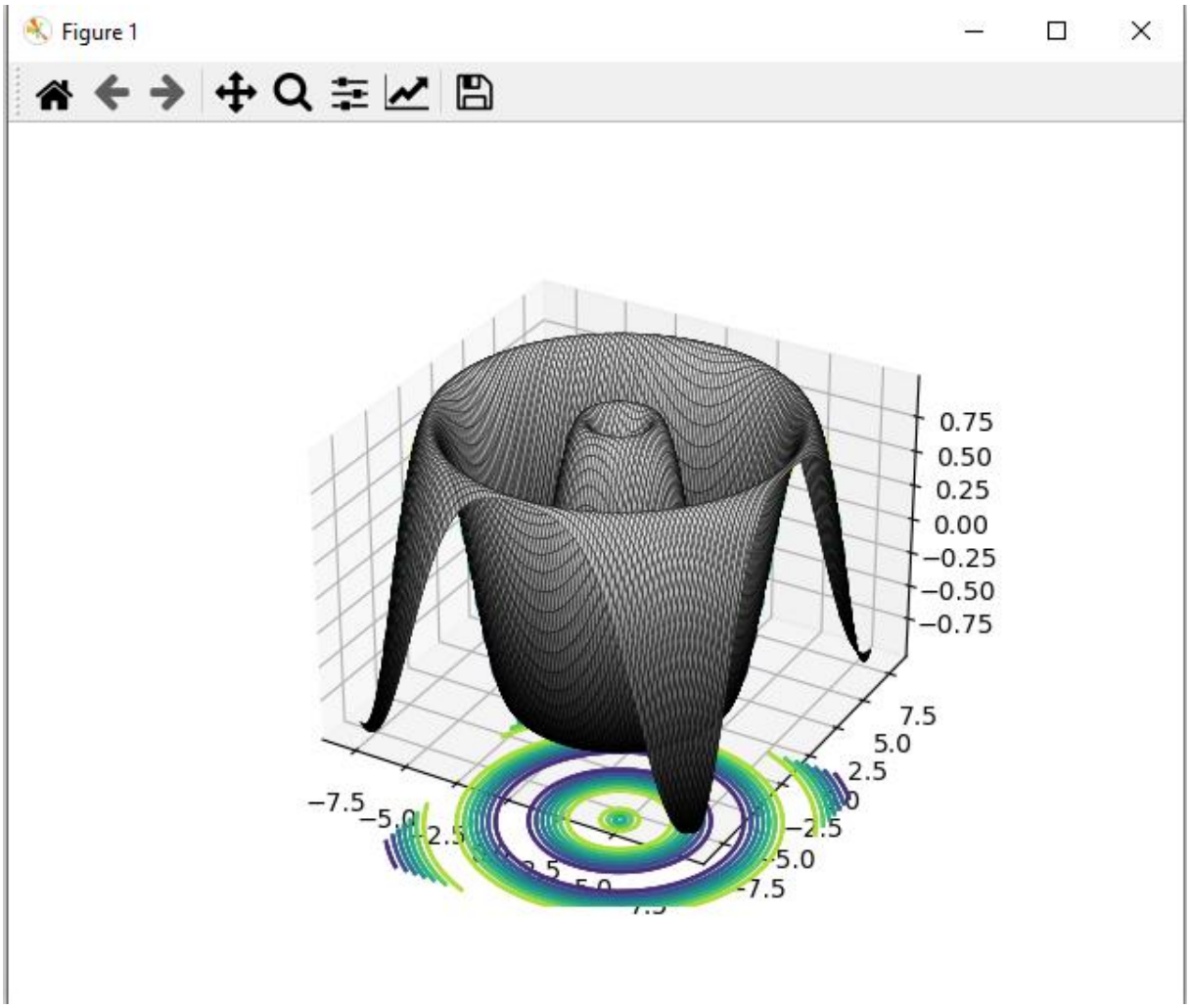
NUMERICAL MATH WITH JULIA

M. Turhan ÇOBAN

Ege University, School of Engineering, department of Mechanical Engineering, Bornova, İZMİR, Turkey

www.turhancoban.com



1. JULIA PROGRAMMING LANGUAGE BASICS

M. Turhan Coban

1.1 LOADING JULIA ENVIRONMENT INTO YOUR COMPUTER

The main source for julia programming language is https://julialang.org

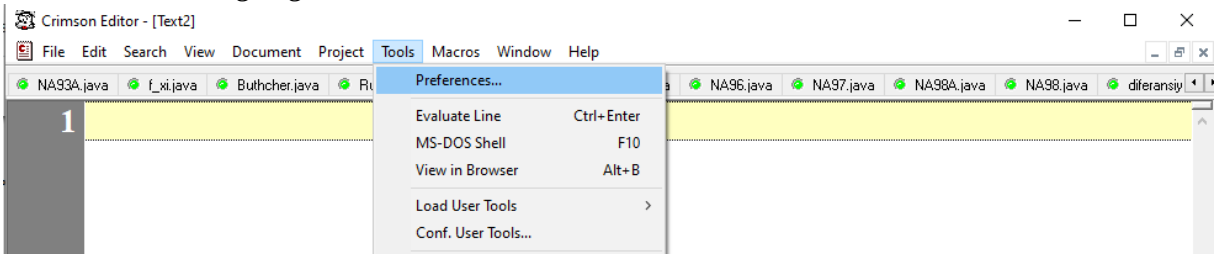
It is a free Access programming language with MIT licence.
After loading up julia programming language, you will find bin command under julia_home/bin directory. When you run this julia command, console environment will be opened. If you wish, you can run your command in console.

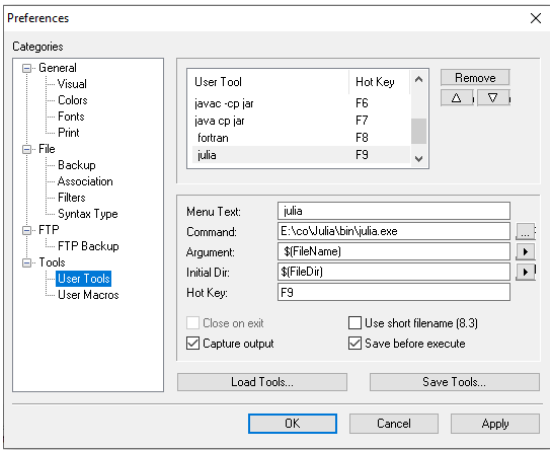
```
E:\co\Julia\bin>julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.5.3 (2020-11-09)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> x=23
23
```

Or alternatively, write your program in a editor environment and then run by using julia command. As

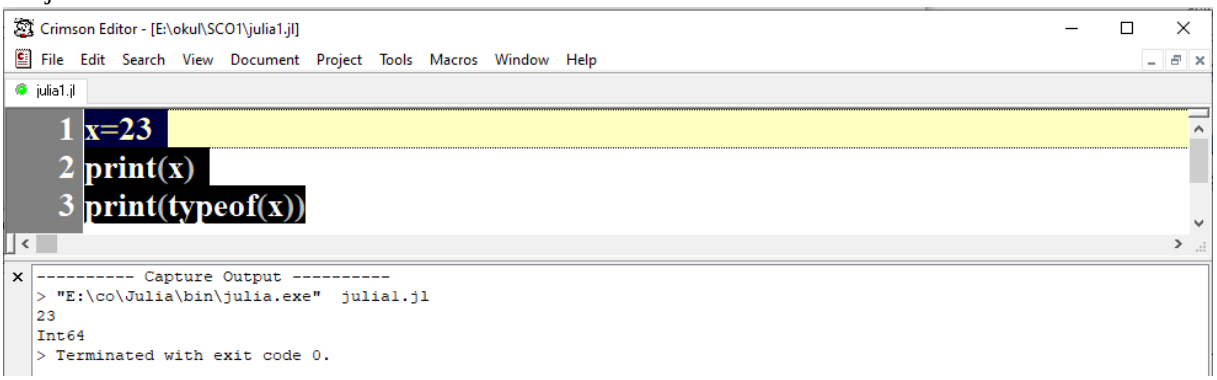
julia>julia script.jl arg1 arg2...

One of my favorite editor is crimson editor(http://www.crimsoneditor.com/) to run the programs. It is a simple editor with build in console environment command runner so you can run your julia commands without going into console.

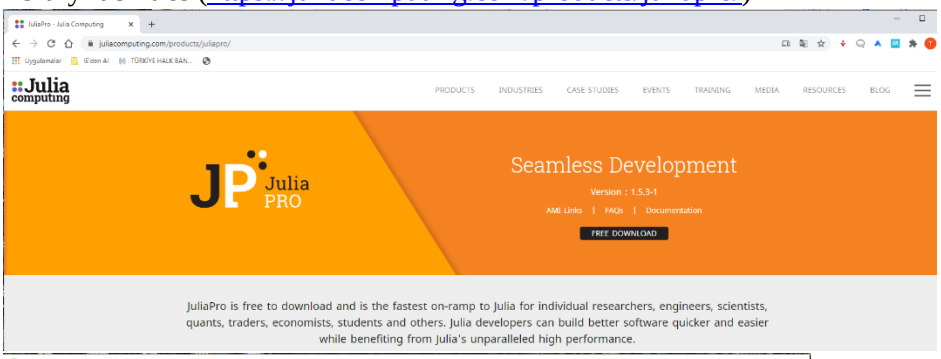


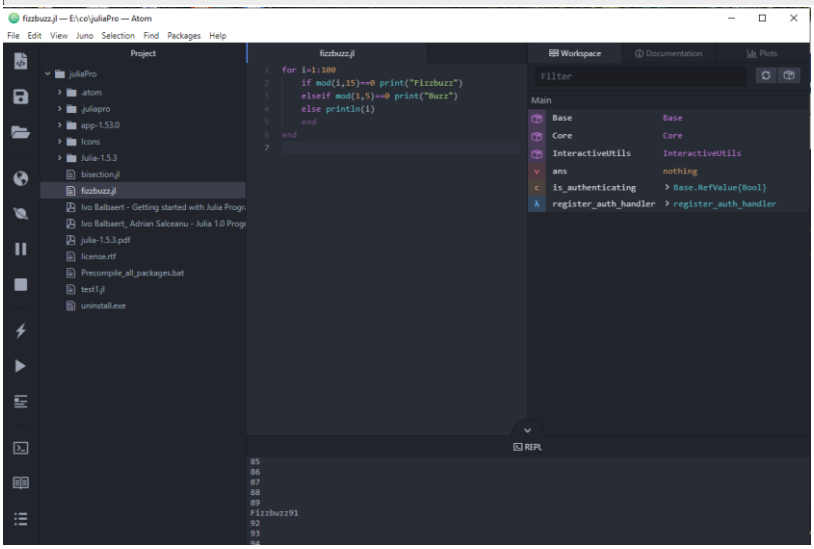


After setting up write your program (name.jl) into the editor and run it by using the selected hot key for julia command



Another good running and editing environment is JuliaPro. It is an environment with editors and more library facilities (https://juliacomputing.com/products/juliapro/)





1.2 BASIC PROCESSES AND VARIABLES

In Julia language when defining the variables, method and classes variable types is not specified. When the data is loaded to a value into the variable language determine the variable type. Julia can use variable base definitions for numbers and strings.

Integer types:

Type	Java/C equivalent	Signed?	Number of Bits	Smallest value	Largest value
Int8	byte	x	8	-2^7=-128	2^7-1=127
UInt8			8	0	$2^8=256$
Int16	short	x	16	$-2^{15}=-32768$	$2^{15}-1$=-32767
UInt16			16	0	2^{16} =65536
Int32	int	x	32	-2^{31} $=-2147483648$	$2^{31}-1$ $=2147483647$
UInt32			32	0	2^{32} =4294967296
Int64	long	x	64	-2^{63} $=-9.223372e18$	$2^{63}-1$ $=9.223372e18$
UInt64			64	0	1.844674e19
Int128		x	128	-2^{127} $=1.701411e38$	$2^{127}-1$ $=1.701411e38$
UInt128			128	0	3.402823669e38
Bool			8	false(0)	true(1)

Floating-point (real) types

Type	Java equivalent	Signed?	Number of Bits	Smallest value	Largest value
Float16		x	16	-2^7=-128	2^7-1=127
Float32	float		32	0	$2^8=256$
Float64	double	x	64	$-2^{15}=-32768$	$2^{15}-1$=-32767

Variable type can be investigated by using typeof method

```
julia> x=277
277
julia> typeof(x)
Int64
julia> y=1.2454353e38
1.2454353e38
julia> typeof(y)
Float64
```

Additionally, full support for Complex and Rational Numbers is built on top of these primitive numeric types. Binary and octoganal types can also be used

```
julia> x=0b01001
0x09
```

```
julia> y=0o14
0x0c
```

Types can be converted by using their variable types

```
julia> x=-1.2
-1.2
julia> y=Float32(x)
-1.2f0
julia> z=0b0010101
0x15
julia> w=Float64(z)
21.0
```

Floating point numbers have two zeros, positive zero and negative zero. Values are equal to each other, but bit values are different

```
julia> x=0.0
0.0
julia> y=-0.0
-0.0
julia> z=x==y
true
julia> bitstring(x)
"0000000000000000000000000000000000000000000000000000000000000000"
julia> bitstring(y)
"1000000000000000000000000000000000000000000000000000000000000000"
julia>
```

Infinities and Notanumber type is also
Positive infinite Infinite numbers: Inf16 Inf32 Inf
Negative infinite numbers: -Inf16 -Inf32 -Inf

Not a number: NaN16 NaN32 NaN

An epsilon value is also defined

```
julia> eps1=eps(Float32)
1.1920929f-7
julia> eps2=eps(Float64)
2.220446049250313e-16
```

The following arithmetic operators are defined in Julia

Expression	Name	Description
+x	Unary plus	The identitiy operation
-x	Unary minus	Map values to their additive inverses
x+y	Binary plus	Perform addition
x-y	Binary minus	Perform substraction
x*y	multiplication	Perform multiplication
x/y	divide	Performs division
x ÷y	integer divide	Division truncated to an integer
x\y	İnverse divide	Equaivalent to y/x

x^y	power	Raises x to the yth power
x%y	remainder	

Negetiation of Bool type

!x	negatiation	Change true to false false to true

Bitwise operators

The following bitwise operators are supported on all primitive integer types:

Expression	Name
~x	bitwise not
x&y	bitwise and
x|y	bitwise or
x ⊻y	bitwise xor(exclusive or)
x>>>y	Logical shift right
x>>y	Arithmetic shift right
x<<y	Logical/arithmetic shift left

```
julia> 123 | 234
251
julia> xor(123,234)
145
julia> 123 & 234
106
julia>
```

Numeric comparison operators:

==	Equality
!=, ≠	Not equal
<	Less than
<=, ≤	Less than or eqaul to
>	Greater than
>= , ≥	Greater than or equal to

```
julia> x=1
1
julia> y=2
2
julia> z=x<=2
true
julia>
```

Some additional functions for comparisons are available

Function	Test if
isequal(x,y)	x and y are identical
isfinite(x)	x is a finite number
isinf(x)	x is an infinite number
isnan(x)	x is a nan

Rounding functions

Function	Description	Return type
round(x)	round x to the nearest integer	typeof(x)
round(T, x)	round x to the nearest integer	T
floor(x)	round x towards -Inf	typeof(x)
floor(T,x)	round x towards -Inf	T
ceil(x)	round x towards +Inf	typeof(x)
ceil(T,x)	round x towards +Inf	T
trunc(x)	round x towards zero	typeof(x)
trunc(T,x)	round x towards zero	T

Division functions

Function	Description
div(x,y)	Truncated division, quotinent rounded towards zero
fld(x,y)	floored division, quotinent rounded towards -Inf
cld(x,y)	ceeling division, quotinent rounded towards Inf
rem(x,y)	remainder
mod(x,y)	modulus
mod1(x,y)	modulus with offset 1
Mod2p(x)	modulus with respect to 2π
divrem(x,y)	returns (div(x,y),rem(x,y)
Fldmod(x,y)	returns (div(x,y),mod(x,y)
gcd(x,y,…	greatest common divisor pf xy,..
Lcm(x,y,…)	least positive common multiple of x,y,…

Sign and absolute value functions

Function	Description
abs(x)	a positive value with the magnitude of x
abs2(x)	the squared magnitude of x
sign(x)	indicates the sign of x, returning -1, 0, or +1
signbit(x)	indicates whether the sign bit is on (true) or off (false)
copysign(x,y)	a value with the magnitude of x and the sign of y
flipsign(xy)	a value with the magnitude of x and the sign of x*y

Powers logs and roots

Function	Description
sqrt(x), √x	Square root of x
cbrt(x), ∛x	Cube root of x
hypot(x,y)	Hypotenuse of right-angled triangle $\sqrt{x^2 + y^2}$
exp(x)	natural exponent of x e^x
expm1(x)	$e^x - 1$
lxdexp(x,n)	$x2^n$ where n integer
log(x)	natural logarithm of x
log(b,x)	base b logarithm
log2(x)	base 2 logarithm
log10(x)	base 10 logarithm
log1p(x)	log(1+x)
exponent(x)	Binary exponent of x
Significant(x)	binary significand (a.k.a. mantissa) of a floating-point number x

```
julia> x=1
1
julia> y=exp(x)
2.718281828459045
julia>
```

Trigonometric and hyperbolic functions
sin cos tan cot sec csc sinh cosh tanh coth sech csch asin acos atan acot asec acsc asinh acosh atanh acoth asech acsch sinc cosc
Trigonometric functions in degree
sind cosd tand cotd secd cscd asind acosd atand acotd asecd acscd

```
julia> y=exp(x)
2.718281828459045
julia> x=sin(pi)
1.2246467991473532e-16
julia> y=cos(pi)
-1.0
julia>
```

Complex Numbers:

```
julia> x=1+2.5im
1.0 + 2.5im
julia> y=1.25+2im
1.25 + 2.0im
julia> z=x*y
-3.75 + 5.125im
julia> w=sin(x)
5.160143667579705 + 3.268939432079549im
julia>
```

Rational numbers

```
julia> x=6//9
2//3
julia> y=-4//8
-1//2
julia> z=x+y
1//6
julia>
```

Special functions

```
Pkg.add("SpecialFunctions")
```

Gamma Function

Function	**Description**
gamma(z)	gamma function \Gamma(z)Γ(z)
loggamma(x)	accurate log(gamma(x)) for large x
logabsgamma(x)	accurate log(abs(gamma(x))) for large x

logfactorial(x)	accurate log(factorial(x)) for large x; same as loggamma(x+1) for x > 1, zero otherwise
digamma(x)	digamma function (i.e. the derivative of loggamma at x)
invdigamma(x)	invdigamma function (i.e. inverse of digamma function at x using fixed-point iteration algorithm)
trigamma(x)	trigamma function (i.e the logarithmic second derivative of gamma at x)
polygamma(m,x)	polygamma function (i.e the (m+1)-th derivative of the loggamma function at x)
gamma(a,z)	upper incomplete gamma function \Gamma(a,z)Γ(a,z)
loggamma(a,z)	accurate log(gamma(a,x)) for large arguments
gamma_inc(a,x,IND)	incomplete gamma function ratio P(a,x) and Q(a,x) (i.e evaluates P(a,x) and Q(a,x)for accuracy specified by IND and returns tuple (p,q))
beta_inc(a,b,x,y)	incomplete beta function ratio Ix(a,b) and Iy(a,b) (i.e evaluates Ix(a,b) and Iy(a,b) and returns tuple (p,q))
gamma_inc_inv(a,p,q)	inverse of incomplete gamma function ratio P(a,x) and Q(a,x) (i.e evaluates x given P(a,x)=p and Q(a,x)=q
beta(x,y)	beta function at x,y
logbeta(x,y)	accurate log(beta(x,y)) for large x or y
logabsbeta(x,y)	accurate log(abs(beta(x,y))) for large x or y
logabsbinomial(x,y)	accurate log(abs(binomial(n,k))) for large n and k near n/2

Exponential and Trigonometric Integrals

Function	**Description**
expint(ν, z)	exponential integral \operatorname{E}_\nu(z)Eν(z)
expinti(x)	exponential integral \operatorname{Ei}(x)Ei(x)
expintx(x)	scaled exponential integral e^z \operatorname{E}_\nu(z)ezEν(z)
sinint(x)	sine integral \operatorname{Si}(x)Si(x)
cosint(x)	cosine integral \operatorname{Ci}(x)Ci(x)

Error Functions, Dawson's and Fresnel Integrals

Function	**Description**
erf(x)	error function at xx
erf(x,y)	erf(*y*)−erf(*x*)
erfc(x)	complementary error function, i.e. the accurate version of 1-
erfcinv(x)	inverse function to erfc()
erfcx(x)	scaled complementary error function, i.e. accurate e^{x^2}
logerfc(x)	log of the complementary error function, i.e.
logerfcx(x)	log of the scaled complementary error function, i.e.
erfi(x)	imaginary error function defined as -i
erfinv(x)	inverse function to erf()
dawson(x)	scaled imaginary error function, a.k.a. Dawson function, i.e.

Airy and Related Functions

Function	**Description**
airyai(z)	Airy Ai function at z

airyaiprime(z)	derivative of the Airy Ai function at z
airybi(z)	Airy Bi function at z
airybiprime(z)	derivative of the Airy Bi function at z
airyaix(z), airyaiprimex(z), airybix(z), airybiprimex(z)	scaled Airy Ai function and kth derivatives at z

Bessel Functions

Function	Description
besselj(nu,z)	Bessel function of the first kind of order nu at z
besselj0(z)	besselj(0,z)
besselj1(z)	besselj(1,z)
besseljx(nu,z)	scaled Bessel function of the first kind of order nu at z
sphericalbesselj(nu,z)	Spherical Bessel function of the first kind of order nu at z
bessely(nu,z)	Bessel function of the second kind of order nu at z
bessely0(z)	bessely(0,z)
bessely1(z)	bessely(1,z)
besselyx(nu,z)	scaled Bessel function of the second kind of order nu at z
sphericalbessely(nu,z)	Spherical Bessel function of the second kind of order nu at z
besselh(nu,k,z)	Bessel function of the third kind (a.k.a. Hankel function) of
hankelh1(nu,z)	besselh(nu, 1, z)
hankelh1x(nu,z)	scaled besselh(nu, 1, z)
hankelh2(nu,z)	besselh(nu, 2, z)
hankelh2x(nu,z)	scaled besselh(nu, 2, z)
besseli(nu,z)	modified Bessel function of the first kind of order nu at z
besselix(nu,z)	scaled modified Bessel function of the first kind of order nu at z
besselk(nu,z)	modified Bessel function of the second kind of order nu at z
besselkx(nu,z)	scaled modified Bessel function of the second kind of order nu at z
jinc(x)	scaled Bessel function of the first kind divided by x. A.k.a. sombrero

Elliptic Integrals

Function	Description
ellipk(m)	complete elliptic integral of 1st kind K(m)K(m)
ellipe(m)	complete elliptic integral of 2nd kind E(m)E(m)

Zeta and Related Functions

Function	Description
eta(x)	Dirichlet eta function at x
zeta(x)	Riemann zeta function at x

```
using SpecialFunctions
x=0:5.0
y1=besselj0.(x)
print("x = $x y = $y1")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" bessel2.jl
x = 0.0:1.0:5.0 y = [1.0, 0.7651976865579666, 0.22389077914123567, -0.2600519549019335, -0.3971498098638474, -
0.1775967713143383]
> Terminated with exit code 0.
```

Strings and character

Strings are finite sequences of characters. Of course, the real trouble comes when one asks what a character is. Chracters is letters, punctuations, numbers etc. In starting of computer times a standart called ASCII (American Standart code for Information Interchange) is developed which basicaly includes control characters, letters and numbers of american alphabet.

ASCII Control code chart

Binary	Oct	Dec	Hex	Abbreviation	[c]	[d]	Name (1967)

				1963	1965	###			
000 0000	0	0	0	NULL	NUL		^@	\0	Null
000 0001	1	1	1	SOM	SOH		^A		Start of Heading
000 0010	2	2	2	EOA	STX		^B		Start of Text
000 0011	3	3	3	EOM	ETX		^C		End of Text
000 0100	4	4	4	EOT			^D		End of Transmission
000 0101	5	5	5	WRU	ENQ		^E		Enquiry
000 0110	6	6	6	RU	ACK		^F		Acknowledgement
000 0111	7	7	7	BELL	BEL		^G	\a	Bell
000 1000	10	8	8	FE0	BS		^H	\b	Backspace[e][f]
000 1001	11	9	9	HT/SK	HT		^I	\t	Horizontal Tab[g]
000 1010	12	10	0A	LF			^J	\n	Line Feed
000 1011	13	11	0B	VTAB	VT		^K	\v	Vertical Tab
000 1100	14	12	0C	FF			^L	\f	Form Feed
000 1101	15	13	0D	CR			^M	\r	Carriage Return[h]
000 1110	16	14	0E	SO			^N		Shift Out
000 1111	17	15	0F	SI			^O		Shift In
10 000	20	16	10	DC0	DLE		^P		Data Link Escape
10 001	21	17	11	DC1			^Q		Device Control
10 010	22	18	12	DC2			^R		Device Control 2
10 011	23	19	13	DC3			^S		Device Control
10 100	24	20	14	DC4			^T		Device Control 4
10 101	25	21	15	ERR	NAK		^U		Negative
10 110	26	22	16	SYNC	SYN		^V		Synchronous Idle
10 111	27	23	17	LEM	ETB		^W		End of Transmission Block
11 000	30	24	18	S0	CAN		^X		Cancel
11 001	31	25	19	S1	EM		^Y		End of Medium
11 010	32	26	1A	S2	SS	SUB	^Z		Substitute
11 011	33	27	1B	S3	ESC		^[	\e[i]	Escape[j]
11 100	34	28	1C	S4	FS		^\		File Separator
11 101	35	29	1D	S5	GS		^]		Group Separator
11 110	36	30	1E	S6	RS		^^[k]		Record Separator
11 111	37	31	1F	S7	US		^_		Unit Separator
1 111 111	177	127	7F	DEL			^?		Delete[l][f]

ASCII Character chart

100 000	40	32	20	space
100 001	41	33	21	!
100 010	42	34	22	"
100 011	43	35	23	#
100 100	44	36	24	$
100 101	45	37	25	%
100 110	46	38	26	&
100 111	47	39	27	'
101 000	50	40	28	(
101 001	51	41	29	)
101 010	52	42	2A	*
101 011	53	43	2B	+
101 100	54	44	2C	,
101 101	55	45	2D	-
101 110	56	46	2E	.
101 111	57	47	2F	/
110 000	60	48	30	0
110 001	61	49	31	1
110 010	62	50	32	2
110 011	63	51	33	3
110 100	64	52	34	4
110 101	65	53	35	5
110 110	66	54	36	6
110 111	67	55	37	7

111 000	70	56	38	8		
111 001	71	57	39	9		
111 010	72	58	3A	:		
111 011	73	59	3B	;		
111 100	74	60	3C	<		
111 101	75	61	3D	=		
111 110	76	62	3E	>		
111 111	77	63	3F	?		
1 000 000	100	64	40	@	`	@
1 000 001	101	65	41	A		
1 000 010	102	66	42	B		
1 000 011	103	67	43	C		
1 000 100	104	68	44	D		
1 000 101	105	69	45	E		
1 000 110	106	70	46	F		
1 000 111	107	71	47	G		
1 001 000	110	72	48	H		
1 001 001	111	73	49	I		
1 001 010	112	74	4A	J		
1 001 011	113	75	4B	K		
1 001 100	114	76	4C	L		
1 001 101	115	77	4D	M		
1 001 110	116	78	4E	N		
1 001 111	117	79	4F	O		
1 010 000	120	80	50	P		
1 010 001	121	81	51	Q		
1 010 010	122	82	52	R		
1 010 011	123	83	53	S		
1 010 100	124	84	54	T		
1 010 101	125	85	55	U		
1 010 110	126	86	56	V		
1 010 111	127	87	57	W		
1 011 000	130	88	58	X		
1 011 001	131	89	59	Y		
1 011 010	132	90	5A	Z		
1 011 011	133	91	5B	[		
1 011 100	134	92	5C	\	~	\
1 011 101	135	93	5D	]		
1 011 110	136	94	5E	↑	^	
1 011 111	137	95	5F	←	_	
1 100 000	140	96	60		@	`
1 100 001	141	97	61		a	
1 100 010	142	98	62		b	
1 100 011	143	99	63		c	
1 100 100	144	100	64		d	
1 100 101	145	101	65		e	
1 100 110	146	102	66		f	
1 100 111	147	103	67		g	
1 101 000	150	104	68		h	
1 101 001	151	105	69		i	
1 101 010	152	106	6A		j	
1 101 011	153	107	6B		k	
1 101 100	154	108	6C		l	
1 101 101	155	109	6D		m	
1 101 110	156	110	6E		n	
1 101 111	157	111	6F		o	
1 110 000	160	112	70		p	
1 110 001	161	113	71		q	
1 110 010	162	114	72		r	
1 110 011	163	115	73		s	
1 110 100	164	116	74		t	

1 110 101	165	117	75		u	
1 110 110	166	118	76		v	
1 110 111	167	119	77		w	
1 111 000	170	120	78		x	
1 111 001	171	121	79		y	
1 111 010	172	122	7A		z	
1 111 011	173	123	7B		{	
1 111 100	174	124	7C	ACK	¬	|
1 111 101	175	125	7D		}	
1 111 110	176	126	7E	ESC	¦	~

ASCII code does not include the characters of the other languages. A more extended coding for characters developed by Unicode organisation (Unicode.org)

Table Some unicode character codes

	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
5	ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
6	ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
7	BEL 0007	ETB 0017	' 0027	7 0037	G 0047	W 0057	g 0067	w 0077
8	BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
9	HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
A	LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
B	VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
C	FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
D	CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
E	SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
F	SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

