |  |  |
| [index |  |  |  |  |
| 0000 | -98.96 | -47.3 | -23.3 | -17.53 |
| ] |  |  |  |  |
| Rank of $A=4$ |  |  |  |  |
| $\operatorname{lsq}(\mathrm{A}, \mathrm{b})=$ vector $->\mathrm{x}(4)$ |  |  |  |  |
| [index |  |  |  |  |
| 0000 | 14.042 | -173.09 | 395.75 | -245.68 |
| ] |  |  |  |  |
| A* $\mathrm{A}=$ vector $->\operatorname{chk}(4)$ |  |  |  |  |
| [index |  |  |  |  |
| 0000 | -98.96 | -47.3 | -23.3 | -17.53 |
| ] |  |  |  |  |

## A Matrix Pseudoinverse Example

The equation for the SVD decomposition is given by $A=U^{*} S^{*} V^{T}$, and the Moore-Penrose pseudo-inverse is given by $A=U^{*} S^{+} V^{T}$. The matrix $S^{+}$is generated by taking the transpose of $S$, then replacing non-zero entries by their reciprocals. For diagonal_entries below the rank, the entries are set to zero. The pseudoinverse of $A$ has properties of the inv function, and is most useful in rank deficient systems of linear equations. For $A y=b$, where matrix $A$ is rank deficient, we can get a solution_by writing $y=p \operatorname{inv}(A)^{*} b$. Note! To verify that $y$ is an exact solution, check with the following statement (variable $b$ declared as a vector), $b=($ vector $)\left(A^{*} y\right)$. If $b$ is equal to the original $b$, then you have an exact solution, otherwise variable $y$ returned from $\operatorname{pinv}(A)^{*} b$ is a least squares solution. Note that the norm of this solution is less than the norm of other solutions, such as that computed by the QR method. In the above statement, variable $b$ can either be a vector or a single column of a matrix. One can also use function lsqX to solve for rank deficient systems. Refer to this function under the help menu_under lsq - Least squares fit. An example follows.

```
#include "matrices.h"
matrix A, y, Chk;
vector b;
A = pascal(8,0); reduce(A, 7, -1); reduce(A, 6, -1); Print(A);
b = zerosV(6); b += 8; Print(b);
printf("Rank of A = %d\n", rank(A));
y = pinv(A)*b; printf("y = pinv(A)*b; "); Print(y);
Chk = A*y; printf("Chk = A*y = "); Print(Chk);
    matrix -> A(6,8)
    [row
    0000 
    00001 1
    0002 1 1 3 3
    rrrrrrrrr
    0005 1 1 6 % 21 
]
    vector -> b(6)
    [index
    0000 
]
    Rank of A = 6
    y = pinv(A)*b; matrix -> y(8,1)
```

```
[row
0000 7.9674
                                0.16783
                -0.32168
                    0.2331
            0.06993
                -0.22378
            0.1352
        -0.027972
]
Chk = A*y = matrix -> Chk (6,1)
[row
0000 8
0002 8
0003 8
0004
0005
    8
    ]
```


## A Matrix Norm-2 Least Squares Example

The following example is a least squares fit for an m-by-n matrix $M x=B$, where the number of rows of $M$ is less then the number of columns, and $B$ is a matrix.

| ```#include "magic.txt" matrix A, Y, B, Chk; float Rank;``` |  |  |  |  |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| printf("A Matrix lsqX Example\n\n"); |  |  |  |  |  |  |  |  |  |  |  |  |
| A $=$ magic (12);reduce $(A, 11, ~-1) ; ~ r e d u c e(A, ~ 10, ~-1) ; ~ P r i n t ~$ |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| $B=\{ \} ; B=\{A[: 0] ; A[: 3] ; A[: 6]\} ; B=B ' ; \operatorname{Print}(B) ;$ printf("Rank of $A=\% d \backslash n ", ~ r a n k(A)) ;$ |  |  |  |  |  |  |  |  |  |  |  |  |
| Rank $=1 \mathrm{lsqX}(\mathrm{A}, \mathrm{B}, \mathrm{Y})$; Print(Y) ; |  |  |  |  |  |  |  |  |  |  |  |  |
| Chk = A*Y; printf("Chk = A*Y "); Print(Chk); |  |  |  |  |  |  |  |  |  |  |  |  |
| matrix -> A(10,12) |  |  |  |  |  |  |  |  |  |  |  |  |
| [row |  |  |  |  |  |  |  |  |  |  |  |  |
| 0000 | 144 | 2 | 3 | 141 | 140 | 6 | 7 | 137 | 136 | 10 | 11 | 133 |
| 0001 | 13 | 131 | 130 | 16 | 17 | 127 | 126 | 20 | 21 | 123 | 122 | 24 |
| 0002 | 25 | 119 | 118 | 28 | 29 | 115 | 114 | 32 | 33 | 111 | 110 | 36 |
| 0003 | 108 | 38 | 39 | 105 | 104 | 42 | 43 | 101 | 100 | 46 | 47 | 97 |
| 0004 | 96 | 50 | 51 | 93 | 92 | 54 | 55 | 89 | 88 | 58 | 59 | 85 |
| 0005 | 61 | 83 | 82 | 64 | 65 | 79 | 78 | 68 | 69 | 75 | 74 | 72 |
| 0006 | 73 | 71 | 70 | 76 | 77 | 67 | 66 | 80 | 81 | 63 | 62 | 84 |
| 0007 | 60 | 86 | 87 | 57 | 56 | 90 | 91 | 53 | 52 | 94 | 95 | 49 |
| 0008 | 48 | 98 | 99 | 45 | 44 | 102 | 103 | 41 | 40 | 106 | 107 | 37 |
| 0009 | 109 | 35 | 34 | 112 | 113 | 31 | 30 | 116 | 117 | 27 | 26 | 120 |
| ] |  |  |  |  |  |  |  |  |  |  |  |  |
| matrix -> B $(10,3)$ |  |  |  |  |  |  |  |  |  |  |  |  |
| [row |  |  |  |  |  |  |  |  |  |  |  |  |
| 0000 |  | 144 |  | 141 |  | 7 |  |  |  |  |  |  |
| 0001 |  | 13 |  | 16 |  | 126 |  |  |  |  |  |  |
| 0002 |  | 25 |  | 28 |  | 114 |  |  |  |  |  |  |
| 0003 |  | 108 |  | 105 |  | 43 |  |  |  |  |  |  |
| 0004 |  | 96 |  | 93 |  | 55 |  |  |  |  |  |  |
| 0005 |  | 61 |  | 64 |  | 78 |  |  |  |  |  |  |
| 0006 |  | 73 |  | 76 |  | 66 |  |  |  |  |  |  |
| 0007 |  | 60 |  | 57 |  | 91 |  |  |  |  |  |  |
| 0008 |  | 48 |  | 45 |  | 103 |  |  |  |  |  |  |
| 0009 |  | 109 |  | 112 |  | 30 |  |  |  |  |  |  |
| ] |  |  |  |  |  |  |  |  |  |  |  |  |


| $\begin{aligned} & \text { Rank of } A=3 \\ & \text { matrix }->Y(12,3) \end{aligned}$ |  |  |  |
| :---: | :---: | :---: | :---: |
| [row |  |  |  |
| 0000 | 0.37821 | 0.26282 | 0.019231 |
| 0001 | -0.17308 | -0.078671 | 0.15093 |
| 0002 | -0.13462 | -0.061189 | 0.15443 |
| 0003 | 0.26282 | 0.21037 | 0.0087413 |
| 0004 | 0.22436 | 0.19289 | 0.0052448 |
| 0005 | -0.019231 | -0.0087413 | 0.16492 |
| 0006 | 0.019231 | 0.0087413 | 0.16841 |
| 0007 | 0.10897 | 0.14044 | -0.0052448 |
| 0008 | 0.070513 | 0.12296 | -0.0087413 |
| 0009 | 0.13462 | 0.061189 | 0.1789 |
| 0010 | 0.17308 | 0.078671 | 0.1824 |
| 0011 | -0.044872 | 0.070513 | -0.019231 |
| ] ${ }^{\text {a }}$ |  |  |  |
| Chk = A*Y matrix -> Chk (10,3) |  |  |  |
| [row |  |  |  |
| 0000 | 144 | 141 | 7 |
| 0001 | 13 | 16 | 126 |
| 0002 | 25 | 28 | 114 |
| 0003 | 108 | 105 | 43 |
| 0004 | 96 | 93 | 55 |
| 0005 | 61 | 64 | 78 |
| 0006 | 73 | 76 | 66 |
| 0007 | 60 | 57 | 91 |
| 0008 | 48 | 45 | 103 |
| 0009 | 109 | 112 | 30 |
| 100 112 |  |  |  |

## An Over-determined Set of Equations

An example of Solving a System of Over determined Equations. We first solve using the QR decomposition function $\boldsymbol{q r}$, then we solve by calling function linequOD. For the $q r$ solution we have,

```
A*P = Q*R definition of qr decompostion
A*P*P' = A = Q****'
A*X = Q*R*P'*X = b
Q'*A*x = Q'*Q*R*P'*x = R*P'*x = Q'*b = y
```

Therefore,
x = linequ(R*P', y);


```
matrix A[][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
vector b[] = { 1, 3, 5, 7 };
vector x, y, z;
matrix Q, R, P, yM, zChk;
printf("Input "); Print(A);
printf("input "); Print(b);
P = qr(A, Q, R); // do qr decomposition
printf("P = qr(A, Q, R); "); Print(P);
printf("Orthogonal Matrix output "); Print(Q);
printf("Upper triangular matrix output "); Print(R);
yM = Q'*b; printf("yM = Q'*b; "); Print(yM);
reduce(yM, 3, -1); Print(yM);
reduce(R, 3, -1); printf("reduce(R, 3, -1); "); Print(R);
y = (vector)yM;
```

```
x = linequ(R*P', y); printf("x = linequ(R*P', y); "); Print(x);
zChk = A*x; printf("zChk = A*x; "); Print(zChk);
z = linequOD(A, b); printf("z= linequOD(A, b); "); Print(z);
Input matrix -> A(4,3)
[row
\begin{tabular}{rrrr}
0000 & 1 & 2 & 3 \\
0001 & 4 & 5 & 6 \\
0002 & 7 & 8 & 9 \\
0003 & 10 & 11 & 12
\end{tabular}
]
input vector -> b(4)
[index
0000 1 3 5 5
]
P = qr(A, Q, R); matrix -> P(3,3)
[row
llll
]
Orthogonal Matrix output matrix -> Q(4,4)
[row
0000 -0.18257 -0.8165 0 0
0001 -0.36515 -0.40825 0. 0 0
rrrrr
]
Upper triangular matrix output matrix -> R(4,3)
[row
0000 -16.432 -12.78 -14.606
0001 0 1.633 0.8165
0002 0003 0
]
yM = Q'*b; matrix -> yM(4,1)
[row
0000 -9.1287
0001 0.8165
0002 0
0003 0
]
reduce(R, 3, -1); matrix -> R(3,3)
[row
0000 -16.432 -12.78 -14.606
0001 0002 0
0002
]
x = linequ(R*P', y); vector -> x(3)
[index
0000 0.5 0 0.16667
]
zChk = A*x; matrix -> zChk(4,1)
[row
0000 1
0001 3
0002 5
0003 7
]
z = linequOD(A, b); vector -> z(3)
[index
0000 0.5 0 0.16667
]
```


## Eigenvalues/Eigenvectors Examples

We can solve eigenvalue and eigenvector problems calling the following functions in Slide-Rule.
Their prototypes are as follows.

```
veccmplx eig(matcmplx MatV, matrix Mat1);
veccmplx eig(matcmplx MatV, matcmplx Mat2);
veccmplx eigX(matcmplx MatV, matrix Mat1);
veccmplx eigX(matcmplx MatV, matcmplx Mat2);
vector eigS(matrix MatV, matrix Mat);
veccmplx eigS(matcmplx MatV, matcmplx Mat2);
veccmplx eigG(matcmplx MatV, matrix A, matrix B);
For functions eig and eigX, the non-symmetric eigenvalue problem. (NEP).
For function eigS , the symmetric eigenvalue problem. (SEP).
    MatV = complex matrix of computed eigenvectors (an output)
    Mat1 = real square matrix input.
    Mat2 = complex square matrix input.
    Returns the complex vector of computed eigenvalues.
Returns the real or complex vector of computed eigenvalues.
Note!! The eigenvectors are stored as complex column vectors.
```

For the first example, we compute the eigenvalues/eigenvectors of a 7 x 7 matrix. Next the eigenvalues/eigenvectors of a $21 \times 21$ matrix, where the matrix is the classical Wilkinson test matrix, whereby two of the eigenvalues are accurate to 15 decimal digits

```
For the 7 x 7 Wilkinson matrix using the (NEP) driver eig
Eigenvalues =
3.732050807568876
3.761557181831890
2.363328237932682
2.000000000000000
0.999999999999999
0.267949192431123
-1.124885419764575
Eigenvectors (on column basis) matrix -> wVectorR(7,7)
[row
\begin{tabular}{lrrrrrrr}
{\([0000\)} & -0.55768 & 0.54025 & -0.3799 & 0.40825 & -0.25 & 0.14943 & 0.03614 \\
0001 & -0.40825 & 0.41143 & 0.24187 & -0.40825 & 0.5 & -0.40825 & -0.14907 \\
0002 & -0.14943 & 0.18451 & 0.46778 & -0.40825 & -0.25 & 0.55768 & 0.4297 \\
0003 & 0.098103 & 0.39587 & 0 & -0.5 & 0 & -0.76398 \\
0004 & 0.14943 & 0.18451 & 0.46778 & 0.40825 & -0.25 & -0.55768 & 0.4297 \\
0005 & 0.40825 & 0.41143 & 0.24187 & 0.40825 & 0.5 & 0.40825 & -0.14907 \\
0006 & 0.55768 & 0.54025 & -0.3799 & -0.40825 & -0.25 & -0.14943 & 0.03614 \\
] \\
For the \(21 \times 21\) Wilkinson matrix using the (NEP) driver eig
\end{tabular}
Eigenvalues =
10.746194182903407
10.746194182903331
9.210678647304920
9.210678647361334
8.038941115814280
8.038941122829032
7.003951798616381
7.003952209528682
6.000217522257097
6.000234031584167
5.000244425001914
```

```
4.999782477742904
4.004354023440857
3.996048201383625
3.043099292578823
2.961058884185725
2.130209219362503
1.789321352695079
0.947534367529293
0.253805817096678
-1.125441522119985
For the 21 x 21 Wilkinson matrix using (SEP) symmetric driver (eigS)
Eigenvalues =
10.746194182903395
10.746194182903324
9.210678647361334
9.210678647304919
8.038941122829021
8.038941115814273
7.003952209528676
7.003951798616376
6.000234031584168
6.000217522257099
5.000244425001912
4.999782477742901
4.004354023440857
3.996048201383624
3.043099292578823
2.961058884185726
2.130209219362503
1.789321352695080
0.947534367529294
0.253805817096678
-1.125441522119987
```


## A Reduced Row Echelon Example

This function returns a reduced row echelon form of a matrix using Gauss Jordan elimination. It uses a default tolerance of $2.2 \mathrm{e}-14$ for negligible column elements. The original input matrix is left intact. The prototype of this function is as follows.

```
matrix rref(matrix mat);
```

An example follows

```
matrix A[][3] = { 1, 1, -2, 3, 2, 4, 4, 3, 3 };
matrix B[][1] = { 1, -4, -4 };
vector y;
matrix Rr, Aug, Chk;
Print(A); Print(B);
Aug = { A, B }; printf("Augmentated matrix = "); Print(Aug);
Rr = rref(Aug); printf("Reduced Row Echelon form = "); Print(Rr);
y = Rr[:3]; printf("Solution = "); Print(y);
Chk = A*y; printf("A*y = "); Print(Chk);
*************************************************************
matrix -> A(3,3)
[row
```

```
\begin{tabular}{rrrr}
0000 & 1 & 1 & -2 \\
0001 & 3 & 2 & 4 \\
0002 & 4 & 3 & 3
\end{tabular}
matrix -> B(3,1)
[row
0000 1
0001 -4
0002 -4
]
Augmentated matrix = matrix -> Aug(3,4)
[row
```



```
]
Reduced Row Echelon form = matrix -> Rr(3,4)
[row
0000 0001 0
lllll
]
Solution = vector -> y(3)
[index
0000 2 -3 -1
]
A*y = matrix -> Chk(3,1)
[row
0000 1
0001 -4
0002 -4
]
```


## Matrix/Vector Norms

Various norms of matrices and vectors can be computed in Slide-Rule. The function prototypes are listed as follows.

```
float normM(matrix A, char type);
float normM(matcmplx A, char type);
float normV(vector B, char type);
float normV(veccmplx B, char type);
float norm(matrix A);
float norm(matcmplx A);
```

for a matrix and function normM

$$
\mathrm{A}=\text { matrix }
$$

'1', largest column sum max (1-norm)
'I' largest row sum (infinity norm)
'F' Frobenius-norm
' M ' largest absolute value
for a vector and function normV
$\mathrm{B}=$ vector
'1' vector sum, sum of absolute values(1-norm)
'2' Euclidean sum, sqrt(sum of values^2)) (2-norm)
'I' Largest absolute value
for function norm, the return value is the norm- 2 value.
The norm-1 of a vector is given by

$$
\|x\|_{1}=\left|x_{1}\right|+\ldots+\left|x_{n}\right|
$$

The norm- 2 of a vector is given by

$$
\|x\|_{2}=\left(\left|x_{1}\right|^{2}+\ldots+\left|x_{n}\right|^{2}\right)^{1 / 2}
$$

The Infinity norm of a vector is given by

$$
\|x\|_{\infty}=\max \quad\left|x_{i}\right|, 1 \leq i \leq n
$$

The norm-1 of a matrix is given by

$$
\|A\|_{1}=\max \sum_{i=1}^{m}\left|a_{i j}\right|, 1 \leq \mathrm{j} \leq \mathrm{n}
$$

The Infinity norm of a matrix is given by

$$
\|A\|_{\infty}=\max \sum_{j=1}^{n}\left|a_{i j}\right|, 1 \leq \mathrm{i} \leq \mathrm{m}
$$

The Frobenius norm of a matrix is given by

$$
\|A\|_{F}=\sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n}\left|a_{i j}\right|^{2}}
$$

The norm- 2 of a matrix is given by

$$
A^{T} A z=\mu^{2} z, \text { where } \mu=\|\mathrm{A}\|_{2}
$$

In particular, the norm- 2 of $A$ is the square root of the largest eigenvalue of $A^{T} A$. It can also be shown that the norm- 2 of $A$ is the largest singular value of the vector returned value from a SVD decomposition, and This is the method used for function norm. An example follows.

```
#include "matrices.h"
matrix M1 [][4] = { 1, 2, 3, 4, 5, 6, 7, 8,
                    9, 10, 11, 12, 13, 14, 15, 16 };
vector V1[] = { 1, -2, 3, -4 };
float ftemp, norm2Val;
vector eigns;
matrix Mx, U, V;
Print(M1);
ftemp = normM(M1, '1'); printf("norm-1 = %g\n", ftemp);
ftemp = normM(M1, 'I'); printf("Infinity norm = %g\n", ftemp);
ftemp = normM(M1, 'F'); printf("frobenius norm = %g\n", ftemp);
ftemp = normM(M1, 'M'); printf("max abs norm = %g\n", ftemp);
Print(V1);
ftemp = normV(V1, '1'); printf("norm-1 = %g\n", ftemp);
ftemp = normV(V1, '2'); printf("norm-2 = %g\n", ftemp);
ftemp = normV(V1, 'M'); printf("max abs norm = %g\n", ftemp);
MX = wilkinson(7); Print(Mx);
eigns = real(eig(eignVecs, MX*Mx)); Print(eigns);
norm2Val = sqrt(max(eigns));
printf("norm-2 = sqrt(max(eigns)) "); Print(norm2Val);
norm2Val = max(max(svd(Mx,U,V)));
printf("norm-2 = max(max(svd(Mx,U,V))) "); Print(norm2Val);
norm2Val = norm(Mx); printf("norm-2 = norm(Mx) "); Print(norm2Val);
matrix -> M1(4,4)
[row
```



```
0002 9 10
```

```
0003 13 14 15
]
norm-1 = 40
Infinity norm = 58
frobenius norm = 38.6782
max abs norm = 16
vector -> V1(4)
[index
[llll
]
norm-1 = 10
norm-2 = 5.47723
max abs norm = 4
matrix -> Mx(7,7)
[row
0000 
0001 
0002 1-0 1 1 1 1 1 0
0003 0
llllllll
llllllll
]
vector -> eigns(7)
```



```
]
norm-2 = sqrt(max(eigns)) norm2Val = 3.76155718183189
norm-2 = max(max(svd(Mx,U,V))) norm2Val = 3.76155718183189
norm-2 = norm(Mx) norm2Val = 3.76155718183189
```