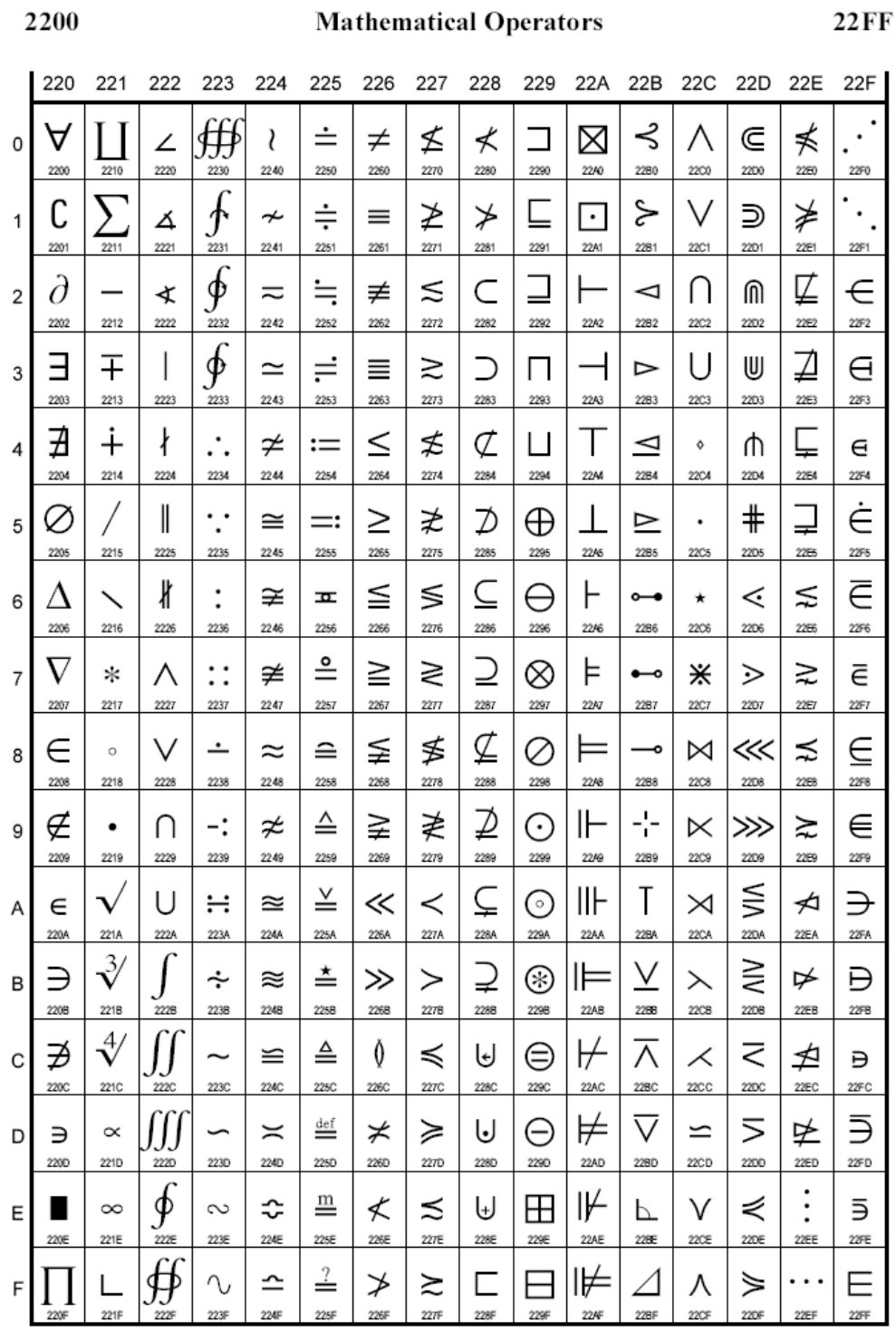
	008	009	00A	00B	00C	00D	00E	00F
0	XXX 0080	DCS 0090	NB SP 00A0	° 00B0	À 00C0	Đ 00D0	à 00E0	ð 00F0
1	XXX 0081	PU1 0091	¡ 00A1	± 00B1	Á 00C1	Ñ 00D1	á 00E1	ñ 00F1
2	BPH 0082	PU2 0092	¢ 00A2	2 00B2	Â 00C2	Ò 00D2	â 00E2	ò 00F2
3	NBH 0083	STS 0093	£ 00A3	3 00B3	Ã 00C3	Ó 00D3	ã 00E3	ó 00F3
4	IND 0084	CCH 0094	¤ 00A4	´ 00B4	Ä 00C4	Ô 00D4	ä 00E4	ô 00F4
5	NEL 0085	MW 0095	¥ 00A5	µ 00B5	Å 00C5	Õ 00D5	å 00E5	õ 00F5
6	SSA 0086	SPA 0096	¦ 00A6	¶ 00B6	Æ 00C6	Ö 00D6	æ 00E6	ö 00F6
7	ESA 0087	EPA 0097	§ 00A7	· 00B7	Ç 00C7	× 00D7	ç 00E7	÷ 00F7
8	HTS 0088	SOS 0098	¨ 00A8	¸ 00B8	È 00C8	Ø 00D8	è 00E8	ø 00F8
9	HTJ 0089	XXX 0099	© 00A9	1 00B9	É 00C9	Ù 00D9	é 00E9	ù 00F9
A	VTS 008A	SCI 009A	ª 00AA	º 00BA	Ê 00CA	Ú 00DA	ê 00EA	ú 00FA
B	PLD 008B	CSI 009B	« 00AB	» 00BB	Ë 00CB	Û 00DB	ë 00EB	û 00FB
C	PLU 008C	ST 009C	¬ 00AC	¼ 00BC	Ì 00CC	Ü 00DC	ì 00EC	ü 00FC
D	RI 008D	OSC 009D	SHY 00AD	½ 00BD	Í 00CD	Ý 00DD	í 00ED	ý 00FD
E	SS2 008E	PM 009E	® 00AE	¾ 00BE	Î 00CE	Þ 00DE	î 00EE	þ 00FE
F	SS3 008F	APC 009F	¯ 00AF	¿ 00BF	Ï 00CF	ß 00DF	ï 00EF	ÿ 00FF

0180 **Latin Extended-B** **024**

	018	019	01A	01B	01C	01D	01E	01F	020	021	022	023	024
0	ƀ 0180	Ɛ 0190	Ơ 01A0	ư 01B0	ǀ 01C0	ǐ 01D0	Ā 01E0	ǰ 01F0	À 0200	Ȑ 0210	ɳ 0220	Ō 0230	ɀ 0240
1	Ɓ 0181	Ƒ 0191	ơ 01A1	Ʊ 01B1	ǁ 01C1	Ǒ 01D1	ā 01E1	DZ 01F1	à 0201	ȑ 0211	ɖ 0221	ō 0231	ɂ 0241
2	Ƃ 0182	ƒ 0192	Ƣ 01A2	Ʋ 01B2	ǂ 01C2	ǒ 01D2	Ē 01E2	Dz 01F2	Â 0202	Ȓ 0212	ȣ 0222	Ȳ 0232	ɐ 0242
3	ƃ 0183	Ɠ 0193	ƣ 01A3	Ƴ 01B3	! 01C3	Ǔ 01D3	ē 01E3	dz 01F3	â 0203	î 0213	ȣ 0223	ȳ 0233	Ƀ 0243
4	Ƅ 0184	ƔƳ 0194	Ƥ 01A4	ƴ 01B4	DŽ 01C4	ǔ 01D4	Ġ 01E4	Ǵ 01F4	È 0204	Ù 0214	Ȝ 0224	ɬ 0234	Ʉ 0244
5	ƅ 0185	ƕ 0195	ƥ 01A5	Ƶ 01B5	Dž 01C5	Ǖ 01D5	ġ 01E5	ǵ 01F5	è 0205	ù 0215	ȝ 0225	ɱ 0235	Ʌ 0245
6	Ɔ 0186	ɬ 0196	Ʀ 01A6	ƶ 01B6	dž 01C6	ǖ 01D6	Ǧ 01E6	Ƕ 01F6	Ê 0206	Û 0216	Ȧ 0226	ɐ 0236	Ɇ 0246
7	Ƈ 0187	Ɨ 0197	Ƨ 01A7	Ʒ 01B7	LJ 01C7	Ǘ 01D7	ǧ 01E7	ƿ 01F7	ê 0207	û 0217	ȧ 0227	ɟ 0237	ɇ 0247
8	ƈ 0188	Ƙ 0198	ƨ 01A8	Ǯ 01B8	Lj 01C8	ǘ 01D8	Ǩ 01E8	Ǹ 01F8	Ì 0208	Ș 0218	Ȩ 0228	ɔ̈ 0238	Ɉ 0248
9	Ɖ 0189	ƙ 0199	Σ 01A9	ǯ 01B9	lj 01C9	Ǚ 01D9	ǩ 01E9	ǹ 01F9	ì 0209	ș 0219	ȩ 0229	ȹ 0239	ɉ 0249
A	Ɗ 018A	ƚ 019A	ƪ 01AA	ƺ 01BA	NJ 01CA	ǚ 01DA	Ǫ 01EA	Ǻ 01FA	Î 020A	Ț 021A	Ɔ̃ 022A	Ⱥ 023A	ɑ̣ 024A
B	Ƌ 018B	ƛ 019B	ƫ 01AB	Ƹ 01BB	Nj 01CB	Ǜ 01DB	ǫ 01EB	ǻ 01FB	î 020B	ț 021B	ȫ 022B	Ȼ 023B	ɑ̤ 024B
C	ƌ 018C	Ɯ 019C	Ƭ 01AC	Ʒ 01BC	nj 01CC	ǜ 01DC	Ǭ 01EC	Ǽ 01FC	Ò 020C	Ȝ 021C	Ɔ̄ 022C	ȼ 023C	Ʀ 024C
D	ƍ 018D	Ɲ 019D	ƭ 01AD	ƽ 01BD	Ǎ 01CD	ǝ 01DD	ǭ 01ED	ǽ 01FD	ò 020D	ȝ 021D	ō̃ 022D	Ł 023D	ɾ 024D
E	Ǝ 018E	ŋ 019E	Ʈ 01AE	ƾ 01BE	ǎ 01CE	Ǟ 01DE	Ǯ 01EE	Ǿ 01FE	Ô 020E	Ȟ 021E	Ȯ 022E	ƚ 023E	Ɏ 024E
F	Ə 018F	Ɵ 019F	Ư 01AF	ƿ 01BF	Ǐ 01CF	ǟ 01DF	ǯ 01EF	ǿ 01FF	ô 020F	ȟ 021F	ȯ 022F	ȿ 023F	ɏ 024F

0370 **Greek and Coptic** **03FF**

	037	038	039	03A	03B	03C	03D	03E	03F
0	Ͱ 0370		ΐ 0390	Π 03A0	ΰ 03B0	π 03C0	ϐ 03D0	Ϣ 03E0	ϰ 03F0
1	ͱ 0371		Α 0391	Ρ 03A1	α 03B1	ρ 03C1	ϑ 03D1	ϣ 03E1	ϱ 03F1
2	Ͳ 0372		Β 0392		β 03B2	ς 03C2	ϒ 03D2	Ϥ 03E2	ϲ 03F2
3	ͳ 0373		Γ 0393	Σ 03A3	γ 03B3	σ 03C3	ϓ 03D3	ϥ 03E3	ϳ 03F3
4	ʹ 0374	΄ 0384	Δ 0394	Τ 03A4	δ 03B4	τ 03C4	ϔ 03D4	Ϧ 03E4	ϴ 03F4
5	͵ 0375	΅ 0385	Ε 0395	Υ 03A5	ε 03B5	υ 03C5	ϕ 03D5	ϧ 03E5	ϵ 03F5
6	Ͷ 0376	Ά 0386	Ζ 0396	Φ 03A6	ζ 03B6	φ 03C6	ϖ 03D6	Ϩ 03E6	϶ 03F6
7	ͷ 0377	· 0387	Η 0397	Χ 03A7	η 03B7	χ 03C7	ϗ 03D7	ϩ 03E7	Ϸ 03F7
8		Έ 0388	Θ 0398	Ψ 03A8	θ 03B8	ψ 03C8	Ϙ 03D8	Ϫ 03E8	ϸ 03F8
9		Ή 0389	Ι 0399	Ω 03A9	ι 03B9	ω 03C9	ϙ 03D9	ϫ 03E9	Ϲ 03F9
A	ͺ 037A	Ί 038A	Κ 039A	Ϊ 03AA	κ 03BA	ϊ 03CA	Ϛ 03DA	Ϭ 03EA	Ϻ 03FA
B	ͻ 037B		Λ 039B	Ϋ 03AB	λ 03BB	ϋ 03CB	ϛ 03DB	ϭ 03EB	ϻ 03FB
C	ͼ 037C	Ό 038C	Μ 039C	ά 03AC	μ 03BC	ό 03CC	Ϝ 03DC	Ϯ 03EC	ϼ 03FC
D	ͽ 037D		Ν 039D	έ 03AD	ν 03BD	ύ 03CD	ϝ 03DD	ϯ 03ED	Ͻ 03FD
E	; 037E	Ύ 038E	Ξ 039E	ή 03AE	ξ 03BE	ώ 03CE	Ϟ 03DE	ϰ 03EE	Ͼ 03FE
F		Ώ 038F	Ο 039F	ί 03AF	ο 03BF	ϗ 03CF	ϟ 03DF	ϱ 03EF	Ͽ 03FF



Characters either ASCII or Unicode, can be defined by single quote sign. Characters can be combined to create String variables

```
julia> c1='ü'
'ü': Unicode U+00FC (category Ll: Letter, lowercase)
julia> c2='\u00c7'
'Ç': Unicode U+00C7 (category Lu: Letter, uppercase)
julia> c3='x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

```
julia> c4='y'
'y': ASCII/Unicode U+0079 (category Ll: Letter, lowercase)

julia> s=string(c1,c2,c3,c4)
"üÇxy"
julia> s1="ü \u00c7 x y"
"ü Ç x y"
julia>
```

```
julia> b1='ü'
'ü': Unicode U+00FC (category Ll: Letter, lowercase)

julia> b2='ğ'
'ğ': Unicode U+011F (category Ll: Letter, lowercase)

julia> b3='ç'
'ç': Unicode U+00E7 (category Ll: Letter, lowercase)

julia> b4='a'
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> b5='b'
'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)

julia> b6='c'
'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)

julia> d="$b1$b2$b3$b4$b5$b6"
"üğçabc"
```

```
julia> b1='\u03B1'
'α': Unicode U+03B1 (category Ll: Letter, lowercase)

julia> b2='\u03B2'
'β': Unicode U+03B2 (category Ll: Letter, lowercase)

julia> b3='\u03B3'
'γ': Unicode U+03B3 (category Ll: Letter, lowercase)

julia> b4='\u03B4'
'δ': Unicode U+03B4 (category Ll: Letter, lowercase)

julia> b5='\u03B5'
'ε': Unicode U+03B5 (category Ll: Letter, lowercase)

julia> b6='\u03B6'
'ζ': Unicode U+03B6 (category Ll: Letter, lowercase)

julia> d="$b1$b2$b3$b4$b5$b6"
"αβγδεζ"
```

Welcome message (String)

```
julia> print("Welcome to Julia programming language")
Welcome to Julia programming language
julia>
```

```
julia> x="Welcome to Julia Programming language"

"Welcome to Julia Programming language"

julia> print(x)

Welcome to Julia Programming language

julia>
```

In order to combine different string together (concatenation) * is used

```
julia> x="Hello"*" world"*"\n"
"Hello world\n"

julia> print(x)
```

Constructing strings using concatenation can become a bit cumbersome, however. To reduce the need for these verbose calls to string or repeated multiplications, Julia allows interpolation into string literals using $,

```
julia> s1="Hello"
"Hello"
julia> s2="World"
"World"
julia> x="$s1,$s2\n"
"Hello,World\n"
```

The shortest complete expression after the $ is taken as the expression whose value is to be interpolated into the string. Thus, you can interpolate any expression into a string using parentheses:

```
julia> x=1
1
julia> y=2
2
julia> s="x+y=$(x+y)"
"x+y=3"

julia> c1='ü'
'ü': Unicode U+00FC (category Ll: Letter, lowercase)

julia> c2='\u00c7'
'Ç': Unicode U+00C7 (category Lu: Letter, uppercase)

julia> x="Hello,$c1,$c2"
"Hello,ü,Ç"
```

Longer than one line of strings can be defined by using tripple quote “””

```
julia> s1="""
      Hello Friend,
      how are you today?
      I am quite fine thanks
      """
"Hello Friend,\nhow are you today?\nI am quite fine thanks\n"
```

Strings can be compare by using the same comparison signs given in the table above.

```
julia> "1+2=3"=="1+2=$(1+2)"
true

julia> "Turhan">"turhan"
false
```

You can search for the index of a particular character using the findfirst and findlast functions:

```
julia> x="Ali veli 49 elli"
"Ali veli 49 elli"
julia> findfirst(isequal('e'),x)
6
julia> findlast(isequal('e'),x)
13
```

You can use the occursin function to check if a substring is found within a string:

```
julia> x="Ali veli 49 elli"
"Ali veli 49 elli"

julia> occursin("veli",x)
true
```

Repeat is a usefull string function

```
julia> repeat("Turhan ",3)
```

Some other useful String functions:

• firstindex(str) gives the minimal (byte) index that can be used to index into str (always 1 for strings, not necessarily true for other containers).
• lastindex(str) gives the maximal (byte) index that can be used to index into str.
• length(str) the number of characters in str.
• length(str, i, j) the number of valid character indices in str from i to j.
• ncodeunits(str) number of code units in a string.
• codeunit(str, i) gives the code unit value in the string str at index i.
• thisind(str, i) given an arbitrary index into a string find the first index of the character into which the index points.
• nextind(str, i, n=1) find the start of the nth character starting after index i. • prevind(str, i, n=1) find the start of the nth character starting before index i.

FUNCTIONS

In Julia, a function is an object that maps a tuple of argument values to a return value.

```
function f(x,y)
  x+y
  end

x=1.2
y=2.3
z=f(x,y)
print("z=",z)
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func1.jl
z=3.5
> Terminated with exit code 0.
```

```
function f(x,y)
    return x*y
end

x=1.2
y=2.3
z=f(x,y)
s="x=$(x) y=$(y) z=$z"
print(s)
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func1.jl
x=1.2 y=2.3 z=2.76
> Terminated with exit code 0.
```

```
julia> s(x,y)=x+y
s (generic function with 1 method)
julia> s(2,3.2)
5.2
```

```
function hypot(x,y)
x = abs(x)
y = abs(y)
if x > y
```

```
r = y/x
return x*sqrt(1+r*r)
end
if y == 0
return zero(x)
end
r = x/y
return y*sqrt(1+r*r)
end

x=1.2
y=2.3
z=hypot(x,y)
s="x=$(x) y=$(y) z=$z"
print(s)
```

```
---------- Capture Output ----------

> "E:\co\Julia\bin\julia.exe"  func2.jl

x=1.2 y=2.3 z=2.5942243542145693

> Terminated with exit code 0.
```

For functions that do not need to return a value (functions used only for some side effects), the Julia convention is to return the value nothing:

```
function printx(x)
println("x = $x")
return nothing
end

x=1.23
printx(x)
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func3.jl
x = 1.23

> Terminated with exit code 0.
```

Function output type can be defined

```
function g(x, y)::Int8
return x * y
end;

z=g(1, 2)
w=typeof(z)
print("z = $z w = $w")
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func5.jl
z = 2 w = Int8
> Terminated with exit code 0.
```

Tupples

Julia has a built-in data structure called a tupple that is closely related to function arguments and return values. A tuple is a fixed-length container that can hold any values, but cannot be modified (it is immutable). Tuples are constructed with commas and parentheses, and can be accessed via indexing: Tupples are in a way similar to array variables in languages like java or c++. But unlike these languages tupples can hold different types of data in its structures.

```
julia> x=(1.2,"hello",6*7)
(1.2, "hello", 42)

julia> x[2]
"hello"

julia> x[1]
```

```
1.2
```

It should also be note that topple references started from 1 (not 0)

The components of tuples can optionally be named, in which case a named tuple is constructed:

```
julia> x=(a=1.2,b="hello",c=6*7)
(a = 1.2, b = "hello", c = 42)

julia> x.a
1.2

julia> x[1]
1.2

julia> x.b
"hello"

julia> x[2]
"hello"

julia>
```

In Julia, one returns a tuple of values to simulate returning multiple values. However, tuples can be created and destructured without needing parentheses, thereby providing an illusion that multiple values are being returned, rather than a single tuple value. For example, the following function returns a pair of values:

```
function g(x, y)
return x * y,x+y
end;

z=g(1, 2)
print("z = $z")
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func6.jl
z = (2, 3)
> Terminated with exit code 0.
```

It is often possible to provide sensible default values for function arguments. This can save users from having to pass every argument on every call. For example, the function Date(y, [m, d]) from Dates module constructs a Date type for a given year y, month m and day d. However, m and d arguments are optional and their default value is 1. This behavior can be expressed concisely as:

```
function Date1(y::Int64, m::Int64=1, d::Int64=1)
err = validargs(Date, y, m, d)
err === nothing || throw(err)
return Date(UTD(totaldays(y, m, d)))
end

using Dates
d1=Date(2020,12,15)
println("d1 $d1")
d2=Date(2020,12)
println("d2 $d2")
d3=Date(2020)
println("d3 $d3")
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func7.jl
d1 2020-12-15
```

```
d2 2020-12-01
d3 2020-01-01

> Terminated with exit code 0.
```

Compound expressions:

If more than one expression should be defined and execute as a block. This can be done two ways:

The first method is begin end block

```
julia> z=begin
       x=1
       y=2
       z=3
       x+y*z
       end
7

julia>
```

The second one is chain synthax

```
julia> w=(x=1;y=2;z=3;x+y*z)
7

julia>
```

Conditional evaluation (if block):

```
function greater(x,y)
b=true
if x<y
   b=false
elseif x==y
   b=false
return b
end
end

x=1
y=2
c=greater(x,y)
println("c =$c")
```

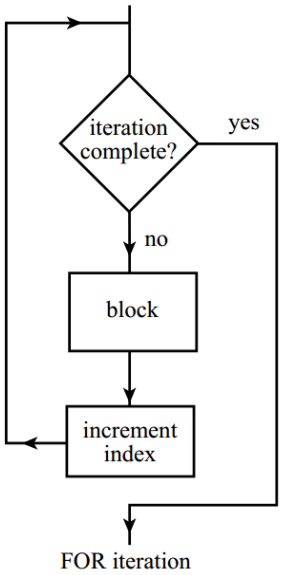
```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func8.jl
c =false

> Terminated with exit code 0.
```

```
function test(x,y)
a="greater than"
if x<y
   a="smaller than"
elseif x==y
   a="equal to"
end
b="$x is $a $y"
return b
end
```

```
x=1
y=2
c=test(x,y)
println(c)
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func9.jl
1 is smaller than 2

> Terminated with exit code 0.
```

For statement and loop structure



Sum of numbers 1 to n

```
function sum(n)
s = 0 # new local
for i = 1:n
s = s + i # assign existing local
end
return s # same local
end

x=100
a=sum(x)
print("s = $a")
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func10.jl
s = 5050
> Terminated with exit code 0.
```

Numbers can be added up by sum function

```
x=[1.1179,2.21,3.31,4.41,5.51,6.61]
a=sum(x)
print("a = $a")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" mean.jl
a = 23.1679
```

```
> Terminated with exit code 0.
```

Average of number set

```
function average(x)
s = 0 # new local
n=length(x)
for i  in x
s = s + i
end
s=s/n
return s # same local
end

x=[1.1179,2.21,3.31,4.41,5.51,6.61]
a=average(x)
print("a = $a")
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func10.jl
a = 3.8613166666666667
> Terminated with exit code 0.
```

The same thing can be achived by using mean function of Statistics library

```
using Statistics
x=[1.1179,2.21,3.31,4.41,5.51,6.61]
a=mean(x)
print("a = $a")
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" mean.jl
a = 3.8613166666666667
> Terminated with exit code 0.
```

Printing string values:

```
a=["ali","veli","49","elli"]
for i in a
   println("i =",i)
end
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func11.jl
i =ali
i =veli
i =49
i =elli

> Terminated with exit code 0.
```

List of greek characters

```
julia> c1='\u03B1'
'α': Unicode U+03B1 (category Ll: Letter, lowercase)

julia> c2='\u03D2'
'ϒ': Unicode U+03D2 (category Lu: Letter, uppercase)

julia> for i=c1:c2
       print(" ",i)
```

```
    end
 α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ ς σ τ υ φ χ ψ ω ϊ ϋ ό ύ ώ Ϗ ϐ ϑ ϒ
```

List of ASCII characters

```
c1='\u0021'
c2='\u007E'
for i =c1:c2
   print("  ",i)
end
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func12.jl
 ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K
L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v
w x y z { | } ~
> Terminated with exit code 0.
```

While statement
While loop statement basically carry out the same duty as for statement

Listing of ASCII characters

```
c1='\u0021'
c2='\u007E'
i=c1
while i<c2
    print("  ",i)
   global i+=1
end
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func12.jl
 ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D
E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f
g h i j k l m n o p q r s t u v w x y z { | }
> Terminated with exit code 0.
```

Listing of greek characters in unicode

```
julia> c1='\u03B1'
'α': Unicode U+03B1 (category Ll: Letter, lowercase)

julia> c2='\u03D2'
'ϒ': Unicode U+03D2 (category Lu: Letter, uppercase)

julia> i=c1
'α': Unicode U+03B1 (category Ll: Letter, lowercase)

julia> while i<=c2
       print(" ",i)
       global i+=1
       end
 α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ ς σ τ υ φ χ ψ ω ϊ ϋ ό ύ ώ Ϗ ϐ ϑ ϒ
julia>
```

Finding average value

```
function average(x)
s = 0 # new local
```

```
n=length(x)
i=1
s=0
while i<=n
s = s + x[i]
i=i+1
end
s=s/n
return s # same local
end

x=[1.1179,2.21,3.31,4.41,5.51,6.61]
a=average(x)
print("a = $a")
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func12.jl
a = 3.8613166666666667
> Terminated with exit code 0.
```

Anonymous function (Lambda variables??)

Array process:

```
fun = (x,y,z)->2x+y-z
x=[1,2,3]
y=[2,3,4]
z=[1,3,5]
w=fun(x, y, z)
print(w)
```

Tupple process

```
fun = (x,y,z)->2x+y-z
x=(1,2,3)
y=(2,3,4)
z=(1,3,5)
w=map(fun,x, y, z)
print(w)
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func14.jl
(3, 4, 5)
> Terminated with exit code 0.
```

Read data from screen and convert it to Float64: (a line of data seperated with ,)

```
print("enter x = ")
x=readline()
y=split(x,",")
z=map(x->(v = tryparse(Float64,x)),y)
print(z)
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func15.jl
enter x1,2,3,4
```

```
[1.0, 2.0, 3.0, 4.0]
> Terminated with exit code 0.
```

Read data from screen and convert it to Float64: (a line of data seperated with space)

```
print("enter x = ")
x=readline()
y=split(x," ")
z=map(x->(v = tryparse(Float64,x)),y)
print(z)
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func26.jl
enter x = 1 2 3 4 5
[1.0, 2.0, 3.0, 4.0, 5.0]
> Terminated with exit code 0.
```

Read a single data from screen and convert it to Float64

```
print("enter x = ")
x=readline()
z=tryparse(Float64,x)
print(z)
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func26.jl
enter x = 1.253453
1.253453
> Terminated with exit code 0.
```

Anonymous function (Lambda variables??)

```
function readdouble(s)
print(s)
x=readline()
y=split(x,",")
z=map(x->(v = tryparse(Float64,x)),y)
return z
end

fun = (x,y,z)->2x+y-z
x=readdouble("x=")
y=readdouble("y=")
z=readdouble("z=")
w=map(fun,x, y, z)
print(w)
```

```
---------- Capture Output ----------
> "E:\co\Julia\bin\julia.exe"  func17.jl
x=1,2,3
y=4,5,6
z=7,8,9
[-1.0, 1.0, 3.0]
> Terminated with exit code 0.
```

x=1.4142135623719696 y=-3.183231456205249e-12
x=1.4142135623719696 y=-3.183231456205249e-12

Arrays and matrices in Julia

Arrays and array dimensions can be defined by using several ways. The most inportant point that should be mentioned here is that array index in Julia start from 1. In languages like C,C++,java,python array index start from zero.

```
B=Array{Float64}(undef,2)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), B)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array1.jl
2-element Vector{Float64}:
 0.0
 0.0
> Terminated with exit code 0.
```

```
B=Array{Float64}(undef,4)
n=size(B)[1]
println("n = $n")
for i=1:n
  y=B[i]
  println("B[$i] = $y")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array2.jl
n = 4
B[1] = 0.0
B[2] = 0.0
B[3] = 0.0
B[4] = 0.0

> Terminated with exit code 0.
```

```
B=zeros(Float64,4)
n=size(B)[1]
println("n = $n")
for i=1:n
  y=B[i]
  println("B[$i] = $y")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array3.jl
n = 4
B[1] = 0.0
B[2] = 0.0
B[3] = 0.0
B[4] = 0.0

> Terminated with exit code 0.
```

```
B=ones(Float64,4)
n=size(B)[1]
println("n = $n")
for i=1:n
  y=B[i]
  println("B[$i] = $y")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array3.jl
n = 4
B[1] = 1.0
B[2] = 1.0
B[3] = 1.0
B[4] = 1.0

> Terminated with exit code 0.
```

```
B=[1.1,2.2,3.3,4.4]
n=size(B)[1]
println("n = $n")
for i=1:n
  y=B[i]
  println("B[$i] = $y")
```

```
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array4.jl
n = 4
B[1] = 1.1
B[2] = 2.2
B[3] = 3.3
B[4] = 4.4

> Terminated with exit code 0.
```

If it is two dimensional array, it is given as:

```
A = Array{Float64,2}(undef, 2, 3)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(),A)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array2D_1.jl
2X3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
> Terminated with exit code 0.
```

```
julia> include("array2D_1.jl")
2×3 Matrix{Float64}:
 1.18628e-314  1.18628e-314  1.18628e-314
 1.18628e-314  1.18628e-314  1.0e-323
julia>
```

```
A = Array{Float64,2}(undef, 2, 3)
n=size(A)[1]
m=size(A)[2]
println("n = $n m = $m")
for i=1:n
  for j=1:m
  y=A[i,j]
  print("A[$i,$j] = $y ")
  end
  println(" ")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array2D_2.jl
n = 2 m = 3
A[1,1] = 0.0 A[1,2] = 0.0 A[1,3] = 0.0
A[2,1] = 0.0 A[2,2] = 0.0 A[2,3] = 0.0
> Terminated with exit code 0.
```

```
A = Array{Float64,2}(undef, 2, 3)
print(A)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array2D_3.jl
[0.0 0.0 0.0; 0.0 0.0 0.0]
> Terminated with exit code 0.
```

```
A = zeros(Float64,2, 3)
print(A)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array2D_4.jl
[0.0 0.0 0.0; 0.0 0.0 0.0]
> Terminated with exit code 0.
```

```
A = ones(Float64,2, 3)
print(A)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array2D_4.jl
[1.0 1.0 1.0; 1.0 1.0 1.0]
> Terminated with exit code 0.
```

```
A = fill(5.34,(2,3))
show(IOContext(stdout, :limit=>false), MIME"text/plain"(),A)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array2D_5.jl
2X3 Matrix{Float64}:
```

```
 5.34  5.34  5.34
 5.34  5.34  5.34
> Terminated with exit code 0.
```

```
A = Matrix{Float64}(undef, 2, 3)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(),A)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" array2D_6.jl
2X3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
> Terminated with exit code 0.
```

If, it is desired that array indis start from some other values rather than 1, class OffsetArrays can be used. If a program originally written in C,C++,java or python array indices started from 1 so sometimes it is easier to use it that way instead of translatign all the index variables form 0 to 1. Package OffsetArrays is used for this purpose.