## Toeplitz Matrices

A matrix whose entries are constant along each diagonal arise in many applications, and are called Toeplitz matrices. Toeplitz matrices are divided into two classes, namely symmetric and positive definite (or Hermitian Toeplitz), and un-symmetric. Slide-Rule has functions to compute a Toeplitz matrix from a vector(s). Their function prototypes are listed as follows.

```
matrix toeplitz(vector p);
matrix toeplitz2(vector p, vector r);
matcmplx toeplitz(veccmplx p);
matcmplx toeplitz2(veccmplx p, veccmplx r);
```

where,
$\mathrm{p}=$ vector input $\mathrm{r}=$ vector input;
returns a Toeplitz matrix
Function toeplitz(p) returns the symmetric or Hermitian Toeplitz matrix formed from vector p , where p defines the first column of the matrix Function toeplitz2( $\mathrm{p}, \mathrm{r}$ ) returns a nonsymmetric Toeplitz matrix having $p$ has its first column and $r$ as its first row. If the first elements of $p$ and $r$ differ, then $p$ wins the diagonal conflict. An example follows.

```
vector b[] = { 1, 2, 3, 4, 5 };
vector c[] = { 6, 7, 8, 9, 10 };
vector a[] = { 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
matrix B;
```

```
printf("Input "); Print(b);
B = toeplitz(b); printf("B = toeplitz(b) "); Print(B);
printf("Input "); Print(c);
B = toeplitz2(b, c); printf("B = toeplitz2(b, c) "); Print(B);
**********************************************************************
Input vector -> b(5)
[index
0000 1 2 0 3 4
B = toeplitz(b) matrix -> B(5,5)
[row
```




```
0002 3 2 2 0
0003 0004 0004 4
```



```
Input vector -> c(5)
[index
\begin{tabular}{|c|c|c|c|c|c|}
\hline 0000 & 6 & 7 & 8 & 9 & 10 \\
\hline \multicolumn{6}{|l|}{] \({ }^{\text {c }}\)} \\
\hline \multicolumn{6}{|l|}{\(\mathrm{B}=\) toeplitz2(b, c) matrix \(\rightarrow\) B \((5,5)\)} \\
\hline \multicolumn{6}{|l|}{[row} \\
\hline 0000 & 1 & 7 & 8 & 9 & 10 \\
\hline 0001 & 2 & 1 & 7 & 8 & 9 \\
\hline 0002 & 3 & 2 & 1 & 7 & 8 \\
\hline 0003 & 4 & 3 & 2 & 1 & 7 \\
\hline 0004 & 5 & 4 & 3 & 2 & 1 \\
\hline
\end{tabular}
]
```


## The Square Root of a Matrix

The square root of a symmetric positive semi-definite matrix is defined as $A=G G^{T} .{ }^{*}$ If we take the Cholesky factorization of $A$, then perform a singular value decomposition of $G$, we have

$$
A=G G^{T}=\left(U S V^{T}\right)\left(U S V^{T}\right)^{T}=U S^{2} V^{T} V U^{T}=U S^{2} U^{T}=\left(U S U^{T}\right)\left(U S U^{T}\right)=X^{2}
$$

or we can call function sqrtM in Slide-Rule. An example follows.

```
#include "pascal.txt"
matrix A, X, U, S, V, G, Chk;
A = pascal(5, 0); printf("Input Matrix = "); Print(A);
printf("Rank of A = %d\n", rank(A));
printf("det(A) = %f\n", det(A));
G = chol(A);
S = svd(G,U,V);
X = U*S*U'; printf("Square root of input matrix = "); Print(X);
Chk = X*X; printf("Check of sqrt(A) = X*X; "); Print(Chk);
X = sqrtM(A); printf("X = sqrtM(A) = "); Print(X);
*************************************************************************
Input Matrix = matrix -> A(5,5)
[row
0000 1 1 1
0001
0002
0003 1-1 4 0
```

[^39]| 0004 | 1 | 5 | 15 | 35 | 70 |
| :---: | :---: | :---: | :---: | :---: | :---: |
| ] |  |  |  |  |  |
| Rank of $\mathrm{A}=5$ |  |  |  |  |  |
| $\operatorname{det}(\mathrm{A})=1.000000$ |  |  |  |  |  |
| Square root of input matrix = matrix -> $\mathrm{X}(5,5)$ |  |  |  |  |  |
| [row |  |  |  |  |  |
| 0000 | 0.86995 | 0.43498 | 0.20827 | 0.094911 | 0.039986 |
| 0001 | 0.43498 | 0.90818 | 0.79483 | 0.52397 | 0.28232 |
| 0002 | 0.20827 | 0.79483 | 1.388 | 1.4704 | 1.1119 |
| 0003 | 0.094911 | 0.52397 | 1.4704 | 2.6824 | 3.2185 |
| 0004 | 0.039986 | 0.28232 | 1.1119 | 3.2185 | 7.637 |
| ] |  |  |  |  |  |
| Check of sqrt(A) $=$ X*X; matrix -> Chk ( 5,5 ) |  |  |  |  |  |
| [row |  |  |  |  |  |
| 0000 | 1 | 1 | 1 | 1 | 1 |
| 0001 | 1 | 2 | 3 | 4 | 5 |
| 0002 | 1 | 3 | 6 | 10 | 15 |
| 0003 | 1 | 4 | 10 | 20 | 35 |
| 0004 | 1 | 5 | 15 | 35 | 70 |
| ] |  |  |  |  |  |
| X = sqrtM $(A)=$ matrix $->\times(5,5)$ |  |  |  |  |  |
| [row |  |  |  |  |  |
| 0000 | 0.86995 | 0.43498 | 0.20827 | 0.094911 | 0.039986 |
| 0001 | 0.43498 | 0.90818 | 0.79483 | 0.52397 | 0.28232 |
| 0002 | 0.20827 | 0.79483 | 1.388 | 1.4704 | 1.1119 |
| 0003 | 0.094911 | 0.52397 | 1.4704 | 2.6824 | 3.2185 |
| 0004 | 0.039986 | 0.28232 | 1.1119 | 3.2185 | 7.637 |
| ] |  |  |  |  |  |

## A Generalized Eigenvalue Problem

The Generalized Eigenvalue Problem is given by $A x=\lambda B x$. Slide-Rule has a function to solve this problem, and its prototype is as follows,

```
veccmplx eigG(matcmplx MatV, matrix A, matrix B);
```

where,
MatV = complex matrix of computed eigenvectors (an output).
$\mathrm{A}=$ real square matrix input.
$B=$ real square matrix input.
Returns the complex vector of computed eigenvalues.
Note!! The eigenvectors are stored as complex column vectors.
An example follows.

```
matrix A[][4] = { 1, 1, 0, 0,
                        1, 0, 0, 0,
    -1, 0, 1, 0,
    0,-1, 0, 1 };
matrix B[][4] = { 1, 0, 1, 0,
    0, 1, 0, 0,
    0, 0, 1, 1,
    0, 0, 1, 0 };
matcmplx eigVects;
matrix eigVectR, chkL, chkR;
vector eigValR;
Print(A); Print(B);
eigValR = real(eigG(eigVects, A, B)); Print(eigValR);
eigVectR = real(eigVects); Print(eigVectR);
chkL = A*eigVectR[:0]; printf("A*eigVect[:0] = "); Print(chkL);
```

```
chkR = eigValR[0]*B*eigVectR[:0];
printf("eigValR[0]*B*eigVect[:0] = ");Print(chkR);
}
matrix -> A(4,4)
[row
0000 1 0
0001 1 0 0 0 0
0002 -1 0 0
0003 0 0-1 0
]
matrix -> B(4,4)
[row
0000 1 0 0
0001 
0002 0
0003 0 0 0
]
vector -> eigValR(4)
[index
0000 2.1889 -2.1889 0.45685 -0.45685
]
matrix -> eigVectR(4,4)
[row
0000 1 0.43952 0.16722 0.34235
0001 0.45685 -0.2008 0.36603 -0.74938
0002 -0.33444 -0.54858 1 0.54858
0003 -0.2752 1 0.82288 -1
]
A*eigVect[:0] = matrix -> chkL(4,1)
[row
0000 1.4569
0001 1
0002 -1.3344
0003 -0.73205
]
eigValR[0]*B*eigVect[:0] = matrix -> chkR(4,1)
[row
0000 1.4569
0001 1
0002 -1.3344
0003 -0.73205
]
```


## Performance

For solving 1000 linear equations in 1000 unknowns with an Intel I5-430M CPU core on a PC, the run time was approximately 2.6 seconds. For 1000 linear equations with 5 solution vectors, the run time was $\sim 3.5$ seconds. For 8000 equations in 8000 unknowns, the run time was $\sim 20$ minutes.

## References

[1] Gene H. Golub and Charles F. Van Loan, MATRIX Computations, Third Edition, John Hopkins University PRESS, 1996.
[2] J Dongarra et al, LAPACK User's Guide, Third Edition, August 22, 1999.

## Chapter 15-Special Mathematical Functions

## Introduction

In the course of engineering and scientific analysis, special mathematical functions pop up such as Bessel functions; the error functions erf and erfc; the Gamma function; Chebyshev polynomials; Legendre Orthogonal polynomials, etc. Slide-Rule has a collection of these special functions, and documentation of these functions can be found under the Functions Reference (F7) under Special Mathematical Functions. This chapter will present plots of some of these functions.

## Bessel Functions of the first kind Jn

The Bessel function $\operatorname{Jn}(x)$ is defined as:

$$
J_{n}(x)=(1 / 2 x)^{n} \sum_{k=01}^{\infty} \frac{\left(-1 / 4 x^{2}\right)^{k}}{k!\Gamma(n+k+1)} \text {, and } \quad J_{n+1}(x)=\frac{2 n}{x} J_{n}(x)-J_{n-1}(x)
$$



Bessel Functions Yn, In, and Kn

$$
Y_{n}(x)=\frac{J_{n}(x) \cos (n \pi)-J_{-n}(x)}{\sin (n \pi)} \quad Y_{n+1}(x)=\frac{2 n}{x} Y_{n}(x)-Y_{n-1}(x)
$$




Incomplete Elliptic Integrals of the $1^{\text {st }}$ and 2nd kind
$F(\varphi \backslash m)=F(\varphi \mid \alpha)=\int_{0}^{\varphi} \frac{1}{\left(1-\sin ^{2} \alpha \sin ^{2} \theta\right)} d \theta, \quad E(\varphi \mid m)=E(\varphi \backslash \alpha\}=\int_{0}^{\varphi}\left(1-m \sin ^{2} \theta\right)^{1 / 2} d \theta, m=\sin ^{2} \alpha$



The error functions $\operatorname{erf}(\mathbf{x})$ and $\operatorname{erfc}(x)$

$$
\operatorname{erf}(x)=\frac{2}{\sqrt{\pi}} \int e^{-t^{2}} d t, \quad \operatorname{erfc}(x)=1-\frac{2}{\sqrt{\pi}} \int e^{-t^{2}} d t
$$



## The Gamma Function

$$
\Gamma(z)=\int_{0}^{\infty} t^{z-1} e^{-t} d t
$$



## The Beta Function B(z,w)

$$
\left.\left.B(z, w)=\frac{\Gamma(z) \Gamma(w)}{\Gamma(z+w)}, \quad b\right) z, w\right)=\int t^{z-1}(1-t)^{w-1} d t
$$

The Beta Function B(z,w)


## The Jacobian Elliptic Function



## The Incomplete Gamma Function P(a, $\mathbf{x}$ )



## The Exponential Integral



## Cbebyshev Polynomials



Jacobi Polynomials


## Legendre Polynomials



## Chapter 16 - Differential Equations and Numerical Integration

## Introduction

Differential equations are commonly used in science and engineering to create a mathematical model of a real world system. Often, however, there is no known analytical solution, and numerical solutions are required. The results of a numerical solution normally result in a plot and a hardcopy printed output of certain parameters. In this chapter we will limit the discussion to the solution of systems of ordinary differential equations, commonly called ODE's, as well as Numerical Integration techniques, and finally Numerical Optimization examples for function's of one or more variables. Note that the shell script code listed in this chapter is without the annotation code, as one can generate this code with mouse and keyboard entries once a given plot is displayed, and have the code automatically inserted into the shell script file (as described in Chapter 3).

Higher-order differential equations involve the higher derivatives $x^{k}(t), x^{k-1}(t) \ldots x^{\prime \prime}(t), x^{\prime}(t)$. They arise in mathematical models for problems in science and engineering. As an example, given the equation

$$
m x^{\prime \prime}(t)+c x^{\prime}(t)+k x(t)+b=g(t)
$$

We can re-write this ODE as a system of equations in the form.

$$
x^{\prime \prime}(t)=f\left(t, x(t), x^{\prime}(t)\right) \quad \text { with } x\left(t_{0}\right)=K_{1}, x^{\prime}\left(t_{0}\right)=K_{2}, x^{\prime \prime}\left(t_{0}\right)=K_{3}
$$

Note that in order to solve this system of equations, we need to know the starting value $\left(\mathrm{K}_{1}\right)$ of x at $\mathrm{t}_{0}$, as well as the first $\left(\mathrm{K}_{2}\right)$ and second $\left(\mathrm{K}_{3}\right)$ derivatives at $\mathrm{t}_{0}$. One of the most powerful methods for solving this system of differential equations is the fourth order Runge-Kutta equations given by,

$$
y_{n+1}=y_{n}+\frac{h}{6}\left(k_{1}+2 k_{2}+2 k_{3}+k_{4}\right)
$$

where,
$k_{1}=f\left(x_{n}, y_{n}\right), \quad k_{2}=f\left(x_{n}+1 / 2 h, y_{n}+1 / 2 h k_{1}\right)$
$k_{3}=f\left(x_{n}+1 / 2 h, y_{n}+1 / 2 h k_{2}\right), \quad k_{4}=f\left(x_{n+1}, y_{n}+h k_{3}\right)$
$h=$ step size
In Slide-Rule, we have an internal function to solve this system of equations, namely sysODE. Its prototype is given as follows.

```
matrix sysODE(void (*derivs)(float, float *, float *),
```

    float \(a\), float \(b\), vector \(Y s\), int \(N\), int Ndiv);
    where,
(*derivs)(float, float ${ }^{*}$, float *)= user supplied routine for computing the derivatives. The first input is the x value,
the next argument is the input vector for the variables $y$, and the last argument are the computed derivatives dydx.
$\mathrm{a}=$ starting point $(\mathrm{a}<\mathrm{b})$
$\mathrm{b}=$ ending point $(\mathrm{b}>\mathrm{a})$
Ys $=$ vector of starting values for each argument at a.
$\mathrm{N}=$ number of data points +1 . The increment between data points is $(\mathrm{b}-\mathrm{a}) / \mathrm{N}$.
Ndiv = integration factor. For non-stiff problems set to 1. For stiff problems increase up to 10 or more. This causes the interval to be divided up by a factor of Ndiv, with output at intervals of (b-a)/N.
Errors:
More than 10 arguments
Arguments times data points exceeds memory
Arithmetic error! signal $=8$. If this occurs, then increase the integration factor, i.e., Ndiv.
Returns:
Matrix of results. First row are abscissa values. Each succeeding row is solution vector for each argument. Refer to example file sysode.txt in sub-directory diffeq.

In order to use this function, the user must supply a subroutine (dervis) in their shell script to calculate the derivatives, as integration takes place through the $4^{\text {th }}$ order Runge-Kutta internal subroutine. As an example, for the differential equation given by,

$$
y^{(3)}+3 y^{\prime \prime}+4 y^{\prime}+12 y=0
$$

The dervis subroutine would be as follows,

```
void derivs(float t, float *y, float *dydt) {
    dydt[2] = -3*y[2] -4*y[1] -12*y[0]; // calculate 3 3
    dydt[1] = y[2]; // calculate 2nd derivative
    dydt[0] = y[1]; // calculate 1st derivative
// Note that y[0] = y, y[1] = y', y[2] = y'
}
```

Note that the variable $t$ (an input) is the abscissa value or the independent variable, which starts out as $t_{0}=a$, and increases to the value of $b$ by the step size, which is equal to $(b-a) / N$. The variable array dydt (an output), are the calculated derivative which are output to the internal function ( the $4^{\text {th }}$ order Runge-Kutta subroutine) to calculate the $y$ values, which come back into this function as the second argument in subroutine dervis. From the above, we note that the second derivative equals $y[2]$ since $y[2]$ is the integration of dydt[2] from the previous iteration (internally), and the first derivative equals y[1] since it's the integration of dydt[1]. Finally, y[0] is a solution value of this particular ODE at a given value of $t$ between $a$ and $b$.

## A Simple Electrical Circuit

Suppose were given the equation of an electric circuit as,

$$
\begin{aligned}
& L I^{\prime \prime}+R I^{\prime}+\frac{1}{C} I=A w E^{\prime}(w t), \mathrm{R}=200, \mathrm{~L}=.1, \mathrm{C}=.006 \\
& \text { and, } \mathrm{E}(\mathrm{wt})=200 \cos (40 \mathrm{t}) \text { with } \mathrm{I}(0)=0, \mathrm{I}^{\prime}(0)=0
\end{aligned}
$$

and we want to solve for the current $I$ over a range of 2 seconds. Re-writing the equation, we have,

$$
I^{\prime \prime}=-80000 \sin (40 t)-2000 I^{\prime}-1666.6667 I
$$

The user supplied function (dervis) then becomes,

```
void derivs(float x, float *y, float *dydt) {
    dydt[1] = -80000.*sin(40*x)-1666.6667*y[1]-2000.*y[0];
    dydt[0] = y[1];
}
```

Putting it all together, the plots for solving this system are shown below.



The shell script code for this example is:

```
void derivs(float x, float *y, float *dydx) {
    dydx[1] = -80000.*sin(40*x) -1666.6667*y[0] -2000*y[1];
    dydx[0] = y[1];
}
vector I, t;
vector ystart[] = { 0, 0, 0 };
vector T, Ic;
matrix M;
int i, N = 2001;
// do classical solution first
t = vecLin(0, 2, N); // Print(t);
I = -1.004*exp(-0.8337*t) + 0.0004*exp(-1999.1663*t) -0.0008*sin(40*t) + cos(40*t);
openPlot("Classical Solution");
plotxy(t, I, Sizeof(t));
pCRT();
// do numerical solution next
M = sysODE(derivs, 0., 2., ystart, 2000, 1);
openPlot("sysODE Solution");
plotxy(M[0], M[1], Sizeof(M[:0]));
pCRT();
```


## A Numerical Integration Example

Numerical integration is a tool used by engineers and scientists to obtain approximate answers for definite integrals that cannot be solved analytically. Two solution methods are Simpson's Rule, and Romberg Integration with Richardson's improvement. Simpson's equations to approximate a definite integral are given by,

$$
\begin{aligned}
& \int_{a}^{b} f(x) d x \cong \frac{h}{3}(f(a)+f(b))+\frac{2 h}{3} \sum_{k=1}^{M-1} f\left(x_{2 k}\right)+\frac{4 h}{3} \sum_{k=1}^{M} f\left(x_{2 k}-1\right) \\
& x_{k}=a+k h, \text { for } k=0,1,2, \cdots 2 M
\end{aligned}
$$

Note that $x_{0}=a$ and $x_{2 M}=b$, and $h$ is the step size
In Slide-Rule, we have two internal functions to approximate a definite integral, namely simpI and rombI. Their prototypes are given as follows.

```
float simpI(float (*func)(float x), float a, float b);
float rombI(float (*func)(float x), float a, float b);
```

where,
(*func)(float, float x ) is a user supplied routine for computing the value of the function at $x$ (note that this function returns a float type).
$\mathrm{a}=$ starting value for the integration process.
$\mathrm{b}=$ ending value for the integration process.
In order to use this function, the user must supply a subroutine (calcPts) in their shell script to calculate the starting (a) and ending (b) values of the desired function to integrate. As an example, for the function $e^{2 t} \cos (3 t)$, we have

```
float calcPts(float t)
{
    return(exp(2*t)* cos(3*t));
}
```

For the function as shown above, to compute the value from 0 to $\mathrm{pi} / 2$, we have the example as shown below.


The shell script code for this example is:

```
float calcPts(float t)
{
    return(exp(2.*t)*}\operatorname{cos(3.*t));
}
float ans,xmax, ftemp1, ftemp2, del, x, array[201];
int i;
xmax = pi/2.; del = xmax/200.;
for(i = 0, x = 0; i <= 200; i++) { // calculate plotting points
    array[i] = calcPts(x); x += del; }
openPlot("Numerical Integration Example");
ploty(0., del, array, 201);
pCRT();
ans = rombI(calcPts, 0, xmax);
printf("ans from rombI = %.12f\n", ans);
ans = simpI(calcPts, 0, xmax);
printf("ans from simpI = %.12f\n", ans);
ftemp1 = exp(pi)*(2.*cos(3.*pi/2.) + 3.*sin(3.*pi/2.))/13.;
ftemp2 = exp(0.)*(2.*cos(0.) + 3.*sin(0.))/13.;
ans = ftemp1 - ftemp2;
printf("ans from formula = %.12f\n", ans);
```


## The First Painleve’ Transcent

For solving a family of a differential equation, a good example is the First Painleve' Transscent given by,

$$
d^{2} y / d^{2} x=y^{2}-x \quad 0 \leq \mathrm{x} \leq 12, \text { and }-3.3 \leq \mathrm{x}^{\prime} \leq 0.7, \mathrm{x}^{\prime}+=0.2
$$

For this ODE as shown above, we have an example as shown below.


The shell script code for this example is:

```
void derivs(float x, float *p, float *dydx)
{
    dydx[1] = p[0]*p[0] - x; dydx[0] = p[1];
}
matrix M;
int NDIVS = 1000;
vector p[] = { 0, -3.3 }; // p[0] = x0, p[1] = x0
float dydx;
openPlot("The First Painleve' Transcendant");
for(dydx = -3.3; dydx <= .7; dydx += .2) {
        p[0] = 0; p[1] = dydx;
        M = sysODE(derivs, 0, 12, p, NDIVS, 1); // compute current case
        plotxy(M[0], M[1], NDIVS+1); // row 0 is x values, row 1 is y values
}
pCRT();
```


## A Spring Shock Absorber

Another simple example is a Sprint Shock Absorber system, where the differential equation is given by:

$$
\mathrm{dy}^{2} / \mathrm{dt}^{2}=5 \mathrm{e}^{-\mathrm{t} / 8} \cos (3 \mathrm{t})-0.5 \mathrm{dy} / \mathrm{dt}-8 \mathrm{y}
$$

The shell script code for this example and a plot of the solution follows.

```
        void derivs(float t, float *p, float *dydt)
        {
            dydt[1] = 5*exp(-t/8)* cos(3*t) -8*p[0] -.5*p[1];
            dydt[0] = p[1];
}
matrix M;
int NDIVS = 1000;
vector p[] = { .25, 1 };
float dydx;
M = sysODE(derivs, 0, 40, p, NDIVS, 1); // compute current case
openPlot("A Spring Shock-Absorber");
plotxy(M[0], M[1], NDIVS+1);
Title("A Spring Shock-Absorber");
xLabel("Time");
yLabel("Position",1);
axisS(0,1,0,0,0);
colorB(223,223,223);
grid();
pCRT();
```



## Van der Pol's Equation

The Van der Pol's differential equation is a very stiff ODE, and it takes special algorithms or techniques to solve this equation. The equation is given by,

$$
d^{2} y / d x^{2}=-\mu\left(1-y^{2}\right) d y / d x+y=0
$$

We can use the sysODE function to solve this equation by setting the last parameter in the calling sequence to a number greater then 1 . The current examples shown up to now have had this parameter set to 1 . The number of plotted points in the calling sequence is equal to N . If we set the last parameter Ndiv to say 100, the sysODE function still computes N points for plotting, but internally the step size is $1 / \mathrm{Ndiv}$ the step size of $(\mathrm{b}-\mathrm{a}) / \mathrm{N}$ of each n point calculated. Internally, sysODE just integrates $1 /$ Ndiv times in a tight loop before returning a value to sysODE. Some would call this a brute force method, but with a modern computer and their precision and speed, we can get a solution, it just takes a little longer. Refer to the plot below for a solution to this ODE with $\mu=1000$.

```
void derivs(float x, float *y, float *dydx) {
    dydx[1] = -y[0] + 1000.*(1.-y[0]*y[0])*y[1];
    dydx[0] = y[1];
}
vector ystart[] = { 1., 1. };
matrix M;
float x[400], y[400], alpha, del;
int i;
echof("This VERY STIFF ODE will take ~ 10 seconds\n");
M = sysODE(derivs, 0., 3000, ystart, 30000, 130);
openPlot("sysODE Solution");
penS(0,1,128,0,255);
plotxy(M[0], M[1], Sizeof(M[:0]));
penS(0,3,210,0,0);
grid();
xLabel("t - axis"); yLabel("y axis",0);
Title("Solution to Van der Pol's Equation");
axisS(0,1,0,0,0);
pCRT();
```



## The Lorenz Strange Attractor

We can also solve ODE's that have $1^{\text {st }}$ derivatives of three or more variables (up to 10) with respect to a 4th variable. In effect solving 3 first order ODE's. As an example of such a system, we have a version of the Lorenz Strange Attractor equations given by,

$$
d x / d t=-10 x+10 y, \quad d y / d t=-x z+30 x-y, \quad d z / d t=x y-3 z
$$

The inital conditions being, $x_{0}=-8, y_{0}=8, z_{0}=64$


The shell script code for this example is:

```
void derivs(float t, float *p, float *dpdt) {
    dpdt[0] = -10*p[0] + 10*p[1];
    dpdt[1] = -p[0]*p[2] + 30*p[0] - p[1];
    dpdt[2] = p[0]*p[1] - 3*p[2];
}
matrix M;
vector yStart[] = {-8, 8, 64 };
vector Vx, Vz;
float tStart = 0, tEnd = 50, numPoints = 10000;
M = sysODE(derivs, tStart, tEnd, yStart, numPoints, 10);
openPlot("Lorenz Strange Attractor");
Vx = M[1]; Vz = M[3];
reduce(Vx, 0, 100); reduce(Vz, 0, 100);
plotxy(Vx, Vz, Sizeof(Vx));
pCRT();
```


## A Projectile Range Calculation

A more sophisticated example is the calculation of a projectile given an initial elevation and velocity using Newtonian mechanics, plus an Air Drag Coefficient and Relative Air Density Model. Were given the following range data for a projectile, and we want to calculate the range is meters, the flight time in seconds, the final velocity in seconds, and the fall angle in degrees.

| Elev. <br> Mils | Range <br> meters | Flight Time <br> seconds | Finalvelocity <br> meters/sec. <br> 240.8Fall <br> deg. |  |
| :--- | :--- | :--- | :--- | :--- |
| 9.0 | 100 | 0.43 | 227.7 | 0.50 |
| 18.6 | 200 | 0.88 | 215.2 | 1.10 |
| 29.1 | 300 | 1.36 | 203.3 | 1.80 |
| 40.4 | 400 | 1.87 | 192.0 | 2.60 |
| 52.6 | 500 | 2.40 | 181.7 | 3.60 |
| 65.9 | 600 | 2.97 | 171.9 | 4.70 |
| 80.3 | 700 | 3.58 | 162.5 | 5.90 |
| 96.0 | 800 | 4.22 | 153.9 | 7.30 |
| 113.1 | 900 | 4.90 | 145.7 | 8.90 |
| 131.9 | 1000 | 5.63 | 138.1 | 10.70 |
| 152.5 | 1100 | 6.41 | 131.1 | 12.80 |
| 175.2 | 1200 | 7.24 | 124.7 | 15.20 |
| 200.4 | 1300 | 8.14 | 118.6 | 17.90 |
| 228.6 | 1400 | 9.11 | 113.4 | 21.00 |
| 260.3 | 1500 | 10.17 | 108.5 | 24.50 |
| 296.7 | 1600 | 11.34 | 104.5 | 28.40 |
| 339.2 | 1700 | 12.65 | 101.5 | 32.90 |
| 390.8 | 1800 | 14.17 | 99.1 | 38.20 |
| 458.3 | 1900 | 16.06 | 98.1 | 44.40 |

We start with Newton's second law of motion, i.e., $\mathrm{f}=\mathrm{ma}=\mathrm{mg}$, where $\mathrm{g}=-9.8 \mathrm{~m} / \mathrm{s}$. We note that mils equals pi $/ 3200$, and the initial muzzle velocity is $240.8 \mathrm{~m} / \mathrm{s}$. The model for air density y versus altitude $u$ in meters is given by $y=(1-2.25577 e-5 * u)^{\wedge} 5.25588$. The air drag coefficient is given by
-0.0005682 . If v is the instantaneous velocity in $\mathrm{m} / \mathrm{s}$, then the de-acceleration of g due to elevation and velocity is given by $\mathrm{dec}=\mathrm{v}^{2}$ (airDen*.0005682). The subroutine to calculate derivatives of this model is as follows, and the code of this shell script can be found in file C: \SlideRulelother\Projectile.txt.

```
void derivs(float t, float *p, float *dydt)
{
    float Vxi, Vyi;
    float Vy, Vx, V2, Speed, sinT, cosT, airDen, decl;
    if(t == 0) { // if time zero, then first entry!!
    Vxi = p[3]; Vyi = p[1]; // initialize initial velocities
    p[1] = 0; p[3] = 0; } // initialize dynamic velocities
    Vy = Vyi+p[1] - 9.8*t; Vx = Vxi+p[3]; // compute curent Y & X
                                    velocities
    dydt[0] = Vy; dydt[2] = Vx; // integrate Y & X velocities to altitude
    V2 = Vy*Vy + Vx*Vx; // Compute current velocity squared
    Speed = sqrt(V2); // Compute current speed
    sinT = Vy/Speed; cosT = Vx/Speed;// Compute sin/cos of current
                                    trajectory
    cosT = Vx/Speed; // Compute relative air density
    airDen = pow(1 - 2.25577e-5*p[0], 5.25588);// p[0] is current altitude
    decl = V2*(airDen*.0005682); // Compute de-acceleration
    dydt[1] = -decl*sinT; // integrate y de-accel. to real time velocity
        dydt[3] = -decl*cosT; // integrate x de-accel. to real time velocity
}
```



The printed results of this simulation are as follows.

| Init | Known | Calculated | Per | Flight | Flight | Per | Final | Final | Per | Fall | Fall |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Elev | Range | Range | cent | Time | Time | cent | Velocity | Velocity | cent | Angle | Angle | Off |
| mils | m | m | error | sec | sec | error | $\mathrm{m} / \mathrm{s}$ | $\mathrm{m} / \mathrm{s}$ | error | degrees | degrees | deg. |
| 9.0 | 100 | 100.606793 | +0.61 | 0.43 | 0.430 | 0.00 | 227.7 | 227.425 | -0.12 | 0.5 | 0.53 | 0.03 |
| 18.6 | 200 | 199.987891 | -0.01 | 0.88 | 0.880 | 0.04 | 215.2 | 214.942 | -0.12 | 1.1 | 1.13 | 0.03 |
| 29.1 | 300 | 300.540427 | +0.18 | 1.36 | 1.362 | -0.11 | 203.3 | 203.016 | -0.14 | 1.8 | 1.83 | 0.03 |
| 40.4 | 400 | 400.503156 | +0.13 | 1.87 | 1.869 | 0.05 | 192.0 | 191.842 | -0.08 | 2.6 | 2.64 | 0.04 |
| 52.6 | 500 | 500. 105197 | +0.02 | 2.40 | 2.405 | -0.22 | 181.7 | 181.354 | -0.19 | 3.6 | 3.57 | -0.03 |
| 65.9 | 600 | 600.163669 | +0.03 | 2.97 | 2.976 | -0.21 | 171.9 | 171.422 | -0.28 | 4.7 | 4.65 | -0.05 |
| 80.3 | 700 | 699.825177 | -0.02 | 3.58 | 3.580 | 0.00 | 162.5 | 162.128 | -0.23 | 5.9 | 5.87 | -0.03 |
| 96.0 | 800 | 799.604566 | -0.05 | 4.22 | 4.222 | -0.05 | 153.9 | 153.418 | -0.31 | 7.3 | 7.28 | -0.02 |
| 113.1 | 900 | 899.178839 | -0.09 | 4.90 | 4.904 | -0.08 | 145.7 | 145.273 | -0.29 | 8.9 | 8.88 | -0.02 |
| 131.9 | 1000 | 999.198502 | -0.08 | 5.63 | 5.634 | -0.07 | 138.1 | 137.671 | -0.31 | 10.7 | 10.72 | 0.02 |
| 152.5 | 1100 | 1098.985683 | -0.09 | 6.41 | 6.412 | -0.03 | 131.1 | 130.639 | -0.35 | 12.8 | 12.82 | 0.02 |
| 175.2 | 1200 | 1198.698393 | -0.11 | 7.24 | 7.244 | -0.06 | 124.7 | 124.171 | -0.42 | 15.2 | 15.21 | 0.01 |
| 200.4 | 1300 | 1298.534329 | -0.11 | 8.14 | 8.141 | -0.01 | 118.6 | 118.288 | -0.26 | 17.9 | 17.92 | 0.02 |
| 228.6 | 1400 | 1398. 562224 | -0.10 | 9.11 | 9.113 | -0.03 | 113.4 | 113.000 | -0.35 | 21.0 | 21.01 | 0.01 |
| 260.3 | 1500 | 1498. 265746 | -0.12 | 10.17 | 10.169 | 0.01 | 108.5 | 108.378 | -0.11 | 24.5 | 24.49 | -0.01 |
| 296.7 | 1600 | 1598. 367020 | -0.10 | 11.34 | 11.339 | 0.01 | 104.5 | 104.428 | -0.07 | 28.4 | 28.46 | 0.06 |
| 339.2 | 1700 | 1698.385774 | -0.09 | 12.65 | 12.652 | -0.02 | 101.5 | 101. 260 | -0.24 | 32.9 | 32.98 | 0.08 |
| 390.8 | 1800 | 1798.644275 | -0.08 | 14.17 | 14.177 | -0.05 | 99.1 | 98.987 | -0.11 | 38.2 | 38.19 | -0.01 |
| 458.3 | 1900 | 1899.370648 | -0.03 | 16.06 | 16.070 | -0.06 | 98.1 | 97.864 | -0.24 | 44.4 | 44.45 | 0.05 |
| 570.1 | 2000 | 2000.864125 | +0.04 | 18.97 | 18.978 | -0.04 | 98.8 | 98.783 | -0.02 | 53.2 | 53.23 | 0.03 |
| Mean |  |  | +0.00 |  |  | -0.05 |  |  | -0.21 |  |  | 0.01 |
| Std De |  |  | +0.16 |  |  | +0.07 |  |  | +0.11 |  |  | 0.03 |

## The Minimum of a Function of a Single Variable

Solving the minimum of a function of a single variable can be accomplished by using the Golden search method.* So given the function $x^{2}-\sin (x)$ between the bounds of 0 and 10 , find the minimum in y and x . Using internal function fmin , the shell script and plot follows.

```
float funcX(float x)
{
    float y;
    y = x*x - sin(x);
    return(y);
}
float xp, yp, x;
int i, N = 1001;
vector Xp, Yp, y[N];
xp = fmin(funcX, 0, 1, 1.e-8); yp = funcX(xp);
printf("xp = %.8f : Yp = %.8f\n", xp, yp);
for(i = 0, x = 0; i < N; i++, x += .001)
    y[i] = x*x - sin(x);
Xp = {xp }; Yp = { yp };
openPlot("The minimum of a function of a single variable");
Title("The Minimum of a Function of a Single Variable");
ploty(0, .001, y, N); plotxy(Xp, Yp, 1);
grid();
pCRT();
```

[^40]

## The Minimum's of a Function of a Several Variables

Solving the minimum of a function of a several variables can be accomplished by using the Nelder-Mead method.* So given the function $f(x, y, z)=3\left(x^{\wedge} 3+y^{\wedge} 3-z^{\wedge} 3\right)+(z-x-$ $y$ ), find the minimum of $x, y, z$. Using internal function $f$ mins, the shell script and print out follows.

```
float FuncX(float P[])
{ // f(x,y,z) = 3(x^3 + y^3 -z^3) + (z - x - y)
    float t;
    t = 3.*P[0]*P[0]*P[0]+3.*P[1]*P[1]*P[1]-3.*P[2]*P[2]*P[2]+P[2]-P[0]-P[1];
    return(t);
}
vector Dat[] = { 0, 0, 0};
vector S;
float x[1000], y[1000], z[1000], P[1000];
float Array[10];
int i;
float t;
printf("A Demo of Function fmins for finding the minimum\n");
printf("of an analytic function of several variables\n\n");
printf("For function f(x,y,z) = 3(x^3 + y^3 - z^3) + (z -x -y)\n\n");
printf("The initial guess for x,y,z "); Print(Dat);
S = fmins(FuncX, Dat, 1000, 1.e-8);
```

[^41]```
printf("The coordinates at the minimum are "); Print(S);
convToA(Array, S);
printf("The value at the minimum is f(x,y,z) = %f\n", FuncX(Array));
*****************************************************************************
    A Demo of Function fmins for finding the minimum
    of an analytic function of several variables
    For the 1st function f(x,y,z) = 3(x^3 + y^3 - z^3) + (z -x - y)
    The initial guess for x,y,z vector -> Dat(3)
    [index
    0000 0 0 0
    ]
    The coordinates at the minimum are vector -> S(3)
    [index
    0000 0.33333 0.33333 -0.33333
    ]
    The value at the minimum is f(x,y,z) = -0.666667
```


## References

[1] John H. Mathews and Kurtis D. Fink, Numerical Methods Using MATLAB [4th. ed.], PrenticeHall, 2004.
[2] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright, Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions, SIAM J. Optim, Vol. 9, No. 1, pp. 112-147.

## Chapter 17-Control Engineering

## Introduction

With the trend moving from analog systems to digital implementations, the Control Engineer must still master the field of Control Engineering in the analog domain. The following topics cover some of the more basic models that occur repeatedly in the field of Control Engineering. This chapter will present some examples using Root-Locus techniques. The Control Engineering Tool Box in Slide-Rule covers the basic functions, and can be considered a SISO (single input, single output) implementation in the analog domain. Note that the shell script code listed in this chapter is without the annotation code, as one can generate this code with mouse and keyboard entries once a given plot is displayed, and have the code automatically inserted into their shell script file.

## A Unit-Step Response

Given the transfer function $16 /\left(s^{2}+3 s+16\right)$, plot the unit step response. Using function stepTF, the shell script and plot is shown below.

```
#include "control.h"
    vector num[] = { 16 };
    vector den[] = { 1, 3, 16 };
vector plotData;
plotData = stepTF(num, den, 5);
openPlot("A Unit Step Response");
Title("Unit-Step response of G(s) = 16/(s^2 +3s + 16)");
ploty(0, 5/500., plotData, 500);
grid();
xLabel("Time(sec)");
yLabel("Amplitude",0);
pCRT();
```



## A Unit-Impulse Response

Given the transfer function $16 /\left(s^{2}+3 s+16\right)$, plot the unit impulse response. Using function impulseTF, the shell script and plot is shown below.

```
    #include "control.h"
    vector num[] = { 16 };
    vector den[] = { 1, 3, 16 };
    vector plotData;
plotData = impulseTF(num, den, 5);
openPlot("A Unit Impulse Response");
Title("Unit-Impulse response of G(s) = 16/(s^2 +3s + 16)");
ploty(0, 5/500., plotData, 500);
grid();
xLabel("Time(sec)");
yLabel("Amplitude",0);
pCRT();
```



## Step Responses of Second-Order Systems

Plot the step responses for the transfer function $1 / s(s+2 z e t a N)$, for values of zetaN from 0 to 2.*

```
matrix Ct[12][500];
float Ont, Odt, Zeta, del, s1, s2;
int i, j;
// generate data for plot
del = 12./500.;
for(i = 0; i < 12; i++) {
    Zeta = i/10.+ .0001;
    if(i == 11) {
    Zeta = 2.;
```

    s1 = Zeta + sqrt(Zeta*Zeta-1.);
    s2 = Zeta - sqrt(Zeta*Zeta-1.);
    printf("s1 = \%f : s2 = \%f\n", s1, s2);
    \}
    for (j = 0, Ont = 0.; \(j<500 ; j++)\) \{
        if(i < 10) \{ // Underdamped case, zeta < 1
            Odt = Ont*sqrt(1.-Zeta*Zeta);
            Ct[i][j] = 1.-exp(-Zeta*Ont)*sin(Odt + atan(sqrt(1.-
                                    Zeta*Zeta)/Zeta))/sqrt(1.-Zeta*Zeta);
    \}
else if(i == 10) \{ // Critically damp case, zeta = 1
$\operatorname{Ct}[\mathrm{i}][\mathrm{j}]=1 .-\exp (-$ Ont $) *(1 .+$ Ont);
\}
else \{
Ct[i][j] = 1. + (exp(-s1*Ont)/s1 - exp(-
s2*Ont)/s2)/(2.*sqrt(Zeta*Zeta-1.));
Ct[i][j] = 1. - $\exp \left(-s 2^{*} O n t\right)$;
\}
Ont += del;
\}
\}
openPlot("Unit-Step Response Curves");
pminmax(0,0,0,2);
for $(\mathrm{i}=0 ; \mathrm{i}<12$; i++)
ploty(0, del, Ct[i], 500);
grid();
xLabel("\omega_nt");
yLabel("c(t)",1);
Title("Unit-Step Response Curves with \omega_n = 1 and values of \zeta");
arrowP(3.104063, 0.530949,3.877106,0.426410,0);
ptext(3.960357,0.401651, "2.0");
pCRT();


## Bode Plots from a Transfer Function

Plot Bode plots for the transfer function $G(s)=K /\left(s^{3}+3 s^{2}+6+K\right)$ for $K=1,4$, and 8 .


The shell script for this example follows (includes annotation functions).