```
import Pkg
Pkg.add("OffsetArrays")
```

```
 using OffsetArrays
A=OffsetArray{Float64}(undef,0:2,0:3)
n=size(A)[1]
m=size(A)[2]
for i=0:n-1
  for j=0:m-1
  y=10*i+j
  A[i,j]=y
  end
end
print("A= $A")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(),A)
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" zeroarray_2.jl
A= [0.0 1.0 2.0 3.0; 10.0 11.0 12.0 13.0; 20.0 21.0 22.0 23.0]
3X4 OffsetArray(::
Matrix{Float64}, 0:2, 0:3) with eltype Float64 with indices 0:2X0:3:
  0.0   1.0   2.0   3.0
 10.0  11.0  12.0  13.0
 20.0  21.0  22.0  23.0
> Terminated with exit code 0.
```

```
function print_OA(A)
 n=size(A)[1]
 m=size(A)[2]
 for i=0:n-1
  for j=0:m-1
    y=A[i,j]
    print("($i,$j)$y ")
  end
  println(" ")
end
end

using OffsetArrays
A=OffsetArray{Float64}(undef,0:2,0:3)
n=size(A)[1]
m=size(A)[2]
for i=0:n-1
  for j=0:m-1
  y=10*i+j
  A[i,j]=y
  end
end
print_OA(A)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" zeroarray_2.jl
(0,0)0.0 (0,1)1.0 (0,2)2.0 (0,3)3.0
```

```
(1,0)10.0 (1,1)11.0 (1,2)12.0 (1,3)13.0
(2,0)20.0 (2,1)21.0 (2,2)22.0 (2,3)23.0

> Terminated with exit code 0.
```

Plotting data

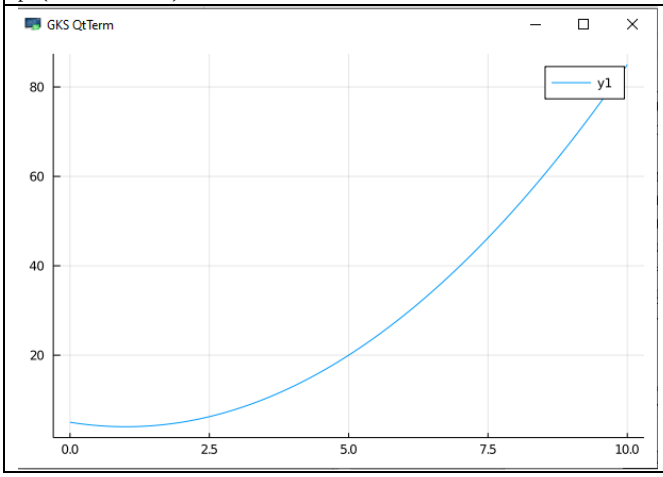
A rich plotting environment is existed in Julia. There are several packages to utilised.The basic Plot Packages are:
Plots
PyPlot
GR
PlotlyJS

In order to use a plot package it should be loaded up. For example, in order to add Plots package to julia use:

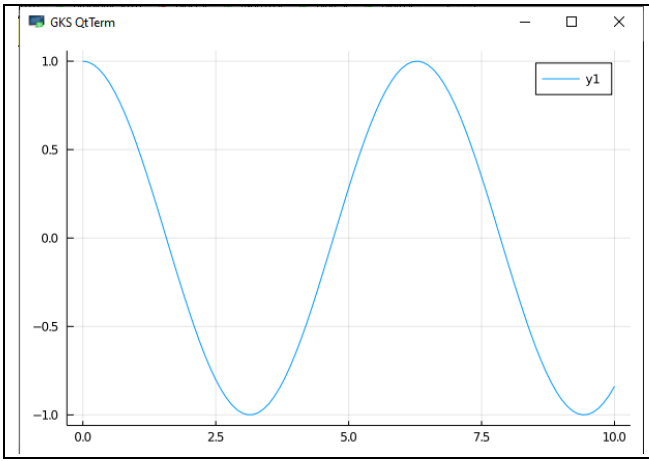
```
using Pkg
Pkg.add("Plots")
```

Julia>include("Plot1.jl")

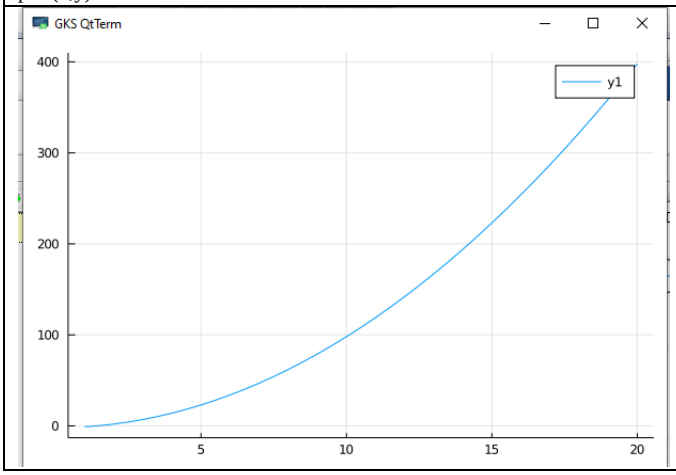
```
function plt(f,x0,x1,dx)
x=x0:dx:x1
y=map(f,x)
plot(x,y)
end

using Plots
f1(x)=x*x-2.0*x+5.0
plt(f1,0.0,10.0,0.1)
```



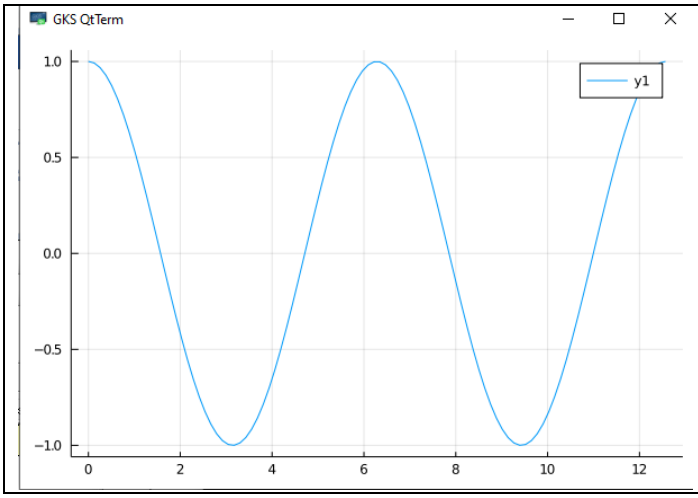

```
function plt(f,x)
y=map(f,x)
plot(x,y)
end

using Plots
f(x)=cos(x)
x=0:0.1:10
plt(f,x)
```



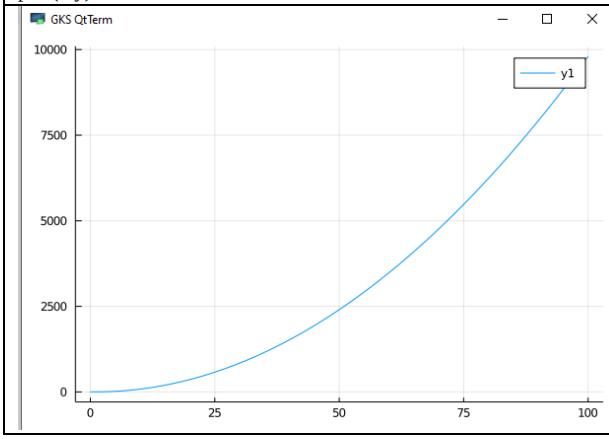

```
using Plots
f(x)=x*x-2.0
x = 1:0.5:20.0
y =map(f,x)
plot(x,y)
```



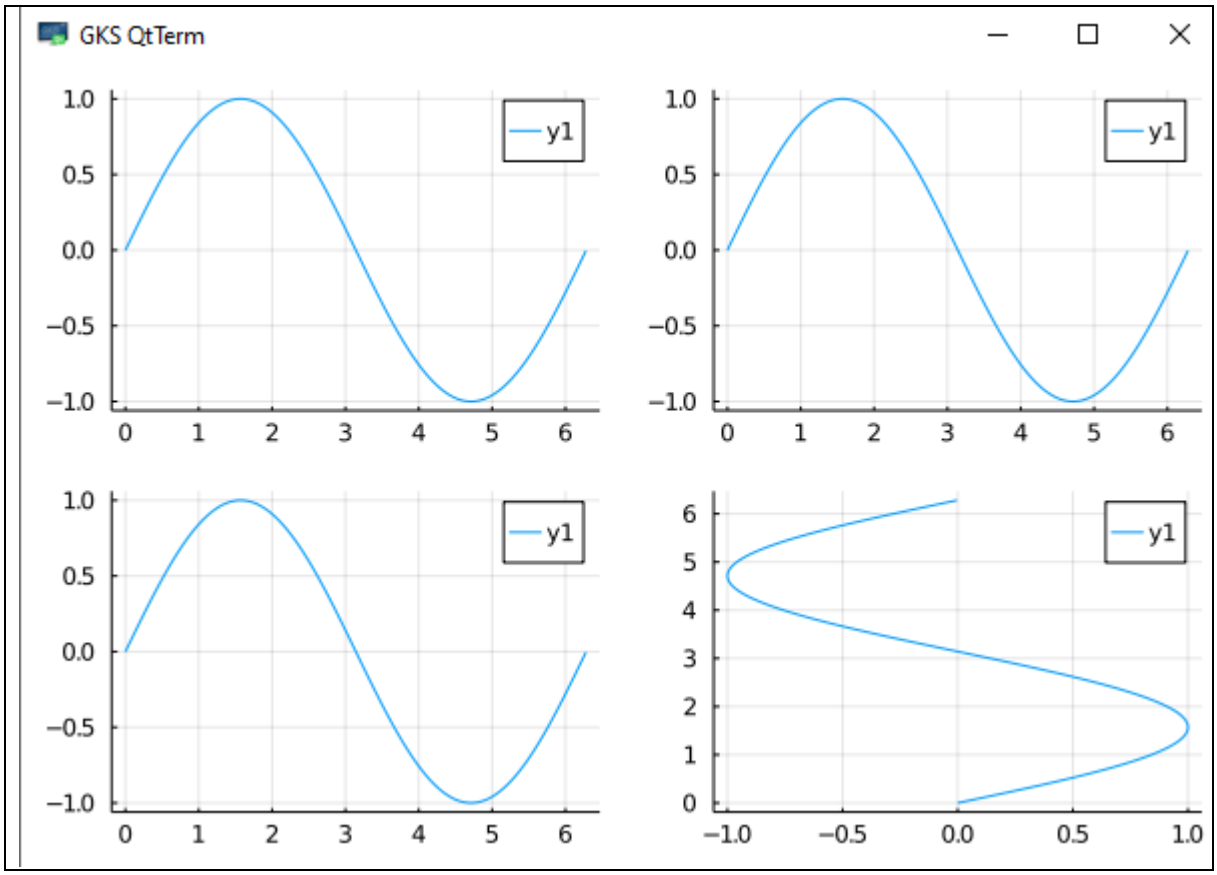

```
using Plots
m= 100
f(x)=cos(x)
x = fill(0.0, (m))
y = fill(0.0, (m))
dx=4.0*pi/(m-1)
for i=1:m
    x[i] = dx*(i-1)
    y[i] = f(x[i])
end
plot(x,y)
```



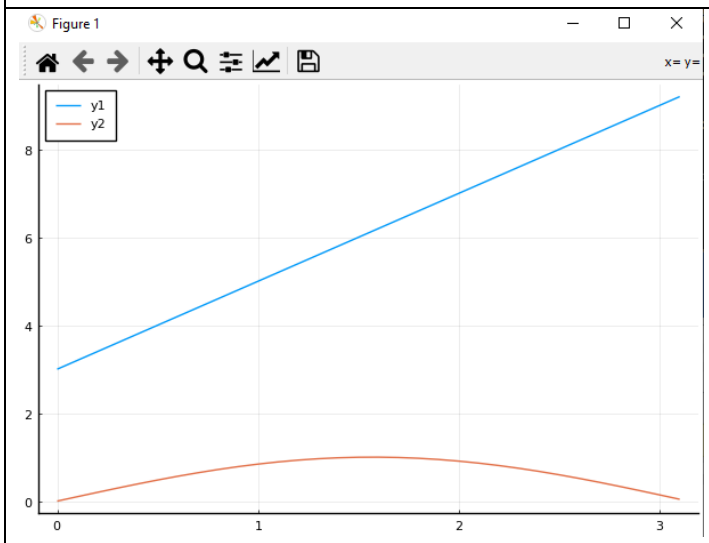

```
using Plots
f(x)=x*x-2.0*x+3.0
x = 0:0.1:100
y=map(f,x)
plot(x,y)
```



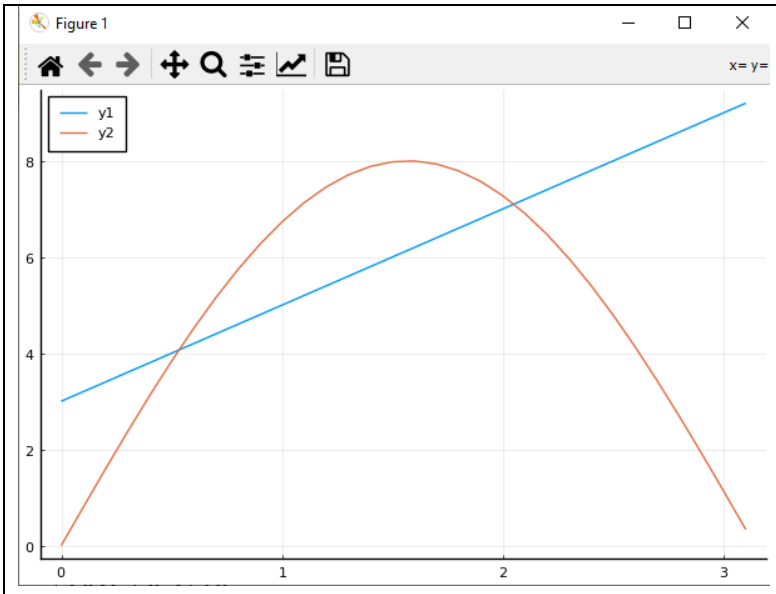

```
using Plots
p1 = plot(sin, 0:0.01:2*pi)
p2 = plot(0:0.01:2*pi, sin)
x = Array(0:0.01:2*pi);
y = sin.(x); # elementwise sin(x)
p3 = plot(x,y)
p4 = plot(y,x)
plot(p1,p2,p3,p4)
```



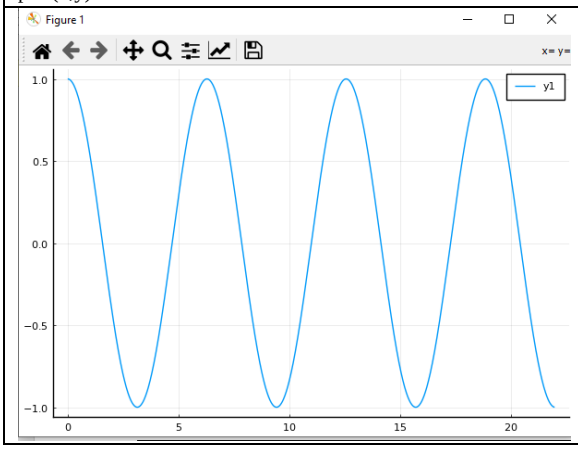

```
using Plots
f1=(x)->2.0*x+3.0
f2=(x)->sin(x)
plot([f1, f2],0:0.1:pi)
```



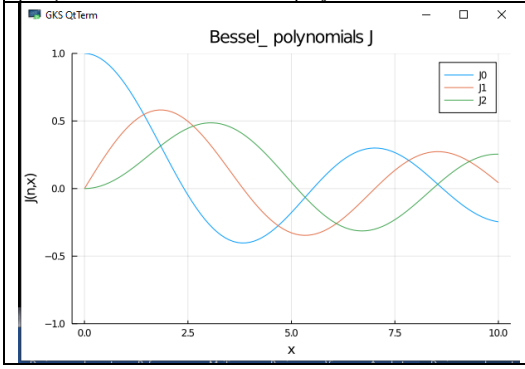

```
using Plots
f3(x)=2.0*x+3.0
f4(x)=8.0*sin(x)
plot([f3, f4],0:0.1:pi)
```




```
using Plots
x=0:pi/200:7*pi
y=cos.(x)
plot(x,y)
```

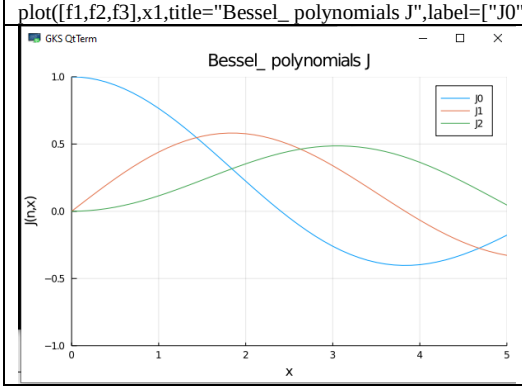



```
using SpecialFunctions
using Plots
x1=0.0:0.1:10.0
f1(x)=besselj(0,x)
f2(x)=besselj(1,x)
f3(x)=besselj(2,x)
plot([f1,f2,f3],x1,title="Bessel_ polynomials J",label=["J0" "J1" "J2"],ylims = (-1.0,1.0),xaxis = ("x"),yaxis = ("J(n,x)"))
```




```
using SpecialFunctions
using Plots
```

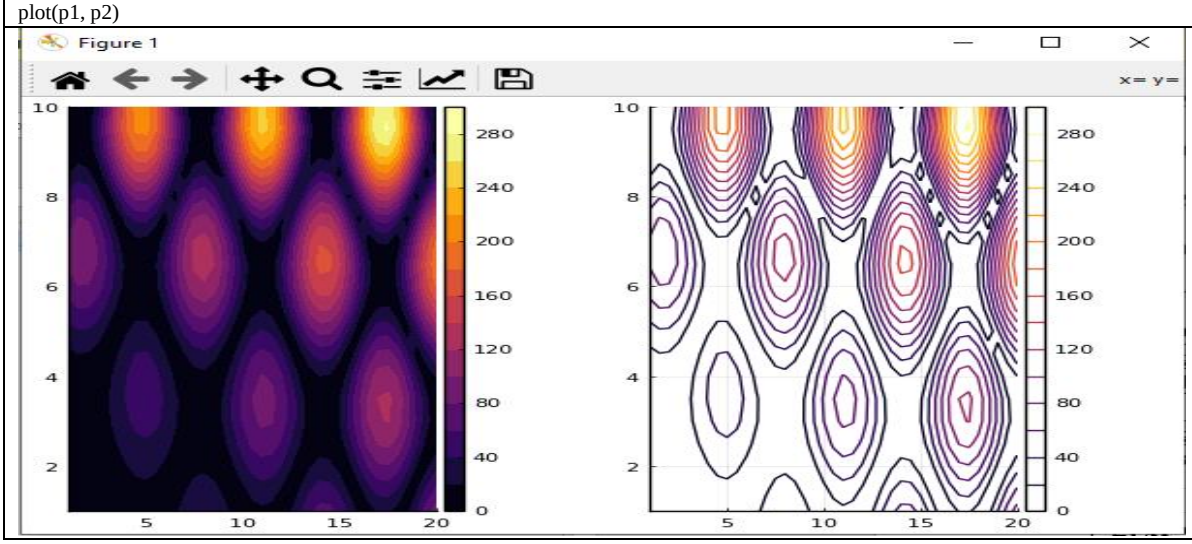
```
x1=0.0:0.1:10.0
f1(x)=besselj(0,x)
f2(x)=besselj(1,x)
f3(x)=besselj(2,x)
plot([f1,f2,f3],x1,title="Bessel_ polynomials J",label=["J0" "J1" "J2"],ylims = (-1.0,1.0),xlims = (0.0,5.0),xaxis = ("x"),yaxis = ("J(n,x)"))
```



Contour Plot:

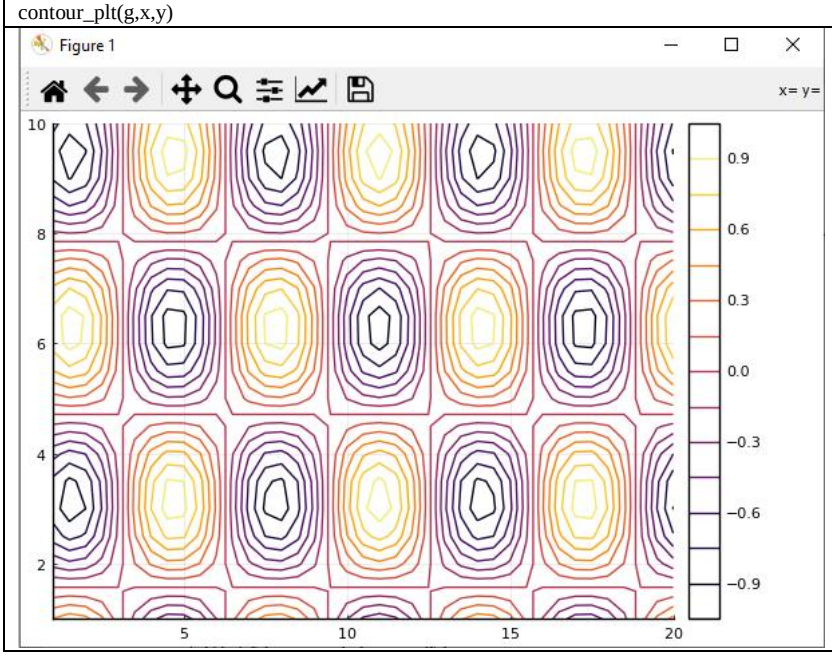
```
using Pkg
Pkg.build("PyCall")
```

```
using Plots
Plots.pyplot() # back to pyplot
x = 1:0.5:20
y = 1:0.5:10
g(x, y) = begin
      (3x + y ^ 2) * abs(sin(x) + cos(y))
   end
X = repeat(reshape(x, 1, :), length(y), 1)
Y = repeat(y, 1, length(x))
Z = map(g, X, Y)
p1 = contour(x, y, g, fill=true)
p2 = contour(x, y, Z)
plot(p1, p2)
```

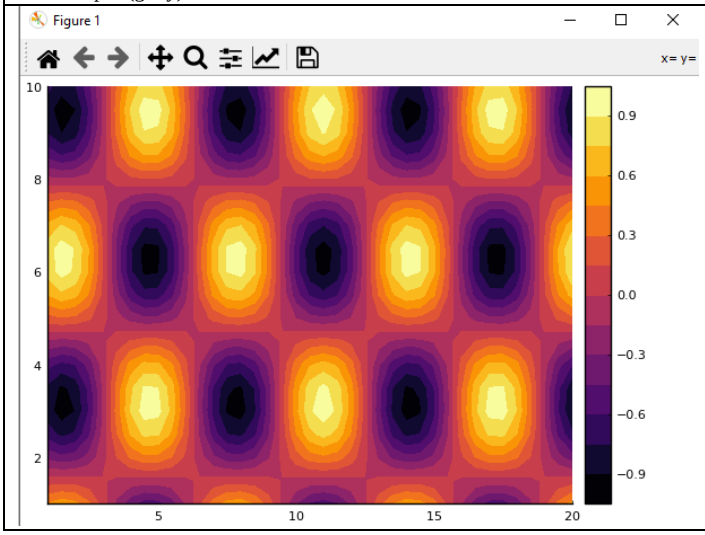



```
function contour_plt(f,x,y)
p1 = contour(x, y, f)
plot(p1)
end


using Plots
Plots.pyplot() # back to pyplot
x = 1:0.5:20
y = 1:0.5:10
```

```
g(x, y) = sin.(x)*cos.(y)
contour_plt(g,x,y)
```



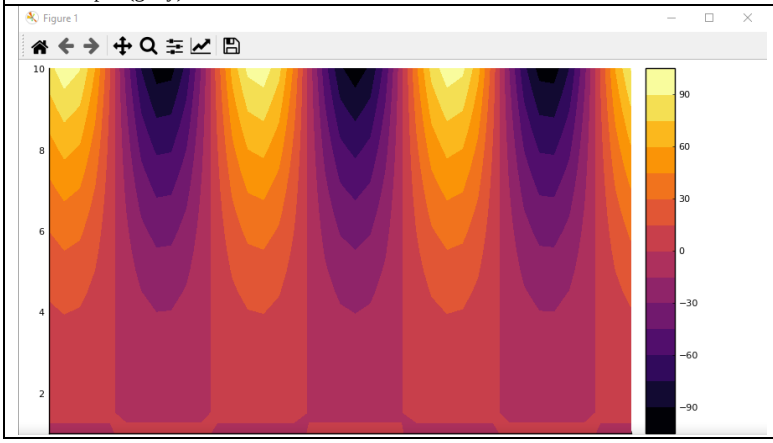

```
function contour_plt1(f,x,y)
p1 = contour(x, y, g,fill=true)
plot(p1)
end

using Plots
Plots.pyplot() # back to pyplot
x = 1:0.5:20
y = 1:0.5:10
g(x, y) = sin.(x)*cos.(y)
contour_plt1(g,x,y)
```

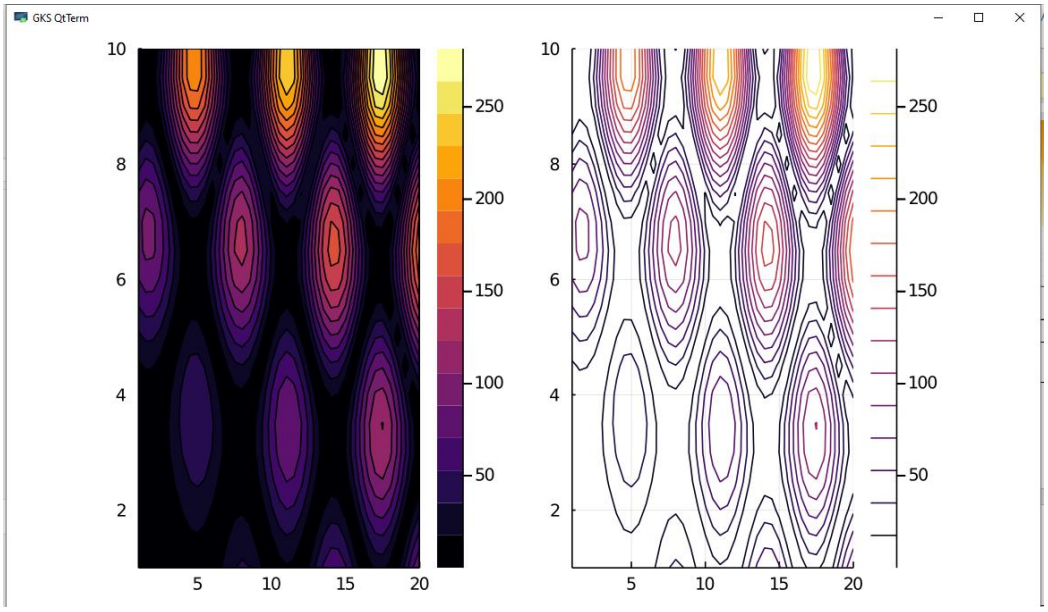


contourplot

```
function contour_plt1(f,x,y)

X = repeat(reshape(x, 1, :), length(y), 1)
Y = repeat(y, 1, length(x))
Z = map(f, X, Y)
p1 = contour(x, y, g,fill=true)
plot(p1)
end
```

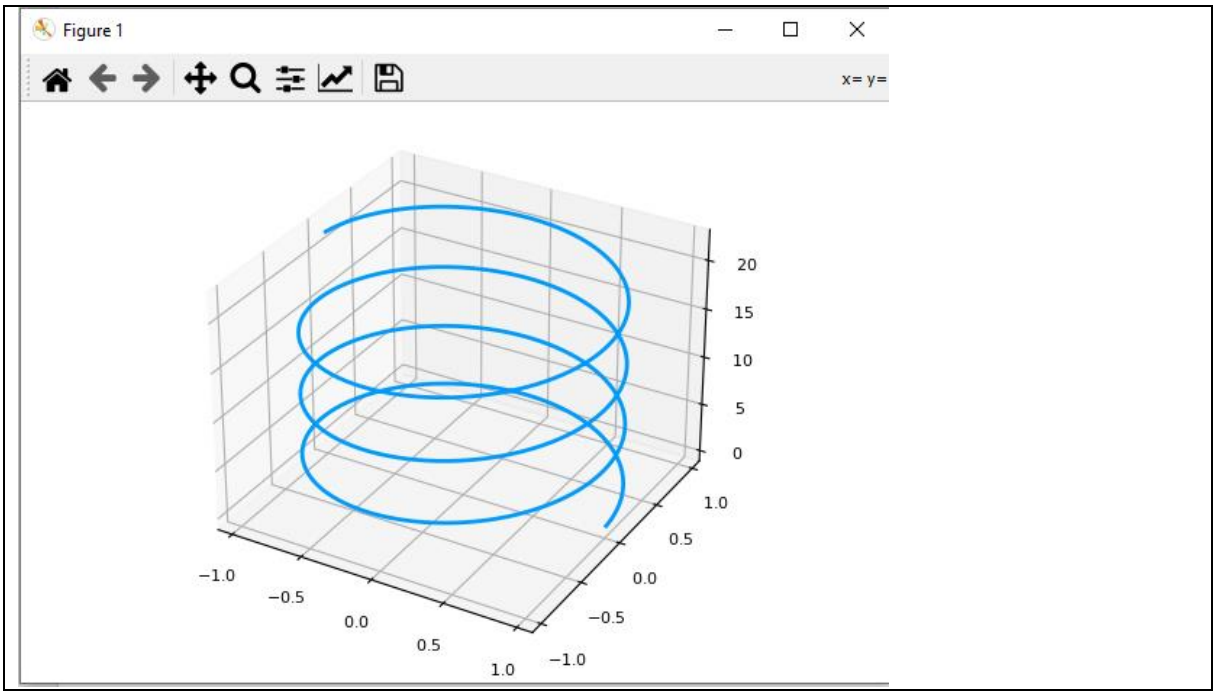
```
using Plots
Plots.pyplot() # back to pyplot
x = 1:0.5:20
y = 1:0.5:10
f(x)=x*x-2.0/x
g(x, y) = sin.(x)*f.(y)
contour_plt1(g,x,y)
```





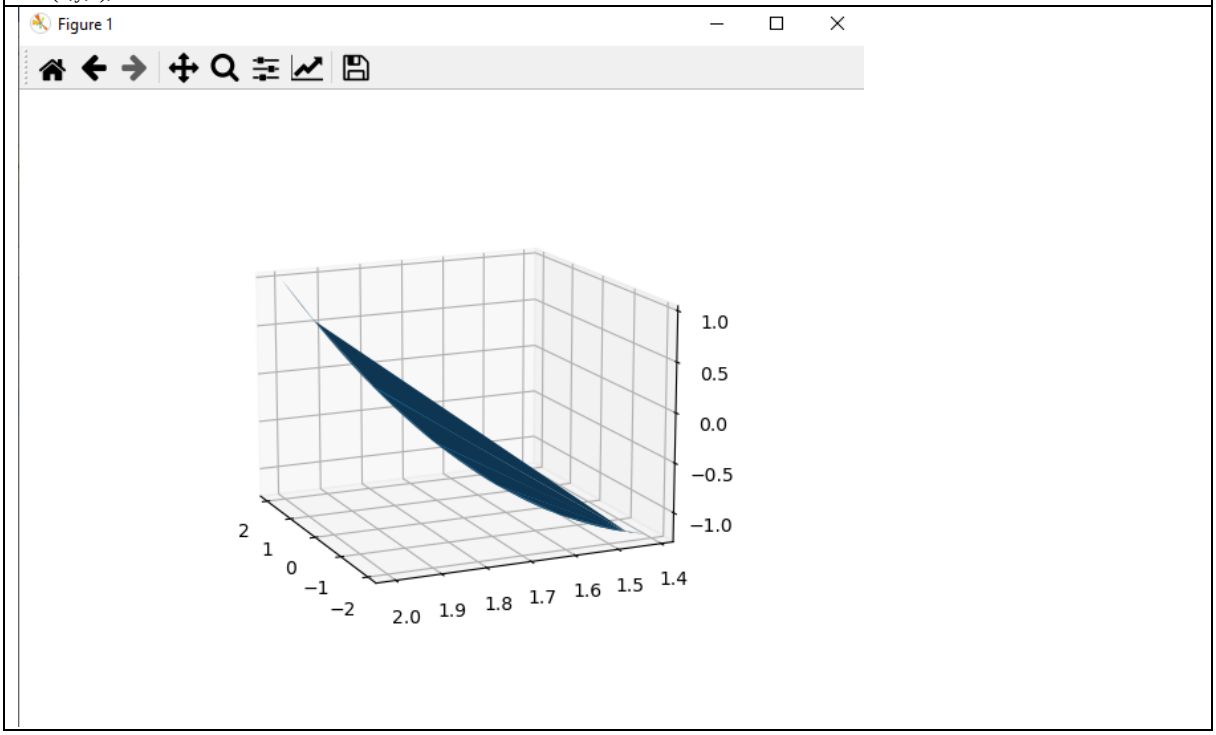
3D plot (3D line plot)