```
#include "control.h"
matrix plotV1, plotV2, plotV3;
vector num[] = { 8 };
vector den[] = { 1, 3, 6, 8 };
plotV1 = bodeTF(num, den, .1, 100, 0, 2);
```

```
num[0] = 4; den[3] = 4;
plotV2 = bodeTF(num, den, .1, 100, 0, 2);
num[0] = 1; den[3] = 1;
plotV3 = bodeTF(num, den, .1, 100, 0, 2);
openPlot("Bode Plot of Phase");
xLabel("Frequency (rad/sec)");
yLabel("Phase deg",1);
plogxy(plotV1[0], plotV1[2], colSize(plotV1));
plogxy(plotV2[0], plotV2[2], colSize(plotV2));
plogxy(plotV3[0], plotV3[2], colSize(plotV3));
grid();
Title("Bode Plot of Phase for G(s) = K/(s^3 + 3s^2 + 6s +K)");
ptext(38.553023,-60.900963,"K = 8");
ptext(13.181368,-82.083906,"K = 1");
ptext(18.731417,-52.579092,"K = 4");
pCRT();
openPlot("Bode plot of Magnitude");
xLabel("Frequency (rad/sec)");
yLabel("Magnitude dB",1);
plogxy(plotV1[0], plotV1[1], colSize(plotV1));
plogxy(plotV2[0], plotV2[1], colSize(plotV2));
plogxy(plotV3[0], plotV3[1], colSize(plotV3));
grid();
Title("Bode Diagam of Closed-Loop System G(s) = K/(s^3 + 3s^2 + 6s + K)");
arrowP(57.879088,-25.226960,68.483647,-8.060523,0);
ptext(69.177403,-9.848693,"K = 8");
arrowP(59.563925,-34.704264,68.483647,-21.292985,0);
ptext(69.177403,-23.081155,"K = 4");
arrowP(59.563925,-46.685007,68.483647,-33.452545,0);
ptext(69.177403,-35.240715,"K = 1");
pCRT();
```


## Nyquist Plots from State-Space

Given the State Space matrices $\mathrm{A}=(-1,-1,12,0), \mathrm{B}=(1,1,1,0), \mathrm{C}=(1,0,0,1), \mathrm{D}=(0,0,0,0)$, plot the Nyquist plots.

```
#include "control.h"
matrix A[][2] = { -1, -1, 12, 0 };
matrix B[][2] = { 1, 1, 1, 0 };
matrix C[][2] = { 1, 0, 0, 1 };
matrix D[][2] = { 0, 0, 0, 0 };
matrix Pdat;
Pdat = nyquistSS(A, B, C, D, 0);
openPlot("Nyquist Plot from U1");
pminmax(-2,2,-2,2);
Title("From: U_1 To: Y_1");
xLabel("Real Axis");
yLabel("Imaginary Axis",0);
penS(0,1,210,0,0);
plotxy(Pdat[0], Pdat[1], Sizeof(Pdat[:0]));
penS(0,1,0,210,0);
plotxy(Pdat[0], Pdat[2], Sizeof(Pdat[:0]));
grid();
aspectR(1);
pCRT();
```

```
Pdat = nyquistSS(A, B, C, D, 1);
openPlot("Nyquist Plot from U1b");
pminmax(-4,4,-4,4);
Title("From: U_1 To: Y_2");
xLabel("Real Axis");
yLabel("Imaginary Axis",0);
penS(0,1,210,0,0);
plotxy(Pdat[0], Pdat[1], Sizeof(Pdat[:0]));
penS(0,1,0,210,0);
plotxy(Pdat[0], Pdat[2], Sizeof(Pdat[:0]));
grid();
aspectR(1);
pCRT();
Pdat = nyquistSS(A, B, C, D, 2);
openPlot("Nyquist Plot from U2");
pminmax(-1,1,-1,1);
Title("From: U_2 To: Y_1");
xLabel("Real Axis");
yLabel("Imaginary Axis",0);
penS(0,1,210,0,0);
plotxy(Pdat[0], Pdat[1], Sizeof(Pdat[:0]));
penS(0,1,0,210,0);
plotxy(Pdat[0], Pdat[2], Sizeof(Pdat[:0]));
grid();
aspectR(1);
pCRT();
Pdat = nyquistSS(A, B, C, D, 3);
openPlot("Nyquist Plot from U2b");
pminmax(-4,4,-4,4);
Title("From: U_2 To: Y_2");
xLabel("Real Axis");
yLabel("Imaginary Axis",0);
penS(0,1,210,0,0);
plotxy(Pdat[0], Pdat[1], Sizeof(Pdat[:0]));
penS(0,1,0,210,0);
plotxy(Pdat[0], Pdat[2], Sizeof(Pdat[:0]));
grid();
aspectR(1);
pCRT();
```




## Gain Adjustment for \% Overshoot*

Given the open-loop system as shown below, determine the value of the Gain K to yield a $25 \%$ overshoot, and plot the step response.


The first step is to plot the Root-Locus of the open loop system. Once the Root-Locus plot is displayed, we then move the mouse cursor along the root-locus line while observing the continuous read out of the OS \% values displayed on the right side of the monitor. When the value is $\sim 25 \%$, we observe the K value to be $\sim 601$ (also displayed on the right side of one's monitor). We then set the numerator of the open-loop transfer function to 601 , call function cloopTF to calculation the closed-loop system, and finally function stepTF to calculate plotting points for the step plot. Note that these functions are defined in the Slide-Rule Functions Reference under the heading of Control Engineering Tool Box. The shell script and plots are shown below.

[^42]


```
#include "control.h"
vector num[] = { 1 };
vector den[] = { 1, 102, 200, 0 };
vector den1, num1, den2, num2, plotD;
matrix sys;
openPlot("Gain Agjustment for 25% over-shoot");
rlocusTF(num, den);
grid();
pCRT();
num = { 601 };
sys = cloopTF(num, den, -1); Print(sys);
num1 = sys[0]; den1 = sys[1];
Print(num1); Print(den1);
openPlot("Step response for 25% over-shoot");
pminmax(0,10,0,1.5);
plotD = stepTF(num1, den1, 10);
ploty(0, 10./Sizeof(plotD), plotD, Sizeof(plotD));
pCRT();
```


## A PI Compensator*

Given the Plant as shown below, design a PI compensator with a damping ratio of 0.2.
Compensator
Plant


[^43]The first step is to plot the Root-Locus of the open loop Plant. Once the Root-Locus plot is displayed, we then move the mouse cursor along the root-locus line while observing the continuous read out of the zeta values displayed on the right side of the monitor. When the value of zeta is $\sim 0.2$, we observe the K value to be $\sim 193.3$ (also displayed on the right side of one's monitor). For a PI Compensator, we add a pole at the origin and a zero at 0.1 . We then plot a Root-Locus plot of this compensated system. Again, moving the mouse cursor along the rootlocus line until zeta is $\sim 0.2$, we read out a K value of $\sim 186.7$. For the uncompensated system, we set the numerator of the plant to 193.9, call function cloopTF to calculation the closed-loop system, and finally function stepTF to calculate plotting points for the step plot. For the PI Compensated system, we set $\mathrm{K}=186.7$, and repeat as above. The shell script and plots for this example are shown below.



\#include "control.h"
vector num[] = \{ 1 \};
vector den[] = \{ 1, 15, 38, 24$\}$;
vector num1[] = \{ 1, 0.1$\}$;

```
vector den1[] = { 1, 15, 38, 24, 0 };
vector den2, num2, num3, den3, plotD;
matcmplx sys;
openPlot("Root Locus of Plant");
rlocusTF(num, den);
pCRT();
openPlot("Root Locus of Plant with PI Compensator");
rlocusTF(num1, den1);
pCRT();
num = { 193.3 };
sys = cloopTF(num, den, -1);
num2 = real(sys[0]); den2 = real(sys[1]); Print(num2); Print(den2);
num1 = { 186.7, 18.67 }; // 186.7*(s + 0.1)
sys = cloopTF(num1, den1, -1);
num3 = real(sys[0]); den3 = real(sys[1]); Print(num3); Print(den3);
openPlot("Step responses");
plotD = stepTF(num2, den2, 40);
ploty(0, 40./Sizeof(plotD), plotD, Sizeof(plotD));
plotD = stepTF(num3, den3, 40);
ploty(0, 40./Sizeof(plotD), plotD, Sizeof(plotD));
pCRT();
```


## A Lag Compensator*

Given the Plant as shown in the PI Compensator, we want to design a Lag Compensator with the same damping ratio 0.2. Note that a PI Compensator with a pole at the origin, requires an active integrator, which requires more current. If we more the pole and zero to the left and close to the origin, we have a Lag Compensator, and we can use passive components to fabricate it. Setting the pole at 0.01 , we calculate the zero to be at 0.112 (refer to equations and discussion in the references). The techniques using Root-Locus plots, are then the same as described in the previous example.

[^44]

Root-Locus of $\mathrm{G}(\mathrm{s})=\mathrm{K}(\mathrm{s}+0.112) /[(\mathrm{s}+.01)(\mathrm{s}+1)(\mathrm{s}+2)(\mathrm{s}+12)]$



```
vector num[] = {1 };
vector den[] = { 1, 15, 38, 24 };
vector num1[] = { 1,0.111 };
vector den1, den2, num2, num3, den3, plotD;
matcmplx sys;
float Kpo;
openPlot("Root Locus of Plant");
rlocusTF(num, den);
pCRT();
openPlot("Root Locus of Plant with Lag Compensator");
den1 = polyM(den,(vector){ 1, .01 }); Print(den1);
rlocusTF(num1, den1);
pCRT();
num = { 194.9 };
sys = cloopTF(num, den, -1);
num2 = real(sys[0]); den2 = real(sys[1]);
Print(num2); Print(den2);
Kpo = 1/gainTF(num, den); Print(Kpo);
num1 = { 190.2, 21.1122 };
sys = cloopTF(num1, den1, -1);
num3 = real(sys[0]); den3 = real(sys[1]);
Print(num3); Print(den3);
openPlot("Step responses");
plotD = stepTF(num2, den2, 40);
ploty(0, 40./Sizeof(plotD), plotD, Sizeof(plotD));
plotD = stepTF(num3, den3, 40);
ploty(0, 40./Sizeof(plotD), plotD, Sizeof(plotD));
pCRT();
```


## An Ideal Derivative Compensator (PD)*

Given the Plant $\mathrm{H}(\mathrm{s})=\mathrm{K} /[\mathrm{s}(\mathrm{s}+3)(\mathrm{s}+8)]$, design a PD Compensator with a $20 \%$ overshoot and a settling three times faster than the un-compensated system. Note that if the closed loop pole (in the S domain) is not on the root-locus for the desired design, then we can re-shape the open-loop transfer function by adding a zero on the real axis at the appropriate point, such that the compensator has the transfer function of $\mathrm{Gc}=\mathrm{s}+\mathrm{z}_{\mathrm{c}}$. We first do a Root-Locus plot for the openloop Plant, and for a $20 \%$ overshoot, we read out $\mathrm{K}=\sim 47.1$ with a settling time (Ts, read out from the right side of the monitor) $=\sim 3.75$ seconds. For a settling time 3 times faster the real part of the pole on the same OS $\%$ line is $-4 /(3.75 / 3)=-3.12$. With the cursor thus placed, we read out the value of the compensating zero of $\sim-4.24$.


[^45]


```
vector num[] = { 1 };
vector den[] = { 1, 11, 24, 0 };
vector num1, den1, den3, num3, plotD;;
vector num2[] = { 1, 4.24 };
vector den2[] = { 1, 11, 24, 0 };
matcmplx sys;
openPlot("Root Locus of Plant");
rlocusTF(num, den);
pCRT();
openPlot("Root Locus of Plant with PD Compensator");
rlocusTF(num2, den2);
pCRT();
num = { 47.1 };
sys = cloopTF(num, den, -1);
num1 = real(sys[0]); den1 = real(sys[1]);
num2 = { 53.4, 53.4*4.25 };
sys = cloopTF(num2, den2, -1);
num3 = real(sys[0]); den3 = real(sys[1]);
openPlot("Step responses");
plotD = stepTF(num1, den1, 5);
ploty(0, 5./Sizeof(plotD), plotD, Sizeof(plotD));
plotD = stepTF(num3, den3, 5);
ploty(0, 5./Sizeof(plotD), plotD, Sizeof(plotD));
pCRT();
```


## A Lead-Lag Compensator*

Given the Plant $G(s)=K /[(s+5)(s+10)]$, add a lead-lag zero/pole such that the system will operate with a $20 \%$ over-shoot, and with a settling time of half that of the un-compensated system. First add a pole at 0.01 to the open-loop Plants transfer function, then trace the rootlocus curve until $\% \mathrm{OS}=20$ percent, and copy down the K value ( $\sim 145$ ), and the x and y coordinates $(\sim-1.6+3.1 \mathrm{j})$. For a reduction in settling time, the real part of the pole must be increased by a factor of 2 , i.e., from -1.6 to -3.2 . Move the cursor to -3.2 along a zeta line such that $\% \mathrm{OS}=\sim 20$ percent and the real value of the cursor location is $\sim-3.2$. Click your left mouse button and enter a zero value of -5 to cancel the pole value of -5 . The compensating pole Pc will be calculated at $\sim-20$ (and displayed on the right side of one's monitor). Construct 2 nd root-locus plot to determine the gain value $\mathrm{K}(\sim 1166.2)$. Finally add a plot for verification. The plot for this design follows, and the shell script code can be found in Slide-Rule sub-directory ControlSystems and file LeadLagcomp.txt.

[^46]



## A Regulator Design in State-Space with Pole-Placement

Given the Plant $G(s)=10(s+6) /[(s+1)(s+5)]$, calculate the gain $K$ for a $20 \%$ overshoot, and a settling time of 1 second. The block diagram of the State-space representation with feedback is shown below.


We first calculate the damping ratio $\zeta$ by calling routine zetaOS in the Control Systems Toolbox, where $\zeta=-\ln (\% O S / 100) / \sqrt{\pi^{2}+\ln (\% O S / 100)}$ (refer to [2], pg 195). Next we calculate $\omega_{n}=4 / \zeta T_{S}, \quad T_{s}=$ settling time (refer to [2], pg 197). Function secondordSys is then called to calculate a second order system vector. The poles of this $2^{\text {nd }}$ order system are then calculated by calling function polyR. Routine TFtoSSControl is then called to convert the transfer function into State-space Controllable Canonical form. Next, function controlMat is called to determine
if the State-space representation is controllable. The feedback vector is then calculated with a pole ( -5.9 ) close to the zero (6) of the Plants transfer function. Function ackerMan is called to calculate the feedback gain Ke (refer to [1], pgs 834-836, or [2], pgs 730-732). Note, function zetaOS can be found in the Slide-Rule's Functions Reference (F7) under the Control Systems Toolbox and 2ndOrdCalc; function TFtoSSControl can be found under TFtoSS. The plot of this example can be found below; and the shell script code can be found in Slide-Rule sub-directory ControlSystems and file regulatorDemo.txt.


## Controller versus Observable Design in State Space

Given the Plant $G(s)=(s+4) /[(s+1)(s+2)(s+6)]$, design the system for a $20 \%$ overshoot and a settling time of 4 seconds using the State-Space Controller method, then re-design using the State-Space Observerable method with a 10 times faster settling time. The reader is referred to [2], pgs $745-757$. The plot of this example can be found below; the shell script code can be found in Slide-Rule sub-directory ControlSystems and file observerDemo.txt.


## A Digital Compensator

Given the closed loop transfer function $601 /\left(s^{\wedge} 3+102 s^{\wedge} 2+100 \mathrm{~s}+601\right)$, convert this to a digital system (using the bilinear transform) with a sampling rate of 100 Hz , then plot the step response. The plot of this example can be found below; and the shell script code can be found in Slide-Rule sub-directory ControlSystems and file digitalComp.txt.


Note that in Chapter 7 we covered how one could change to a different custom menu. If one selects under the Config tile User Menu Selection, then the Click for File Menu selection, an enters OgatasControl, one can observe 126 examples given in [1] that pertain to this software. This was done to verify the integrity of this design tool.

## References

[1] Katsuhiko Ogata, Modern Control Engineering, $4^{\text {th }}$ Edition, Prentice-Hall, 2002.
[2] Norman S. Nise, Control Systems Engineering, $4^{\text {th }}$ Edition, John Wiley \& Sons, 2004.
[3] Katsuhiko Ogata, Modern Control Engineering, $5^{\text {th }}$ Pearson Education Inc., 2010.

## Chapter 18 - Microwave Transistor Amplifier Design

## Introduction

This module of Slide-Rule has been designed as a basic Graphical RF design package using the Smith Chart as the basic overlay. It allows the Engineer or Student of Microwave engineering a quick and fast method of designing simple RF circuits up to and including complex circuits such as a multistage wideband amplifier. Because of the graphical nature of this design package, the Engineer can get an intuitive feel in the design stages because of the visual presentation of displayed data. If you're new to Smith Chart techniques or need a refresher course, read the first 3 references at the end of this chapter. This chapter takes you through several design examples with step by step procedures along with a discussion of theory and equations where appropriate Note: The user needs to create a sub-directory (refer to (INFO) Script Directory $=(\ldots)$ in Chapter 2).under the SlideRule directory to store the shell script files that will be created by their designs using one's mouse and keyboard. Also note that in Chapter 7 we covered how one could change to a different custom menu. If one selects under the Config tile User Menu Selection, then the Click for File Menu selection, an enters gonzalez, one can observe 36 examples given in that text book that pertain to this software. This was done to verify the integrity of this design tool.

## A Simple Impedance Match*

Design an impedance network that matches a 15 - j15 source to a $100-\mathrm{j} 50$ ohm load at 100 MHz .The matching network must also act as a low-pass filter between the source and the load.

The requirement that the matching network act as a low-pass filter forces an L-configuration with a series-L, shunt-C configuration. Since the source impedance is a complex impedance, it wants to see the load impedance as its complex conjugate. Thus, we want to force the $100-\mathrm{j} 50 \mathrm{ohm}$ load to look like an impedance of $15+\mathrm{j} 15$ ohms.

Solution:

1. Under the File tile, select New...->Smith Chart..., and enter a file name. Then change the grid to Imped+Admit reduced. Then, under the Functions menu, select Point type... Plot... and fill in the dialog box with the conjugate source impedance of $15+\mathrm{j} 15$ ohms in Cartesian coordinates. Repeat for the load impedance of $100-\mathrm{j} 50$ ohms and note the two plotted points. Label these points as $\mathrm{Z}^{*}$, and $\mathrm{Z}_{\mathrm{H}}$ using the text insert mode.
2. Under the Functions menu, select Impedance Match... Const Conductance - Load -> Source, and fill in the load impedance of 100 - j 50 ohms in Cartesian coordinates, and depress the enter key. Note the change in the cursor. Move the cursor down and note the line that is drawn along a constant conductance contour. At the same time note the component value that is being displayed in the upper left-hand corner of the monitor. When the drawn line along the constant conductance line intersects a constant resistance line that will pass through the conjugate source point, click the right most button on your mouse, and note that a solid line is drawn along with the component value of the shunt-C capacitor in the upper left corner of your monitor. You can monitor the impedance by observing the impedance on the third line in the upper right hand corner as you move your mouse. When this value is close to 15 ohms, click the right most button on your mouse. If you mess up, go to the edit tile, and click the undo tile to start over.

[^47]3. Again under the $\underline{\text { Functions menu, select Impedance Match... Const Resistance - Load -> }}$ Source. At this point a dialog box will be presented requesting that you enter the appropriate impedance point. However, the end impedance point of the series leg has been remembered, so just depress the enter key. Note that if the end impedance point is slightly off, you can make a correctly in the dialog box. Now, move the cursor up until the cursor window encloses the conjugate source impedance point, then click the right most mouse button.
At this point the design is complete. Note the Series-L inductor value as the second value listed in the upper left-hand corner. The design circuit is shown in Figure 18.2a, and .plot for the above example is shown in Figure 18.2b.

The equations used to determine the component values are as follows:
For a series-C component:

$$
C=\frac{1}{\omega X Z_{0}}
$$

For a series-L component:

$$
L=\frac{X Z_{0}}{\omega}
$$

For a shunt-C component:

$$
C=\frac{B}{\omega Z_{0}}
$$

For a shunt-L component:

$$
L=\frac{Z_{0}}{\omega B}
$$

where,

$$
\omega=2 \pi f
$$

$X=$ the reactance from the chart
$\mathrm{B}=$ the susceptance from the chart
$\mathrm{Z} 0=$ the Characteristic Impedance value


Figure 18.2b
$\mathbf{Z 0}=\mathbf{5 0 . 0} \mathbf{\Omega}$
Freq. $=100.0 \mathrm{Mhz}$
shunt- $\mathrm{C}=\mathbf{2 8 . 2 \mathrm { pF }}$ series-L=88.30nH


Figure 18.2a

## Small Transistor Design for MAG*\$

You're given a Motorola MPS571 High-Frequency Transistor. Design the input and output impedance matching network for Maximum Available Gain (MAG) using discrete components with 50 ohm Source and Load impedance's at 500 MHz with Vce equal 5 volts and Ic equal 15 mA . The S-parameters for these conditions are as follows:

$$
\begin{aligned}
& \text { S11 } 1=0.28 @-175 \mathrm{deg} . \\
& \text { S } 21=5.62 @ 79 \text { deg. } \\
& \text { S12 } 12.10 @ 65 \text { deg. } \\
& \text { S2 } 2=0.18 @-67 \text { deg. }
\end{aligned}
$$

Solution:

[^48]1. Under the File tile, select New...->Smith Chart..., and enter a file name. Then modify the frequency to 500 MHz by selecting the Frequency Dialog box under the Functions Menu. Under the Grids tile, change the grid to Imped+Admit reduced.
2. We next select the Small Transistor Calc... Manual... under the Functions menu and input the S-parameters as listed above. This calculation is necessary to check the stability of the transistor at these operating conditions. From this calculation, we observe the following info box as follows:

$$
\begin{aligned}
& \mathrm{K}=1.029 \\
& \mathrm{DEL}=0.517 \\
& \mathrm{MAG}=16.45 \mathrm{~dB} \\
& \mathrm{RefC}(\mathrm{~L})=0.657 @ 54.3 \mathrm{deg} . \\
& \mathrm{RefC}(\mathrm{~S})=0.693 @ 168.0 \mathrm{deg} . \\
& \mathrm{Gs}=2.85 \mathrm{~dB}, \mathrm{Go}=14.99 \mathrm{~dB}, \mathrm{Gl}=-1.39 \mathrm{~dB}, \mathrm{Gt}, \max =16.45 \mathrm{~dB} \\
& \mathrm{U} \quad=.03,-0.27<\text { MaxErr }) \mathrm{dB})<0.28
\end{aligned}
$$

From this info box, since $\mathrm{K}>=1.0$, and (DEL) is $<=1.0$, we know that the transistor has stability under these conditions. We note the Source and Load Reflections for MAG, and proceed with the design. Note also, the unilateral figure of merit (U) and Gt/Gtu (MaxErr - dB).
3. Under the Functions menu, select Point type... Plot... and fill in the dialog box with the source reflection coefficient of 0.693@168 degrees in Polar coordinates. Repeat for the load reflection coordinate of $0.657 @ 54.3$ degrees and notes the two plotted points. Label these points as $\backslash$ Gamma_S and $\backslash$ Gamma_L using the text insert mode as described in Chapter 2.
4. Now, under the Functions menu, select Impedance Match... Const R - Source -> Load, and fill in the source reflection coefficient of $0.693 @ 168$ deg. in Polar coordinates and depress the enter key. Note the change in the cursor. Move the cursor down and note the line that is drawn along a constant resistance contour. At the same time note the component value that is being displayed in the upper left-hand corner of the monitor. When the drawn line along the constant resistance line intersects a constant conductance line that will pass through the 50 ohms source load, click the right most button on your mouse, and note that a solid line is drawn along with the component value of the series inductor in the upper left corner of your monitor. Note that you can monitor the admittance by observing the admittance on the forth line in the upper right hand corner as you move your mouse. When this value is close to 20 mmhos, click the right most-button on your mouse. If you mess up, go to the edit tile, and click the undo tile to start over.
5. Again under the Functions menu, select Impedance Match... Const G-Source -> Load. At this point a dialog box will be presented requesting that you enter the appropriate impedance point. However, the end impedance point of the series leg has been remembered, so just depress the enter key. Now, move the cursor up until the cursor window encloses the source load of 50 ohms, then click the right most mouse button.
6. Again, under the Functions menu, select Impedance Match... Const G-Source -> Load, and fill in the load reflection coefficient of $0.657 @ 54.3$ deg. in Polar coordinates. Move the cursor down and note the line that is drawn along a constant conductance contour. At the same time note the component value that is being displayed in the upper left-hand corner of the monitor. When the drawn line along the constant conductance line intersects a constant resistance line that will pass through the 50 ohm load, click the right most button on your mouse, and note that a solid line is drawn along with the component value of the shunt inductor in the upper left corner of your monitor. Note that you can monitor the impedance by observing the impedance on the third line in the upper right hand corner as you move your mouse. When this value is close to 50 ohms, click the right most button on your mouse. If you mess up, go to the edit tile, and click the undo tile, then repeat this step.
7. Again under the Functions menu, select Impedance Match... Const R-Source -> Load. At this point a dialog box will be presented requesting that you enter the appropriate impedance point. However, the end impedance point of the shunt leg has been remembered, so just depress the enter key. Now, move the cursor up until the cursor window encloses the load of 50 ohms, then click the right most mouse button. At this point the design is complete. Note the Series-C component value as the fourth value listed in the upper left-hand corner.

The design plot is shown in Figure18.3a, and the design circuit from the above example is shown in Figure 18.3b.


Figure 18.3a


Figure 18.3b

## Small Transistor Design for Specified Gain*

You're given a Motorola LP1001A High-Frequency Transistor. Design the input and output impedance matching network for a gain of 8 dB using discrete components with 50 ohm Source and Load impedance's at 1000 MHz with Vce equal 10 volts and Ic equal 10 mA . The S-parameters for these conditions are as follows:

$$
\begin{aligned}
& \text { S11 } 11=0.05 @ 127 \mathrm{deg} . \\
& \text { S } 21=2.87 @ 58 \mathrm{deg} . \\
& \text { S } 12=0.16 @ 58 \text { deg. } \\
& \text { S22 } 20.45 @-41 \text { deg. }
\end{aligned}
$$

Solution:

1. Under the File tile, select New...->Smith Chart..., and enter a file name. Then modify the frequency to 1000 MHz by selecting the Frequency Dialog box under the Functions menu.
2. We next select the Small Transistor Calc... Manual... under the Functions menu and input the S-parameters as listed above. This calculation is necessary to check the stability of the transistor at these operating conditions. From this calculation, we observe the following info box as follows:

$$
\begin{aligned}
& \mathrm{K}=1.076 \\
& \mathrm{DEL}=0.440 \\
& \mathrm{MAG}=10.85 \mathrm{~dB} \\
& \operatorname{RefC}(\mathrm{~L})=0.683 @ 39.6 \mathrm{deg} . \\
& \operatorname{RefC}(\mathrm{S})=0.498 @-152.3 \mathrm{deg} . \\
& \mathrm{Gs}=1.24 \mathrm{~dB}, \mathrm{Go}=9.16 \mathrm{~dB}, \mathrm{Gl}=0.46 \mathrm{~dB}, \mathrm{Gt}, \max =10.85 \mathrm{~dB} \\
& \mathrm{U} \quad=0.01,-0.11<\operatorname{MaxErr}(\mathrm{dB})<0.11
\end{aligned}
$$

3. From this info box, since $\mathrm{K}>=1.0$ and (DEL) $<1.0$, , we know that the transistor has stability under these conditions. Since we want only 8 dB of gain for this design, we go to the Constant Gain... dialog under the Functions menu to plot a constant gain circle of 8 dB . In this case will select the Operating Power Gain option (Gp), and then select a Load Reflection coefficient on this

[^49]circle. Note that we also selected the corresponding locus of matched input reflection points in the opposite plane.
Once we select a Load Reflection coefficient, we can go to the Source Refl. Calc... under the Functions menu to calculate the Source Reflection coefficient or select the Load Reflection coefficient with the mouse crosshair and select the Source Reflection Calc with the right most button on the mouse. The plot for this procedure in shown in Figure 18.4a.


Figure 18.4a
5. From the above we select the Load Reflection coefficient as $.455 @-55.9$ by positioning our cursor over this point and selecting the Source Reflection coefficient calculation with the right mouse button to calculate the input reflection coefficient as .216@-62 and proceed to the impedance match to match the source and load impedance values to 50 ohms. Note that we also plotted an input VSWR circle around the input reflection point of 1.5. The impedance matching plot is shown in Figure 18.4c and the circuit for this design is shown in Figure 18.4c.


Figure 18.4c


Figure 18.4c

## Small Transistor Design for Minimum Noise*

You're given a Motorola MRF571 High-Frequency Transistor. Design the input and output impedance matching network for a front end amplifier using discrete components with $\mathrm{Zs}=35-\mathrm{j} 60$ ohms and $\mathrm{Zl}=$ $50-\mathrm{j} 50$ ohms at 1000 MHz with Vce equal 6 volts and Ic equal 5 mA . The gain is specified at 10 dB with a minimum noise figure of 2 dB . The noise parameters for these conditions are $\mathrm{Nf}=1.5 \mathrm{~dB}$, equivalent noise resistance $(\mathrm{Rn})=7.5 \mathrm{ohms}$, and $\Gamma_{\text {OPT }}=0.48 @ 134$. The S-parameters for these conditions are as follows:

$$
\begin{aligned}
& \mathrm{S} 11=0.61 @ 178 \mathrm{deg} . \\
& \mathrm{S} 21=3.0 @ 78 \mathrm{deg} . \\
& \mathrm{S} 12=0.09 @ 37 \mathrm{deg} . \\
& \mathrm{S} 22=0.28 @-69 \mathrm{deg} .
\end{aligned}
$$

Solution:

1. Under the File tile, select New...->Smith Chart..., and enter a file name. We then modify the frequency to 1000 MHz by selecting the Frequency Dialog box under the Functions menu.
2. We next select the Small Transistor Calc... under the Functions menu and input the Sparameters as listed above. From this calculation, we observe the following info box as follows:

$$
\begin{aligned}
& \mathrm{K}=1.037 \\
& \mathrm{DEL}=0.102 \\
& \mathrm{MAG}=14.05 \mathrm{~dB} \\
& \operatorname{RefC}(\mathrm{~L})=0.806 @ 66.1 \mathrm{deg} . \\
& \operatorname{RefC}(\mathrm{S})=0.891 @-178.7 \mathrm{deg} . \\
& \mathrm{Gs}=6.85 \mathrm{~dB} \\
& \mathrm{Go}=9.54 \mathrm{~dB} \\
& \mathrm{Gl}=-2.24 \mathrm{~dB} \\
& \mathrm{Gt}, \max =14.05 \mathrm{~dB} \\
& \mathrm{U} \quad=-10.99 \mathrm{~dB} \\
& -0.67<\operatorname{MaxErr}(\mathrm{dB})<0.72
\end{aligned}
$$

3. From this info box, since $\mathrm{K}>=1.0$ and (DEL) $<=1.0$, we know that the transistor has stability under these conditions.
4. Since we want the minimum noise figure to be less then 2 dB , we process to the Noise Circle... dialog under the Functions menu to plot some noise circles.
5. We next proceed to the Constant Gain... dialog menu under the Functions menu in order to plot a constant gain circle of 10 dB . In this case we opted to plot the Gain Available circles (Ga). From this, an input Reflection Coefficient of 0.235@109.3 was selected.
6. The Load Reflection coefficient of $0.304 @ 81$ is calculated by using the Load Refl. Calc... under the Functions menu. The plot for this procedure is shown in Figure 18.5a.

[^50]

Figure 18.5a
7. Using the above reflection coefficients, the impedance matching plot is shown in Figure 18.5b and the circuit is shown in Figure 18.5c


Figure 18.5b


Figure 18.5c

## Small Transistor Design - Stability*\$

You're given a Motorola MRF5711LT1 High-Frequency Transistor. Design the input and output impedance matching network using discrete components with $\mathrm{Zs}=35+\mathrm{j} 50$ ohms and $\mathrm{Zl}=50-\mathrm{j} 50$ ohms at 500 MHz with Vce equal 6 volts and Ic equal 10 mA . The gain is specified at 16 dB with a minimum noise figure of 1.6 dB . The noise parameters for these conditions are $\mathrm{Nf}=1.2 \mathrm{~dB}$, equivalent noise resistance $(\mathrm{Rn})=7 \mathrm{ohms}$, and $\Gamma_{\text {OPT }}=0.36 @ 104$. The S-parameters for these conditions are as follows:

$$
\begin{aligned}
& \text { S11 } 11=0.69 @-160 \text { deg. } \\
& \text { S } 21=6.9 @ 92 \text { deg. } \\
& \text { S12 }=0.06 @ 34 \text { deg. } \\
& \text { S } 22=0.30 @-92 \text { deg. }
\end{aligned}
$$

Solution:

1. Under the File tile, select New...->Smith Chart..., and enter a file name. We then modify the frequency to 500 MHz by selecting the Frequency Dialog box under the Functions menu.
2. We next select the Small Transistor Calc... under the Functions menu and input the Sparameters as listed above. From this calculation, we observe the following info box as follows:

$$
\begin{aligned}
& \mathrm{K}=0.59 \\
& \text { Proceed to Stability } \\
& \text { Circle calculation }
\end{aligned}
$$

3. We next plot the stability circles by selecting the Stability Circles... dialog under the Functions menu. The S-parameters will be remembered from the Small Transistor Calc... dialog procedure, so just depress the enter key.
4. We then proceed to the Constant Gain... dialog procedure under the Functions menu in order to plot the constant gain circle of 16 dB . In this case we opted to plot the Gain Available circles (Ga). From this an input Reflection Coefficient of $0.167 @-12.8$ is selected.
5. The Load Reflection coefficient of $0.245 @ 98$ is then calculated by using the Load Refl. Calc... dialog procedure under the Functions menu. The plot for this procedure is shown in Figure 18.5a.

[^51]

Figure 18.6a
6. Using the above reflection coefficients, the impedance matching plot is shown in Figure 18.6b and the circuit is shown in Figure 18.6c.


Figure 18.6c


Figure 18.6b

## Micro-strip Matching - A Simple Example\$

The amplifier block diagram shown in Figure18.7a shows the input and output reflection coefficients $\mathrm{G}_{\mathrm{S}}$ and $\mathrm{G}_{\mathrm{L}}$. Design the input and output matching networks in a 50 ohm system with a frequency of 2.4 GHz . The substrate material is Duriod.

$$
\varepsilon_{r}=2.23, h=0.75 \mathrm{~mm}
$$



Figure 18.7a

1. We start with an Impedance grid with Freq. $=2400 \mathrm{MHz}$.
2. Starting with the Input Matching Network, we first plot $\Gamma_{\mathrm{S}}$ by plotting this point using the Point type... Plot... dialog procedure under the Functions menu.
3. We next plot Ys by using the Point type... 1/Imped. point dialog menu under the Functions menu. Refer to reference [2] as to why using the Y Smith is convenient when designing microstrip impedance matching networks.
4. We next plot the VSWR circle by using the Point type... VSWR Circle... dialog procedure under the Functions menu.
5. Label these points using the text annotation feature.
6. Now, under the Functions menu, select Micro-strip Match... Stub Open Circuit... , and input a value of 50 and also click the Direction radio button to select Towards Generator and depress the enter key. Note the line drawn along a constant resistance line as you push the cursor up and to the right. Click the right most mouse button when the small cursor intersects the VSWR circle, and note the open-Stub $=0.172$ statement displayed in the upper left hand corner.
7. Next select the Micro-strip Match... Series Line... , and click the Direction radio button to select Towards Generator and just depress the enter key since the endpoint of the last match is remembered and is the starting point of the next match. Now pull the mouse down and note the line that is drawn along the constant VSWR circle. Click the right most mouse button when the small cursor is over the
Admittance point $\mathrm{Y}_{\mathrm{S}}$. Note the series-Line $=0.111$ as displayed in the upper left of your monitor. At this point, the input network has been designed except for the calculation of the actual micro-strip dimensions. The design plot for the Input Matching Network is shown in Figure 18.7b.

[^52]

Figure 18.7b
8. Repeating the above procedure for the Output Matching Network except we use a short circuit stub, we get a short-Stub $=0.073$, and a series-Line $=0.067$ for the micro-strip parameters. The design plot for the Output Matching Network is shown in Figure 18.7c.
9. Using the Micro-strip Calc... dialog under the Functions menu, the line lengths including a quarter wave transformer (/4) to supply the $\mathrm{V}_{\mathrm{BB}}$ input voltage are as follows:

$$
\begin{aligned}
& 0.173=1.571 \mathrm{~mm} \\
& 0.111=1.008 \mathrm{~mm} \\
& 0.073=0.663 \mathrm{~mm} \\
& 0.067=0.608 \mathrm{~mm} \\
& 0.250=2.270 \mathrm{~mm}
\end{aligned}
$$

The circuit diagram including bypass and coupling capacitors for the above example in shown in Figure18.7d. The above design could have used balanced shunt stubs which minimizes the transition between the shunt stubs and the series transmission line. In this case, each side of the balanced stub must equal half of the total admittance. For the input matching network, the open circuit balanced stubs are 0.121 , and for the output network, the short circuit balanced stubs are 0.123 (at 50 ohms ).


Figure 18.7c


Figure 18.7d

## Micro-strip Matching - Different Characteristic Impedance's

We repeat the input network design of the last example, except will use micro-strip lines with different characteristic impedance's.

## Solution 1:

1. Repeat the first 5 steps in the previous example.
2. Move your cursor over the $Z_{S}$ plotted point, and read off the value of $Y_{S}=33-j 46.8$ milli-mhos in the upper right hand corner of your display. Lets use a quarter wave transformer to transform the source impedance of 50 to a resistance of $1 / 0.033=30.2$ ohms. The resistance of this micro-strip line is given by $Z_{0}=\sqrt{50(30.2)}=38.8 \Omega$
3. Under the Functions menu, select Micro-strip Matching using a Quarter Wave Transformer - Y-Chart. Move the cursor right until the resistance value in the upper left hand corner reads $\sim 38.8$ ohms, then right click on your mouse.
4. We can now use an open circuit shunt stub of 0.375 or a short circuit stub of 0.125 to transform the susceptance of -j46.79 milli-mhos. The resistance of this line is $1 / 0.04679=21.372$.
5. Select the Micro-strip Match... Stub Open Circuit... dialog procedure, and change the resistance value of the strip to 21.372 ohms. The direction is Towards -> Generator (Source) and the from point is the admittance point of the 30.13 impedance point as remembered from step 4. Move the cursor down until the cursor is over $\mathrm{Y}_{\mathrm{S}}$ and note the open-Stub= 0.375 statement displayed in the upper left-hand corner of the monitor (as expected). The design plot for the Input Matching Network is shown in Figure 18.7e, and the input network circuit for the above design is shown in Figure18.7f.


Figure 18.7e


Figure 18.7f

Solution 2:

1. Plot the $\mathbf{Z}_{\mathrm{S}}$ point using the Point type... Plot... dialog as found under the Functions menu.
2. Under the Functions menu, select Point type... VSWR, to plot a VSWR circle through $\mathbf{Z}_{\mathrm{s}}$.
3. Now select Micro-strip Matching using a Quarter Wave Transformer - Z Chart, and move the cursor left until the cursor intersects the VSWR circle, then right click on your mouse.
4. Select the Micro-strip Match... Series Line... from under the Functions menu with direction Towards ->Generator, and move the cursor up until the cursor is over the $\mathbf{Z}_{\mathbf{S}}$ point, then right click on your mouse. The length of this line is 0.046 and the resistance at this point is 9.28 ohms

The design plot for the Input Matching Network is shown in Figure 18.7 g , and the input network circuit for the above design is shown in Figure18.7h

```
Z0=50.0 \Omega
Freq. = 1000.0 Mhz
\lambda/4, Z = =21.6 \Omega
series-Line=0.046\lambda, Z }\mp@subsup{\textrm{Z}}{0}{=50\Omega
```



Figure 18.7g


Figure 18.7h

## A Single Stub Tuner*

You're given a 50 ohm transmission line with a load of $125-\mathrm{j} 150$ ohms at a frequency of 100 MHz . The transmission line has an effective relative dielectric constant $\varepsilon_{f f}$ of 2.56 . Determine how far back from the load (d) you need to go to insert an open circuit stub of the same transmission medium and the length (l) of the stub such that you create a match to the 50 ohm line.

## Solution:

1. The first step is to plot the impedance point of the load by using the Point type... Plot... dialog menu under the Functions menu.
2. We next plot the conjugate match circle with $\lambda=0$, and then plot the admittance point of the load by using the Point type... 1/Imped... dialog selection under the Functions menu.
3. Next, select Micro-strip strip... Match Series Line... under the Functions menu, and depress the enter key. Move the small cursor up along the VSWR circle towards the conjugate match circle. When the small cursor intersects this point, click the right mouse button to determine the fraction of wave length traveled.
4. Next select the Micro-strip Match... Open Circuit Stub... under the Functions menu, and depress the enter key. Move the cursor down and to the left until the small cursor intersects the center point on the chart. Then click the right most-button on your mouse to determine the fraction of wave length traveled for an open circuit stub. From this we have the series-line as 0.158 , and the open circuit line as 0.320 . The equation for the actual distances traveled is

$$
d=x \lambda=\frac{x v_{p}}{f}=\frac{x c}{f \sqrt{\varepsilon_{f f}}}
$$

[^53]From the above equation,

$$
\begin{aligned}
& d=\frac{0.158\left(3 \times 10^{10} \mathrm{~cm}\right)}{10^{8} \sqrt{2.56}}=29.625 \mathrm{~cm} \\
& l=\frac{0.320\left(3 \times 10^{10} \mathrm{~cm}\right)}{10^{8} \sqrt{2.56}}=60 \mathrm{~cm}
\end{aligned}
$$

The plot of the above solution is shown is Figure18.8.