```
using Plots
t=0:pi/200:7*pi
x=cos.(t)
y=sin.(t)
plot3d(x,y,t,lw=2,leg=false)
```



Pyplot surf (3D surface plot)

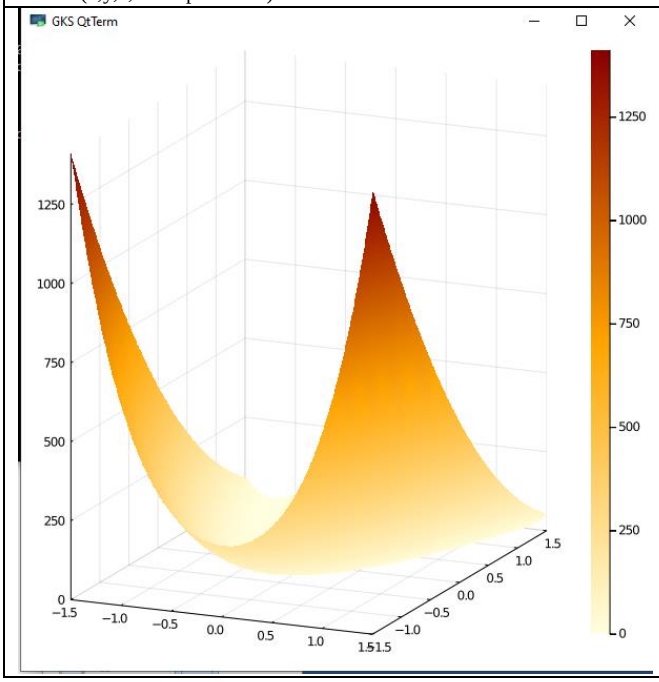
```
using PyPlot;
x = collect(Float16, range(-2,length=100,stop=2));
y = collect(Float16, range(sqrt(2),length=100, stop=2));
z = (x.*y).-y.-x.+1;
surf(x,y,z);
```



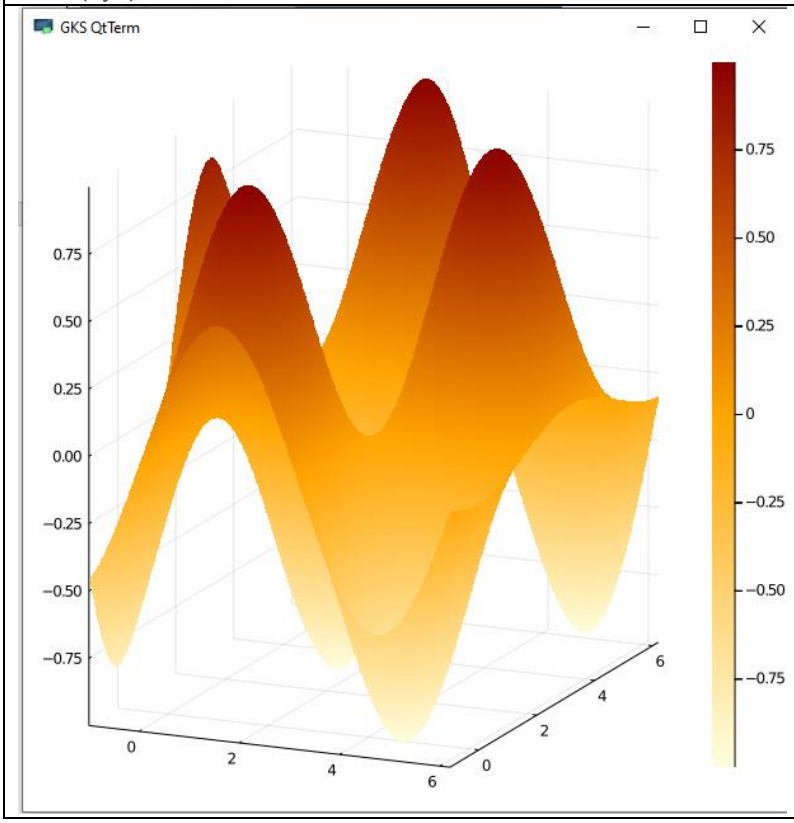
Plots surface (3D surface plot)

```
using Plots

function rosenbrock(x::Vector)
  return (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
end
```

```
default(size=(600,600), fc=:heat)
x, y = -1.5:0.1:1.5, -1.5:0.1:1.5
z = Surface((x,y)->rosenbrock([x,y]), x, y)
surface(x,y,z, linealpha = 0.3)
```

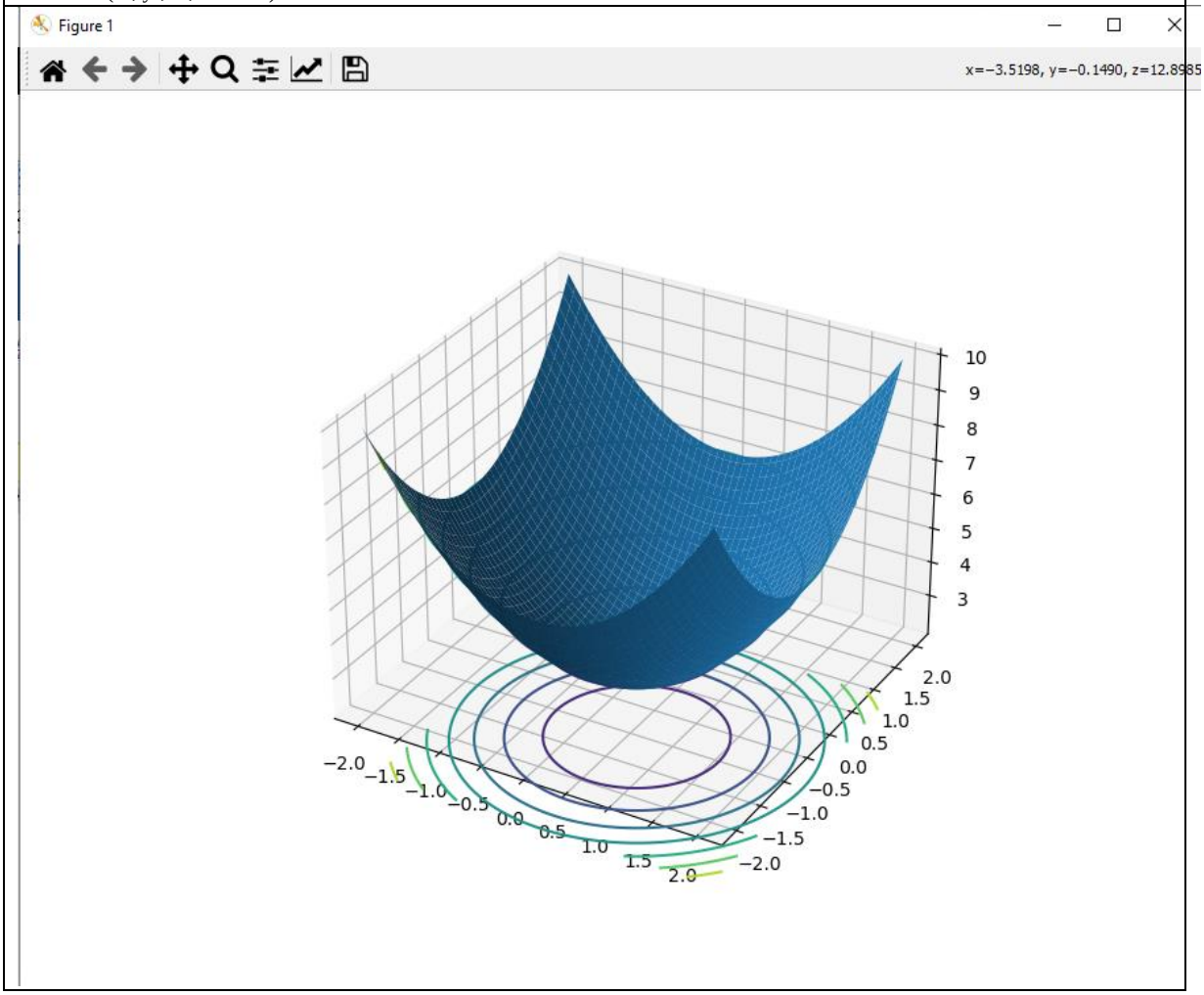



```
using Plots
x=-1:0.1:2.0*pi
y=-1:0.1:2.0*pi
h(x,y)=sin.(x)*cos.(y);

surface(x,y,h)
```

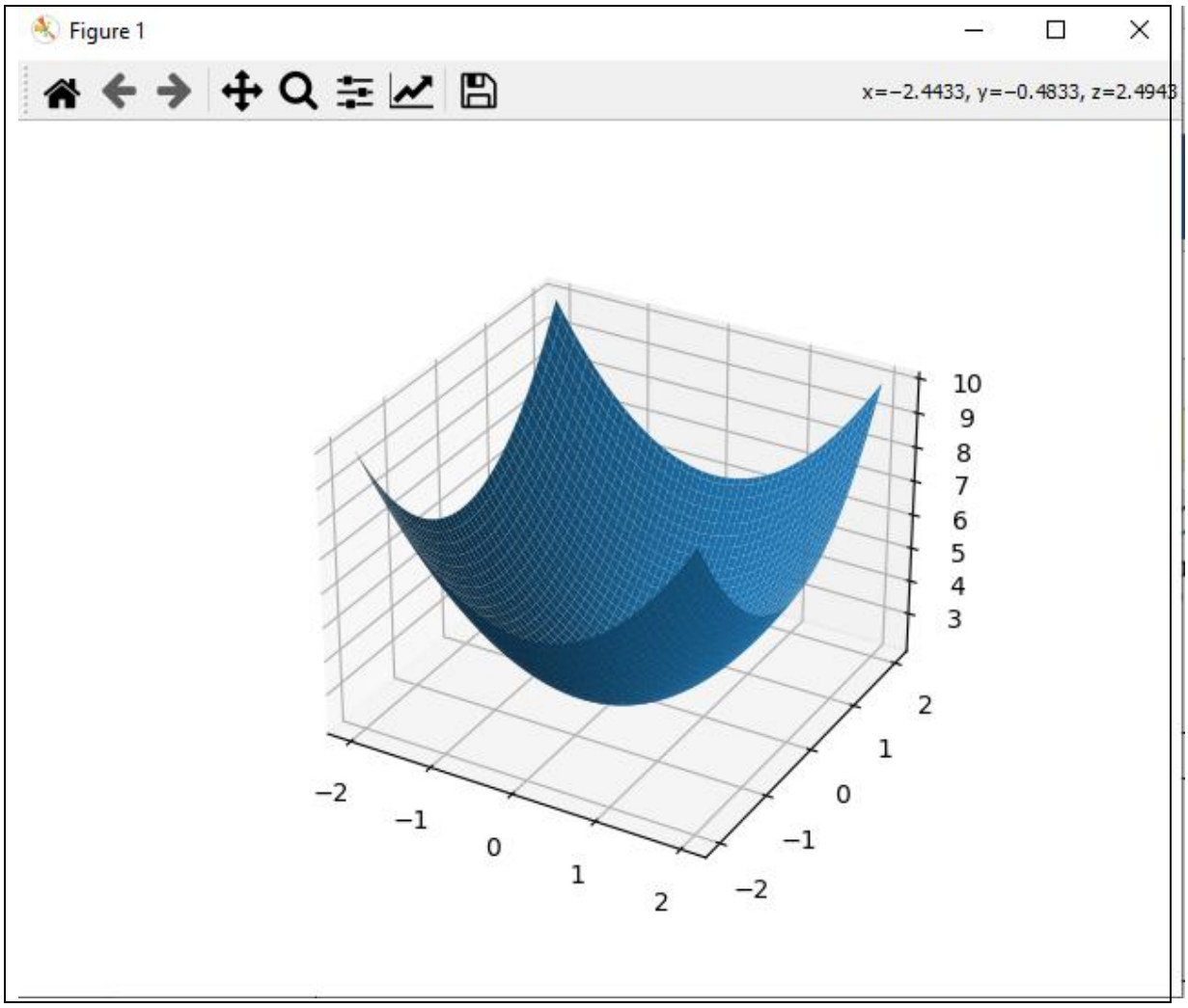



```
using PyPlot
```

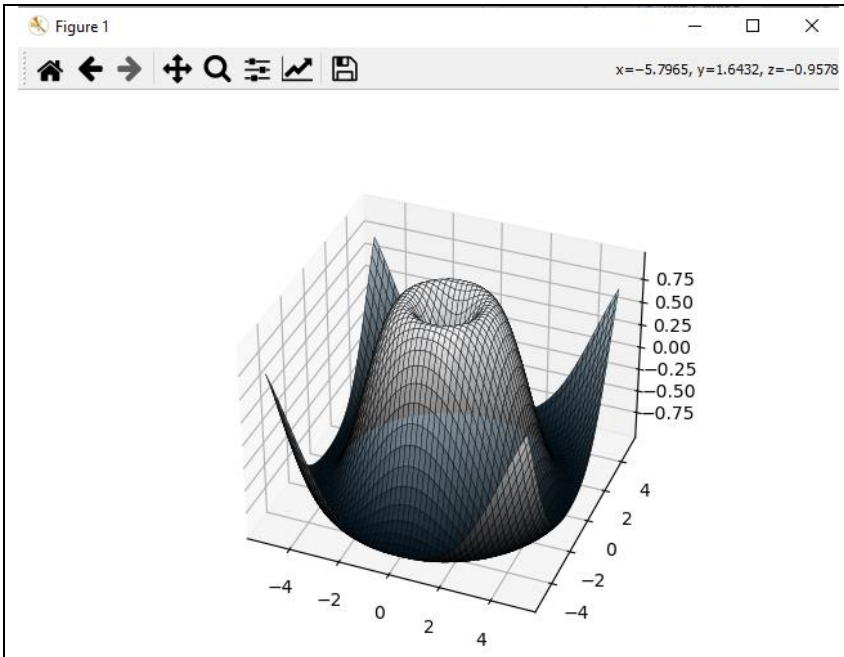
```
xs = ys = range(-2, stop=2, length=100)
f(x,y) = 2 + x^2 + y^2
zs = [f(x,y) for y in ys, x in xs]
PyPlot.plot_surface(xs, ys, zs)
PyPlot.contour3D(xs, ys, zs)
ax = PyPlot.gca()
ax.contour(xs, ys, zs, offset=0)
```



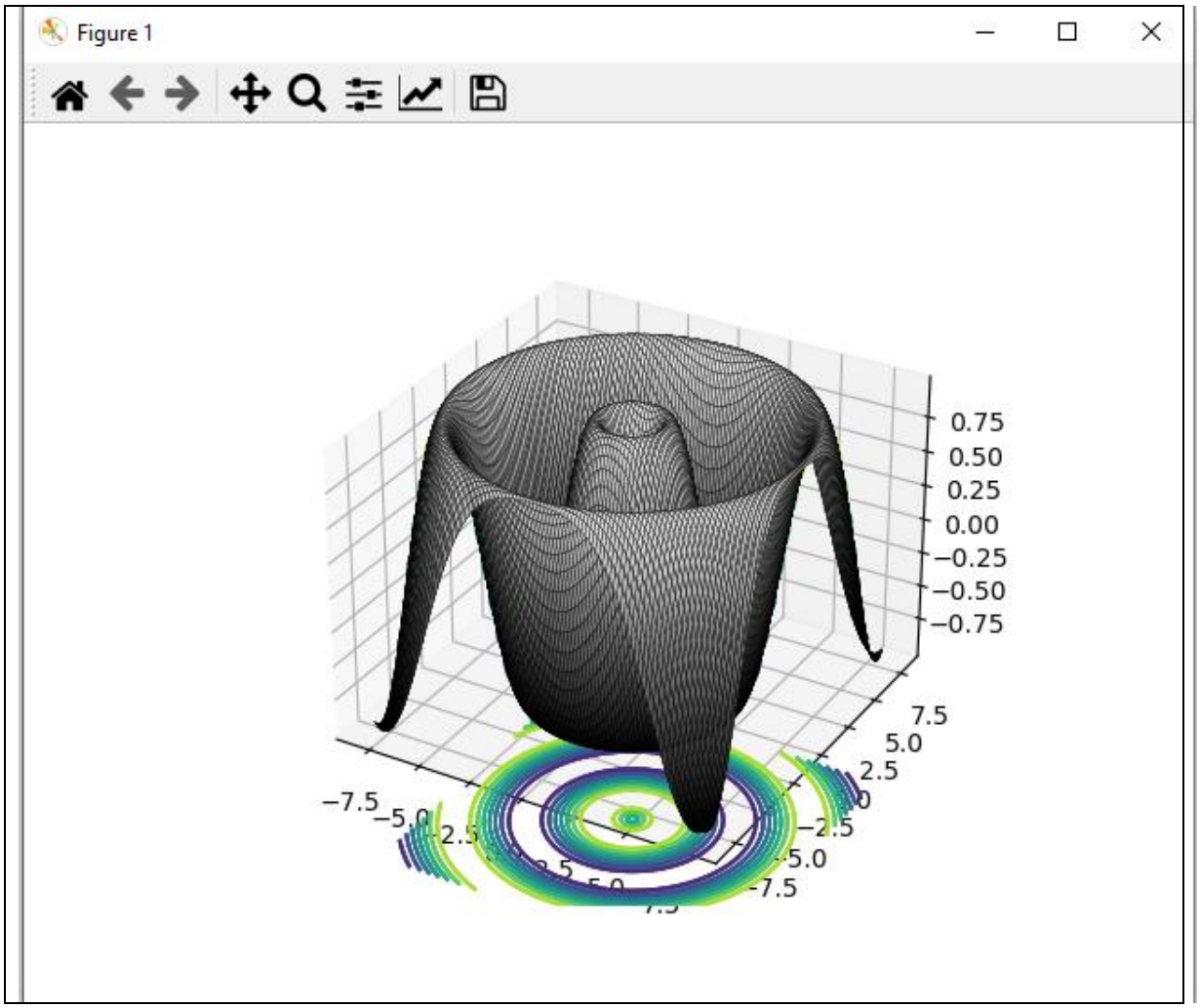

```
using PyPlot
xs = ys = range(-2, stop=2, length=100)
f(x,y) = 2 + x^2 + y^2
zs = [f(x,y) for y in ys, x in xs]
PyPlot.plot_surface(xs, ys, zs)
```




```
using PyPlot
xs = ys = range(-5, stop=5, length=100)
f(x,y) = sin(sqrt(x^2+y^2))
zs = [f(x,y) for y in ys, x in xs]
PyPlot.plot_surface(xs, ys, zs, rstride=2,edgecolors="k", cstride=2,
  cmap=ColorMap("gray"), alpha=0.8, linewidth=0.25)
```

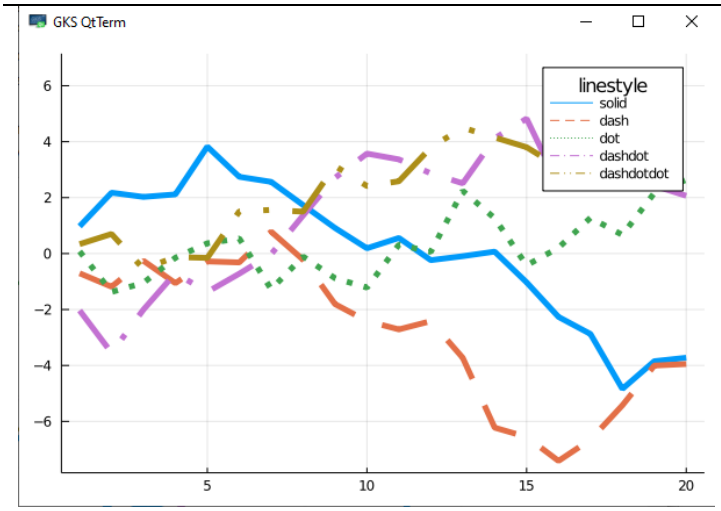



```
using PyPlot
xs = ys = range(-8, stop=8, length=50)
f(x,y) = sin(sqrt(x^2+y^2))
zs = [f(x,y) for y in ys, x in xs]
PyPlot.plot_surface(xs, ys, zs, rstride=2,edgecolors="k", cstride=2,
   cmap=ColorMap("gray"), alpha=0.8, linewidth=0.25)
 PyPlot.contour3D(xs, ys, zs)
ax = PyPlot.gca()
ax.contour(xs, ys, zs, offset=-2)
```

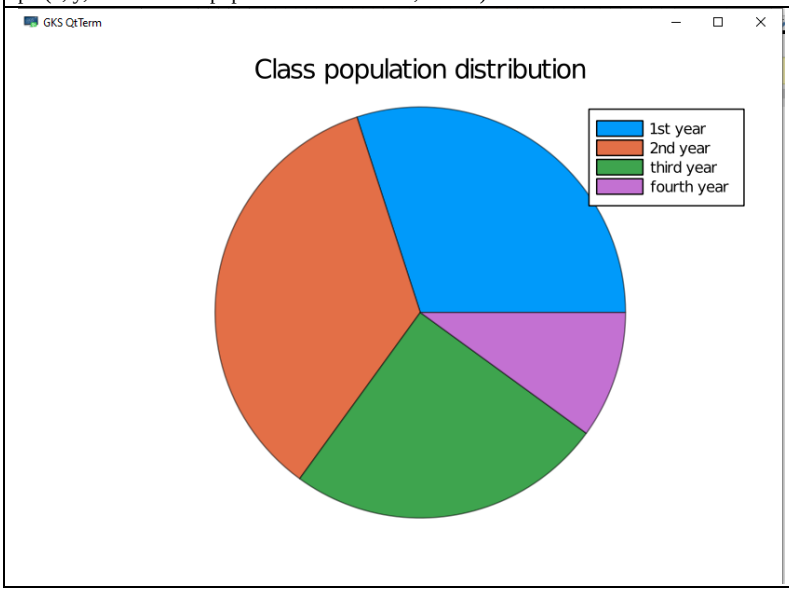


Multiline plot with GR

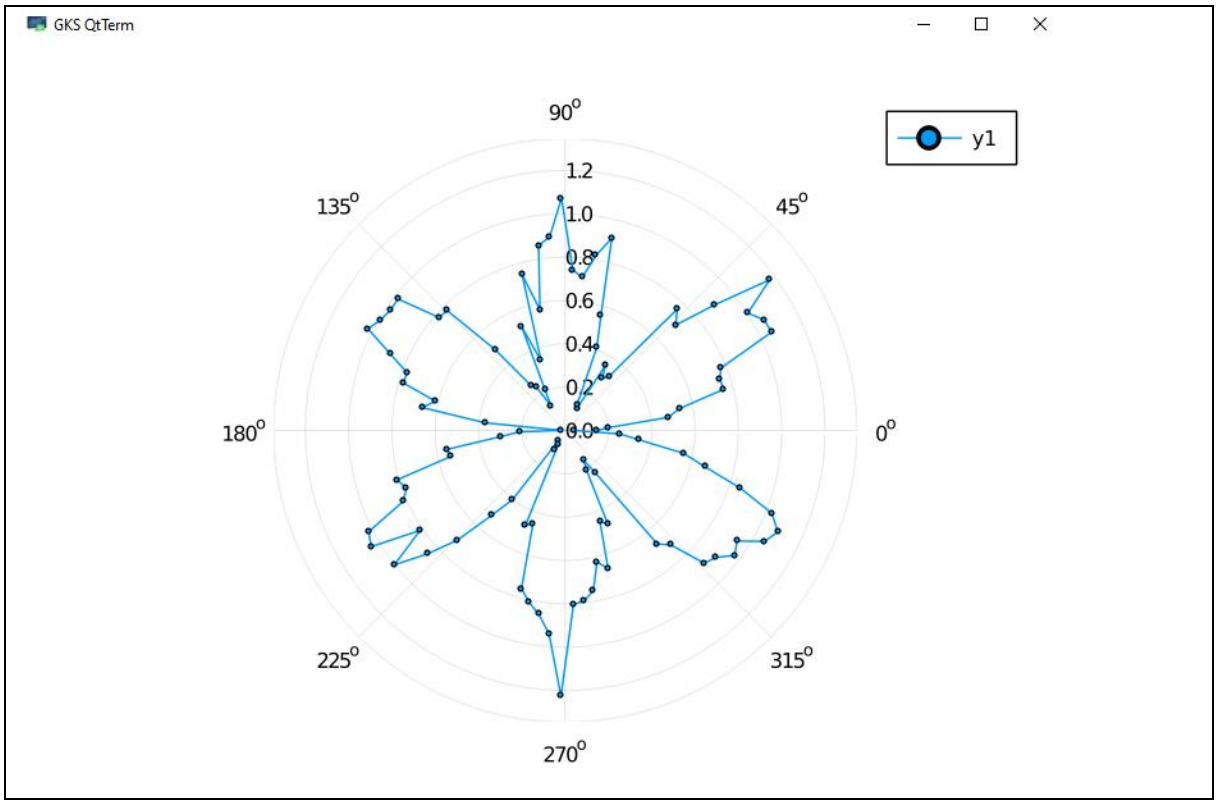
```
using Plots
gr()
styles = filter((s->begin
            s in Plots.supported_styles()
        end), [:solid, :dash, :dot, :dashdot, :dashdotdot])
styles = reshape(styles, 1, length(styles))
n = length(styles)
y = cumsum(randn(20, n), dims = 1)
plot(y, line = (5, styles), label = map(string, styles), legendtitle = "linestyle")
```




```
using Plots
gr()
x = ["1st year", "2nd year", "third year","fourth year"]
y = [0.3, 0.35, 0.25,0.1]
pie(x, y, title = "Class population distribution", l = 0.5)
```




```
using Plots
gr()
tetha = range(0, stop = 2.0*pi, length = 100)
r = abs.(0.1 * randn(100) + sin.(3*tetha))
plot(tetha, r, proj = :polar, m = 2)
```



READING AND WRITING DATA FROM A SEQUENTIAL FILE

In order to read/write data into a sequaential file open statement will be used. This statement has the following form:

open(filename)

Where “filename” is the name of input/output file

```
open("test.txt") do f

  # line_number
  line = 0

  # read till end of file
  while ! eof(f)
    # read a new / next line for every iteration
    s = readline(f)
    line += 1
    println("$line . $s")
  end

end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" file1.jl
1 . once upon a time in a country far far a way
2 . there was a little princess

> Terminated with exit code 0.
```

```
julia> include("file1.jl")
1 . once upon a time in a country far far a way
2 . there was a little princess

julia>
```

Reading file b.txt (first two varibles as x,y in each line)

File b.txt

1.234 3.24 5.67 7.78
1.345 5.25 5.67 7.78
3.748 6.38 5.67 7.78
2.175 5.24 5.67 7.78
1.222 3.45 5.67 7.78

```
 function convert_to_float(column)
   new_column = map(x -> (
         x = ismissing(x) ? "" : x;  # no method matching tryparse(::Type{Float64}, ::Missing)
         x = tryparse(Float64, x);   # returns: Float64 or nothing
         isnothing(x) ? missing : x; # options: missing, or "", or 0.0, or nothing
         ), column)  # input
   # returns Array{Float64,1} OR Array{Union{Missing, Float64},1}
   return new_column
end

function readDouble(name)
  name="b.txt"
  open(name) do f
  # read till end of file
  m=0
  while ! eof(f)
    m+=1
    line = readline(f)
    s=split(line," ")
    n=length(s)
    global m1=m
    global n1=n
  end  #while
  close(f)
  end  #open
  global x=Matrix{Float64}(undef,n1,m1)
  open(name) do f
  # read till end of file
  m=0
  global z=Array{Float64}(undef,n1,0)
  while ! eof(f)
    m+=1
    line = readline(f)
    s=split(line," ")
    n=length(s)
    x1=convert_to_float(s)
    z=hcat(z,x1)
    global x=z
  end  #while
  close(f)
  end  #open
  #transpose matrix
  global y=x'
  return y
  end

function readOffsetDouble(name)
  y=readDouble(name)
  n=size(y)[1]
  m=size(y)[2]
  global x=OffsetArray(y, 0:n-1,0:m-1)
  return x
  end

using OffsetArrays
name="b.txt"
y=readDouble(name)
n=size(y)[1]
m=size(y)[2]
print("n = $n m = $m")
print("array indis starting from 1")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), y)
print("array indis starting from 0")
x=readOffsetDouble(name)
```

```
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), x)
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" readouble.jl
n = 5 m = 4array indis starting from 1
5X4 adjoint(::Matrix{Float64}) with eltype Float64:
 1.234  3.24  5.67  1.0
 1.345  5.25  5.67  2.0
 3.748  6.38  5.67  3.0
 2.175  5.24  5.67  4.0
 1.222  3.45  5.67  5.0array indis starting from 0
5X4 OffsetArray(adjoint(::Matrix{Float64}), 0:4, 0:3) with eltype Float64 with indices 0:4X0:3:
 1.234  3.24  5.67  1.0
 1.345  5.25  5.67  2.0
 3.748  6.38  5.67  3.0
 2.175  5.24  5.67  4.0
 1.222  3.45  5.67  5.0
> Terminated with exit code 0.
```

Writing into a text file:

```
io = open("myfile.txt", "w")
write(io, "Hello world!")
close(io)
```

Text f,le myfile.txt contains:

```
Hello world!
```

2. TAYLOR SERIES AND SERIES APPROXIMATION TO FUNCTIONS

2.1 TAYLOR SERIES

Taylor equation can be considered the roots of many numerical analysis methods. It will give us serial. **Taylor series** is a representation of a function as an infinite sum of terms calculated from the values of its derivatives at a single point. It is named after the English mathematician Brook Taylor. If the series is centered at zero, the series is also called a **Maclaurin series**, named after the Scottish mathematician Colin Maclaurin. It is common practice to use a finite number of terms of the series to approximate a function. Taylor series around a point a can be written as

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f"(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \cdots$$

Or if it is written in summation notation

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

Or if a is replaces with x^k and x is replaced with x^{k+1} equation becomes

$$f(x^{k+1}) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x^k)}{n!}(x^{k+1}-x^k)^n$$

+..

Computers could not calculate anything more complex than addition, the rest usually accomplish through Taylor series approximations. For example function $f(x)=e^x$ can be approximated as

$$f(x) = e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120}$$

Some of the formulation used in Numerical analysis is direct derivation from the Taylor series. For example for the Newton root finding equation, above linear terms of the equation is ignorede and function value is replaced with 0

$f(x^{k+1}) = 0 = f(x^k) + f'(x^k)(x^{k+1}-x^k)$ Will yield to

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}$$

Taylor series can also be used to estimate error. The series usually used up to a certain term and error will be proportinal to one upper term in the series. For example If exponetial function is taken as

$f(x) = e^x = 1 + \mathrm{x} + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$

The error term will be proportional to the next term of the series

$$Error \cong \frac{x^5}{120}$$

2.2 SERİES SOLUTIONS OF FUNCTIONS

Digital computers can only calculate addition process, therefore most of the mathematical expressions are calculated by using series. Series solutions of a few functions will be listed in here.

1. $\ln(x) = 2\sum_{k=1}^{\infty} \frac{1}{2k-1}\left(\frac{x-1}{x+1}\right)^{2k-1} \quad [0 < x]$
2. $\exp(\mathrm{x}) = \sum_{k=1}^{\infty} \frac{x^k}{k!}$
3. $\exp(-\mathrm{x}) = \sum_{k=1}^{\infty} (-1)^k \frac{x^k}{k!}$
4. $\exp(-\mathrm{x}^2) = \sum_{k=1}^{\infty} (-1)^k \frac{x^{2k}}{k!}$
5. $\exp(\mathrm{x})(1+x) = \sum_{k=1}^{\infty} \frac{x^{k(k+1)}}{k!}$
6. $a^x = e^{(xln(a))} = \sum_{k=1}^{\infty} \frac{[xln(a)]^k}{k!}$
7. $\sum_{k=0}^{\infty} a^{kx} = \frac{1}{1-a^x}$ [a>1 and x<0 or 0<a<1 x>0]
8. $\sin(\mathrm{x}) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$
9. $\sin h(\mathrm{x}) = \sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)!}$
10. $\cos(\mathrm{x}) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$
11. $\cos \mathrm{h}(\mathrm{x}) = \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!}$
12. $\tan(\mathrm{x}) = \frac{\sin(x)}{\cos(x)}$
13. $\tanh(\mathrm{x}) = \frac{\sinh(x)}{\cos \mathrm{h}(x)}$
14. $\arcsin(\mathrm{x}) = \sum_{k=0}^{\infty} \frac{(2k)!x^{2k+1}}{2^{2k}(k!)^2(2k+1)} \quad x^2 \leq 1$
15. $\mathrm{arcsinh}(\mathrm{x}) = \sum_{k=0}^{\infty} (-1)^k \frac{(2k)!x^{2k+1}}{2^{2k}(k!)^2(2k+1)} \quad x^2 \leq 1$
16. $\arctan(\mathrm{x}) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)} \quad x^2 \leq 1$
17. $\mathrm{arctanh}(\mathrm{x}) = \sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)} \quad x^2 \leq 1$
18. $\mathrm{erf}(\mathrm{x}) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{n!(2n+1)}$
19. $\pi = \sum_{k=0}^{\infty} \frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{2}{8k+6}\right)$ Bailey, Borwein Plouffe(BBP) formula[90]
20. $J_n(z) = \left(\frac{1}{2z}\right) \sum_{k=1}^{\infty} (-1)^k \frac{\left(\frac{1}{2z}\right)^{2k}}{k!\,\Gamma(n+1)}$ Bessel function
21. $\Gamma(z) = \int_{z=0}^{\infty} t^{z-1}e^{-t}dt = \frac{e^{-\gamma z}}{z} \prod_{n=1}^{\infty} \left(1 + \frac{z}{n}\right)^{-1} e^{z/n}$ where
    $\gamma = 0.5772156649015328 6\ldots$ is the Euler-Mascheroni constant
22. $Y_n(z) = \frac{J_n(z)\cos(n\pi) - J_{-n}(z)}{\sin(n\pi)}$ Bessel function

Exponential function $\exp(x) = \sum_{k=1}^{\infty} \frac{x^k}{k!}$