```
Z0}=50.0
Freq. = 100.0 Mhz
series-Line=0.158\lambda, Z}\mp@subsup{\textrm{Z}}{0}{}=50
open-Stub=0.320\lambda, Z
```



Figure 18.8

## A Double Stub Tuner*

We repeat the Single Stub Tuner example as given above, but add the requirement that two short open circuit stubs be used, with the first one a distance of 0.30 from the load, while the second stub is $3 / 8$ from the first. All other conditions remain the same. Find the distances d 1 and d 2 that each stub is from the load, and the length of each stub (11 and 12).

[^54]1. The first step is to plot the impedance point and admittance point of the load. First select the Point type... Plot... dialog from under the Functions menu, and enter the impedance value of 125 - j150 ohms.
2. Repeat the above, but select the $\mathbf{1}$ /Imped... selection to plot the admittance point.
3. Next, select Micro-strip Match... Series Line... under the Functions menu, and depress the enter key. Move the small cursor to the right along the VSWR circle until the read out in the upper left hand corner reads series-Line $=0.300$. This is shown as point 1 in Figure18.9a.
4. Next, select the Conj Match Circle... from the Functions menu and enter a value of zero to plot the $1+j$ conjugate match circle.
5. Repeat the above, but enter 0.375 to plot the $1+j$ conjugate match circle as rotated towards the generator by $3 / 8$.
6. Next, select Micro-strip Match... Open Circuit Stub... under the Functions menu, and depress the enter key. Push the small cursor down until the small cursor intersects the rotated $1+\mathrm{j}$ conjugate match circle, then click the right most button on your mouse. The read out in the upper left of your monitor should read open-Stub $=0.157$.
7. Again, select Micro-strip Match... Series Line... under the Functions menu, and depress the enter key. Push the small cursor around (clockwise) until the small cursor intersects the $1+j$ conjugate match circle, then click the right most button on your mouse. The read out in the upper left of your monitor should read series-Line $=0.375$.
8. Now, select the Micro-strip Match... Open Circuit Stub... under the Functions menu, and depress the enter key. Move the cursor down and to the right until the small cursor intersects the impedance value of 50 ohms as displayed in the upper right hand corner of your monitor. Then click the right most button on your mouse to determine the fraction of wave length traveled for an open circuit stub. From this we have the open circuit stub as 0.077 . The equation for the actual distances traveled is

$$
d=x \lambda=\frac{x v_{p}}{f}=\frac{x c}{f \sqrt{\varepsilon_{f f}}}
$$

From the above,

$$
\begin{aligned}
& d_{1}=\frac{0.300\left(3 \times 10^{10} \mathrm{~cm}\right)}{10^{8} \sqrt{2.56}}=56.25 \mathrm{~cm} \\
& l_{1}=\frac{0.158\left(3 \times 10^{10} \mathrm{~cm}\right)}{10^{8} \sqrt{2.56}}=29.6255 \mathrm{~cm} \\
& d_{2}=d_{1}+\frac{0.375\left(3 \times 10^{10} \mathrm{~cm}\right)}{10^{8} \sqrt{2.56}}=126.56 \mathrm{~cm} \\
& l_{2}=\frac{0.076\left(3 \times 10^{10} \mathrm{~cm}\right)}{10^{8} \sqrt{2.56}}=14.25 \mathrm{~cm}
\end{aligned}
$$

$\mathbf{Z 0}=\mathbf{5 0 . 0} \boldsymbol{\Omega}$
Freq. $=100.0 \mathrm{Mhz}$
series-Line $=0.300 \lambda, \mathrm{Z}_{0}=50 \Omega$ open-Stub $=0.158 \lambda, \mathrm{Z}_{0}=50 \Omega$ series-Line $=0.375 \lambda, \mathrm{Z}_{0}=50 \Omega$ open-Stub $=0.076 \lambda, Z_{0}=50 \Omega$


Figure 18.9a

## Plotting of Performance Data

Were given a Motorola MPS571 NPN Silicon High-Frequency Transistor. Plot the S-Parameter data for $\mathrm{V}_{\mathrm{CE}}=5$ Volts and $\mathrm{I}_{\mathrm{C}}=5 \mathrm{~mA}$ as given under directory MOTOROLA\MPS【MPS571a.S2P.

1. We first start with an Impedance only grid with a characteristic impedance of 50 ohms and at any frequency to plot the $S_{11}$ and $S_{22}$ data.
2. We list this file by selecting under the Functions menu, Parametric Data... List... , and select the file to open under the path MOTOROLA\MPS\MPS571a.S2P. The format of this data will look as follows. Refer to Parametric Data in chapter 2 of this manual for a description.

| 3 | MPS571A.S2P |  |  |  |  |  |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $!$ | MPS571 |  |  |  |  |  |  |  |
| $!$ | VCE=5V; | IC=5mA |  |  |  |  |  |  |
| \# | GHZ S MA R | 50 |  |  |  |  |  |  |
| $!$ | S-PARAMETER |  |  |  |  |  |  |  |
| 0.2 | 0.62 | -80 | 8.22 | 122 | 0.07 | 56 | 0.63 | -44 |
| 0.5 | 0.40 | -148 | 4.52 | 87 | 0.11 | 50 | 0.36 | -58 |
| 1 | 0.39 | 155 |  | 2.51 | 54 | 0.16 | 48 | 0.23 |
| 1.5 | 0.46 | 122 | 1.86 | 32 | 0.23 | 42 | 0.15 | -114 |
| 2 | 0.59 | 100 | 1.50 | 14 | 0.31 | 33 | 0.14 | 173 |

3. Under the Functions menu, select Parametric Data... Plot... , and select the file to open. When the next dialog box appears, select S11 and S22, then reflection data.
4. Open up a new plotting window by selecting File $\underline{\text { New... under the File menu selection. Repeat }}$ the above for S21 and S12, and reflection data.

The plots of the above can be found in figures 3.10a \& 3.10b. The annotation was adding interactively by using the text annotation function. Note! Plot types of S21 and S12 are of polar type and can't be mixed with S11 and S22, which are plotted on a Reduced Smith Chart overlay. By using a network analyzer and recording S-parameter data as a file in Touchstone format, one can make similar plots as shown below. Note that these plots were created, then a metafile was generated as described in Chapter 2, and inserted into this manual by using the insert picture function in Microsoft Word for Windows.


Figure 18.10a


Figure 18.10b

## Handy notes for Impedance matching

In the process of performing an impedance match, we need a few rules to keep in mind so that we come up with the right answer

1. For a simple impedance match as illustrated in the first example of this chapter, take the conjugate of the source impedance, and then either:

- Start at the Conjugate Source Impedance and proceed to the Load such that the direction is Source->Load.
- Start at the Load Impedance and proceed to the Conjugate Source Impedance such that the direction is Load->Source.

2. For impedance matching the input or output of a transistor, the reflection coefficients $\Gamma_{\mathrm{S}}$ and $\Gamma_{\mathrm{L}}$ are considered the Source points.

- For component values, if you start at these impedance values, then the direction is Source$>$ Load, else if you start with the Load point and move to the Source point $\left(\Gamma_{\mathrm{S}}\right.$ or $\left.\Gamma_{\mathrm{L}}\right)$, then the direction is Load->Source.
- For micro-strip matching using shunt (series) stubs, first plot the impedance point of $\Gamma_{\mathrm{S}}$ or $\Gamma_{\mathrm{L}}$, then (for shunt stubs only) its corresponding admittance point. Then the direction (for stubs only) is Towards->Load from $\Gamma_{\mathrm{S}}$ or $\Gamma_{\mathrm{L}}$, else Towards->Generator from the load. For series line, take the shortest path, where Towards->Generator is clockwise and Towards$>$ Load is counter-clockwise.

3. For micro-strip matching a load to a transmission line using shunt (series) stubs, first plot the impedance point of the Load, then (for shunt stubs only) its corresponding admittance point. Then in all cases (shunt stubs, series stubs, or series line):

- Start at the admittance (impedance) point of the Load and move to the impedance of the line where the direction is Towards Generator.
- Start at the impedance of the line and move towards the admittance (impedance) point of the Load where the direction is Towards Load.


## References

[1] G. Gonzalez, Microwave Transistor Amplifiers Analysis and Design, Prentice-Hall, Second Edition, 1997.
[2] Chris Bowick, RF Circuit Design, Sams, 1982.
[3] David M. Pozar, Microwave Engineering, Addison-Wesley, 1993.
[4] Phillip H. Smith, ELECTRONIC APPLICATIONS OF THE SMITH CHART in Waveguide, Circuit, and Component Analysis, Robert E. Krieger Publishing Company, 1983.
[5] Hewlett Packard, S-Parameter Design, Application Note 154, April 1972.
[6] Hewlett Packard, S-Parameter Techniques for faster More Accurate Network Design, Application Note 95-1

## Appendix A-Functions Reference List

The following is a list of the keywords used in the grammar and imbedded control along with the internal and shell script functions that the user can call (as subroutines). As stated previously, all of the internal function names (case sensitive) are reserved and may not be used by the user for variable naming. Please note that for Matrix/Vector types, that many of the mathematical functions as well as the assignment statements handle these types (refer to Chapter 4, Matrix Vector Grammar). A description of the following functions may be found under the Help Menu/Functions Reference.... Note that for groups; Special Matrices, the Signal Processing Toolbox, the Control Engineering Toolbox, the Adaptive Filters Toolbox, the Statistical Signal Processing Toolbox, and the Forward Error Correction (FEC) Toolbox, that these are shell script files. When using routines in these groups, the user must insert the following statements at the top of their shell script code. Note, that if all 4 of these include directives are included in a given shell script, that the code will be compiled in approximately 125 milliseconds for $\sim 8500$ lines of code, and will be stored in block 3 as shown in figure 3.2 of Chapter 3 until the given shell script finishes.

For Special Matrices:
For the Signal Processing Toolbox:
For the Control Engineering Toolbox:
For the Adaptive Toolbox:
For the Statistical Signal Processing Toolbox:
For the Forward Error Correction Toolbox:

```
#include "matrices.h"
#include "sigsys.h"
#include "control.h"
#include "sigsys.h"
#include "sigsys.h"
#include "commsys.h"
```

|  | Grammar and Imbedded Control |
| :--- | :--- |
| \#define | Define constant for conditional \#ifdef |
| \#else | Else for conditional \#ifdef/\#ifndef |
| \#endif | End statement for \#ifdef or \#ifndef |
| \#ifdef | Conditional if (True) for compile |
| \#ifndef | Conditional if (False) for compile |
| break | Break out of loop( for, do, while, switch) |
| case | Part of switch statement |
| char | Data type(8 bits) for ASCII strings definition |
| colSize | Returns column size |
| complex | Real and Imag. data type (64-bit floats) |
| const | Definition of read only constants |
| continue | Continue with a loop (for, do, while) |
| default | Part of switch statement |
| dialog | Generic Dialog Procedure |
| do while | The do while statement |
| double | Real data type (64-bit float) |
| echof | Same as printf, but to display monitor also |
| else | False part of if statement |
| exit | Terminate shell with message |
| float | Real data type (64-bit float) |
| for | First part of for loop |
| if | Conditional statement |
| int | Fixed data type (32-bit, signed) |
| long | Fixed data type (32-bit, signed) |


| matcmplx | Matrix of complex data types |
| :--- | :--- |
| matrix | Matrix of double data types |
| pause | Output message to display monitor and pause |
| printf | printf statement (as standard in C) |
| Print | Print of Matrix/Vector (Real/Complex arrays) |
| return | Return from user subroutine |
| rowSize | Returns row size |
| Sizeof | Returns size of array in type elements |
| sprintf | sprintf statement (as standard in C) |
| switch | First part of complex multiple if/else |
| veccmplx | Vector of complex data types |
| vector | Vector of double data types |
| void | The void return type |
| while | The while statement |
|  |  |

## Mathematical Functions

| abs | Absolute value |
| :---: | :---: |
| acos | Inverse cosine |
| acosh | Inverse hyperbolic cosine |
| arg | Angle of |
| asin | Inverse sine |
| asinh | Inverse hyperbolic sine |
| atan | Inverse tangent |
| atanh | Inverse hyperbolic tangent |
| $\operatorname{atan} 2$ | Inverse tangent y , x |
| ceil | Rounds up to nearest integer |
| cmplx | Complex number of |
| conj | Complex conjugate |
| cos | Cosine |
| cosh | Hyperbolic cosine |
| $\exp$ | Exponential |
| fabs | Absolute value (for real argument) |
| floor | Rounds down to nearest integer |
| fmod | Modulo function |
| imag | Imaginary part of complex number |
| In | Natural logarithm |
| $\log$ | Natural logarithm |
| $\log 10$ | Logarithm base 10 |
| $\log 2$ | Logarithm base 2 |
| max | Maximum function |
| min | Minimum function |
| norm | Square of |
| polar | Complex of magnitude and angle |
| pow | Power $x$ of y |
| real | Real part of complex num. |
| round | Round to nearest integer |
| sigNum | The Signum function |
| sin | Sine |


| $\boldsymbol{\operatorname { s i n h }}$ | Hyperbolic sine |
| :--- | :--- |
| sqrt | Square root |
| $\boldsymbol{\operatorname { t a n }}$ | Tangent |
| $\boldsymbol{\operatorname { t a n h }}$ | Hyperbolic tangent |


|  | Special Mathematical Functions |
| :--- | :--- |
| besselIn | Bessel function $\operatorname{In}(\mathrm{x}), \mathrm{n}=0,1,2 \ldots$ |
| besselJn | Bessel function $\mathrm{Jn}(\mathrm{x}), \mathrm{n}=0,1,2, \ldots$ |
| besselKn | Bessel function $\mathrm{Kn}(\mathrm{x}), \mathrm{n}=0,1,2, \ldots$ |
| besselYn | Bessel function $\mathrm{Yn}(\mathrm{x}), \mathrm{n}=0,1,2, \ldots$ |
| beta | Beta function $\mathrm{B}(\mathrm{z}, \mathrm{w})$ |
| betaI | Incomplete beta function $\mathrm{Ix}(\mathrm{a}, \mathrm{b})$ |
| cosint | Cosine Integral function |
| ellipticC | Elliptic Integral of 1st kind |
| ellipticJ | Jacobian Elliptic function |
| ellipticS | Elliptic Integral of 2nd kind |
| erf | Error function erf(x) |
| erfc | Complementary of error function erf(x) |
| expi | Exponental Integral Ei(x) |
| expn | Exponental Integral En(x) |
| factorial | Factorial function |
| gamma | Gamma function |
| gammaln | Natural log of gamma(x) |
| gammaP | Incomplete gamma function $\mathrm{P}(\mathrm{a}, \mathrm{x})$ |
| gammaQ | Incomplete gamma function $\mathrm{Q}(\mathrm{a}, \mathrm{x})$ |
| sinint | Sine Integral function |


|  | Random Numbers and Distributions |
| :--- | :--- |
| chisqrPDF | Chi-squared distribution (Central) |
| chisqrProb | Chi-squared probabilities (Central) |
| chisqr2PDF | Chi-squared distribution (Non-central) |
| chisqr2Prob | Chi-squared probabilities (Non-central) |
| F PDF | F Distribution (Central) |
| F_Prob | F probabilities (Central) |
| normal | Gaussian noise generation (vector/matrix) |
| normalPDF | Normal (Gaussian) Distribution |
| normalProb | Probabilities from a Gaussian PDF |
| rand | Uniform Random number generators |
| randn | Gaussian number |
| rayleighPDF | Rayleigh Distribution of a random variable |
| rayleighProb | Rayleigh probabilities |
| ricianPDF | Rician Distribution of a random variable |
| ricianProb | Rician probabilities |
| srand | Initialize randf with seed |
| uniform | White noise generation (vector/matrix) |


| Matrix/Vector Operations |  |
| :---: | :---: |
| cod | Complete Orthogonal Decomposition |
| chol | Cholesky factorization |
| det | Matrix Determinant |
| eig | Matrix Eigenvalues/Eigenvectors |
| $\exp \mathrm{M}^{\prime}$ | Matrix Exponential |
| hess | Hessenburg Decomposition |
| inv | Matrix Inverse |
| kron | Calculates the Kronecker tensor product |
| kurt | Calculates kurtosis of vector |
| linequ | Matrix linear equation solver |
| linequOD | Matrix equation solver (over determined) |
| Isq | Least squares fit |
| lu | Matrix LU factorization |
| , | The matrix transpose operator |
| .$^{\wedge}$ | The matrix/vector power operator |
| * | The Matrix Product operator |
| norms | Matrix/Vector norms |
| pinv | Moore-Penrose pseudoinverse of a matrix |
| qr | QR Decomposition |
| rank | Rank of a Matrix |
| rcond | Matrix reciprocal condition number |
| rref | Reduced row echelon form |
| schur | Schur Matrix Decomposition |
| skew | Calculates skewness of vector |
| svd | Singular value Decomposition |
| toeplitz | Generates a Toeplitz matrix |


|  | Matrix/Vector Auxiliary Functions |
| :--- | :--- |
| 'casting' | Convert matrix/vector to vector/matrix |
| cmplx | Create complex matrix or complex vector |
| colLen | Column size of a matrix (length) |
| colSize | Column dimension of a matrix |
| convToA | Matrix/Vector elements to array elements |
| copyV | Copy a portion of a vector to a new vector |
| diag | Diagonal of Matrix and Diagonal Matrices |
| dnSamp | Down sample a matrix or vector |
| expand | Matrix/Vector expansion |
| even | Even function |
| eye | Square matrix to singular type |
| flip | Reverse elements function |
| matGen | Generate Matrix Sequence |
| matLin | Generate Matrix Sequence |
| odd | Odd function |
| ones | Generate 1's matrix or vector |


| Print | Matrix/Vector print |
| :--- | :--- |
| reduce | Matrix reduction |
| reshape | Matrix shaping |
| resizeM | Matrix resize |
| resizeV | Vector resize |
| rotM | Matrix rotation |
| rowLen | Row size of a matrix (length) |
| rowSize | Row dimension of a matrix |
| sigNum | The Signum function for real matrices/vectors |
| sort | Sort a vector in ascending order |
| sortRows | Sort matrix rows on column specification |
| trace | Sum of diagonal elements of a matrix |
| triL | Lower triangular part of matrix |
| triU | Upper triangular part of matrix |
| upSamp | Up sample a matrix or vector |
| vecGen | Generate Vector Sequence |
| vecLin | Generate Vector Sequence |
| vecLog | Generate Vector Logarithmic Sequence |
| zero | Zeros elements of matrix or vector |
| zeros | Generate zero matrix or vector |
| zeros3D | Generate 3D zero matrix or vector |


| Special Matrices |  |
| :--- | :--- |
| compan | Companion matrix |
| gallery | Wilson/Rosser matrices |
| hadamard | Hadamard matrix |
| hankel | Hankel matrix |
| hilb | Hilbert matrix |
| magic | Magic matrix |
| pascal | Pascal matrix |
| toeplitz | Toeplitz matrix |
| vander | Vandermonde matrix |
| wilkinson | Wilkinson test matrix |


|  | Digital Filters |
| :--- | :--- |
| butterF | Butterworth Filter Design |
| chebyIF | Chebyshev I Filter Design |
| chebyIIF | Chebyshev II Filter Design |
| elliptF | Elliptic Filter Design |
| filtadj | Filter section gain adjustment |
| filter | Vector filter function in direct form |
| filterD | General filter function |
| firpm | Parks McCellan FIR filter Design with vectors |
| firwin | FIR Windowed Filter Design with vectors |
| firwindF | FIR Windowed Filter Design |
| gainF | Complex gain over range |


| groupD | Group Delay over range |
| :--- | :--- |
| hogSym | Hogenauer Filter Design |
| iifixedR | IIR Fixed Point Coefficients Response |
| impulseF | Impulse Response |
| interpFIR | Interpolated multi-FIR Filter design |
| MaxFlat | MaxFlat FIR Symmetric Filter design |
| meteorFIR | FIR constraint based design |
| ParksMc | Parks McCellan FIR Filter Design |
| phaseF | Complex phase over range |
| SavGov | Savitzky-Golay Smoothing Filter |
| stepF | Step Response |
| unwrap | Phase unwrap |


| Spectral Analysis |  |
| :--- | :--- |
| blackharrisW | 4-term Blackman harris window |
| blackWin | Blackman window |
| chebyWin | Dolph-Chebyshev window |
| chirpZ | Chirp Z transform |
| dct | Discrete Cosine Transform (DCT) |
| dct2 | Two-dimensional DCT |
| dft | Discrete Fourier Transform (DFT) |
| fft | Fast Fourier Transform(FFT) |
| fft2 | Two-dimensional FFT |
| fftshift | Shift zero freq. of FFT to center of spectrum |
| gaussianWin | Gaussian window |
| hammWin | Hamming window |
| hannWin | Hanning window |
| hilbert | Hilbert transform |
| idct | Inverse Discrete Cosine Transform (DCT) |
| idct2 | Inverse two-dimensional DCT |
| idft | Inverse Discrete Fourier Transform (IDFT) |
| ifft | Inverse FFT |
| ifft2 | Inverse of two-dimensional FFT |
| kaiserWin | Kaiser window |
| taprectWin | Tappered rectangular window |
| triangWin | Triangular window |


| Correlation/Convolution |  |
| :--- | :--- |
| autocor | The Auto Correlation function |
| autocorMat | Generates the Autocorrelation matrix |
| autocovar | The Auto Covariance function |
| autocovarMat | Generates the Auto covariance matrix |
| conv | The Convolve function |
| convMat | Computes the convolution matrix |
| covarMat | Computes the covariance matrix |
| corCoef | Calculates the correlation coefficient |


| crosscor | The Cross Correlation function |
| :--- | :--- |
| crosscovar | The Cross Covariance function |


|  | Signal Processing Toolbox |
| :--- | :--- |
| bilinear | Bilinear transform |
| butterAP | Butterworth analog prototype |
| butterDF | Butterworth digital filter design |
| cas2dir | Second-order cascade to direct form |
| casfilter | Second-order cascade filter |
| chebyIAP | Chebyshev I analog prototype |
| chebyIDF | Chebyshev I digital filter design |
| cheby2AP | Chebyshev II analog prototype |
| Cheby2DF | Chebyshev II digital filter design |
| circonv | Circular convolution |
| cirFold | Circular folding of a sequence |
| cirshift | Circular shift of samples in time domain |
| dfs | Compute discrete Fourier Series Coefficients |
| dir2cas | Direct form of filter to second-order sections |
| dir2fs | Direct form of filter to frequency form |
| dir2par | Direct form of filter to parallel 2nd order |
| ellipticAP | Elliptic analog prototype |
| ellipticDF | Elliptic digital filter design |
| evenodd | Signal decomposition to even/odd parts |
| freqZ | Frequency response of digital filters |
| FIR2Lat | Direct form of FIR filter to all-zero Lattice form |
| FIRIIR2Lat | Direct form of all pole/zero IIR filter to Lattice/Latter |
|  | form |
| groupDelay | Group delay calculation |
| IIR2Lat | Direct form of IIR filter to lattice IIR form |
| Lat2FIR | All-zero Lattice form to FIR direct form conversion |
| Lat2IIR | Lattice IIR form to IIR direct form conversion |
| Lat2FIRIIR | Lattice/Ladder form to pole/zero direct form |
|  | conversion |
| LatFiltIIR | Lattice/Ladder module to filter input sequence |
| LatFilt | All-zero Lattice filter |
| ovrlpsav | Overlap-save method of block convolution |
| par2dir | Parallel 2nd order form to direct filter form |
| residueZ | Z-transform partial fraction expansion |
| sigadd | Signal addition |
| sigmult | Signal multiplication |
| sigshift | Signal shift |
| sinc | Sinc signal generation |
| STrans | Prototype frequency band transformation |
| Smapping | Analog frequency band transformation |
| stepseq | Step sequence generation |
| Zmapping | Digital frequency band transformation |
|  |  |


| Adaptive Filters |  |
| :---: | :---: |
| apa | Affine Projection Algorithm. for an adaptive FIR filter |
| cma1 | Constant Modulus Algorithm (CMA1-2) for an adaptive FIR filter |
| cma2 | Constant Modulus Algorithm (CMA2-2) for an adaptive FIR filter |
| fastAlg | The Fast Array algorithm for an adaptive FIR filter |
| fbnlms | The fast Block (FFT) NLMS algorithm for an adaptive FIR filter |
| Ims | The LMS Algorithm for an adaptive FIR filter |
| nlms | The Normalized LMS Algorithm for an adaptive FIR filter |
| pnlms | The NLMS Alg. with power normalization |
| rea | The Reduced-Constellation Algorithm (RCA) for an adaptive FIR filter |
| rls | The Exponentially Weighted RLS Algorithm for an adaptive FIR filter |
| selms | The Sign-Error LMS Algorithm for an adaptive FIR filter |
| steepestD | The Steepest Descent Algorithm for an adaptive FIR filter |
| wienerF | The Wiener FIR filter |


|  | Statistical Signal Processing Toolbox |
| :--- | :--- |
| AllPoleCor | All Pole Signal Model using the Auto-Correlation |
| AllPoleCovar | Method |
|  | All Pole Signal Model using the Covariance Method |
| AllPoleModCo | All Pole Signal Model using the Modified Covariance |
| var | Method |
| BTpds | Computes the Blackman-Tukey Periodogram |
| cepstralC | LPC coefficients to Cepstral Coefficients |
| EigVecpds | The Eigenvector Method for Power Density Spectrum |
| harrisPDS | Computes a Periodogram using the Window Overlap |
|  | Method |
| lar2rc | Log area ratios to reflection coefficients |
| LevDurbin | The Levinson-Durbin Recursion |
| LevRecur | The Levinson Recursion |
| lineSpec | Line Spectral Pairs from LPC coefficients |
| LSIfilter | FIR Least Squares Inverse Filter |
| MUSICpds | The MUSIC Algorithm for Power Density Spectrum |
| MinNormpds | The Minimum Norm Algorithm for Power Density |
|  | Spectrum |
| rc2lar | Reflection coefficients to Log Area Ratio's |
| Welchpds | Computes the Modified or Welch Periodogram |


|  | Control Engineering Toolbox |
| :--- | :--- |
| ackerMan | State feedback gain matrix calculation |
| bodeSS | Bode plot data from State-space |
| bodeTF | Bode plot data from transfer function |
| feedbackTF | Feedback transfer functions connection |
| gainTF | Gain to force step function to zero dB |
| gridRL | Plot Root-Locus grid |
| impulseSS | Impulse response from State-space |
| impulseTF | Impulse response from transfer function |
| INITsySSS | State-space response to initial condition |
| LTIsysSS | State-space response from arbitrary input |
| LTIsysTF | Transfer func. response from arbitrary input |
| margin | Gain and Phase margins from Bode plot |
| NicholsChart | Plot of Nichols Chart |
| NicholsTF | Nyquist locus plotted on Nichols Chart |
| nyquistPolar | Nyquist plot data for Polar Plot |
| nyquistSS | Nyquist plot data from state-space |
| nyquistTF | Nyquist plot data from transfer function |
| parallelTF | Parallel transfer functions connection |
| PZtoSS | Pole-zero to state-space |
| PZtoTF | Pole-zero to transfer function |
| residueS | Partial fraction expansion from transfer func. |
| rlocusSS | Root-locus plot from state-space |
| rlocusTF | Root-locus plot from transfer function |
| RootLocusEna | Enable root-locus cursor read-out |
| ble |  |
| secondOrdSys | Computes a second order transfer function |
| seriesTF | Series transfer functions connection |
| SStoPZ | State-space to pole-zero |
| SStoTF | State-space to transfer function |
| stepSS | Step response from state-space |
| stepTF | Step response from transfer function |
| TFtoPZ | Transfer function to pole-zero |
| TFtoSS | Transfer function to state-space |
|  |  |


| Forward Error Correction (FEC) ToolBox |  |
| :--- | :--- |
| decodeBCH | A BCH decoder |
| decodeGolay | A Golay (24.12) decoder |
| decodeRs | A Reed-Solomon decoder |
| decodeViterbi | A Rate $1 ⁄ 2$ Viterbi decoder |
| encodeBCH | A BCH encoder |
| encodeGolay | A Golay (24,12) encoder |
| encodeRS | A Reed-Solomon encoder |
| encodeViterbi | A Rate $1 / 2$ Viterbi encoder |
| initBCH | Initialization of BCH decoder/encoder |
| initViterbi | Initialization of a Viterbi decoder/encoder |
| logmap | Turbo encoder/decoder |


|  | Polynomials |
| :--- | :--- |
| chebyNodes <br> chebypol | Chebyshev nodes over interval |
| lagrange | Chebyshev interpolating polynomial |
| poly | Lagrange interpolating polynomial |
| polyC | Polynomial coefficients from its roots |
| polyD | Characteristic polynomial of square matrix |
| polyDiv | Polynomial derivative |
| polyFit | Polynomial division |
| polyM | Polynomial curve fit |
| polyR | Polynomial multiplication for vector input |
| polyV | Roots of polynomial for vector input |
| proots | Value of polynomial for vector input |
|  | Roots of polynomial |


|  | Curve Fitting |
| :--- | :--- |
| interpL | Data interpolation with Lagrange approximation |
| interpS | Data interpolation using natural cubic splines |
| lsqLine | Least Squares Fit by data Linearization |
| polyFit | Polynomial curve fit |
| spline | Cubic Spline Interpolation |


|  | Statistics |
| :--- | :--- |
| delTime | Measures execution times to within 1 msec. |
| cumsum | Cumulative sum |
| cumprod | Cumulative product |
| max | Maximum value of array elements |
| $\boldsymbol{\operatorname { m a x I }}$ | Max. value index of array elements |
| $\boldsymbol{\operatorname { m e a n }}$ | Mean value of array elements |
| $\boldsymbol{\operatorname { m i n }}$ | Minimum value of array elements |
| $\boldsymbol{\operatorname { m i n I }}$ | Min. value index of array elements |
| sum | Summation of array elements |
| var | Variance of array elements |


| Differential Equations and Numerical Integration |  |
| :--- | :--- |
| abmODE | Adams-Bashforth-Moultin Method |
| hamODE | Hamming Method |
| msmODE | Milne-Simpson Method |
| rkf45 | Runge-Kutta-Fehlberg Method |
| sysODE | System of Equations ODE solver, 4 ${ }^{\text {th }}$ order Runge- |
|  | Kutta |
| sysODE2 | System of Equations ODE solver, $5{ }^{\text {th }}$ order Runge- |
|  | Kutta |
| lsmBVP | Linear Shooting Method B.V.P. |


| rombI | Romberg Integration |
| :--- | :--- |
| simpI | Integration by Simpson's rule |
| diffExt | Differentiation of $\mathrm{F}(\mathrm{x})$ |
| diffExt2 | Partial differentiation of $\mathrm{F}(\mathrm{x}, \mathrm{y})$ |


|  | Non-Linear Numerical Methods |
| :--- | :--- |
| fmin | Local minimum of function $f(x)$ |
| fmins | Local minimum of function $f(x, y, z, \ldots)$ |


|  | File I/O and String Functions |
| :--- | :--- |
| atof | Convert ASCI string to floating value |
| atoi | Convert ASCII string to integer value |
| close | Close file |
| coordTextFile | Read Coordinate Text File format to matrix |
| getc | Get next ASCII character |
| input | Input dialog for a variable |
| open | Open file |
| read | Read file to array(int, float, complex) |
| readM | Read file to matrix type |
| readV | Read file to vector type |
| readExcel | Read Excel data to matrix |
| readLine | Read ASCII line to character buffer |
| rewind | Rewind file |
| skip | Skip data elements in file |
| skipLines | Skip lines in ASCII opened file |
| sprintf | sprintf statement (as standard in C) |
| strcat | String catenation |
| strchr | Search string for a character |
| strcpy | String copy function |
| strcmp | String compare function |
| strlen | String length function |
| strstr | String sub-compare |
| write | Write data to file |

## Plotting Routines

| arrowP | Plot arrow, $\mathrm{x} / \mathrm{y}$ plot |
| :--- | :--- |
| aspectR | Aspect ratio for x/y plots |
| axisS | Axis style, x/y plot |
| camloc3D | Camera location -3D plot |
| closePlot | Close plot |
| color3D | Set color value in palette |
| colorB | Background color |
| colorBar | Color palette selection - 3D plot |
| colorT | Text color, x/y plot |
| confil3D | Specifications for contour fill mode |


| conlab3D | Options for 3D contour labels |
| :--- | :--- |
| consurf3D | 3D contour and/or surface plot |
| contourP | Contour x/y plot with color mapping |
| contour | A line Contour x/y plot |
| grid | Plot grid lines, x/y plot |
| gridH | Plot horizontal grid lines, x/y plot |
| histoP | Histogram plot |
| linethick3C | Line thickness for 3D line plot (plot3C) |
| movieP | Set movie mode for multi x-y plots |
| openPlot | Open plot |
| opts3D | Options for 3D plot |
| pause | Pause for keyboard or mouse input |
| pCRT | Output plot to the display monitor |
| penS | Pen style, x/y plot |
| ploglog | Plot x and y array on log scale |
| plogxy | Plot x array on log scale, y linear |