```
function exp(x)
    total=0;
    factorial=1;
    power=1;
    for i =1:50
      total=total+power/factorial;
      power=power*x
      factorial=factorial*i
      end
    return total
    end
x=1.0
r=exp(x);
print("exp($x) = $r");
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" exp.jl
exp(1.0) = 2.7182818284590455
> Terminated with exit code 0.
```

Natural logarithm $\ln(x) = 2\sum_{k=1}^{\infty} \frac{1}{2k-1}\left(\frac{x-1}{x+1}\right)^{2k-1} \quad [0 < x]$

```
function ln(x)
   total=0;
   y=(x-1.0)/(x+1.0);
   power=y;
   for k =1:100
      total=total+power/(2.0*k-1.0);
      power=power*y*y
      end
   return 2.0*total
   end
e=exp(1.0);
r=ln(e);
print("ln($e) = $r");
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" ln.jl
ln(2.718281828459045) = 0.9999999999999997
> Terminated with exit code 0.
```

ln(x)

k		x=	2,7182818285
		y=(x-1)/(x+1)	0,4621171573
		power	total
	0	0,4621171573	0
	1	0,0986861666	0,9242343145
	2	0,0210746546	0,9900250922
	3	0,0045005403	0,9984549541
	4	0,0009611006	0,9997408227
	5	0,0002052452	0,9999544006
	6	4,38306E-005	0,9999917179
	7	9,36012E-006	0,9999984611
	8	1,99887E-006	0,9999997091
	9	4,26864E-007	0,9999999443
	10	9,11578E-008	0,9999999892
	11	1,94670E-008	0,9999999979
	12	4,15721E-009	0,9999999996
	13	8,87782E-010	0,9999999999
	14	1,89588E-010	1
	15	4,04869E-011	1
	16	8,64608E-012	1
	17	1,84639E-012	1
	18	3,94301E-013	1
	19	8,42038E-014	1
	20	1,79819E-014	1

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

```
function sin1(x)
   total=0.0
   factorial=1.0
   power=x;
   plusminus=1
   i=0;
   for k=1:80
      total=total+power/factorial*plusminus;
      power=power*x*x;
      i=2*k+1;
      factorial=factorial*(i-1)*i;
      plusminus*=-1;
      end
   return total
   end

x=pi/3.0;
y=sin(x)
y1=sin1(x)
print("sin=$y1 math library sin = $y")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" sin1.jl
sin=0.8660254037844385 math library sin = 0.8660254037844386
> Terminated with exit code 0.
```

$$\arcsin(x) = \sum_{k=0}^{\infty} \frac{(2k)!\, x^{2k+1}}{2^{2k}(k!)^2(2k+1)} \qquad x^2 \le 1$$

```
function asin1(x)
   total=0.0
   factorial1=1.0
   factorial2=1.0
   power=x;
   power2=1.0;
   i=0;
   for k=1:80
      total=total+power*factorial1/(power2*factorial2*factorial2)/(2.0*(k-1)+1.0);
      power*=x*x;
      power2*=4;
      i=2*k;
      factorial1=factorial1*(i-1)*i
      factorial2=factorial2*k
      end
   return total
   end

x1=pi/3.0
x2=sin(x1);
y2=asin1(x2)
y3=asin(x2)
print("asin =$y2 math.asin = $y3")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" asin.jl
asin =1.047197551196467 math.asin = 1.0471975511965976
> Terminated with exit code 0.
```

$$a^x = e^{xln(a)}$$

```
function exp1(x)
     total=0.0;
     factorial=1.0;
     power=1.0;
     for i =1:100
       total=total+power/factorial;
       power=power*x
       factorial=factorial*i
       end
     return total
     end
```

```
function ln1(x)
   total=0.0;
   y=(x-1.0)/(x+1.0);
   power=y;
   for k =1:100
      total=total+power/(2.0*k-1.0);
      power=power*y*y
      end
   return 2.0*total
   end

function pow1(x,a)
  y=x*ln1(a)
  return exp1(x*ln1(a))
  end

x=2.0
a=3.0
r=pow1(x,a)
print("pow($x,$a) = $r")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" pow.jl
pow(2.0,3.0) = 8.999999999999986
> Terminated with exit code 0.
```

$$\text{erf(x)} = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{n!\,(2n+1)}$$

```
function erf1(x)
   A=2.0/sqrt(pi)
   total=0.0
   plusminus=1
   power=x
   factorial=1.0
   for n=1:200
        total=total+power*plusminus/factorial/(2.0*(n-1)+1.0)
        power=power*x*x
        factorial=factorial*n
        plusminus*=plusminus*(-1)
        end
   return A*total
   end

using SpecialFunctions
x=0.1
y1=erf1(x)
y=erf(x)
print("x=$x,y1=$y y=$y")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" erf.jl
x=0.1,y1=0.1124629160182849 y=0.1124629160182849
> Terminated with exit code 0.
```

Some of the algorithms might not be as clear as a directly given function. For example let us look at the **Russian peasant multiplication problem:**

Take two integer for example 21 and 37 then double the first one(left hand side) and divide by two the second one(right side) ignore remaining in division process and stop when the right hand side reached to 1 and then cross each line where the right side number is even add up remaining left hand side numbers for example. If the given integer numbers to be multiplied is 21 and 17, the process will follow the path given below:

21 37
~~42 18~~ cross out (18 is even)
84 9
~~168 4~~ cross out(4 is even)
~~336 2~~ cross out(2 is even)
672 1
then 21+84+672=777
21*37=**777**

Now let us convert this to an algorithmic structure:

```
function  isOdd(x)
  return bool(x %2 != 0 )
  end

function multiply(x1,x2)
  y2=x2;
  y1=x1;
  b2=0;
  total=0;
  while y2>=1
    b2=(y2 %2 != 0 )
    if y2 %2 != 0
      total=total+y1
    end
    println("total = $total, y1= $y1,y2= $y2")
    y1=y1*2;
    y2=Int64(floor(y2/2))
    end
  return total
  end

x1=21
x2=37
x=multiply(x1,x2)
print("russian multiplication $x1*$x2 = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" russian.jl
total = 21, y1= 21,y2= 37
total = 21, y1= 42,y2= 18
total = 105, y1= 84,y2= 9
total = 105, y1= 168,y2= 4
total = 105, y1= 336,y2= 2
total = 777, y1= 672,y2= 1
russian multiplication 21*37 = 777
> Terminated with exit code 0.
```

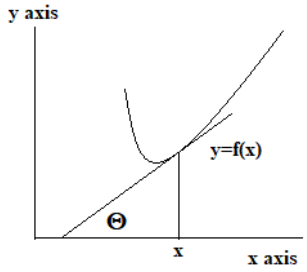
1. DERIVATIVES

3.1 DEFINITIONS

Definition of derivatives:

$$\frac{df(x)}{dx} = lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

Other definition



Derivative is the slope of tangent to the function at a given x value

$$\frac{df(x)}{dx} = \tan(\theta)$$

Some common analytic formulas for derivatives:

f(x)	$f'(x) = \frac{df(x)}{dx}$
x^n	nx^{n-1}
x^2	$2x$
x^3	$3x^2$
e^{ax}	ae^{ax}
$e^{u(x)}$	$u'(x)e^u$
e^{-x}	$-e^{-x}$
a^{bx}	$bln(a)a^{bx}$
a^x	$ln(a)a^x$
2^{3x}	$3ln(2)2^{3x}$
$ln(u(x))$	$\frac{u'(x)}{u(x)}$
cos(u(x))	-u'(x)sin(u(x))
sin(u(x))	u'(x)cos(u(x))
tan(u(x))	u'(x)$sec^2(u(x))$
arcsin$(u(x))$	$\frac{u'(x)}{\sqrt{1-u(x)^2}}$
arccos$(u(x))$	$-\frac{u'(x)}{\sqrt{1-u(x)^2}}$
arctan$(u(x))$	$\frac{u'(x)}{1+u(x)^2}$
cosh(u(x))	u'(x)sinh(u(x))
sinh(u(x))	u'(x)cosh(u(x))
tanh(u(x))	u'(x)$sech^2(u(x))$
arcsinh$(u(x))$	$\frac{u'(x)}{\sqrt{1+u(x)^2}}$
arccosh$(u(x))$	$-\frac{u'(x)}{\sqrt{u(x)^2-1}}$
arctanh$(u(x))$	$\frac{u'(x)}{1-u(x)^2}$
$u(x) \pm v(x)$	$u'(x) \pm v'(x)$
$u(x)v(x)$	$u'(x)v(x) + v'(x)u(x)$
$\frac{1}{u(x)}$	$-\frac{u'(x)}{u^2(x)}$
$\frac{u(x)}{v(x)}$	$\frac{u'(x)v(x) - v'(x)u(x)}{v^2(x)}$
Denklemi buraya yazın.	

Examples of analytical derivations

EX1: take derivative of $y = f(x) = 2x^3 + 3x^2 + 5$

Answer: $f'(x) = 6x^2 + 6x$

EX2: take derivative of $y = f(x) = u(x)v(x) = (2x^3 + 3x^2 + 5)(x^2 - 2)$

$f'(x) = u'(x)v(x) + v'(x)u(x)$

$u'(x) = 6x^2 + 6x \qquad v'^{(x)} = 2x$

$f'(x) = (6x^2 + 6x)(x^2 - 2) + (2x^3 + 3x^2 + 5)(2x)$

EX3: take derivative of $y = f(x) = \frac{u(x)}{v(x)} = \frac{(2x^3+3x^2+5)}{(x^2-2)}$

$$f'(x) = \frac{u'(x)v(x) - v'(x)u(x)}{v^2(x)}$$

$$f'(x) = \frac{(6x^2+6x)(x^2-2) - (2x^3+3x^2+5)(2x)}{(x^2-2)^2}$$

EX4: Use the chain rule to calculate $\frac{dy}{dx}$

$y(x) = u(x)^2 - 2u(x) + 1$ where $u(x) = (x^2+4)^2$

$\frac{dy(x)}{dx} = \frac{dy(x)}{du(x)}\frac{du(x)}{dx} = [2u(x)-2][2(x^2+4)2x] = [2(x^2+4)^2-2][2(x^2+4)2x]$

EX5: Use the chain rule to calculate $\frac{dy}{dx}$

$y(x) = 2u(x)^3 \quad u(v) = \sqrt{v^2+3} \quad v(x) = \frac{1}{x}$

$\frac{dy(x)}{du(x)} = 6u(x)^2 \quad \frac{du(v)}{dv} = (v^2+3)^{1/2} = \frac{1}{2}\frac{2v}{\sqrt{v^2+3}} \quad \frac{dv(x)}{dx} = -\frac{1}{x^2}$

$$\frac{dy(x)}{dx} = \frac{-6u(x)^2\, v(x)}{x^2\sqrt{v^2+3}}$$

EX6: Calculate the derivative of $y(x) = x^{(x^2+2x+1)}$

$$\ln(y(x)) = (x^2+2x+1)\ln(x)$$

$$\frac{\frac{dy(x)}{dx}}{y(x)} = (2x+2)\ln(x) + \frac{(x^2+2x+1)}{x}$$

$$\frac{dy(x)}{dx} = y(x)\left[(2x+2)\ln(x) + \frac{(x^2+2x+1)}{x}\right]$$

$$\frac{dy(x)}{dx} = x^{(x^2+2x+1)}\left[(2x+2)\ln(x) + \frac{(x^2+2x+1)}{x}\right]$$

NUMERICAL DERIVATIONS (DIFFERENCE FORMULAS)

In order to approximate derivation difference equations can be used.Difference equations can basically be derived from Taylor formulations

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2}h^2 + \frac{f^{(3)}(x_i)}{3!}h^3 + \frac{f^{(4)}(x_i)}{4!}h^4 + \ldots$$

If the first derivative is solved from this equation it is found as

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f''(x_i)}{2}h - \frac{f^{(3)}(x_i)}{3!}h^2 - \frac{f^{(4)}(x_i)}{4!}h^3 + O(h^4)\ldots$$

or

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} + O(h)$$

Term O(h) is the order of error. Error is proportional to the difference parameter h. When h is getting smaller, error wil also shrinked. If the second derivative is written the same way:

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + O(h)$$

In order to decrease the error, if the second derivative difference equation is substituted into thefirst derivative Taylor equation a first derivative equation with a second order error can be obtained.

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{2h^2}h + O(h^2)$$

$$f'(x_i) = \frac{2f(x_{i+1}) - 2f(x_i) - f(x_{i+2}) + 2f(x_{i+1}) - f(x_i)}{h} + O(h^2)$$

$$f'(x_i) = \frac{-f(x_{i+2}) + 4f(x_{i+1}) - 3f(x_i)}{2h} + O(h^2)$$

Difference equation can be written as forward or backward way. The backward difference equation: $f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h} + O(h)$. By taken avarage of forward and bakward formulas can create a central formula

$$f'(x_i) = \frac{f'(x_i)ileri + f'(x_i)geri}{2} = \frac{\frac{f(x_i) - f(x_{i-1})}{h}}{2} + \frac{\frac{f(x_i) - f(x_{i-1})}{h}}{2} = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + O(h^2)$$

Difference formulas for calculating derivatives are given (as coefficients) in the following tables

Derivative (Difference) equations

Central finite difference

Derivativ	Accurac	−5	−4	−3	−2	−1	0	1	2	3	4	5
1	2					−1/2	0	1/2				
	4				1/12	−2/3	0	2/3	−1/12			
	6			−1/60	3/20	−3/4	0	3/4	−3/20	1/60		
	8		1/280	−4/10	1/5	−4/5	0	4/5	−1/5	4/105	−1/28	
2	2					1	−2	1				
	4				−1/12	4/3	−5/2	4/3	−1/12			
	6			1/90	−3/20	3/2	−49/18	3/2	−3/20	1/90		
	8		−1/56	8/315	−1/5	8/5	−205/72	8/5	−1/5	8/315	−1/56	
3	2				−1/2	1	0	−1	1/2			
	4			1/8	−1	13/8	0	−13/8	1	−1/8		
	6		−7/24	3/10	−169/120	61/30	0	−61/3	169/120	−3/10	7/240	
4	2				1	−4	6	−4	1			
	4			−1/6	2	−13/2	28/3	−13/2	2	−1/6		
	6		7/240	−2/5	169/60	−122/	91/8	−122/	169/60	−2/5	7/240	
5	2			−1/2	2	−5/2	0	5/2	−2	1/2		
	4		1/6	−3/2	13/3	−29/6	0	29/6	−13/3	3/2	−1/6	
	6	−13/2	19/36	−87/3	13/2	−323/	0	323/48	−13/2	87/32	−19/3	13/28
6	2			1	−6	15	−20	15	−6	1		
	4		−1/4	3	−13	29	−75/2	29	−13	3	−1/4	
	6	13/240	−19/2	87/16	−39/2	323/8	−1023/2	323/8	−39/2	87/16	−19/2	13/24

Forward finite difference

Derivative	Accuracy	0	1	2	3	4	5	6	7	8
1	1	−1	1							
	2	−3/2	2	−1/2						
	3	−11/6	3	−3/2	1/3					
	4	−25/12	4	−3	4/3	−1/4				
	5	−137/60	5	−5	10/3	−5/4	1/5			
	6	−49/20	6	−15/2	20/3	−15/4	6/5	−1/6		
2	1	1	−2	1						
	2	2	−5	4	−1					
	3	35/12	−26/3	19/2	−14/3	11/12				
	4	15/4	−77/6	107/6	−13	61/12	−5/6			
	5	203/45	−87/5	117/4	−254/9	33/2	−27/5	137/180		
	6	469/90	−223/10	879/20	−949/18	41	−201/10	1019/180	−7/10	
3	1	−1	3	−3	1					
	2	−5/2	9	−12	7	−3/2				
	3	−17/4	71/4	−59/2	49/2	−41/4	7/4			
	4	−49/8	29	−461/8	62	−307/8	13	−15/8		
	5	−967/120	638/15	−3929/40	389/3	−2545/24	268/5	−1849/120	29/15	
	6	−801/80	349/6	−18353/120	2391/10	−1457/6	4891/30	−561/8	527/30	−469/240
4	1	1	−4	6	−4	1				
	2	3	−14	26	−24	11	−2			
	3	35/6	−31	137/2	−242/3	107/2	−19	17/6		
	4	28/3	−111/2	142	−1219/6	176	−185/2	82/3	−7/2	
	5	1069/80	−1316/15	15289/60	−2144/5	10993/24	−4772/15	2803/20	−536/15	967/240

Backward finite difference

Derivative	Accuracy	0	-1	-2	-3	-4	-5	-6	-7	-8
1	1	1	-1							
	2	3/2	-2	1/2						

	3	11/6	-3	3/2	-1/3					
	4	25/12	-4	3	-4/3	1/4				
	5	137/60	-5	5	-10/3	5/4	-1/5			
	6	49/20	-6	15/2	-20/3	15/4	-6/5	1/6		
2	1	1	−2	1						
	2	2	−5	4	−1					
	3	35/12	−26/3	19/2	−14/3	11/12				
	4	15/4	−77/6	107/6	−13	61/12	−5/6			
	5	203/45	−87/5	117/4	−254/9	33/2	−27/5	137/180		
	6	469/90	−223/10	879/20	−949/18	41	−201/10	1019/180	−7/10	
3	1	1	-3	3	1					
	2	5/2	-9	12	-7	3/2				
	3	17/4	-71/4	59/2	-49/2	41/4	-7/4			
	4	49/8	-29	461/8	-62	307/8	-13	15/8		
	5	967/120	-638/15	3929/40	-389/3	2545/24	-268/5	1849/120	-29/15	
	6	801/80	-349/6	18353/120	-2391/10	1457/6	-4891/30	561/8	-527/30	469/240
4	1	1	−4	6	−4	1				
	2	3	−14	26	−24	11	−2			
	3	35/6	−31	137/2	−242/3	107/2	−19	17/6		
	4	28/3	−111/2	142	−1219/6	176	−185/2	82/3	−7/2	
	5	1069/80	−1316/15	15289/60	−2144/5	10993/24	−4772/15	2803/20	−536/15	967/240

In order to Show utilisation of the table, the derivative of f(x)=sin(x)

y=f(x)=sin(x)

Derivative	Accuracy	-5	-4	-3	-2	-1	0	1	2	3	4	5
1	2	0.00000000	0.00000000	0.00000000	0.00000000	-0.50000000	0.00000000	0.50000000	0.00000000	0.00000000	0.00000000	0.00000000
	4	0.00000000	0.00000000	0.00000000	0.08333333	-0.66666667	0.00000000	0.66666667	-0.08333333	0.00000000	0.00000000	0.00000000
	6	0.00000000	0.00000000	-0.01666667	0.15000000	-0.75000000	0.00000000	0.75000000	-0.15000000	0.01666667	0.00000000	0.00000000
	8	0.00000000	0.00357143	-0.03809524	0.20000000	-0.80000000	0.00000000	0.80000000	-0.20000000	0.03809524	-0.00357143	0.00000000

Δx	0.01	-5	-4	-3	-2	-1	0	1	2	3	4	5		
x	0	-0.05	-0.04	-0.03	-0.02	-0.01	0	0.01	0.02	0.03	0.04	0.05		
y=f(x)	Accuracyy	-0.049979	-0.039989	-0.029996	-0.019998667	-0.01	0	0.009999833	0.0199987	0.0299955	0.0399893	0.049979	Σ	f'(x)
	2	0	0	0	0	0.0049999	0	0.004999917	0	0	0	0	0.01	0.999983
	4	0	0	0	-0.001666556	0.0066666	0	0.006666556	-0.001667	0	0	0	0.01	1
	6	0	0	0.0004999	-0.0029998	0.0074999	0	0.007499875	-0.003	0.000499925	0	0	0.01	1
	8	0	-0.000143	0.0011427	-0.003999733	0.0079999	0	0.007999867	-0.004	0.001142686	-0.000143	0	0.01	1

Derivative	Accuracy	-5	-4	-3	-2	-1	0	1	2	3	4	5
	2					1	-2	1				
	4				-0.083333333	1.3333333	-2.5	1.333333333	-0.083333			
2	6			0.0111111	-0.15	1.5	-2.722222	1.5	-0.15	0.011111111		
	8		-0.001786	0.0253968	-0.2	1.6	-2.847222	1.6	-0.2	0.025396825	-0.001786	

Δx	0.01	-5	-4	-3	-2	-1	0	1	2	3	4	5		
x	0	-0.05	-0.04	-0.03	-0.02	-0.01	0	0.01	0.02	0.03	0.04	0.05		
y=f(x)	Accuracy	-0.049979	-0.039989	-0.029996	-0.019998667	-0.01	0	0.009999833	0.0199987	0.0299955	0.0399893	0.049979	Σ	f''(x)
	2	0	0	0	0	-0.01	0	0.009999833	0	0	0	0	0	0
	4	0	0	0	0.001666556	-0.013333	0	0.013333111	-0.001667	0	0	0	-2.2E-19	-2.2E-17
	6	0	0	-0.000333	0.0029998	-0.015	0	0.01499975	-0.003	0.000333283	0	0	2.71E-19	2.71E-17
	8	0	7.141E-05	-0.000762	0.003999733	-0.016	0	0.015999733	-0.004	0.00076179	-7.14E-05	0	2.17E-19	2.17E-17

Derivative	Accuracy	-5	-4	-3	-2	-1	0	1	2	3	4	5
	2	0	0	0	-0.5	1	0	-1	0.5	0	0	0
3	4	0	0	0.125	-1	1.625	0	-1.625	1	-0.125	0	0
	6	0	-0.029167	0.30000000	-1.40833333	2.03333333	0.00000000	-2.03333333	1.40833333	-0.30000000	0.02916667	0

Δx	0.0001	-5	-4	-3	-2	-1	0	1	2	3	4	5		Third
x	0	-0.05	-0.04	-0.03	-0.02	-0.01	0	0.01	0.02	0.03	0.04	0.05		Derivative
y=f(x)	Accuracy	-0.049979	-0.039989	-0.0299955	-0.019998667	-0.0099998	0	0.009999833	0.01999867	0.0299955	0.03998933	0.049979	Σ	f'''(x)
	2	0	0	0	0.009999333	-0.0099998	0	-0.009999833	0.00999933	0	0	0	-1E-06	-0.999975
	4	0	0	-0.0037494	0.019998667	-0.0162497	0	-0.016249729	0.01999867	-0.003749438	0	0	-1E-06	-1
	6	0	0.0011664	-0.0089987	0.028164789	-0.020333	0	-0.020332994	0.02816479	-0.00899865	0.00116636	0	-1E-06	-1

First derivatives:

(central difference Formula)

$$f'(x_n) \cong \frac{f(x_n+\Delta x)-f(x_n-\Delta x)}{2\Delta x}$$

(Forward difference formula-same as in derivative definition except the limit)

$$f'(x_n) \cong \frac{f(x_n+\Delta x)-f(x_n)}{\Delta x}$$

(Backward difference formula)

$$f'(x_n) \cong \frac{f(x_n)-f(x_n-\Delta x)}{\Delta x}$$

In order to minimize error Δx should be relatively small, to have a further approximation a second or higher order derivative (difference) formulas can also be used

$$f'(x_n) \cong \frac{-f(x_n+2\Delta x)+8f(x_n+\Delta x)-8f(x_n-\Delta x)+f(x_n-2\Delta x)}{12\Delta x} \quad (2.5.5)$$

$$f'(x_n) \cong \frac{f(x_n+3\Delta x)-9f(x_n+2\Delta x)+45f(x_n+\Delta x)-45f(x_n-\Delta x)+9f(x_n-2\Delta x)-f(x_n-3\Delta x)}{60\Delta x} \quad (2.5.6)$$

$$f'(x_n) \cong \frac{-3f(x_n+4\Delta x)+32f(x_n+3\Delta x)-168f(x_n+2\Delta x)+672f(x_n+\Delta x)-672f(x_n-\Delta x)+168f(x_n-2\Delta x)-32f(x_n-3\Delta x)+3f(x_n-4\Delta x)}{840\Delta x}$$

Second derivatives:

$$f''(x_n) \cong \frac{f(x_n+\Delta x)-2f(x_n)+f(x_n-\Delta x)}{\Delta x^2}$$

$$f''(x_n) \cong \frac{-f(x_n+2\Delta x)+16f(x_n+\Delta x)-30f(x_n)+16f(x_n-\Delta x)-f(x_n-2\Delta x)}{12\Delta x^2}$$

```
function f(x)
   y=x*x-2.0*x+5.0
   return y
   end
function df(f,x)
   dx=0.001
   y=(f(x+dx)-f(x-dx))/(2.0*dx)
   return y
   end
function d2f(f,x)
   dx=0.001
   y=(f(x+dx)-2.0*f(x)+f(x-dx))/(dx*dx)
   return y
   end
function read(s)
  print("enter x = ")
  x=readline()
  z=tryparse(Float64,x)
  print("z = $z")
  return z
```

```
  end

x=read("x=")
y=f(x);
dy=df(f,x)
d2y=d2f(f,x)
print("x=$x,y=$y,dy=,$dy,d2y= $d2y")
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" derivative1.jl
enter x = 1
z = 1.0x=1.0,y=4.0,dy=,0.0,d2y= 2.000000000279556
> Terminated with exit code 0.
```

```
 function f(x)
   y=x*x-2.0*x+5.0
   return y
   end

function f(x)
   y=x*x-2.0*x+5.0
   return y
   end
function df(f,x)
   dx=0.001
   y=(-f(x+2.0*dx)+8.0*f(x+dx)-8.0*f(x-dx)+f(x-2.0*dx))/(12.0*dx)
   return y
   end
function d2f(f,x)
   dx=0.001
   y=(-f(x+2.0*dx)+16.0*f(x+dx)-30.0*f(x)+16.0*f(x-dx)-f(x-2.0*dx))/(12.0*dx*dx)
   return y
   end
function read(s)
  print("enter x = ")
  x=readline()
  z=tryparse(Float64,x)
  print("z = $z")
  return z
  end


x=read("x=")
y=f(x);
dy=df(f,x)
d2y=d2f(f,x)
print("x=$x,y=$y,dy=,$dy,d2y= $d2y")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" derivative2.jl
enter x = 1
z = 1.0x=1.0,y=4.0,dy=,-7.401486830834377e-14,d2y= 2.00000000064963
> Terminated with exit code 0.
```

The derivative formula coefficients can be calculated numerically by using Taylor expansion. The general derivation formula is applied into abstract class f_x in order to achieve a general derivation format of any given function. Algorithm used in here is given by Bength Fornberg[47] article "Generation of Finite Difference Formulas on Arbitrary Spaced Grids" an has the following form:

$$\left.\frac{d^m f(x)}{dx^m}\right|_{x=x_0} \approx \sum_{\nu=0}^{n} \delta_{n,\nu}^{m} f(x+\alpha_\nu h) \quad m=0,1,...,M; n=m,m+1,...,N$$

Enter M,x_0,α_0 ,α_1 ,α_2 ,α_3 ,... ,α_N

$\delta_{0,0}^{0} := 1$

$c1 := 1$

for n:=1 to N do

  c2:=1

for ν:=0 to n-1 do
  c3:=α_n - α_ν
  c2:=c2*c3
  if $n \le M$ *then* $\delta^n_{n-1,\nu} := 0$
  For m:=0 to min(n,M) do
    $\delta^m_{n,\nu} := ((\alpha_n - x_0)\delta^n_{n-1,\nu} - m\delta^{m-1}_{n-1,\nu})/c3$
    next m
  next ν
For m:=0 to min(n,M) do
    $\delta^m_{n,n} := \frac{c1}{c2}\left[m\delta^{m-1}_{n-1,n-1} - (\alpha_{n-1} - x_0)\delta^m_{n-1,n-1}\right]$
    next m
c1:=c2
next n

3. INTEGRAL

4.1 DEFINITIONS

Integral defines the area under the curve. It can also be defined as antiderivative(inverse derivative) It can be defined as without limits, or with integration boundaries

$$I = \int f(x)dx$$

Or with limits

$$I = \int_a^b f(x)dx$$

Some analytical integration formulas are given below:

f(x)	$\int f(x)dx$
$\int x^n\,dx$	$\frac{x^{n+1}}{n+1}$
$\int x^3\,dx$	$\frac{x^4}{4}$
$\int e^x dx$	e^x
$\int e^{bx} dx$	$\frac{e^x}{b}$
$\int a^{bx}\,dx$	$\frac{a^{bx}}{\mathrm{bln}(a)}$
$\int \cos(x)\,dx$	$\sin(x)$
$\int \sin(x)\,dx$	$-\cos(x)$
$\int sec^2(x)\,dx$	$\tan(x)$
$\int \cosh(x)\,dx$	$\sinh(x)$
$\int \sinh(x)\,dx$	$\cos\mathrm{h}(x)$
$\int sech^2(x)\,dx$	$\tanh(x)$
$\int \frac{dx}{x}$	$\ln(\lvert x\rvert)$
$\int \frac{dx}{\sqrt{1-x^2}}$	$\arcsin(x)$
$\int \frac{dx}{\sqrt{a^2-x^2}}$	$\arcsin\left(\frac{x}{a}\right)$

$\int -\frac{dx}{\sqrt{1-x^2}}$	arccos(x)
$\int \frac{dx}{1+x^2}$	arctan(x)+C
$\int \frac{dx}{a^2+x^2}$	$\frac{1}{a}arctan\left(\frac{x}{a}\right)$
$\int \frac{dx}{x\sqrt{x^2-a^2}}$	$\frac{1}{a}arcsec\left(\left\|\frac{x}{a}\right\|\right)$
$\int \frac{dx}{\sqrt{a^2+x^2}}$	$ln\left(x+\sqrt{a^2+x^2}\right)$
$\int \frac{dx}{\sqrt{a^2-x^2}}$	$ln\left(\left\|x+\sqrt{a^2-x^2}\right\|\right)$
$\int \frac{dx}{a^2-x^2}$	$\frac{1}{2a}ln\left(\left\|\frac{x+a}{x-a}\right\|\right)$
$\int \frac{dx}{x\sqrt{a^2-x^2}}$	$-\frac{1}{a}ln\left(\left\|\frac{a+\sqrt{a^2-x^2}}{x}\right\|\right)$
$\int \frac{dx}{x\sqrt{a^2+x^2}}$	$-\frac{1}{a}ln\left(\left\|\frac{a+\sqrt{a^2+x^2}}{x}\right\|\right)$
$\int u(x)v'(x)\,dx$	$u(x)v(x)-\int u'(x)v(x)\,dx$
$\int ku(x)\,dx$	$k\int u(x)\,dx$
$\int ln(x)\,dx$	$xln(x)-x$

Examples of analytical integrations

EX1: take the integral of $y=f(x)=2x^3+3x^2+5$

Answer: $I=\frac{2}{4}x^4+\frac{3}{3}x^3+5x+C$

EX2: take integral of $y=f(x)=u'(x)v(x)=(2x^3+3x^2+5)(2x-2)$

$v(x)=\frac{2}{4}x^4+\frac{3}{3}x^3+5x \qquad u'(x)=2$

$\int u(x)v'(x)\,dx=u(x)v(x)-\int u'(x)v(x)\,dx$

$$\int u(x)v'(x)\,dx=\left(\frac{2}{4}x^4+\frac{3}{3}x^3+5x\right)(2x-2)-\int 2\left(\frac{2}{4}x^4+\frac{3}{3}x^3+5x\right)dx$$

$$\int u(x)v'(x)\,dx=\left(\frac{2}{4}x^4+\frac{3}{3}x^3+5x\right)(2x-2)-2\left(\frac{2}{20}x^5+\frac{3}{12}x^4+\frac{5}{2}x^2\right)+C$$

EX3: take integral of

$$I=\int x^2e^x dx$$

$$\frac{d}{dx}(uv)=udv+vdu$$

$$\int udv=uv-\int vdu$$

$u=x^2 \quad dv=e^x dx \quad du=2xdx \quad v=e^x$

$\int x^2e^x dx=x^2e^x-\int 2xe^x dx$

$u=x \quad dv=e^x dx \quad du=dx \quad v=e^x$

$$\int x^2e^x dx=x^2e^x-2\left[xe^x-\int e^x dx\right]=x^2e^x-2xe^x+2e^x+C$$

EX4

Calculate $\int \frac{3x^2+2x+3}{(x^2+1)^2}$

$$\frac{3x^2+2x+3}{(x^2+1)^2} = \frac{Ax+B}{x^2+1} + \frac{Bx+D}{(x^2+1)^2} = \frac{(Ax+B)(x^2+1)+Bx+D}{(x^2+1)^2}$$

$$\frac{3x^2+2x+3}{(x^2+1)^2} = \frac{Ax^3+Bx^2+(A+C)x+(B+D)}{(x^2+1)^2}$$

So $A=0 \;\; B=3 \;\; C=2 \;\; D=0$

$$\int \frac{3x^2+2x+3}{(x^2+1)^2} = \int \frac{3dx}{(x^2+1)} + \int \frac{2xdx}{(x^2+1)^2}$$

$u=(x^2+1) \;\; du=2xdx$

$$\int \frac{2xdx}{(x^2+1)^2} = \int \frac{du}{u^2} = \int u^{-2}du = -\frac{1}{u} = -\frac{1}{(x^2+1)}$$

$\int \frac{3x^2+2x+3}{(x^2+1)^2} = 3\text{atan}(x) - \frac{1}{(x^2+1)}$+c

EX5

Calculate $\int cos^2(x)dx$

$cos^2(x)+= 1$

$cos^2(x) - sin^2(x) = \cos(2x)$

$cos^2(x) = \frac{1+\cos(2x)}{2}$

$$\int cos^2(x)dx = \int \frac{1+\cos(2x)}{2}\, dx = \frac{x}{2} + \frac{\sin(2x)}{4}$$

EX6

Calculate $\int xsin^2(x)dx$

$$\int xsin^2(x)dx = \int \frac{x(1-\cos(2x)}{2}dx = \int \frac{x}{2}dx - \int \frac{\text{xcos}(2x)}{2}dx = \frac{x^2}{4} - \int \frac{\text{xcos}(2x)}{2}dx$$

$$\int \frac{\text{xcos}(2x)}{2}dx$$

$$\int udv = uv - \int vdu$$

u=x dv=cos(2x)dx $du=dx$ $v=\int \cos(2x)\,dx = \frac{\sin(2x)}{2}$

$$\int \frac{\text{xcos}(2x)}{2}dx = x\frac{\sin(2x)}{2} - \int \frac{\sin(2x)}{2}dx = x\frac{\sin(2x)}{2} + \frac{\cos(2x)}{4}$$

$$\int xsin^2(x)dx = \frac{x^2}{4} - \left[x\frac{\sin(2x)}{2} + \frac{\cos(2x)}{4}\right] + C$$

4.2 NEWTON-COTES NUMERICAL INTEGRAL

If function f(x) is given and ,it is desired to calculate integral of the function between integration limits a and b. Definition of the integral is the area under the function between integration limits. The simplest approximation is to connect f(a) and f(b) points with a linear line and calculate the area of the trapezoidal under the line. This approach is called trapezoidal rule of Newton-Cotes formulations (Area finding formulations)

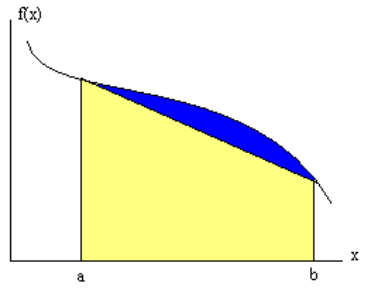


Figure 4.2 Finding integration with trapezoidal rule (yellow area) and error created by the method(blue area)

Integration formula :

$$I = \int_a^b f(x)dx = (b-a)\frac{f(a)+f(b)}{2}$$

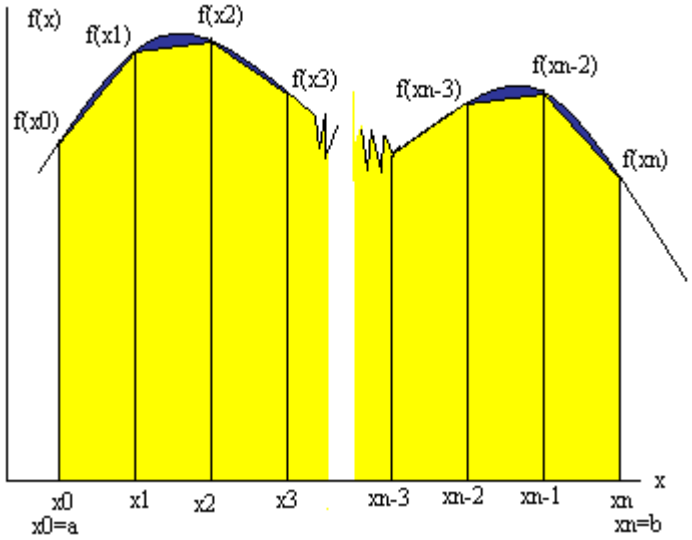


Figure 4.3 Finding integration with trapezoidal rule (yellow area) with multiple trapezoidal rule

If total area is divided into smaller areas amount of error will be reduced due to the fact that smaller piece of functions are closer to the line compare to a big piece of function. The same integration Formula for multipart equation will be:

$$I = \int_a^b f(x)dx = (x_n - x_0)\frac{f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)}{2n}$$

Now assume an example integration function of

$$I = \int_0^1 \frac{4}{\pi}\sqrt{1-x^2}\,dx = 1.0$$

Since the value of the function is known exactly, it will be easy to evaluate the error of numerical function evaluation process.

Trapezoidal intergral rule of Newton-Cotes approximation n=1

```
function  newton_cotes2(ff,a,b,n)
      #Newton-cotes integral 2 points
       #trapezoidal rule
       # ff integrating function
       # a,b integral limits
       # n : number of sub division h=(b-a)/n;
       h=(b-a)/n
       h1=h;
       sum=0.0;
       x=zeros(3)
       f=zeros(3)
       for p=1:n
         for i=1:2
           x[i]=a+(i-1)*h1+(p-1)*h
           f[i]=ff(x[i])
           end
         sum=sum+h*(f[1]+f[2])/2.0
       end
       return sum
       end
f(x)=4.0/pi*sqrt(1.0-x*x)
I=newton_cotes2(f,0.0,1.0,1)
print("Integral = $I")
```

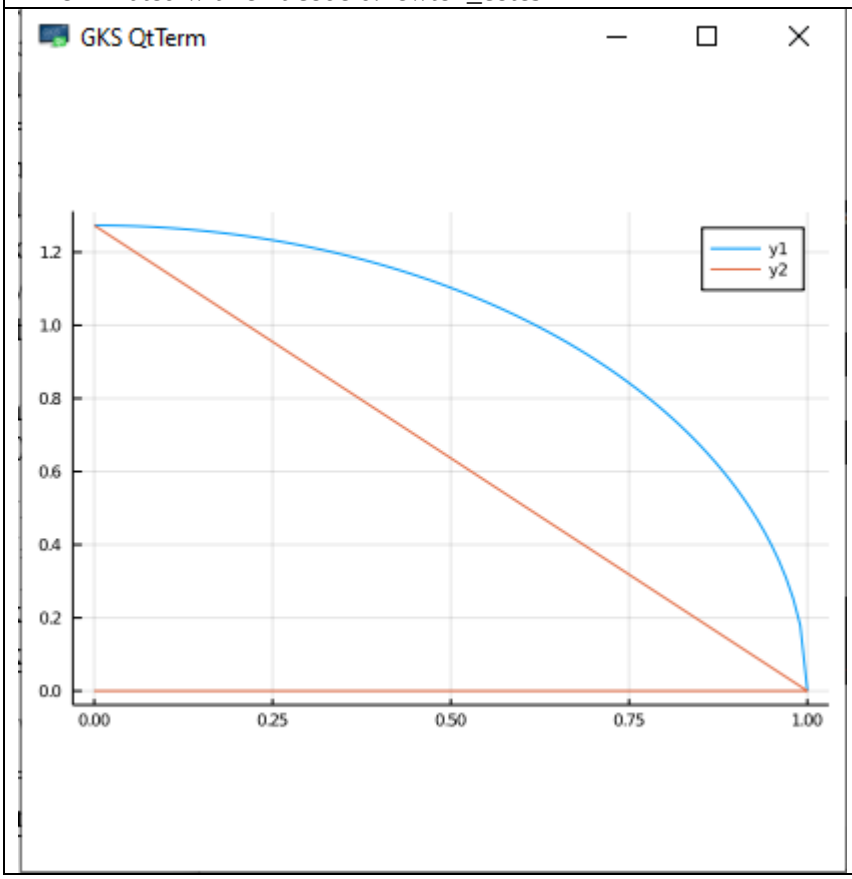
```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes2.jl
Integral = 0.6366197723675814
> Terminated with exit code 0.
```

In order to see the error happened in the function visiually a graphical output form of the same program is created. n=1

```
 function  newton_cotes2(ff,a,b,n)
      #Newton-cotes integral 2 points
       #trapezoidal rule
       # ff integrating function
       # a,b integral limits
       # n : number of sub division h=(b-a)/n;
       h=(b-a)/n
       h1=h;
       sum=0.0;
       x=zeros(3)
       f=zeros(3)
       xi=zeros(n+1)
       fi=zeros(n+1)
       xi[1]=a
       fi[1]=ff(a)
       for p=1:n
```

```
        for i=1:2
          x[i]=a+(i-1)*h1+(p-1)*h
          f[i]=ff(x[i])
          xi[p+1]=x[i]
          fi[p+1]=f[i]
          end
        sum=sum+h*(f[1]+f[2])/2.0
      end
      xx=[sum,xi,fi]
      return xx
      end
using Plots
f(x)=4.0/pi*sqrt(1.0-x*x)
xx=newton_cotes2(f,0.0,1.0,1)
I=xx[1]
print("Integral = $I")
x1=0:0.01:1
y1=f.(x1)
x2=xx[2]
y2=xx[3]
p=plot(x1,y1)
plot!(p,x2,y2)
```

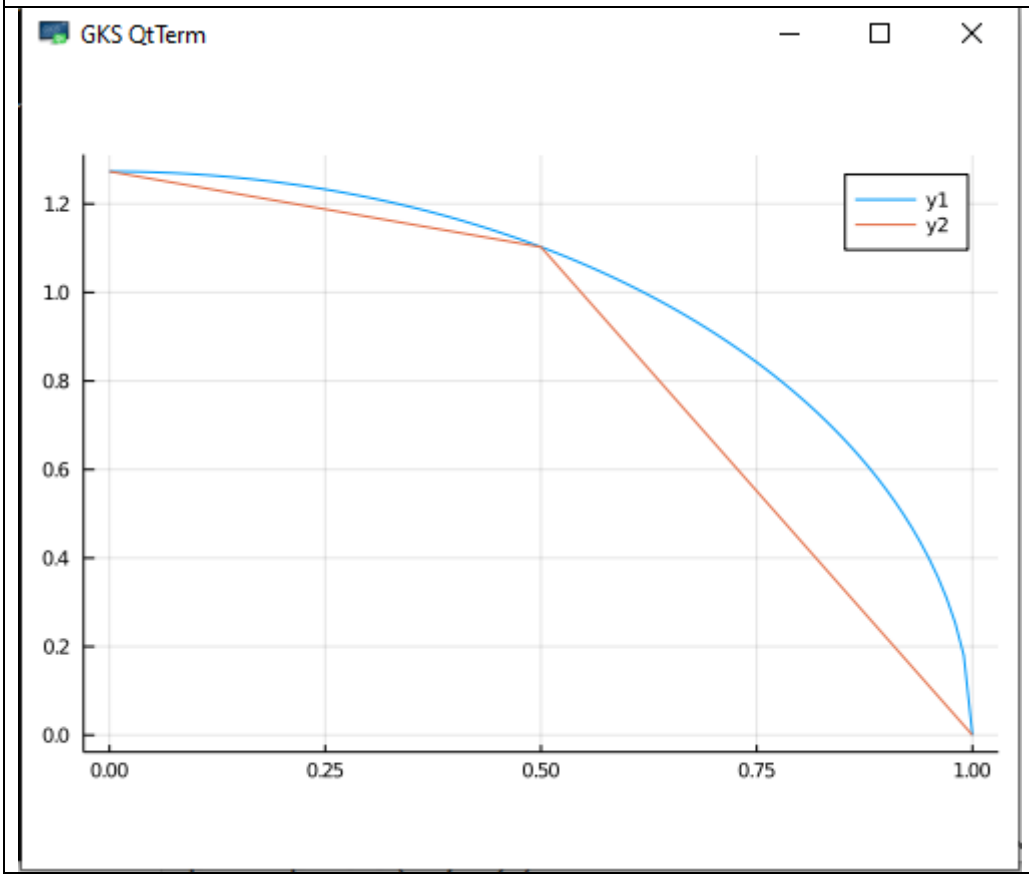
```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes2.jl
Integral = 0.6366197723675814
> Terminated with exit code 0.newton_cotes2
```



n=2

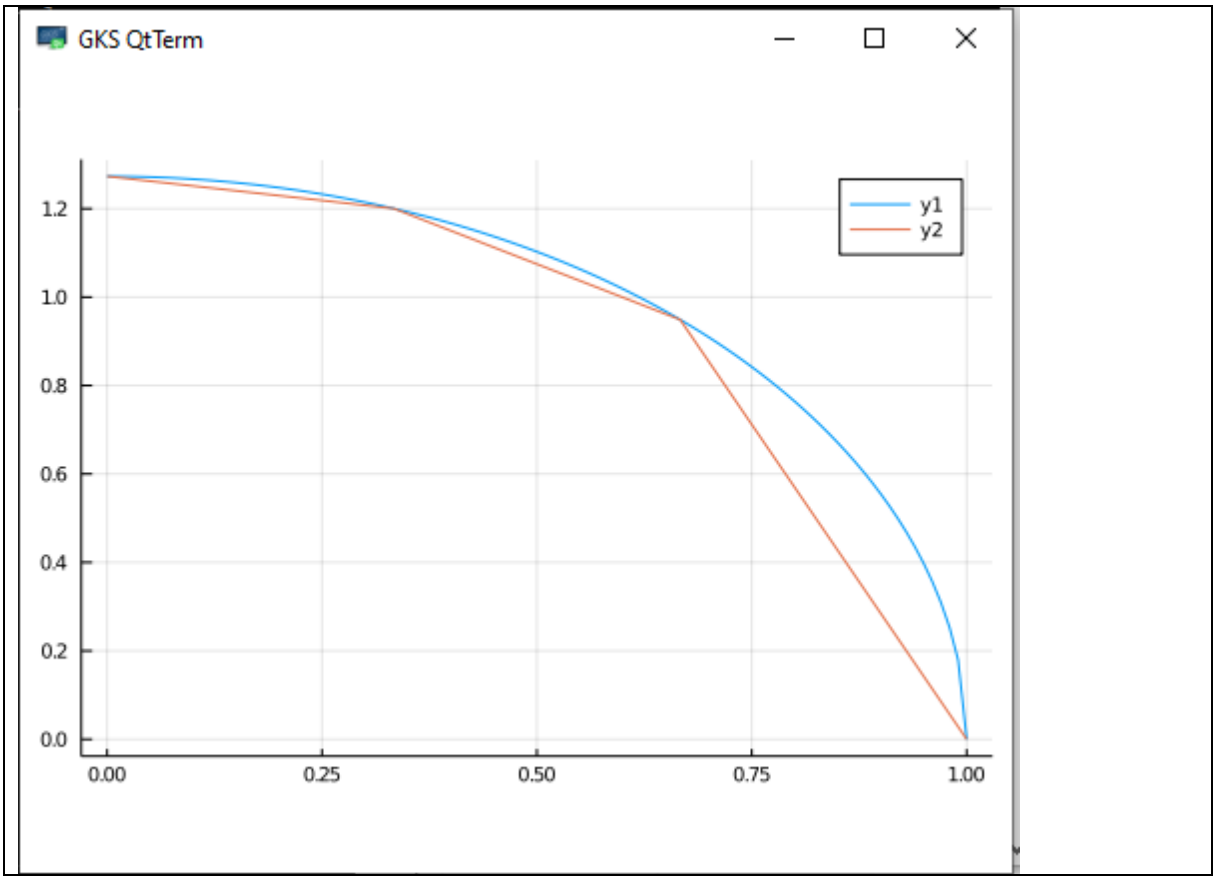
```
---------- Capture Output ----------
```

```
> "C:\co\Julia\bin\julia.exe" newton_cotes2.jl
Integral = 0.8696387816055828
> Terminated with exit code 0.
```



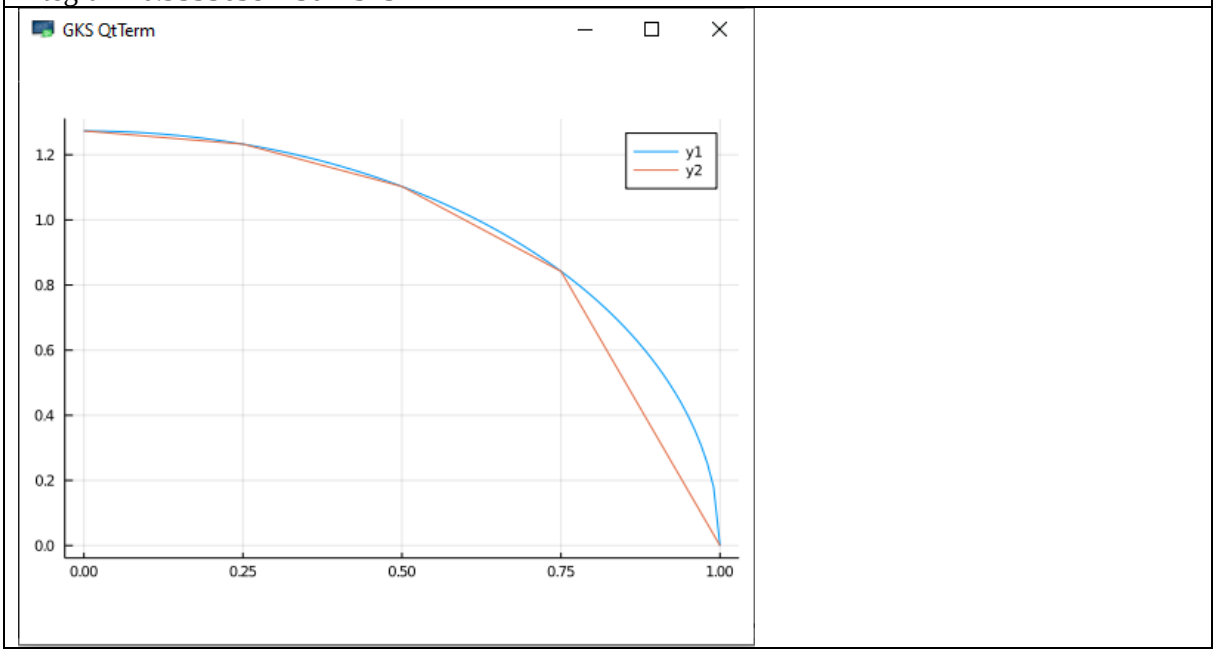
n=3

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes2.jl
Integral = 0.9286860839333196
> Terminated with exit code 0.
```



n=4

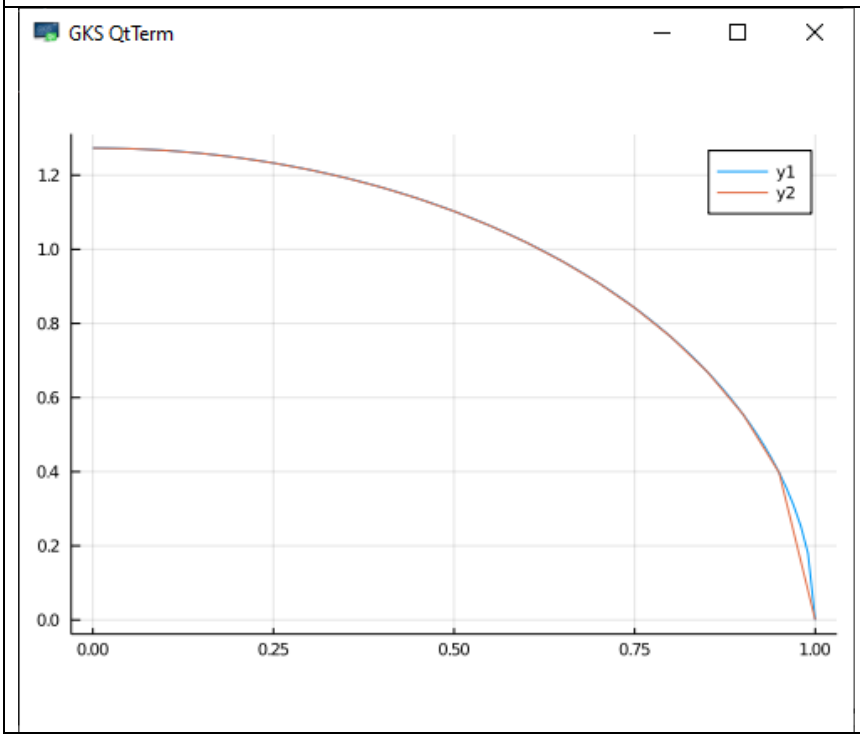
```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes2.jl
Integral = 0.9535638125074375
```



n=20

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes2.jl
Integral = 0.9958212998047945
```

> Terminated with exit code 0.



As it is seen from the example when trapezoidal steps are increased, approximation is getting better but even with 100 steps there are stil some error in the approximation. The next step in the approach is to use higher degrees of polinomials to connect points of integration. Quadratic (second degree) polinomial rule is called **Simpson 1/3 rule**. If step size of formulation is taken as h

h= (b – a)/n

Quadratic polynomial Newton-Cotes formula(Simpson 1/3 rule)

$$I=\int_a^b f(x)dx=\frac{(b-a)}{6}[f(x_0)+4f(x_1)+f(x_2)]$$ or

$$h=\frac{(b-a)}{n}=\frac{(b-a)}{2} \qquad I=\int_a^b f(x)dx=\frac{h}{3}[f(x_0)+4f(x_1)+f(x_2)]$$ (n=2 or three points are given in this Formula)

Quadratic polynomial Newton-Cotes formula(Simpson 1/3 rule) with multipoint approach

$$I=\int_a^b f(x)dx=(b-a)\frac{f(x_0)+4\sum_{i=1,3,5}^{n-1}f(x_i)+\sum_{j=2,4,6}^{n-2}f(x_j)+f(x_n)}{6}$$

Cubic polynomial Newton-Cotes formula(Simpson 3/8 rule)

$$I=\int_a^b f(x)dx=\frac{(b-a)}{8}[f(x_0)+3f(x_1)+3f(x_2)+f(x_3)]$$ or

$$h=\frac{(b-a)}{n}=\frac{(b-a)}{3} \qquad I=\int_a^b f(x)dx=\frac{3h}{8}[f(x_0)+3f(x_1)+3f(x_2)+f(x_2)]$$

(n=3 or four points are required for this formula)

Fourth order polynomial Newton-Cotes formula(Bode rule)

$$I=\int_a^b f(x)dx=\frac{(b-a)}{90}[7f(x_0)+32f(x_1)+12f(x_2)+32f(x_3)+7f(x_4)]$$

$$h=\frac{(b-a)}{n}=\frac{(b-a)}{4} \qquad I=\int_a^b f(x)dx=\frac{2h}{45}[7f(x_0)+32f(x_1)++12f(x_2)+32f(x_3)+7f(x_4)]$$

(n=4 or five points are used in this Formula)

Fifth order polynomial Newton-Cotes formula(Boole rule)

$$I=\int_a^b f(x)dx=\frac{(b-a)}{288}[19f(x_0)+75f(x_1)+50f(x_2)+50f(x_3)+75f(x_4)+19f(x_5)]$$

or

$$h=\frac{(b-a)}{n}=\frac{(b-a)}{5} \qquad I=\int_a^b f(x)dx=\frac{5h}{288}[19f(x_0)+75f(x_1)+50f(x_2)+50f(x_3)+75f(x_4)+19f(x_5)]$$

(n=5 or six points are used in this Formula)

Eight order polynomial Newton-Cotes formula

$$I=\int_a^b f(x)dx=\frac{(b-a)}{28350}[989f(x_0)+5888f(x_1)-928f(x_2)+10496f(x_3)-4540f(x_4)$$
$$+10496f(x_5)-928f(x_6)+5888f(x_7)+989f(x_8)]$$

or

$$h=\frac{(b-a)}{n}=\frac{(b-a)}{8} \qquad I=\int_a^b f(x)dx=\frac{4h}{14175}[989f(x_0)+5888f(x_1)-928f(x_2)+10496f(x_3)-4540f(x_4)$$
$$+10496f(x_5)-928f(x_6)+5888f(x_7)+989f(x_8)]$$

(n=8 or nine points are used in this Formula)

Programs for this approaches are given below:

quadratic polynomial Newton - Cotes Formula (Simpson 1/3 rule)

```
function  newton_cotes3(ff,a,b,n)
      #Newton-cotes integral 2 points
       #trapezoidal rule
       # ff integrating function
       # a,b integral limits
       # n : number of sub division h=(b-a)/n;
       h=(b-a)/n
       h1=h/2.0;
       sum=0.0;
       x=zeros(3)
       f=zeros(3)
       for p=1:n
         for i=1:3
           x[i]=a+(i-1)*h1+(p-1)*h
           f[i]=ff(x[i])
           end
```

```
        sum=sum+h1*(f[1]+4.0*f[2]+f[3])/3.0
      end
      return sum
      end
using Plots
f(x)=4.0/pi*sqrt(1.0-x*x)
I=newton_cotes3(f,0.0,1.0,1)
print("Integral Newton-cotes 1/3 = $I")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes3.jl
Integral Newton-cotes 1/3 = 0.9473117846849166
> Terminated with exit code 0.
```

If the quadratic Newton-Cotes Formula (Simpson 1/3 rule) applied as two times integral zone (0 to 1)

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes3.jl
Integral Newton-cotes 1/3 = 0.9815388228080559
> Terminated with exit code 0.
```

cubic polynomial Newton - Cotes Formula (Simpson 3/8 rule)

```
function  newton_cotes4(ff,a,b,n)
      #Newton-cotes integral 2 points
       #trapezoidal rule
       # ff integrating function
       # a,b integral limits
       # n : number of sub division h=(b-a)/n;
       h=(b-a)/n
       h1=h/3.0;
       sum=0.0;
       x=zeros(4)
       f=zeros(4)
       for p=1:n
         for i=1:4
           x[i]=a+(i-1)*h1+(p-1)*h
           f[i]=ff(x[i])
           end
         sum=sum+h*(f[1]+3.0*f[2]+3.0*f[3]+f[4])/8.0
       end
       return sum
       end
using Plots
f(x)=4.0/pi*sqrt(1.0-x*x)
I=newton_cotes4(f,0.0,1.0,1)
print("Integral Newton-cotes 3/8 = $I")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes4.jl
Integral Newton-cotes 3/8 = 0.965194372879037
> Terminated with exit code 0.
```

If the cubic Newton-Cotes Formula (Simpson 3/8 rule) applied as two times integral zone (0 to 1) and divided up into 2

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes4.jl
Integral Newton-cotes 3/8 = 0.9877870530380952
> Terminated with exit code 0.
```

fourth order polynomial Newton - Cotes Formula (Boole rule)

```
function  newton_cotes5(ff,a,b,n)
      #Newton-cotes integral 2 points
       #trapezoidal rule
       # ff integrating function
       # a,b integral limits
       # n : number of sub division h=(b-a)/n;
       h=(b-a)/n
       h1=h/4.0;
       sum=0.0;
       x=zeros(5)
       f=zeros(5)
```

```
        for p=1:n
          for i=1:5
            x[i]=a+(i-1)*h1+(p-1)*h
            f[i]=ff(x[i])
            end
          sum=sum+h*(7.0*f[1]+32.0*f[2]+12.0*f[3]+32.0*f[4]+7.0*f[5])/90.0
        end
        return sum
        end
using Plots
f(x)=4.0/pi*sqrt(1.0-x*x)
I=newton_cotes5(f,0.0,1.0,1)
print("Integral Newton-cotes (boole rule) = $I")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes5.jl
Integral Newton-cotes (boole rule) = 0.9838206253495985
> Terminated with exit code 0.
```

If the fourth order polynomial Newton-Cotes Formula (Boole rule) applied as two times integral zone (0 to 1) and divided up into 2

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes5.jl
Integral Newton-cotes (boole rule) = 0.9943031391613758
> Terminated with exit code 0.
```

fifth order polynomial Newton - Cotes Formula

```
function  newton_cotes6(ff,a,b,n)
       #Newton-cotes integral 2 points
        #trapezoidal rule
        # ff integrating function
        # a,b integral limits
        # n : number of sub division h=(b-a)/n;
        h=(b-a)/n
        h1=h/5.0;
        sum=0.0;
        x=zeros(6)
        f=zeros(6)
        for p=1:n
          for i=1:6
            x[i]=a+(i-1)*h1+(p-1)*h
            f[i]=ff(x[i])
            end
          sum=sum+h*(19.0*f[1]+75.0*f[2]+50.0*f[3]+50.0*f[4]+75.0*f[5]+19.0*f[6])/288.0
        end
        return sum
        end
using Plots
f(x)=4.0/pi*sqrt(1.0-x*x)
I=newton_cotes6(f,0.0,1.0,1)
print("Integral Newton-cotes (boole rule) = $I")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes6.jl
Integral Newton-cotes (boole rule) = 0.9872489399864514
> Terminated with exit code 0.
```

If the fifth order polynomial Newton-Cotes Formula applied as two times integral zone (0 to 1) and divided up into 2

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes6.jl
Integral Newton-cotes (boole rule) = 0.9955081956385212
> Terminated with exit code 0.
```

RİCHARDSON AND AITKEN INTERPOLATION FORMULAS

RİCHARDSON EXTRAPOLATION (ROMBERG INTEGRATION) FORMULA

Richardson extrapolation formula can be applied to decrease integration error. Richardson extrapolation Formula has the folloving form :

$$I \cong I(h_2) + \frac{1}{(h_1/h_2)^2 - 1}[I(h_2) - I(h_1)]$$

In this Formula h_2 should be smaller than h_1 . If$h_2=h_1/2$ equation becomes:

$$I \cong \frac{4}{3} I(h_2) - \frac{1}{3} I(h_1)]$$

As a starting interpolation any of the Newton-Cotes formulations can be taken. A General form of the above equation can be written as

$$I_{j,k} \cong \frac{4^{k-1} I_{j+1,k-1} - I_{j,k-1}}{4^{k-1} - 1}$$

If trapezoidal rule is used as starting point of this iterative formulation, it is called **Romberg Integration**

Consider the previous integral $I = \int_0^1 \frac{4}{\pi} \sqrt{1-x^2}\, dx = 1.0$ that solved by excel, if the results are integrated into the romberg integration formula:

Romberg: Trapezoidal

2*h	I0	I1	I2
0.125	0.98352	0.99771	0.999685
0.0625	0.994162	0.999191	
0.03125	0.997934		

Romberg: Simpson 1/3

2*h	I0	I1	I2
0.125	0.993505	0.999111	0.999877
0.0625	0.99771	0.999685	
0.03125	0.999191		

Romberg integration with trapezoidal rule (Newton-Cotes 2)f(x)= 2.0/(1.0+2.0*x*x)

```
function  newton_cotes2(ff,a,b,n)
    #Newton-cotes integral 2 points
     #trapezoidal rule
     # ff integrating function
     # a,b integral limits
     # n : number of sub division h=(b-a)/n;
     h=(b-a)/n
     h1=h;
     sum=0.0;
     x=zeros(2)
     f=zeros(2)
     for p=1:n
       for i=1:2
         x[i]=a+(i-1)*h1+(p-1)*h
         f[i]=ff(x[i])
         end
       sum=sum+h1*(f[1]+f[2])/2.0
     end
     return sum
     end
function romberg_integral(f, a,b,n)
   m = n
   R = zeros(Float64,n,m)
   # calculate Newton-cotes 2
   h=(a - b)/2.0
   for j=1:n
     m1=2*j
     R[j,1]=newton_cotes2(f,a,b,m1)
     y=R[j,1]
     println("R[$j,1] = $y")
```