| plot3C | Plot x/y/z curve(s) in 3D |
| plotx | Plot x array |
| plotxy | Plot x and y arrays |
| ploty | Plot y array |
| pminmax | Set min/max for x/y axes |
| pminmax3D | Set min/max for x/y/z axes |
| polarP | Polar plot |
| polarPdb | Polar plot -dB |
| polezeroP | Pole-Zero plot |
| pPrint | Output plot to printer |
| ptext | Plot text string, x/y plot at x/y point |
| Ptext3C | Plot text string, x/y/z 3D plot (plot3C) |
| Ptics | Force selection of number of tic marks |
| pxlogy | Plot y array on log scale, x linear |
| stripsP | Plot y array as strip chart |
| surfil3D | Specifications for surface fill mode |
| surwir3D | Specifications for wire mesh mode |
| Title | Plot Title Label, x/y plot |
| vectorP | Vector gradient plot |
| waterFallP | 3D waterfall plot |
| xLabel | Plot x-label |
| yLabel | Plot y-label |
| zLabel | Plot z-label |
|  |  |
|  |  |

## Appendix B-Equations Reference for the Smith Chart Tool

## VSWR Circle

The standing wave ratio of an impedance point is given by,

$$
S=\frac{1+\rho}{1-\rho}
$$

where $\rho$ is equal to the reflection coefficient at the impedance point. The value $\rho$ is given by,

$$
\rho=\frac{Z-Z_{0}}{Z+Z_{0}}
$$

where Z 0 is the Characteristic Impedance. The complex Characteristic Impedance is given by,

$$
Z_{0}=\sqrt{\frac{R+j \omega L}{G+j \omega C}}
$$

or to simplify

$$
Z_{0}=\sqrt{\frac{L}{C}}
$$

Then the locus of all impedance points with the same VSWR is thus a circle centered at the normalized impedance point of 1.0 with a constant radius equal to,

$$
r=\sqrt{R_{0}^{2}+X_{0}^{2}}
$$

where
$R_{0}$ equals the normalized impedance resistance
$X_{0}$ equals the normalized impedance reactance

## Circuit Q Contour

The Circuit Q contour is all those impedance points whose ratio of absolute reactance to absolute resistance is a constant. In equation form this is give by,

$$
Q_{c}=\frac{|X|}{|R|}
$$

where an impedance value is given by,

$$
Z=R+j X
$$

## Admittance Point

The equation for an Admittance point from a given Impedance point is given by,

$$
Y=\frac{1}{Z}
$$

## Conjugate Impedance Point

Given an impedance point by the equation,

$$
Z=R+j X
$$

The conjugate impedance point is give by,

$$
Z^{*}=R-j X
$$

## Impedance Match - Component Values

When performing an impedance match along constant resistance lines or constant conductance lines, the equations to determine the component values are as follows:

For a series-C component:

$$
C=\frac{1}{\omega X Z_{0}}
$$

For a series-L component:

$$
L=\frac{X Z_{0}}{\omega}
$$

For a shunt-C component:

$$
C=\frac{B}{\omega Z_{0}}
$$

For a shunt-L component:

$$
L=\frac{Z_{0}}{\omega B}
$$

where,

$$
\omega=2 \pi f
$$

$X=$ the reactance from the chart
$B=$ the susceptance from the chart
$Z_{0}=$ the Characteristic Impedance value

## Small Transistor - Stability Circles

A stability circle on a Smith Chart represents the boundary between values of source or load impedance values that can cause instability. The boundary of the circle represents the locus of points that causes the Rowlett Stability Factor (K) to equal 1. The center locations and the radii of the input and output stability circles are calculated as follows:

1. Calculate $D_{S}$ as follows:

$$
\begin{equation*}
D_{S}=S_{11} S_{22}-S_{12} S_{21} \tag{Eq.4-1}
\end{equation*}
$$

2. Calculate $C_{1}$

$$
\begin{equation*}
C_{1}=S_{11}-D_{s} S_{22}{ }^{*} \tag{Eq.4-2}
\end{equation*}
$$

3. Calculate $C_{2}$

$$
C_{2}=S_{22}-D_{s} S_{11}{ }^{*}
$$

4. Calculate the center of the input stability circle

$$
\begin{equation*}
c_{s}=\frac{C_{1}^{*}}{\left|S_{11}\right|^{2}-\left|D_{s}\right|^{2}} \tag{Eq.4-4}
\end{equation*}
$$

5. Calculate the radius of the input stability circle

$$
\begin{equation*}
r_{s}=\left|\frac{S_{12} S_{21}}{\left|S_{11}\right|^{2}-\left|D_{s}\right|^{2}}\right| \tag{Eq.4-5}
\end{equation*}
$$

6. Calculate the center of the output stability circle

$$
\begin{equation*}
c_{o}=\frac{C_{2}^{*}}{\left|S_{22}\right|^{2}-\left|D_{s}\right|^{2}} \tag{Eq.4-6}
\end{equation*}
$$

7. Calculate the radius of the output stability circle

$$
\begin{equation*}
r_{o}=\left|\frac{S_{12} S_{21}}{\left|S_{22}\right|^{2}-\left|D_{s}\right|^{2}}\right| \tag{Eq.4-7}
\end{equation*}
$$

## Small Transistor - Constant Gain Circles

## Operating Power Gain - $\boldsymbol{G}_{\boldsymbol{p}}$

The operating power gain is independent of the source impedance. The center location and the radius of the circle for a given gain is calculated as follows:

1. Calculate $D_{2}$.

$$
D_{2}=\left|S_{22}\right|^{2}-|\Delta|^{2}
$$

2. Calculate $C_{2}$ using equation 4-3.
3. Calculate desired gain

$$
\begin{equation*}
g_{p}=\frac{\text { DesiredGain(absolute })}{\left|S_{21}\right|^{2}} \tag{Eq.4-8}
\end{equation*}
$$

4. Calculate the center location of the gain circle

$$
\begin{equation*}
c_{p}=\frac{g_{p} C_{2}^{*}}{1+D_{2} g_{p}} \tag{Eq.4-9}
\end{equation*}
$$

5. Calculate the radius of the gain circle

$$
\begin{equation*}
r_{p}=\frac{\sqrt{1-2 K\left|S_{12} S_{21}\right| g_{p}+\left|S_{12} S_{21}\right|^{2} g_{p}{ }^{2}}}{1+D_{2} g_{p}} \tag{Eq.4-10}
\end{equation*}
$$

## Available Power Gain - $\boldsymbol{G}_{\boldsymbol{A}}$

The center location and the radius of the circle for a given gain is calculated as follows:

1. Calculate $\mathrm{D}_{2}$.

$$
D_{2}=\left|S_{11}\right|^{2}-|\Delta|^{2}
$$

2. Calculate $C_{1}$ using equation 4-2.
3. Calculate desired gain

$$
\begin{equation*}
g_{A}=\frac{\text { DesiredGain(absolute })}{\left|S_{21}\right|^{2}} \tag{Eq.4-11}
\end{equation*}
$$

4. Calculate the center location of the gain circle

$$
\begin{equation*}
c_{A}=\frac{g_{A} C_{1}^{*}}{1+D_{2} g_{A}} \tag{Eq.4-12}
\end{equation*}
$$

5. Calculate the radius of the gain circle

$$
\begin{equation*}
r_{A}=\frac{\sqrt{1-2 K\left|S_{12} S_{21}\right| g_{A}+\left|S_{12} S_{21}\right|^{2} g_{A}{ }^{2}}}{1+D_{2} g_{A}} \tag{Eq.4-13}
\end{equation*}
$$

## Unilateral Cases - S12 equals zero

When S12 is small, we can represent the transducer power gain of the transistor by the following block diagram.


The equation for the unilateral transducer gain is then given by, (Eq. 4-14)
$G_{T u}=G_{S} G_{O} G_{L}$
The equations for each block are given by,
$G_{S}=\frac{1-\left|\Gamma_{S}\right|^{2}}{\left|1-S_{11} \Gamma_{S}\right|^{2}}$
$G_{O}=\left|S_{21}\right|^{2}$
$G_{L}=\frac{1-\left|\Gamma_{L}\right|^{2}}{\left|1-S_{22} \Gamma_{L}\right|^{2}}$

## Small Transistor - Constant VSWR Circles

The input VSWR of a microwave amplifier is given by:

$$
\operatorname{VSWR}_{\mathrm{IN}}=\frac{1+\left|\Gamma_{a}\right|}{1-\left|\Gamma_{a}\right|} \text {, where }\left|\Gamma_{\mathrm{a}}\right|=\left|\frac{\Gamma_{\mathrm{IN}}-\Gamma_{S}^{*}}{1-\Gamma_{\mathrm{IN}} \Gamma_{S}}\right|
$$

$\Gamma_{a}$ can also be computed by

$$
\begin{aligned}
& \Gamma_{\mathrm{a}}=\sqrt{1-\mathrm{M}_{\mathrm{s}}}, \text { where } \mathrm{M}_{\mathrm{S}} \text { is the Source Mismatch Factor given by } \\
& \mathrm{M}_{\mathrm{S}}=\frac{\left(1-\left|\Gamma_{S}\right|^{2}\right)\left(1-\left|\Gamma_{\mathrm{IN}}\right|^{2}\right)}{\left|1-\Gamma_{S} \Gamma_{\mathrm{IN}}\right|^{2}}
\end{aligned}
$$

The output VSWR of a microwave amplifier is given by:

$$
\operatorname{VSWR}_{\text {OUT }}=\frac{1+\left|\Gamma_{\mathrm{b}}\right|}{1-\left|\Gamma_{\mathrm{b}}\right|} \text {, where }\left|\Gamma_{\mathrm{b}}\right|=\left|\frac{\Gamma_{\text {OUT }}-\Gamma_{\mathrm{L}}^{*}}{1-\Gamma_{\text {OUT }} \Gamma_{\mathrm{L}}}\right|
$$

$\Gamma_{\mathrm{b}}$ can also be computed by

$$
\begin{aligned}
& \Gamma_{\mathrm{b}}=\sqrt{1-\mathrm{M}_{\mathrm{L}}}, \text { where } \mathrm{M}_{\mathrm{L}} \text { is the Load Mismatch Factor given by } \\
& \mathrm{M}_{\mathrm{L}}=\frac{\left(1-\left|\Gamma_{\mathrm{L}}\right|^{2}\right)\left(1-\left|\Gamma_{\mathrm{out}}\right|^{2}\right)}{\left|1-\Gamma_{\mathrm{L}} \Gamma_{\text {out }}\right|^{2}}
\end{aligned}
$$

## Small Transistor - Noise Circles

The center location and radius of a given noise circle for a two-port amplifier is calculated as follows:

1. Calculate a noise parameter $N_{p}$

$$
\begin{equation*}
N_{p}=\frac{F_{p}-F_{\min }}{4 R_{N} / Z_{0}}\left|1+\Gamma_{0}\right|^{2} \tag{Eq.4-18}
\end{equation*}
$$

where,
$F_{p}$ is the desired noise circle contour in dB
$F_{\text {min }}$ is the minimum noise figure (from the manufacture)
$Z_{0}$ is the characteristic impedance
$\Gamma_{0}$ is the reflection coefficient at the input port for minimum noise
$R_{N}$ is the effective noise resistance (from the manufacture)
2. Calculate the center of the noise circle
(Eq. 4-19)

$$
c_{n}=\frac{\Gamma_{0}}{1+N_{p}}
$$

3. Calculate the radius of the noise circle

$$
\begin{equation*}
r_{n}=\frac{1}{1+N_{p}} \sqrt{N_{p}^{2}+N_{p}\left(1-\left|\Gamma_{0}\right|^{2}\right)} \tag{Eq.4-20}
\end{equation*}
$$

## Small Transistor Calculations

The Rowlett Stability Factor - K
The Rowlett Stability factor is calculated as follows.

1. Calculate $D s$ from equation 4-1.
2. Calculate $K$ as
(Eq. 4-21)
$K=\frac{1+\left|D_{S}\right|^{2}-\left|S_{11}\right|^{2}-\left|S_{22}\right|^{2}}{2\left|S_{21}\right|\left|S_{12}\right|}$

## The Maximum Available Gain - MAG

The Maximum Available Gain is only defined when K is $>=1$.

1. First calculate $B$
(Eq. 4-22)
$B=1+\left|S_{11}\right|^{2}-\left|S_{22}\right|^{2}-\left|D_{s}\right|^{2}$
2. Then MAG is calculated as follows.
if $B>=0$
$M A G=10 \log \frac{\left|S_{21}\right|}{\left|S_{12}\right|}+10 \log \left|K-\sqrt{K^{2}-1}\right|$
if $\mathrm{B}<0 \quad$ Potential Unstable

$$
M A G=10 \log \frac{\left|S_{21}\right|}{\left|S_{12}\right|}+10 \log \left|K+\sqrt{K^{2}-1}\right|
$$

The Load Reflection Coefficient at MAG

1. First calculate $B_{L}$
(Eq. 4-23)

$$
B_{L}=1+\left|S_{22}\right|^{2}-\left|S_{11}\right|^{2}-\left|D_{s}\right|^{2}
$$

2. Next calculate $C_{L}$
(Eq. 4-24)

$$
C_{L}=S_{22}-D_{S} S_{11}{ }^{*}
$$

3. Then $\left|\Gamma_{\mathrm{L}}\right|$ is calculated as follows.

$$
\text { if } B_{L}>=0
$$

$$
\begin{aligned}
& \left|\Gamma_{L}\right|=\frac{B_{L}-\sqrt{B_{L}^{2}-4\left|C_{L}\right|^{2}}}{2\left|C_{L}\right|} \\
& \text { if } B_{L}<0 \\
& \left|\Gamma_{L}\right|=\frac{B_{L}+\sqrt{B_{L}^{2}-4\left|C_{L}\right|^{2}}}{2\left|C_{L}\right|} \\
& \operatorname{ARG}\left(\Gamma_{L}\right)=-\operatorname{ARG}\left(C_{L}\right)
\end{aligned}
$$

The Source Reflection Coefficient at MAG
(Eq. 4-27)

$$
\Gamma_{S}=\left[S_{11}+\frac{S_{12} S_{21} \Gamma_{L}}{1-\Gamma_{L} S_{22}}\right]^{*}
$$

## Source Reflection Coefficient Calculation

The Source Reflection Coefficient is calculated from a given Load Reflection Coefficient by the following equation.

$$
\Gamma_{S}=\left[S_{11}+\frac{S_{12} S_{21} \Gamma_{L}}{1-\Gamma_{L} S_{22}}\right]^{*}
$$

The input reflection coefficient is given by:

$$
\Gamma_{\mathrm{IN}}=\left[\mathrm{S}_{11}+\frac{\mathrm{S}_{12} \mathrm{~S}_{21} \Gamma_{\mathrm{L}}}{1-\Gamma_{\mathrm{L}} \mathrm{~S}_{22}}\right]
$$

## Load Reflection Coefficient Calculation

The Load Reflection Coefficient is calculated from a given Source Reflection Coefficient by the following equation.

$$
\begin{equation*}
\Gamma_{L}=\left[S_{22}+\frac{S_{12} S_{21} \Gamma_{S}}{1-\Gamma_{S} S_{11}}\right]^{*} \tag{Eq.4-28}
\end{equation*}
$$

The output reflection coefficient is given by:

$$
\Gamma_{\mathrm{ouT}}=\left[\mathrm{S}_{22}+\frac{\mathrm{S}_{12} \mathrm{~S}_{21} \Gamma_{\mathrm{S}}}{1-\Gamma_{\mathrm{S}} \mathrm{~S}_{11}}\right]
$$

## Small Transistor - Power Gain Equations

The Transducer Gain is the gain of the amplifier stage plus the gains from the effects of input and output matching networks. From the S-parameters defined at a given operating state and for a given set of source and load reflection coefficients, the Transducer Gain is given by,

$$
G_{T}=\frac{1-\left|\Gamma_{S}\right|^{2}}{\left|1-\Gamma_{I N} \Gamma_{S}\right|^{2}}\left|S_{21}\right|^{2} \frac{1-\left|\Gamma_{L}\right|^{2}}{\left|1-S_{22} \Gamma_{L}\right|^{2}}
$$

The Operating Power gain given by:

$$
G_{P}=\frac{1}{1-\left|\Gamma_{I N}\right|^{2}}\left|S_{21}\right|^{2} \frac{1-\left|\Gamma_{L}\right|^{2}}{\left|1-S_{22} \Gamma_{L}\right|^{2}}
$$

The Available Power gain given by:

$$
G_{A}=\frac{1-\left|\Gamma_{S}\right|^{2}}{\left|1-S_{11} \Gamma_{S}\right|^{2}}\left|S_{21}\right|^{2} \frac{1}{1-\left|\Gamma_{\text {oUT }}\right|^{2}}
$$

## Y-Parameters to S-Parameters Conversion

The Y-param -> S-param... dialog menu under the Functions menu is given to convert Yparameters to S-parameters. The parameters once converted to S-parameters are saved internally and the S-parameters for other dialog procedures are initialized from these saved values. Note that if you must first multiply each individual Y parameter by $\mathrm{Z}_{0}$ before enter the value in the dialog box. The equations for these conversions are as follows:

$$
\begin{aligned}
& S_{11}=\frac{\left(1-y_{i}\right)\left(1+y_{0}\right)+y_{r} y_{f}}{\left(1+y_{i}\right)\left(1+y_{0}\right)-y_{r} y_{f}} \\
& S_{12}=\frac{-2 y_{r}}{\left(1+y_{i}\right)\left(1+y_{0}\right)-y_{r} y_{f}} \\
& S_{21}=\frac{-2 y_{f}}{\left(1+y_{i}\right)\left(1+y_{0}\right)-y_{r} y_{f}} \\
& S_{22}=\frac{\left(1+y_{i}\right)\left(1-y_{0}\right)+y_{r} y_{f}}{\left(1+y_{i}\right)\left(1+y_{0}\right)-y_{r} y_{f}}
\end{aligned}
$$

## Micro-strip Calculation

The Micro-strip Calc... dialog menu under the Functions menu is given to calculate either the width and length of a micro-strip line or it's characteristic impedance. It is based on the quasiTEM assumption and negligible thickness of the strip conductor.

For the Width and Length calculation the inputs are as follows:

1. The Characteristic impedance of the line.
2. The thickness of the substrate.
3. The dielectric constant of the substrate material $\mathrm{E}_{\mathrm{R}}$.
4. The fractional wavelength of the micro-strip (lambda)
5. The operating frequency.

The calculations are performed as follows: (Reference[1], pp 143-152)

1. First calculate parameters $A$ and $B$ as follows:

$$
\begin{aligned}
& A=\frac{Z_{0}}{60} \sqrt{\frac{\varepsilon_{r}+1}{2}}+\frac{\varepsilon_{r}-1}{\varepsilon_{r}+1}\left(0.23+\frac{0.11}{\varepsilon_{r}}\right) \\
& B=\frac{377 \pi}{2 Z_{0} \sqrt{\varepsilon_{r}}}
\end{aligned}
$$

2. Next calculate W , where W is the width of the line, and h is the thickness of the substrate.

$$
\begin{aligned}
& \text { for } \mathrm{W} / \mathrm{h}<2 \\
& W=\frac{8 e^{A} h}{e^{2 A}-2} \\
& \text { for } \mathrm{W} / \mathrm{h}>=2
\end{aligned}
$$

$$
W=\frac{2}{\pi}\left\{B-1-\ln (B-1)+\frac{\varepsilon_{r}-1}{2 \varepsilon_{r}}\left[\ln (B-1)+0.39-\frac{0.61}{\varepsilon_{r}}\right]\right\}
$$

3. Calculate $\lambda$

$$
\text { for } \mathrm{W} / \mathrm{h}>=0.6
$$

$$
\lambda=\frac{\lambda_{0}}{\sqrt{\varepsilon_{r}}}\left[\frac{\varepsilon_{r}}{1+0.63\left(\varepsilon_{r}-1\right)(W / h)^{0.1255}}\right]^{1 / 2}
$$

for $\mathrm{W} / \mathrm{h}<0.6$

$$
\lambda=\frac{\lambda_{0}}{\sqrt{\varepsilon_{r}}}\left[\frac{\varepsilon_{r}}{1+0.6\left(\varepsilon_{r}-1\right)(W / h)^{0.0297}}\right]^{1 / 2}
$$

4. Calculate $\varepsilon_{f f}$

$$
\varepsilon_{f f}=\left(\frac{1}{\lambda}\right)^{2}
$$

5. Calculate the length of the micro-strip.

$$
l=\frac{\lambda}{k_{0} \sqrt{\varepsilon_{f f}}}
$$

where

$$
k_{0}=\frac{2 \pi f}{c}
$$

For the Characteristic Impedance calculation the inputs are as follows:

1. The thickness of the substrate (h).
2. The dielectric constant of the substrate material.
3. The width of the micro-strip in millimeters (W).

The calculations are performed as follows:

1. First calculate the effective dielectric constant.

For W/h $>=1$

$$
\varepsilon_{f f}=\frac{\varepsilon_{r}+1}{2}+\frac{\varepsilon_{r}-1}{2}\left(1+12 \frac{h}{W}\right)^{-1 / 2}
$$

For W/h $<1$

$$
\varepsilon_{f f}=\frac{\varepsilon_{r}+1}{2}+\frac{\varepsilon_{r}-1}{2}\left[\left(1+12 \frac{h}{W}\right)^{-1 / 2}+0.04\left(1-\frac{W}{h}\right)^{2}\right]
$$

2. Next, calculate $\mathrm{Z}_{0}$

For W/h >= 1

$$
Z_{0}=\frac{120 \pi \sqrt{\varepsilon_{f f}}}{W / h+1.393+0.667 \ln (W / h+1.444)}
$$

For W/h $<1$

$$
Z_{0}=\frac{60}{\sqrt{\varepsilon_{f f}}} \ln \left(8 \frac{h}{W}+0.25 \frac{W}{h}\right)
$$

## JCP Software \& Company License Agreement

This is a legal agreement between you, the end user, and JCP Software \& Company.

## USE of the SOFTWARE

The term "SOFTWARE", as used throughout this License Agreement will refer to the Slide-Rule ${ }^{\text {TM }}$ software package that is shipped with this package or container.

GRANT OF LICENSE. JCP Software \& Company grants the original purchaser ("Licensee") the limited rights to possess and use the SOFTWARE and Reference Manual for its intended purposes. You may not network the SOFTWARE or otherwise use it on more than one computer or computer terminal at the same time. Licensee's who are corporations, companies, government entities, schools systems, or any other non-personal user, must have a license for each and every computer that the software is installed on. Personal users, i.e., when the software is registered in a person's name, may have the software installed on their assigned computer at work, on their personal PC, and also on their personal notebook.

COPYRIGHT. The SOFTWARE is owned by and remains the property of JCP Software \& Company, is protected by United States copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE like any other copyrighted material (e.g., a book or musical recording) except that you may either (a) make one copy of the SOFTWARE solely for backup or archival purposes, and the copy shall display all propriety notices, and be labeled externally to show that the back-up copy is the property of JCP Software \& Company, and that use is subject to this License, or (b) transfer the SOFTWARE to a single hard disk provided you keep the original solely for backup or archival purposes. You may copy the written materials that accompany the SOFTWARE, however, and distribute them freely.

WARRANTY. The material is sold "as is". JCP Software \& Company makes no warranties, either expressed or implied, regarding the enclosed computer software package, its merchantability, or its fitness for any particular purpose.

DISCLAIMER. License fees for the SOFTWARE do not include any consideration for assumption of risk by JCP Software \& Company, and JCP Software \& Company disclaims any and all liability for incidental or consequential damages arising out of the use or operation or inability to use the SOFTWARE, even if any of these parties have been advised of the possibility of such damages. Furthermore, Licensee indemnifies and agrees to hold JCP Software \& Company harmless from such claims. The entire risk as to the results and performance of the SOFTWARE is assumed by the Licensee. The warranties expressed in this License are the only warranties, expressed or implied, including but not limited to implied warranties of merchantability and of fitness for a particular purpose. Some jurisdictions do not allow the exclusion or limitations of warranties, so the above limitations or exclusions may not apply to you.


[^0]:    * [2] pp 464-488, [8] pp 678-699

[^1]:    * [6] pp 207-210

[^2]:    * [8] pp 693-699

[^3]:    * [8] pp 691-693

[^4]:    * [1] pp 205-292, [8] pp 701-735, [2] 415-438

[^5]:    * [6] pp 370-383, [7] pp 134-140

[^6]:    * [6] pp 323-367, [11], [[12] pp 397-487

[^7]:    * [6] pp 127-149

[^8]:    * [3], [1] pp 88-105, [2] pp 444-464, [4] 43-51, [5] pp 664-670

[^9]:    * [7] pp 451-469

[^10]:    * [2] 623-628, [5] pp 544-549

[^11]:    * [1] pp 791-727, [2] 301-362, [3] 205-292

[^12]:    * [1] pp 574-582, 594-602, [2] pp 208-218
    ${ }^{\$}$ [5] pp 225-232

[^13]:    * [3] pp 81-82, [2] pg 41

[^14]:    * [1] pp 20-24, [3] pp 82-83, [2] pp 41-44

[^15]:    * [1] pp 24-26, [2] pp 45-46

[^16]:    * [1] pp 26-28, [2] pp 46-48
    * [2] pp 48-50, [1[ pp30-31

[^17]:    * [2] pp 50-52, [1[ pp 31-32

[^18]:    * [3] pp 194-236, [2] pp 49-82, [1] pp 60-110, [3] 50-56

[^19]:    * [1] pp 178-182, [2] pp 139-143, [3] pp 367-372, [3] pp 50-56

[^20]:    * [1[ 174-177

[^21]:    * [1] pp 142-147

[^22]:    * [1] pp 440-447, [2] pp 247-264, [3] pp 56-59

[^23]:    * [3] pp 20-22

[^24]:    * [1] pp 186-189

[^25]:    * [1] pp 191-197

[^26]:    * [1] pp 186-190

[^27]:    * [1] pp 125-128, pp 175-179, [2] pp 369-370

[^28]:    * [1] pp 194-209, [2] 370-374

[^29]:    * [1] pp 237-254, [2] 437-469

[^30]:    * [2] pp 469-475

[^31]:    * [1] pp 515-598, [2] 383-430, [5] 153-209, [4]

[^32]:    * [5] pp 202-209

[^33]:    * [1] pp 826-844, [5] 213-245

[^34]:    * Gaussian noise added at Eb/N0 level

[^35]:    * [1] pp223-233, 248-250

[^36]:    * [1] pp143-146

[^37]:    * [1] pp 253-254

[^38]:    * [1] pp 152-153

[^39]:    * [1] p 149

[^40]:    * [1] pp 410-414

[^41]:    * [1] pp 431-436
    ${ }^{\$}$ [2]

[^42]:    * [2] pp 199-200

[^43]:    * [2] pp 504-509

[^44]:    * [2] pp 509-515

[^45]:    * [2] pp 515-524

[^46]:    * [2] pp 537-546

[^47]:    * [2] pp 66-96, [1[ pp 92-141

[^48]:    * [1] pp 22-70
    \$ [2[ pp 103-107, pp 127-149

[^49]:    * [2] pp 132-133

[^50]:    * [2[ 138-149

[^51]:    * [2] pg 121, pp 133-138
    \$ [1] pp 217-228

[^52]:    \$ [1] pp 141-175

[^53]:    * [3] pp 97-102

[^54]:    * [3] pp 110-112