```
    end
    for k=2:n
      for j=1:m-1
        n1=4^(k-1)
        n2=n1-1
         R[j,k]=(n1*R[j+1,k-1]-R[j,k-1])/n2
      end
    end
    return R
end

function output(R,nn)
   for i=1:nn
     for j=i:nn-1
         print(R[j,i])
         print(" ")
       end
       println(" ")
     end
     return nothing
   end

f(x)=2.0/(1.0+2*x*x)
a=-3.0
b=3.0
n=8
R=romberg_integral(f, a,b,n)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), R)
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_romberg.jl
R[1,1] = 6.315789473684211
R[2,1] = 4.248803827751196
R[3,1] = 3.883040935672515
R[4,1] = 3.8058295411062657
R[5,1] = 3.789612338685158
R[6,1] = 3.786638312954102
R[7,1] = 3.7864174566839495
R[8,1] = 3.786675582381121
8X8 Matrix{Float64}:
 6.31579  3.55981   3.77454    3.78147     3.78454     3.78577     3.7864       3.7868
 4.2488   3.76112   3.78136    3.78453     3.78577     3.7864      3.7868      -0.337943
 3.88304  3.78009   3.78448    3.78576     3.7864      3.7868     -0.337691     0.00569917
 3.80583  3.78421   3.78574    3.7864      3.7868     -0.336684    0.00567821  -2.25949e-5
 3.78961  3.78565   3.78639    3.7868     -0.332657    0.00559463 -2.22469e-5   2.18218e-8
 3.78664  3.78634   3.78679   -0.316566    0.0052643  -2.08756e-5   2.04626e-8  -5.00041e-12
 3.78642  3.78676  -0.252451   0.00400716 -1.57143e-5   1.5361e-8   -3.75117e-12  2.28967e-16
 3.78668  0.0       0.0        0.0         0.0         0.0         0.0          0.0
> Terminated with exit code 0.
```

QUADRATURES: GAUSS-LEGENDRE, GAUSS - CHEBYCHEV, GAUSS -JACOBİ, GAUSS -HERMİT, GAUSS -LEQUERRE INTEGRATİON FORMULAS

A problem known from antique times, is to create a rectangle with the same area with a polynomial. IT is called a quadrature problem.

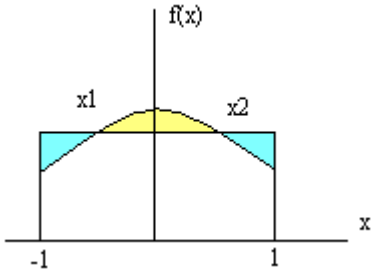


Figure : a rectangle with the same area with a polynomial

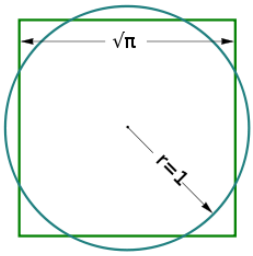


Figure : A rectangle with the same area with a circle

If point x1 and x2 points intersecting the polynomial and rectangle is known, area calculation of the polynomial can be interchange with the area calculation of the rectangle. Or in more general means instead of integration process of summation can be substituted. As a general definition:

$$I_w(-1,1) \cong \int_{-1}^{1} f(x)w(x)dx \cong \sum_{K01}^{n} c_k f(x_k)$$

Can be written. In this equation w(x) is the weight factor. In the first of a such formulation, Gauss-Legendre integral formulation, weight factor can be taken as 1.

$$I_w(-1,1) \cong \int_{-1}^{1} f(x)dx \cong \sum_{K01}^{n} c_k f(x_k)$$

If this integration is solved for a general polynamial equation. Following relation is found:

$$\int_{-1}^{1} x^k dx \cong \frac{1-(-1)^{k+1}}{k+1}$$

This equation can be written in open form as:

$$c_1 x_1^k + c_2 x_2^k + \cdots + c_n x_n^k = \frac{1-(-1)^{k+1}}{k+1} \qquad 0 \le k \le 2n$$

For a special case of n=2 equation becomes :

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

$$I(-1,1) = \int_{-1}^{1} f(x)dx \cong c_1 f(x_1) + c_2 f(x_2) = \int_{-1}^{1} [a_0 + a_1 x + a_2 x^2 + a_3 x^3]dx$$

$$c_1[a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^2] + c_2[a_0 + a_1 x_2 + a_2 x_2^2 + a_3 x_2^2]$$

$$= a_0 \int_{-1}^{1} dx + a_1 \int_{-1}^{1} x dx + a_2 \int_{-1}^{1} x^2 dx + a_3 \int_{-1}^{1} x^3 dx$$

$$c_1[a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^2] + c_2[a_0 + a_1 x_2 + a_2 x_2^2 + a_3 x_2^2]$$

$$= a_0(1-(-1)) + \frac{a_1}{2}(1^2-(-1)^2) + \frac{a_2}{3}(1^3-(-1)^3) + \frac{a_4}{4}(1^4-(-1)^4)$$

$$a_0(c_1+c_2) + a_1(c_1 x_1 + c_2 x_2) + a_2(c_1 x_1^2 + c_2 x_2^2) + a_3(c_1 x_1^3 + c_2 x_2^3)$$

$$= a_0(2) + a_1(0) + a_2\left(\frac{2}{3}\right) + a_3(0)$$

Coefficients of a_0, a_1, a_2 and a_3 must be equal to each other so

$$c_1 + c_2 = 2$$

$$c_1 x_1 + c_2 x_2 = 0$$

$$c_1 x_1^2 + c_2 x_2^2 = \frac{2}{3}$$

$$c_1 x_1^3 + c_2 x_2^3 = 0$$

If this system of equation is solved roots and coefficients can be found as:

Roots: $x_{1,2} = \pm\frac{1}{\sqrt{3}}$ and Coefficients : $c_{1,2} = 1$

The general solution of this problem can be defined with the Legendre polynomials. Legendre polynomials has a general definition as:

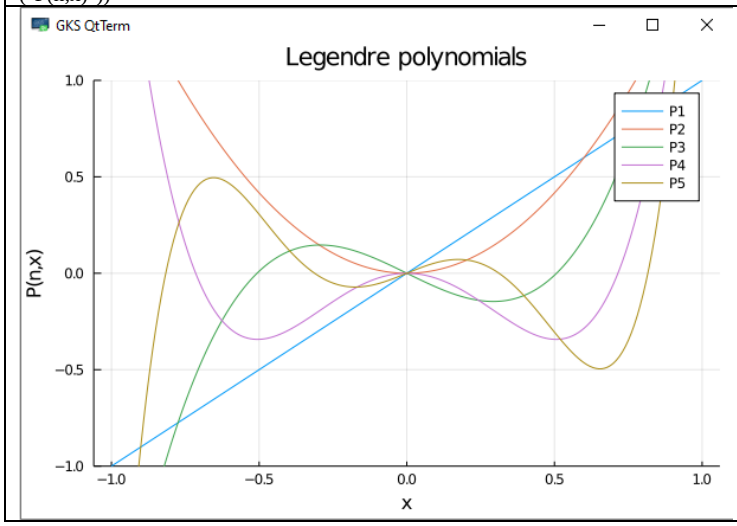
$$(k+1)P_{k+1}(x) = (2k+1)xP_k(x) - kP_{k-1}(x) \qquad k \geq 1$$

And

$$P_0(x) = 1$$
$$P_1(x) = x$$

The first 5 Legendre polynomials are giving in graphic form.

```
function P(n,x)
   P1=0.0
   if n==0 P1=0.0
   elseif n==1 P1=x
   else
     P1=((2.0*n+1.0)*x*P(n-1,x)-n*P(n-2,x))/(n+1.0)
   end
   return P1
   end

using Plots
x1=-1:0.01:1
f1(x)=P(1,x)
f2(x)=P(2,x)
f3(x)=P(3,x)
f4(x)=P(4,x)
f5(x)=P(5,x)
plot([f1,f2,f3,f4,f5],x1,title="Legendre polynomials",label=["P1" "P2" "P3" "P4" "P5"],ylims = (-1.0,1.0),xaxis = ("x"),yaxis =
("P(n,x)"))
```



Adrien Marie Legendre

The roots of Legendre Polynomials are the roots of the Gauss-Legendre integration Formula. For example if $P_2(x)$ value is calculated from the above general form:

$$2P_2(x) = (2 * 2 + 1)xP_1(x) - 1P_0(x)$$
$$2P_2(x) = 3x * x - 1$$
$$P_2(x) = \frac{3x^2 - 1}{2}$$

The root of this is equal to

$$P_2(x) = \frac{3x^2 - 1}{2} = 0$$
$$x_{1,2} = \pm\frac{1}{\sqrt{3}}$$

In general roots can be calculated from Legendre polynomials as follows:

$$\theta_{n,k} = \frac{n - k + 3/4}{n + 1/2}\pi$$
$$x_{n,k} = \left[1 - \frac{1}{8n^2} + \frac{1}{8n^3} - \frac{1}{384n^4}\left(39 - \frac{28}{sin^2(\theta_{n,k})}\right)\right]\cos(\theta_{n,k}) + E(n^{-5})$$

This equation contains an error level of $\mathbf{n^{-5}}$. In order to decrease the error Newton-Raphson root finding method can be usefull. The above equation can be used as first estimate in Newton-Raphson formula:

$$x_{v+1} = x_v - \frac{1 - x_v^2}{n}\frac{P_n(x_v)}{P_{n-1}(x_v) - x_v P_n(x_v)}$$

After finding the roots coeficients can be calculated as

$$c_k = \frac{2(1 - x_k^2)}{[nP_{n-1}(x_k)]^2}$$

As a last concept, consider that Gauss-Legendre integration limits are -1 and 1
Region x=[-1,1] can be converted to z=[a,b] by changing the variables

$$z = \left(\frac{b - a}{2}\right)x + \left(\frac{b + a}{2}\right) = \alpha x + \beta$$

$$\alpha = \left(\frac{b - a}{2}\right)$$
$$\beta = \left(\frac{b + a}{2}\right)$$

In this case, integration Formula will became :

$$I_w(a,b) \cong \int_a^{b1} f(z)dz \cong \alpha \sum_{K01}^{n} c_k f(\alpha x_k + \beta)$$

First 10 Gauss-Legendre integration formulas are given in the table

Table Gauss – Legendre integration roots and coefficients

N	x_k	c_k
2	-0.577350269189625	1.000000000000000
	0.577350269189625	1.000000000000000
3	-0.774596669241483	0.555555555555552
	0.000000000000000	0.888888888888888
	0.774596669241483	Format long
4	-0.861136311594052	0.347854845137447
	-0.339981043584856	0.652145154862546
	0.339981043584856	0.652145154862546
	0.861136311594052	0.347854845137447

5	**-0.906179845938664**	**0.236926885056181**
	-0.538469310105683	**0.478628670499366**
	0.000000000000000	**0.568888888888888**
	0.538469310105683	**0.478628670499366**
	0.906179845938664	**0.236926885056181**
6	**-0.932469514203152**	**0.171324492379162**
	-0.661209386466264	**0.360761573048138**
	-0.238619186083196	**0.467913934572691**
	0.238619186083196	**0.467913934572691**
	0.661209386466264	**0.360761573048138**
	0.932469514203152	**0.171324492379162**
7	**-0.949107912342758**	**0.129484966168862**
	-0.741531185599394	**0.279705391489276**
	-0.405845151377397	**0.381830050505119**
	0.000000000000000	**0.417959183673469**
	0.405845151377397	**0.381830050505119**
	0.741531185599394	**0.279705391489276**
	0.949107912342758	**0.129484966168862**
8	**-0.960289856497536**	**0.101228536290369**
	-0.796666477413626	**0.222381034453374**
	-0.525532409916329	**0.313706645877887**
	-0.183434642495649	**0.362683783378362**
	0.183434642495649	**0.362683783378362**
	0.525532409916329	**0.313706645877887**
	0.796666477413626	**0.222381034453374**
	0.960289856497536	**0.101228536290369**
9	**-0.968160239507626**	**0.081274388361569**
	-0.836031107326635	**0.180648160694857**
	-0.613371432700590	**0.260610696402935**
	-0.324253423403808	**0.312347077040002**
	0.000000000000000	**0.330239355001259**
	0.324253423403808	**0.312347077040002**
	0.613371432700590	**0.260610696402935**
	0.836031107326635	**0.180648160694857**
	0.968160239507626	**0.081274388361569**
10	**-0.973906528517171**	**0.066671344308684**
	-0.865063366688984	**0.149451349150580**
	-0.679409568299024	**0.219086362515982**
	-0.433395394129247	**0.269266719309996**
	-0.148874338981631	**0.295524224714752**
	0.148874338981631	**0.295524224714752**
	0.433395394129247	**0.269266719309996**
	0.679409568299024	**0.219086362515982**
	0.865063366688984	**0.149451349150580**
	0.973906528517171	**0.066671344308684**

EXAMPLE : Calculate integral of $\boldsymbol{f(x) = \frac{\pi}{4}\sqrt{1 - x^2}}$ function between limits 0 and 1 with three points Gauss-Legendre integration formula by hand and by computer program

For n=3 $\alpha=(1-0)/2=0.5$ $\beta=(1+0)/2=0.5$

I=0.5*(0.555555555555553 *f(0.5*-0.774596669241483+0.5)+ 0.888888888888889*f(0.5*0+0.5)+ 0.555555555555553 *f(0.5*0.774596669241483+0.5))

I=1.004609

excel versions of the same integral

For n=4

Gauss Integral Formula

a		0.0000000000
b		1.0000000000
a	(b-a)/2	0.5000000000
b	(b+a)/2	0.5000000000

rk	**Ck**	**xk=a+b*rk**	**yk=f(xk)**	**z=a*Ck*yk**
-0.861136312	**0.347854845**	0.0694318442	1.2701668325	0.2209168434
-0.339981044	**0.652145155**	0.3300094782	1.2019096333	0.3919097720
0.339981044	**0.652145155**	0.6699905218	0.9452143982	0.3082084950
0.861136312	**0.347854845**	0.9305681558	0.4661568155	0.0810774534

	Integral	1.002112564

For n=6

Gauss Integral Formula

a		0.0000000000
b		1.0000000000
a	(b-a)/2	0.5000000000
b	(b+a)/2	0.5000000000

rk	Ck	xk=a+b*rk	yk=f(xk)	z=a*Ck*yk
-0.932469514	**0.171324492**	0.0337652429	1.2725135329	0.1090063675
-0.661209386	**0.360761573**	0.1693953068	1.2548389257	0.2263488324
-0.238619186	**0.467913935**	0.3806904070	1.1773678593	0.2754534138
0.238619186	**0.467913935**	0.6193095930	0.9996800599	0.2338821151
0.661209386	**0.360761573**	0.8306046932	0.7090193556	0.1278934690
0.932469514	**0.171324492**	0.9662347571	0.3280671473	0.0281029687
			Integral	1.000687166

For n=10

Gauss Integral Formula

a		0.0000000000
b		1.0000000000
a	(b-a)/2	0.5000000000
b	(b+a)/2	0.5000000000

rk	Ck	xk=a+b*rk	yk=f(xk)	z=a*Ck*yk
-0.973906529	**0.066671344**	0.0130467357	1.2731311764	0.0424406835
-0.865063367	**0.149451349**	0.0674683167	1.2703383629	0.0949268911
-0.679409568	**0.219086363**	0.1602952159	1.2567754345	0.1376711792
-0.433395394	**0.269266719**	0.2833023029	1.2210757598	0.1643975319
-0.148874339	**0.295524225**	0.4255628305	1.1521912348	0.1702502107
0.148874339	**0.295524225**	0.5744371695	1.0422085462	0.1539989363
0.433395394	**0.269266719**	0.7166976971	0.8879368571	0.1195459222
0.679409568	**0.219086363**	0.8397047841	0.6914240280	0.0757407876
0.865063367	**0.149451349**	0.9325316833	0.4597517249	0.0343552578
0.973906529	**0.066671344**	0.9869532643	0.2050004799	0.0068338288
			Integral	1.000161229

For n=60

$$f(x) = \frac{\pi}{4}\sqrt{1 - x^2}$$

Gauss Integral Formula

a		0.0000000000
b		1.0000000000
a	(b-a)/2	0.5000000000
b	(b+a)/2	0.5000000000

rk	Ck	xk=a+b*rk	yk=f(xk)	z=a*Ck*yk
-0.999210123227436	0.002026811968873	0.0003949384	1.2732394454	0.0012903085
-0.995840525118838	0.004712729926953	0.0020797374	1.2732367912	0.0030002106
-0.989787895222221	0.007389931163346	0.0051060524	1.2732229468	0.0047045150
-0.981067201752598	0.010047557182288	0.0094663991	1.2731824942	0.0063961870
-0.969701788765052	0.012678166476816	0.0151491056	1.2730934350	0.0080702453
-0.955722255839996	0.015274618596784	0.0221388721	1.2729274803	0.0097217409
-0.939166276116423	0.017829901014207	0.0304168619	1.2726504171	0.0113456155
-0.920078476177627	0.020337120729457	0.0399607619	1.2722225443	0.0129366717
-0.898510310810046	0.022789516943998	0.0507448446	1.2715991671	0.0144895654
-0.874519922646898	0.025180477621521	0.0627400387	1.2707311395	0.0159988085
-0.848171984785929	0.027503556749925	0.0759140076	1.2695654443	0.0174587826
-0.819537526162145	0.029752491500789	0.0902312369	1.2680457997	0.0188637609
-0.788693739932264	0.031921219019296	0.1056531300	1.2661132804	0.0202079397
-0.755723775306585	0.034003892724946	0.1221381123	1.2637069454	0.0214854777
-0.720716513355730	0.035994898051084	0.1396417433	1.2607644611	0.0226905441
-0.683766327381355	0.037888867569243	0.1581168363	1.2572227133	0.0238173724
-0.644972828489477	0.039680695452381	0.1775135858	1.2530184000	0.0248603208

-0.604440597048510	0.041365551235585	0.1977797015	1.2480886002	0.0258139365
-0.562278900753944	0.042938892835936	0.2188605496	1.2423713133	0.0266730243
-0.518601400058569	0.044396478795787	0.2406993000	1.2358059656	0.0274327167
-0.473525841761707	0.045734379716114	0.2632370791	1.2283338822	0.0280885441
-0.427173741583078	0.046948988848912	0.2864131292	1.2198987211	0.0286365057
-0.379670056576798	0.048037031819971	0.3101649717	1.2104468712	0.0290731374
-0.331142848268448	0.048995575455757	0.3344285759	1.1999278109	0.0293955768
-0.281722937423261	0.049822035690550	0.3591385313	1.1882944300	0.0296016238
-0.231543551376029	0.050514184532509	0.3842282243	1.1755033149	0.0296897957
-0.180739964873425	0.051070156069856	0.4096300176	1.1615149982	0.0296593761
-0.129449135396945	0.051488451500981	0.4352754323	1.1462941736	0.0295104560
-0.077809333949537	0.051767943174910	0.4610953330	1.1298098787	0.0292439668
-0.025959772301248	0.051907877631221	0.4870201138	1.1120356457	0.0288617051
0.025959772301248	0.051907877631221	0.5129798862	1.0929496235	0.0283663477
0.077809333949537	0.051767943174910	0.5389046670	1.0725346712	0.0277614570
0.129449135396945	0.051488451500981	0.5647245677	1.0507784251	0.0270514770
0.180739964873425	0.051070156069856	0.5903699824	1.0276733412	0.0262417190
0.231543551376029	0.050514184532509	0.6157717757	1.0032167145	0.0253383371
0.281722937423261	0.049822035690550	0.6408614687	0.9774106755	0.0243482948
0.331142848268448	0.048995575455757	0.6655714241	0.9502621669	0.0232793209
0.379670056576798	0.048037031819971	0.6898350283	0.9217829005	0.0221398573
0.427173741583078	0.046948988848912	0.7135868708	0.8919892958	0.0209389978
0.473525841761707	0.045734379716114	0.7367629209	0.8609024020	0.0196864187
0.518601400058569	0.044396478795787	0.7593007000	0.8285478029	0.0183923025
0.562278900753944	0.042938892835936	0.7811394504	0.7949555081	0.0170672547
0.604440597048510	0.041365551235585	0.8022202985	0.7601598294	0.0157222152
0.644972828489477	0.039680695452381	0.8224864142	0.7241992435	0.0143683648
0.683766327381355	0.037888867569243	0.8418831637	0.6871162440	0.0130170282
0.720716513355730	0.035994898051084	0.8603582567	0.6489571814	0.0116795738
0.755723775306585	0.034003892724946	0.8778618877	0.6097720929	0.0103673124
0.788693739932264	0.031921219019296	0.8943468700	0.5696145247	0.0090913950
0.819537526162145	0.029752491500789	0.9097687631	0.5285413466	0.0078627110
0.848171984785929	0.027503556749925	0.9240859924	0.4866125631	0.0066917881
0.874519922646898	0.025180477621521	0.9372599613	0.4438911240	0.0055886953
0.898510310810046	0.022789516943998	0.9492551554	0.4004427432	0.0045629483
0.920078476177627	0.020337120729457	0.9600392381	0.3563357382	0.0036234215
0.939166276116423	0.017829901014207	0.9695831381	0.3116409221	0.0027782634
0.955722255839996	0.015274618596784	0.9778611279	0.2664316177	0.0020348207
0.969701788765052	0.012678166476816	0.9848508944	0.2207839805	0.0013995680
0.981067201752598	0.010047557182288	0.9905336009	0.1747782074	0.0008780470
0.989787895222221	0.007389931163346	0.9948939476	0.1285028199	0.0004748135
0.995840525118838	0.004712729926953	0.9979202626	0.0820735936	0.0001933953
0.999210123227436	0.002026811968873	0.9996050616	0.0357805413	0.0000362602
			Integral	1.000000837

A julia version of Gauss-Legendre integration Formula will be written next to use n=10 formula given in the table. Our function is $f(x) = \frac{4}{\pi}\sqrt{1-x^2}$

10 Point Gauss-Legendre integration

```
 function  integral(f_xnt, a, b)
#integral f(x)dx
#integral of a function by using gauss-legendre quadrature
#coefficients are pre-calculated for 60 terms for [-1,1]
#band then utilises variable transform
 r=zeros(Float64,10)
 c=zeros(Float64,10)
 r[1]=-0.973906528517171
 r[2]=-0.865063366688984
 r[3]=-0.679409568299024
 r[4]=-0.433395394129247
 r[5]=-0.148874338981631
 r[6]=0.148874338981631
```

```
 r[7]=0.433395394129247
 r[8]=0.679409568299024
 r[9]=0.865063366688984
 r[10]=0.973906528517171
 c[1]=0.066671344308684
 c[2]=0.149451349150580
 c[3]=0.219086362515982
 c[4]=0.269266719309996
 c[5]=0.295524224714752
 c[6]=0.295524224714752
 c[7]=0.269266719309996
 c[8]=0.219086362515982
 c[9]=0.149451349150580
 c[10]=0.066671344308684
 n=length(r)
 z=0.0
 k1=(b-a)/2.0
 k2=(b+a)/2.0
 y1=0.0
 for i=1:n
   x=k2+k1*r[i]
   y=f_xnt(x)
   y1=c[i]*y
   z+=y1
 end
return k1*z
end

f(x) = 4.0/pi*sqrt(1.0-x*x)
r=integral(f,0.0,1.0)
print("integral  : $r")
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss_integral1.jl
integral  : 1.0001612291825637
> Terminated with exit code 0.
```

More terms can be used to achieve more accurate results. In the next program 60 terms of the Legendre coefficcients and roots are entered tho the program.

Program 60 Point Gauss-Legendre integration

```
 function  integral(f_xnt, a, b)
#integral f(x)dx
#integral of a function by using gauss-legendre quadrature
#coefficients are pre-calculated for 60 terms for [-1,1]
#band then utilises variable transform
r=[-0.99921012322743600,
-0.99584052511883800,
-0.98978789522222100,
-0.98106720175259800,
-0.96970178876505200,
-0.95572225583999600,
-0.93916627611642300,
-0.92007847617762700,
-0.89851031081004600,
-0.87451992264689800,
-0.84817198478592900,
-0.81953752616214500,
-0.78869373993226400,
-0.75572377530658500,
-0.72071651335573000,
-0.68376632738135500,
-0.64497282848947700,
-0.60444059704851000,
-0.56227890075394400,
```

-0.51860140005856900,
-0.47352584176170700,
-0.42717374158307800,
-0.37967005657679800,
-0.33114284826844800,
-0.28172293742326100,
-0.23154355137602900,
-0.18073996487342500,
-0.12944913539694500,
-0.07780933394953650,
-0.02595977230124780,
0.02595977230124780,
0.07780933394953650,
0.12944913539694500,
0.18073996487342500,
0.23154355137602900,
0.28172293742326100,
0.33114284826844800,
0.37967005657679800,
0.42717374158307800,
0.47352584176170700,
0.51860140005856900,
0.56227890075394400,
0.60444059704851000,
0.64497282848947700,
0.68376632738135500,
0.72071651335573000,
0.75572377530658500,
0.78869373993226400,
0.81953752616214500,
0.84817198478592900,
0.87451992264689800,
0.89851031081004600,
0.92007847617762700,
0.93916627611642300,
0.95572225583999600,
0.96970178876505200,
0.98106720175259800,
0.98978789522222100,
0.99584052511883800,
0.99921012322743600]
c=[0.00202681196887338,
0.00471272992695334,
0.00738993116334568,
0.01004755718228800,
0.01267816647681590,
0.01527461859678380,
0.01782990101420730,
0.02033712072945710,
0.02278951694399780,
0.02518047762152120,
0.02750355674992480,
0.02975249150078890,
0.03192121901929630,
0.03400389272494640,
0.03599489805108440,
0.03788886756924340,
0.03968069545238070,
0.04136555123558470,
0.04293889283593560,
0.04439647879578710,
0.04573437971611440,
0.04694898884891210,
0.04803703181997110,
0.04899557545575670,
0.04982203569055010,
0.05051418453250940,
0.05107015606985560,
0.05148845150098080,
0.05176794317491010,
0.05190787763122060,
0.05190787763122060,
0.05176794317491010,
0.05148845150098080,
0.05107015606985560,
0.05051418453250940,

```
0.04982203569055010,
0.04899557545575670,
0.04803703181997110,
0.04694898884891210,
0.04573437971611440,
0.04439647879578710,
0.04293889283593560,
0.04136555123558470,
0.03968069545238070,
0.03788886756924340,
0.03599489805108440,
0.03400389272494640,
0.03192121901929630,
0.02975249150078890,
0.02750355674992480,
0.02518047762152120,
0.02278951694399780,
0.02033712072945710,
0.01782990101420730,
0.01527461859678380,
0.01267816647681590,
0.01004755718228800,
0.00738993116334568,
0.00471272992695334,
0.00202681196887338]
 n=length(r)
 z=0.0
 k1=(b-a)/2.0
 k2=(b+a)/2.0
 y1=0.0
 for i=1:n
   x=k2+k1*r[i]
   y=f_xnt(x)
   y1=c[i]*y
   z+=y1
 end
return k1*z
end

f(x) = 4.0/pi*sqrt(1.0-x*x)
r=integral(f,0.0,1.0)
print("integral  : $r")
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss_integral2.jl
integral  : 1.000000837330801
> Terminated with exit code 0.
```

In the next example coeeficients are directly calculated from the Legendre polynomials, so storage of any coefficients is unnecessary. Of course it will have additional cost of calculating the coefficients.

variable Point Gauss-Legendre integration

```
   #calculates legendre gauss-coefficients as coefficients of the integral
  #for n terms
  function gauss_legendre_coefficients(x1,x2,n)
            A = zeros(Float64,2,n)
            eps=3.0e-15
            mx=(n+1)/2
            m=floor(Int,mx)
            xm=0.5*(x2+x1)
            xl=0.5*(x2-x1)
            nmax=20
            pp=0.0
            for i=1:m
                    z=cos(pi*((i-0.25)/(n+0.5)))
                    for ii=1:nmax
                            p1=1.0
                            p2=0.0
                            for j=1:n
```

```
                                    p3=p2
                                    p2=p1
                                    p1=(((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j)
                                  end
                                  pp=((z*p1-p2)*n/(z*z-1.0))
                                  z1=z
                                  z=z1-p1/pp
                                  if abs(z-z1) < eps
                                    break
                                  end
                          end
                          y1=xm-xl*z
                          y2=xm+xl*z
                          i1=n+1-i
                          A[1,i]=y1
                          A[1,i1]=y2
                          y3=2.0*xl/((1.0-z*z)*pp*pp)
                                  A[2,i1]=y3
                           A[2,i]=y3
                end
                show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A)
                println(" ")
                return A
    end
    #Gauss-Legendre integration
    function integral(f,x1,x2,n)
      #integral f1(x)dx
      #integral of a function by using gauss-legendre quadrature
      #between x1 and x2
      A=gauss_legendre_coefficients(x1,x2,n)
      z=0.0
      for i=1:n
        xx1=A[1,i]
        xx2=A[2,i]
        z=z+xx2*f(xx1)
      end
      return z
      end

f(x) = 4.0/pi*sqrt(1.0-x*x)
r=integral(f,0.0,1.0,10)
print("integral  : $r")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_gauss.jl
2X10 Matrix{Float64}:
 0.0130467  0.0674683  0.160295  0.283302  0.425563  0.574437  0.716698  0.839705  0.932532   0.986953
 0.0333357  0.0747257  0.109543  0.134633  0.147762  0.147762  0.134633  0.109543  0.0747257  0.0333357
integral  : 1.0001612291825648
> Terminated with exit code 0.
```

In the Gauss-Legendre integral formulation weight function was taken as 1.

$$I_w(-1,1) \cong \int_{-1}^{1} f(x)w(x)dx \cong \sum_{K01}^{n} c_k f(x_k)$$

If the weight function is taken as: $w(x) = \frac{1}{\sqrt{1-x^2}}$ Gauss-Chebychev integration formula is created. This equation has the form of

$$I_w(-1,1) \cong \int_{-1}^{1} f(x)w(x)dx \cong \int_{-1}^{1} f(x)\frac{1}{\sqrt{1-x^2}}dx \cong \sum_{K01}^{n} c_k f(x_k)$$

x_k in the equation is the roots of chebcychev polynomials. General definition of the Chbychev polynomials are given as:

$$T_0(x) = 1$$
$$T_1(x) = x$$

$$T(x) = 2x^2 - 1$$
$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x) \ , n = 3,4,5..$$

It should be note that Chebychev polinomials are orthogonal polnomials. The roots of the polynomials are defined as:

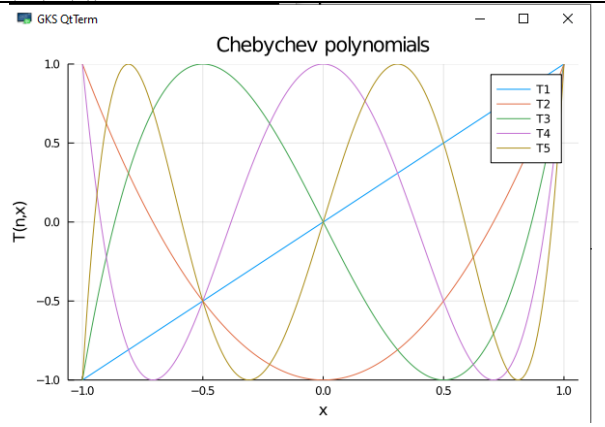
$$x_k = cos\left(\frac{\left(k+\frac{1}{2}\right)\pi}{n}\right), \qquad k = 0,1,2,\dots n-1$$

Coeeficients of the Formula can be calculated as:

$$c_k = \frac{\pi}{n}, \quad k = 0,1,2,\dots n-1$$

The first 5 Chebychev polynomial plotted below:

```
 #Chebychev polynomials
function T(n,x)
   P1=0.0
   if n==0 P1=1.0
   elseif n==1 P1=x
   else
     P1=2.0*x*T(n-1,x)-T(n-2,x)
   end
   return P1
   end

using Plots
x1=-1:0.01:1
f1(x)=T(1,x)
f2(x)=T(2,x)
f3(x)=T(3,x)
f4(x)=T(4,x)
f5(x)=T(5,x)
plot([f1,f2,f3,f4,f5],x1,title="Chebychev polynomials",label=["T1"
"T2" "T3" "T4" "T5"],ylims = (-1.0,1.0),xaxis = ("x"),yaxis =
("T(n,x)"))
```



x=[-1,1] integration range in Gauss-Chebychev Formula can be change to more general z=[a,b] range by using similar formulations used in Gauss-Legendre formulations.

$$z = \left(\frac{b-a}{2}\right)x + \left(\frac{b+a}{2}\right) = \alpha x + \beta$$

$$\alpha = \left(\frac{b-a}{2}\right)$$
$$\beta = \left(\frac{b+a}{2}\right)$$

As an example program if following integral is to be solved:

$$I = \int_{-1}^{1} \frac{e^x}{(1-x^2)} dx$$

Program Gauss-Chebychev integral

```
 function  gauss_chebychev_coefficients(n)
   #Gauss-Chebychev quadrature
    sgngam=1
    a=zeros(Float64,2,n)
    for j=1:n
      a[1,j]=cos((2.0*j-1.0)*pi/(2.0*n));
      a[2,j]=pi/n
    end
    return a
    end

function gauss_chebychev_integral(f_xnt,n)
#n : number of integral coefficients
# this routine first generates gauss chebychev coefficients
# for [xa,xb] band
# then calculates gauss chebychev integral
a=zeros(Float64,2,n)
a=gauss_chebychev_coefficients(n)
toplam=0.0
#ck
#x,y
z1=0.0
for k=1:n
  toplam+=a[2,k]*f_xnt(a[1,k])
  end
return toplam
end

f(x) = exp(x)
r=gauss_chebychev_integral(f,50)
print("gauss_chebychev_integral  : $r")
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_gauss_chebychev.jl
gauss_chebychev_integral  : 3.9774632605064237
> Terminated with exit code 0.
```

Gauss-Hermit formula : this formulation uses the weight function and calculates integrals from minus infinity to plus infinity.

$$w(x) = e^{-x^2}$$

$$I_w(-\infty,\infty) \cong \int_{-\infty}^{\infty} f(x)w(x)dx \cong \int_{-\infty}^{\infty} e^{-x^2} f(x)dx \cong \sum_{K01}^{n} c_k f(x_k)$$

In this equation x_k is the roots of the Hermite polynomials. Hermit Polinomials defined as:

$H_{-1}(x)=0$

$H_0(x)=1$

$H_1(x)=2x$

$H_2(x)=4x^2-2$

$H_3(x)=8x^3-12x$

$H_n(x)=2xP_{n-1}(x) - 2(n-1)P_{n-2}(x)$, n=1,2,3,4,5..

Coefficients are given as:

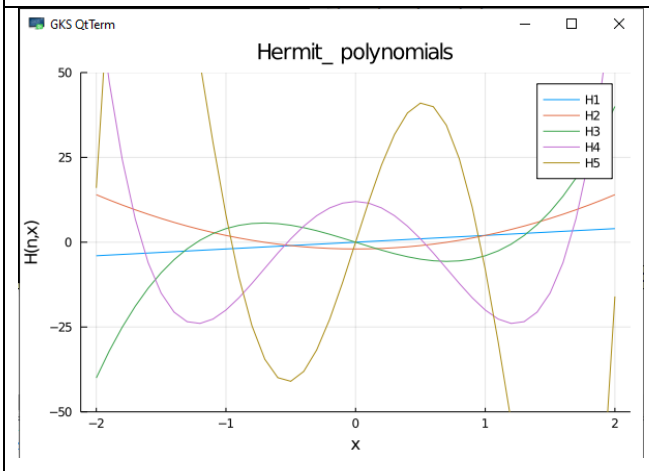
$$c_k = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 P_{n-1}(x_k)^2}$$

Coefficients for the first 7 sets are given in the following table

Table 7.3-6 Gauss –Hermit integration roots and coefficients

N	x_k	c_k
2	+/-0.707167811	0.886226926
3	0	1.181635901
	+/- 1.224744871391589	0.295408975
4	+/- 0.524647623275290	0.80491409
	+/- 1.650680123885784	0.081312835
5	0	0.945308721
	+/-0.9585724646	0.393619323
	+/-2.0201828705	0.019953242
6	+/- 2.350604973674492	0.00453001
	+/- 1.335849074013697	0.15706732
	+/- 0.436077411927616	0.724629595
7	+/- 2.651961356835233	0.000971781
	+/- 1.673551628767471	0.054515583
	+/- 0.816287882858965	0.425607253
	0	0.810264618

```
#Hermit polynomials
function H(n,x)
   P1=0.0
   if n==0 P1=1.0
   elseif n==1 P1=2.0*x
   else
     P1=2.0*x*H(n-1,x)-2.0*(n-1)*H(n-2,x)
   end
   return P1
   end

using Plots
x1=-2.0:0.1:2.0
f1(x)=H(1,x)
f2(x)=H(2,x)
f3(x)=H(3,x)
f4(x)=H(4,x)
f5(x)=H(5,x)
plot([f1,f2,f3,f4,f5],x1,title="Hermit_ polynomials",label=["H1" "H2" "H3" "H4"
"H5"],ylims = (-50.0,50.0),xaxis = ("x"),yaxis = ("H(n,x)"))
```



Gauss_Hermite integral equation example code :

Gauss-Hermite integration

```
 #Gauss- Hermit integral
function gauss_hermite_coefficients(n)
   A=OffsetArray{Float64}(undef,0:1,0:n-1)
   r=0.0
   r1=0.0
   p1=0.0
   p2=0.0
   p3=0.0
   dp3=0.0
   j=0
   i=Int64(0)
   eps=1e-15
   pipm4 =1.0/pi^0.25
   n1=floor(Int64,(n+1)/2-1)
   for i = 0:n1
      if i==0
        r =sqrt(Float64(2*n+1.0))-1.85575*(Float64(2*n+1))^(-1.0/6.0)
      elseif i==1
        r = r-1.14*Float64(n)^0.426/r
      elseif i==2
        r = 1.86*r-0.86*A[0,0]
      elseif i==3
        r = 1.91*r-0.91*A[0,1]
      else
        r = 2*r-A[0,i-2]
      end # if
      while(true)
         p2 = 0.0
         p3 = pipm4;
         for j = 0:n-1
            p1 = p2
            p2 = p3
            p3 = p2*r*sqrt(2.0/Float64(j+1))-p1*sqrt(Float64(j)/Float64(j+1))
            end # for j
         dp3 = sqrt(Float64(2.0*n))*p2
         r1 = r
         r = r-p3/dp3
         yy1=abs(r-r1)
         yy2=eps*(1.0+abs(r))*100.0
         if yy1<yy2
           break
         end
         end #while
   y1=r
   y2=2.0/(dp3*dp3)
   A[0,i] = y1
   A[1,i] = y2
   A[0,n-1-i] = -y1
   A[1,n-1-i] = y2
  end # for i
  return A
  end #function

function gauss_hermite_integral(f_xnt,n)
# n : number of integral coefficients
# this routine first generates gauss hermite coefficients
# for [x1,x2] band
# then calculates gauss hermite integral
# a=OffsetArray{Float64}(undef,0:1,0:n-1)
a=gauss_hermite_coefficients(n)
 show(IOContext(stdout, :limit=>false), MIME"text/plain"(), a)
 println(" ")
```

```
z=0
for i=0:n-1
 z+=a[1,i]*f_xnt(a[0,i])*exp(a[0,i]*a[0,i])
 end # for
return z
end # function


using OffsetArrays
f(x)=1.0/(1.0+x*x)
I=gauss_hermite_integral(f,20)
print("I = $I")
```

Gauss-Hermite integrasyonu

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss_hermite_integral.jl
2X20 OffsetArray(::
Matrix{Float64}, 0:1, 0:19) with eltype Float64 with indices 0:1X0:19:
 5.38748     4.60368     3.94476     3.34785     2.78881     2.25497     1.73854     1.23408   0.737474  0.245341  -0.245341  -0.737474  -
1.23408   -1.73854    -2.25497    -2.78881     -3.34785    -3.94476    -4.60368     -5.38748
 2.22939e-13  4.39934e-10  1.08607e-7  7.80256e-6  0.000228339  0.00324377  0.0248105  0.109017  0.286676  0.462244   0.462244
0.286676   0.109017   0.0248105   0.00324377   0.000228339   7.80256e-6   1.08607e-7   4.39934e-10   2.22939e-13
I = 2.804683334304932
> Terminated with exit code 0.
```

Gauss-Laguerre formulation has the weight factor in the form of

$$w(x) = x^a e^{-x}$$

The integral limits of this formulation is in between o and infinity. The equation is in the form of

$$I_w(0,\infty) = \int_{x=0}^{\infty} f(x)w(x)dx = \int_{x=0}^{\infty} x^a e^{-x} f(x)dx \cong \sum_{k=0}^{n-1} c_k f(x_k)$$

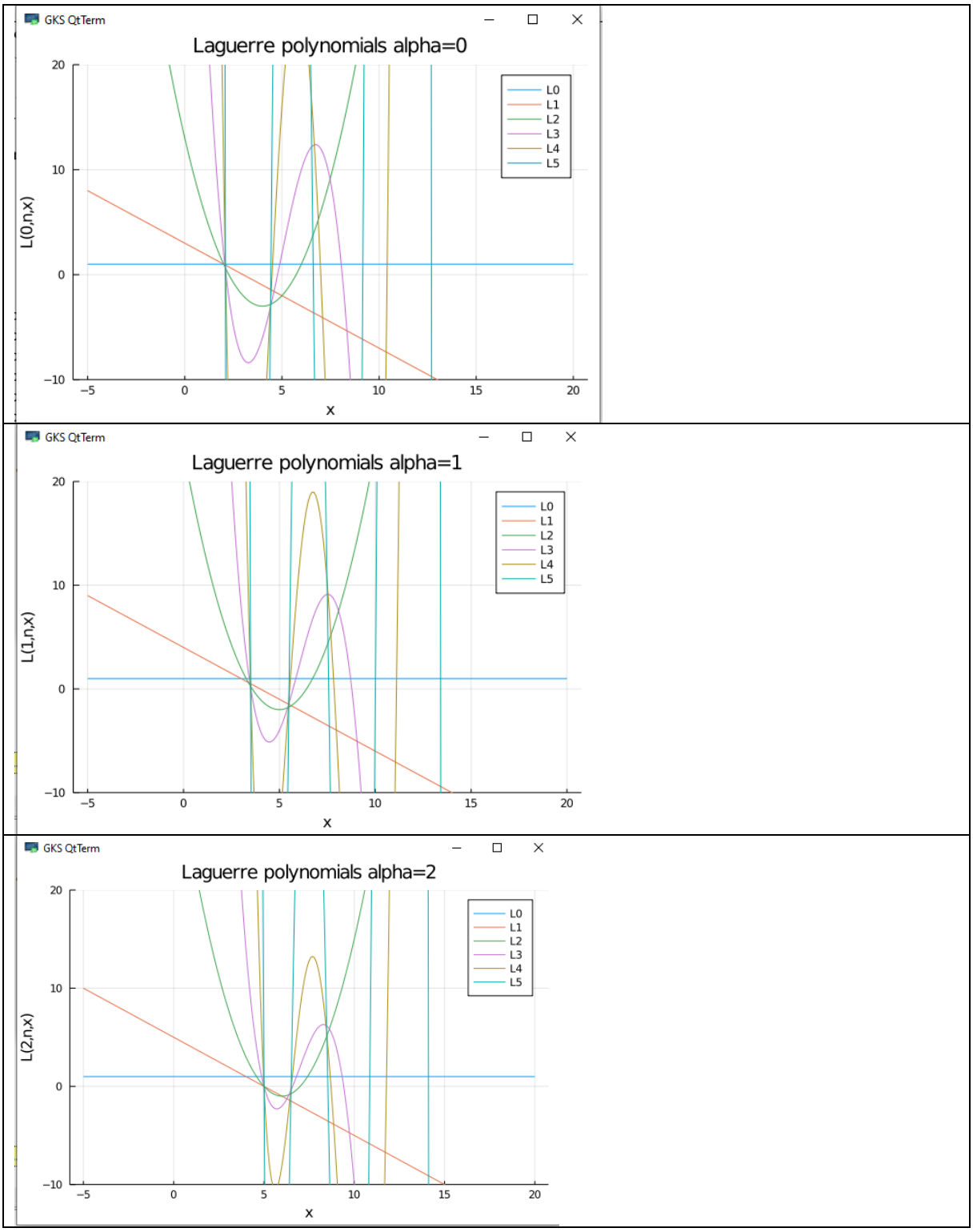
In this equation x_k is the root of Laguerre polinomials. Leguerre polynomial has the form of:

$$L_{-1}(x) = 0$$

$$L_o(x) = 1$$

$$(n+1)L_{n+1}^{\alpha}(x) = (-x + 2n + \alpha + 1)L_n^{\alpha}(x) - (n+\alpha)L_{n-1}^{\alpha}(x), \;\; n = 1,2,3,\ldots$$

```
#Laguerre polynomials
function L(alpha,n,x)
   P1=0.0
   if n==-1 P1=0.0
   elseif n==0 P1=1.0
   else
     P1=(-x+2.0*n+alpha+1.0)*L(alpha,n-1,x)-(n-alpha)*L(alpha,n-2,x)
   end
   return P1
   end

using Plots
x1=-5.0:0.1:20.0
alpha=0.0;
f0(x)=L(alpha,0,x)
f1(x)=L(alpha,1,x)
f2(x)=L(alpha,2,x)
f3(x)=L(alpha,3,x)
f4(x)=L(alpha,4,x)
f5(x)=L(alpha,5,x)
plot([f0,f1,f2,f3,f4,f5],x1,title="Laguerre polynomials alpha=0",label=["L0" "L1" "L2" "L3" "L4" "L5"],ylims = (-10.0,20.0),xaxis = ("x
```



$$C_k = \frac{1}{x_k \left[\frac{dL_n^{\alpha}(x_k)}{dx_k}\right]^2}$$

Coefficients,

For the first 5 values, roots and coefficient of the equation is given in the tables for (α=0, α=1 and α=2) .

In Gauss-Laguerre integral Formula for α=0 integral equation can be changed as

$$I_w(a,\infty) \cong \int_a^{\infty} f(x)w(x)dx \cong \int_a^{\infty} e^{-(x+a)} f(x+a)dx \cong e^{-a}\sum_{k=0}^{n-1} c_k f(x_k+a)$$

Table Gauss – Laquerre integral roots and coefficients (α=0)

N	x_k	c_k
2	0.585786437626905	0.853553390592534
	3.414213562373090	0.146446609406723
3	0.415774556783479	0.711093009915089
	2.294280360279040	0.278517733568676
	6.289945082937470	0.010389256497354
4	0.322547689619392	0.603154104288259
	1.745761101158340	0.357418692437802
	4.536620296921120	0.038887908507759
	9.395070912301130	0.000539294705556
5	0.263560319718141	0.521755610467045
	1.413403059106510	0.398666821407258
	3.596425771040720	0.075942449681708
	7.085810005858830	0.003611758663878
	12.640800844275700	0.000023369972386

Table Gauss – Laquerre integral roots and coefficients (α=1)

N	x_k	c_k
2	1.267949192431120	0.788675133123411
	4.732050807568870	0.211324865405185
3	0.935822227524088	0.588681472855445
	3.305407289332270	0.391216059222280
	7.758770483143630	0.020102459732679
4	0.743291927981432	0.446870579513482
	2.571635007646270	0.477635774492073
	5.731178751689090	0.074177784726293
	10.953894312683100	0.001315849683447
5	0.617030853278271	0.348014523429866
	2.112965958578520	0.502280674138866
	4.610833151017530	0.140915919102187
	8.399066971204840	0.008719893025972

Table Gauss – Laquerre integral roots and coefficients (α=2)

N	x_k	c_k
2	2.000000000000000	1.500000000000000
	6.000000000000000	0.500000000000000
3	1.517387080677410	1.037494961490390
	4.311583133719520	0.905750004703039
	9.171029785603060	0.056755033772202
4	1.226763263500300	0.725524997698604
	3.412507358696940	1.063424292391060
	6.902692605851610	0.206696130999709
	12.458036771951100	0.004354579188558
5	1.031109144093380	0.520917396835042
	2.837212823953820	1.066705933159050
	5.620294272598700	0.383549720007093
	9.682909837664020	0.028564233510280
	15.828473921690000	0.000262712802303

If we investigate Gauss-Leguerre Formula with an example. We will calculate the integral

$$I = \int_0^{\infty} \frac{x}{e^x + 1} dx$$ Exact integral value I= 0.822467.

Program Gauss-Laguerre integration

```
function gauss_laguerre_coefficients(n,alpha)
  #Gauss-Laguerre quadrature
  a=OffsetArray{Float64}(undef,0:1,0:n-1)
  i=0
  j=0
  r=0.0
  r1=0.0
  p1=0.0
```

```
    p2=0.0
    p3=0.0
    dp3=0.0
    eps=1.0e-10
    for i=0:n-1
       if i==0
          r = (1.0+alpha)*(3.0+0.92*alpha)/(1+2.4*n+1.8*alpha)
       else
          if i==1
            r = r+(15.0+6.25*alpha)/(1.0+0.9*alpha+2.5*n)
          else
            r = r+((1.0+2.55*(i-1))/(1.9*(i-1))+1.26*(i-1)*alpha/(1+3.5*(i-1)))/(1.0+0.3*alpha)*(r-a[0,i-2])
          end # if i==1
       end  # if i==0
       while(true)
          p2 = 0
          p3 = 1
          for j = 0:n-1
             p1 = p2
             p2 = p3
             p3 = ((-r+2*j+alpha+1)*p2-(j+alpha)*p1)/(j+1)
          end # for
          dp3 = (n*p3-(n+alpha)*p2)/r
          r1 = r
          r = r-p3/dp3
       if abs(r-r1)<eps*(1+abs(r))*100
         break
       end #if abs
       end # while
       a[0,i] = r
       lng=loggamma(Float64(n))
       lng1=loggamma(Float64(alpha+n))
       a[1,i] = -exp(lng1-lng)/(dp3*n*p2)
   end #for i
   show(IOContext(stdout, :limit=>false), MIME"text/plain"(), a)
   println(" ")
return a
end #function

function gauss_laquerre_integral(f_xnt,xa,n,alpha)
# n : number of integral coefficients
# this routine first generates gauss legendre coefficients
# for [x1,x2] band
# then calculates gauss hermite integral
# a=OffsetArray{Float64}(undef,0:1,0:n-1)
a=gauss_laguerre_coefficients(n,alpha)

z=0.0
for i=0:n-1
  x1=a[0,i]+xa
  z+=a[1,i]*f_xnt(x1)*exp(a[0,i])*a[0,i]^(-alpha)
end # for
return z
end # func

using OffsetArrays
using SpecialFunctions
f(x)=x/(exp(x)+1.0)
I=gauss_laquerre_integral(f,0.0,30,0.0)
print("integral  : $I  exact= 0.822467")
```

Gauss-Laguerre integration

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss_laguerre_integral.jl
2X30 OffsetArray(::
Matrix{Float64}, 0:1, 0:29) with eltype Float64 with indices 0:1X0:29:
 0.0474072  0.249924  0.614833  1.1432   1.83645  2.69652   3.72581   4.92729    6.30452   7.86169    9.60378    11.5365
13.6667     16.0022    18.5521    21.3272   24.34      27.6056    31.1416    34.9697    39.1161    43.6137    48.504
53.8414     59.6991    66.1806    73.4412   81.7368    91.5565    104.158
```

```
 0.116044   0.220851  0.2414    0.194637  0.123728  0.0636788  0.0268605  0.00933807  0.0026807  0.000635129  0.000123907
1.98288e-5  2.58935e-6  2.74094e-7  2.33283e-8  1.58075e-9  8.42748e-11  3.48516e-12  1.09902e-13  2.58831e-15  4.43784e-17
5.36592e-19  4.39395e-21  2.31141e-23  7.27459e-26  1.23915e-28  9.83238e-32  2.84232e-35  1.87861e-39  8.74598e-45
integral  : 0.822467013985489  exact= 0.822467
> Terminated with exit code 0.
```

Gauss-Kronroad built-in integration of Julia language

```
using Pkg; Pkg.add("QuadGK")
```

```
using QuadGK
f(x)=x/(exp(x)+1.0)
I1=quadgk(f,0.0,Inf)[1]
print("integral  = $I1 I_exact= 0.822467")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_gk.jl
integral  = 0.8224670334241129 I_exact= 0.822467
> Terminated with exit code 0.
```

INTEGRATION FORMULATION WITH ADJUSTABLE ERROR

If integration in region **[a,b]** to be defined for a continious integrable function f(x), Integration can be written as:

$$I = \int_a^b f(x)dx$$

If a sub-region is defined for the integration. For example,

$$m = \frac{b-a}{2}$$

definition is given integral can be redefined as

$$I = \int_a^m f(x)dx + \int_m^b f(x)dx$$

For numerical integration it is generally assumed that the error of the second definition will be less than the first definition. It is possible to continue this process by dividing each integral to subregions again. The important concept in here is to stop dividing integral into smaller parts when the desired accuracy is achieved. In this section several integration methods with error control will be investigated.

7.4.1 Adjustable Simpson 1/3 integral

If simpson 1/3 integration equation is written in [a,b] region

$$I(x_0, x_4) = \int_a^b f(x)dx = \int_{x_0}^{x_4} f(x)dx \cong \frac{h}{6}[f(x_0) + 4f(x_2) + f(x_4)]$$

$$\int_a^b f(x)dx = I(x_0, x_4) - h^5 \frac{f^{(4)}(d_1)}{90}$$

Where h =(b-a)/2, x_0=a,x_2=(a+b)/2,x_4=b and d_1 is a number between a and b. and $h^5 \frac{f^{(4)}(d_1)}{90}$ equation gives amount of error in the integral equation. $f^{(4)}$ term in the error equation is the fourth derivative of the function. If the integration region is divided into two subregion, equation becomes

$$I(x_0, x_2) = \frac{h}{6}[f(x_0) + 4f(x_1) + f(x_2)]$$

$$I(x_2, x_4) = \frac{h}{6}[f(x_2) + 4f(x_3) + f(x_4)]$$

$$\int_{x_0}^{x_2} f(x)dx + \int_{x_2}^{x_4} f(x)dx = I(x_0,x_2) + I(x_2,x_4) - \frac{h^5}{16}\frac{f^{(4)}(d_2)}{90}$$

In this equation $x_1=(a+b)/4$, $x_3=3(a+b)/4$. d_2 is stil a value in between a and b.

If it is assumed that $f^{(4)}(d_1) = f^{(4)}(d_2)$Equation becomes

$$h^5\frac{f^{(4)}(d_1)}{90} \cong \frac{16}{15}[I(x_0,x_2) + I(x_2,x_4) - I(x_0,x_4)]$$

$$\frac{h^5}{16}\frac{f^{(4)}(d_1)}{90} \cong \frac{1}{15}[I(x_0,x_2) + I(x_2,x_4) - I(x_0,x_4)]$$

If we substitute this into integration equation

$$\left|\int_{x0}^{x_4} f(x)dx - I(x_0,x_2) - I(x_2,x_4)\right| \approx \frac{1}{15}\left|I(x_0,x_2) + I(x_2,x_4) - I(x_0,x_4)\right|$$

In this case for error $\varepsilon>0$, it can be written

$$\frac{1}{15}|I(x_0,x_2) + I(x_2,x_4) - I(x_0,x_4)| < \varepsilon$$

If this condition is met, then approximately

$$\int_a^b f(x)dx = \int_{x_0}^{x_4} f(x)dx \cong [I(x_0,x_2) + I(x_2,x_4)]$$

can be written. If the given condition can not be met, division process will be continue till the condition met. Described algorithm is used in the next program.

Simson 1/3 integration with adjustable error

```
function  newton_cotes2(ff,a,b,n)
    #Newton-cotes integral 2 points
     #trapezoidal rule
     # ff integrating function
     # a,b integral limits
     # n : number of sub division h=(b-a)/n;
     h=(b-a)/n
     h1=h;
     sum=0.0;
     x=zeros(3)
     f=zeros(3)
     xi=zeros(n+1)
     fi=zeros(n+1)
     xi[1]=a
     fi[1]=ff(a)
     for p=1:n
       for i=1:2
         x[i]=a+(i-1)*h1+(p-1)*h
         f[i]=ff(x[i])
         xi[p+1]=x[i]
         fi[p+1]=f[i]
         end
       sum=sum+h*(f[1]+f[2])/2.0
     end
     xx=[sum,xi,fi]
     return xx
     end

function adaptive_simpson(ff,a,b,eps,MI)
```

```
 sum1=0.0
 sum2=0.0
 for i=1:MI
  sum1=newton_cotes2(ff,a,b,2*i)[1]
  m=(b-a)/2.0;
  sum2=newton_cotes2(ff,a,m,2*i)[1]+newton_cotes2(ff,m,b,2*i)[1]
  dsum=1.0/15.0*abs(sum2-sum1)
  if dsum<eps
   break
  end
  end
 return sum2
end

f(x)=x^1.5
I=adaptive_simpson(f,0.0,1.0,1.0e-8,500)
print("I = $I")
```

integration with the adaptive simpson integration

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton_cotes2.jl
I = 0.40000005006813727
> Terminated with exit code 0.
```

Adaptive Gauss-Legendre Integration

The error for the Gauss-Legendre formulas is specified generally by

$$E_t = \frac{2^{2n+3}[(n+1)!]^4}{(2n+3)[(2n+2)!]^3} f^{(2n+2)}(\xi)$$

where n = the number of points minus one and $f^{(2n+2)}(\xi)$ is the $(2n+2)$th derivative of the function after the change of variable with ξ located somewhere on the interval from -1 to 1. It is clear that error level of Gauss-Legendre integration is far smaller than the Newton-Cotes formulas, therefore it will take less iteration to reach the desired error levels. Adjustable error version of the Gauss-Legendre integration is given below. The main idea here is to increase terms in the Gauss-Legendre integration untill the error level between two levels decrease below the desired error level.

Program: Gauss-Legendre integration with the adjustable error

```
 #calculates legendre gauss-coefficients as coefficients of the integral
 #for n terms
 function gauss_legendre_coefficients(x1,x2,n)
        A = zeros(Float64,2,n)
        eps=3.0e-15
        mx=(n+1)/2
        m=floor(Int,mx)
        xm=0.5*(x2+x1)
        xl=0.5*(x2-x1)
        nmax=20
        pp=0.0
        for i=1:m
            z=cos(pi*((i-0.25)/(n+0.5)))
            for ii=1:nmax
                p1=1.0
                p2=0.0
                for j=1:n
                 p3=p2
                 p2=p1
                 p1=(((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j)
                end
                pp=((z*p1-p2)*n/(z*z-1.0))
                z1=z
                z=z1-p1/pp
                if abs(z-z1) < eps
                  break
```

```
                                end
                            end
                            y1=xm-xl*z
                            y2=xm+xl*z
                            i1=n+1-i
                            A[1,i]=y1
                            A[1,i1]=y2
                            y3=2.0*xl/((1.0-z*z)*pp*pp)
                                    A[2,i1]=y3
                             A[2,i]=y3

                end
                #show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A)
                #println(" ")
                return A
  end
  #Gauss-Legendre integration
  function integral(f,x1,x2,n)
    #integral f1(x)dx
    #integral of a function by using gauss-legendre quadrature
    #between x1 and x2
    A=gauss_legendre_coefficients(x1,x2,n)
    z=0.0
    for i=1:n
      xx1=A[1,i]
      xx2=A[2,i]
      z=z+xx2*f(xx1)
    end
    return z
    end

function adaptive_gauss_legendre_integral(f,x1,x2,eps,MAXITER)
global n=4
ans1=integral(f,x1,x2,n)
ans2=0.0
while(true)
  ans2=ans1
  n=n+1
  ans1=integral(f,x1,x2,n)
    dans=abs(ans2-ans1)
  #println("n = $n ans1 = $ans1 dans = $dans eps = $eps")

  if dans<eps || n>MAXITER
    return ans2
  elseif n==MAXITER
    print("required error level can not be obtained n= $n")
  end # if
end # while
return ans2
end


f(x) = 4.0/pi*sqrt(1.0-x*x)
eps=1.0e-10
MAXITER=500
r=adaptive_gauss_legendre_integral(f,0.0,1.0,eps,MAXITER)
print("integral  : $r")
```

Adaptive Gauss-Legendre integration

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_adaptive_gauss_legendre.jl
integral  : 1.0000000090616536
> Terminated with exit code 0.
```

Gauss-Kronrod integration formula

$$I_w(-1,1) \cong \int_{-1}^{1} f(x)dx \cong \sum_{k=1}^{n} c_k f\left(x_k^G\right) + \sum_{k=1}^{n+1} c_{n+k} f\left(x_k^K\right)$$

Gauss Kronrod integration formula uses two terms. First term is standard Gauss-Legendre integration term. The second term coefficients are taken as same as the Gauss-Legendre term, but roots are different. It is called Kronrod integration. In our example code constant integration of 15 terms will be used for each equation (total of 30 terms). Program utilises Gauss-Legendre terms as first, and then uses Gauss-Kronrod terms. To improve error reduction equation can be continued with half dividing technique. Gauss-Kronrod Formula is prefferred method for many package programs such as Mathlab. Julia also has Gauss-Kronrod integration formula as build in method

Program Gauss-Kronrod integration formula

```
function  gauss_kronrad_integral(ff,a,b,eps,n)
    Aw = 4
    global A=OffsetArray{Float64}(undef,0:n-1,0:Aw-1)
    nn = 61
    ng = 15
    h=0
    i=0
    j=0
    v=0.0
    k1=0
    k2=0
    intg=0.0
    intk=0.0
    ta=0.0
    tb=0.0
    c = OffsetArray{Float64}(undef,0:nn-1)
    xk = OffsetArray{Float64}(undef,0:nn-1)
    xg =OffsetArray{Float64}(undef,0:nn-1)
    # Gauss-Legendre coefficients
    xg[0] = 0.00796819249616660561546588347467 4
    xg[1] = 0.01846646831109095914230213191204 7
    xg[2] = 0.02878470788332336934971917961129 2
    xg[3] = 0.03879919256962704959680193644634 8
    xg[4] = 0.04840267283059405290293814042280 8
    xg[5] = 0.05749315621761906648172168940205 6
    xg[6] = 0.06597422988218049512812851511596 2
    xg[7] = 0.07375597473770520626824385002219 1
    xg[8] = 0.08075589522942021535469493846053 0
    xg[9] = 0.08689978720108297980238753071512 6
    xg[10] = 0.09212252223778612871763270708761 9
    xg[11] = 0.09636873717464425963946862635181 0
    xg[12] = 0.09959342058679526706278028210356 9
    xg[13] = 0.10176238974840550459642895216855 4
    xg[14] = 0.10285265289355884034128563670541 5
    c[0] = 0.99948441005049063757132589570581 1
    c[1] = 0.99689348407464954027163005091869 5
    c[2] = 0.99163099687040459485862836610948 6
    c[3] = 0.98366812327974720997003258160566 3
    c[4] = 0.97311632250112626837469386842370 7
    c[5] = 0.96002186496830751221687102558179 8
    c[6] = 0.94437444474855997941583132403743 9
    c[7] = 0.92620004742927432587932427708047 4
    c[8] = 0.90557330769990779854652255892595 8
    c[9] = 0.88256053579205268154311646253022 6
    c[10] = 0.85720523354606109895865851065894 4
    c[11] = 0.82956576238276839744289811973250 2
```

```
c[12] = 0.799727835821839083013668942322683
c[13] = 0.767777432104826194917977340974503
c[14] = 0.733790062453226804726171131369528
c[15] = 0.697850494793315796932292388026640
c[16] = 0.660061064126626961370053668149271
c[17] = 0.620526182989242861140477556431189
c[18] = 0.579345235826361691756024932172540
c[19] = 0.536624148142019899264169793311073
c[20] = 0.492480467861778574993693061207709
c[21] = 0.447033769538089176780609900322854
c[22] = 0.400401254830394392535476211542661
c[23] = 0.352704725530878113471037207089374
c[24] = 0.304073202273625077372677107199257
c[25] = 0.254636926167889846439805129817805
c[26] = 0.204525116682309891438957671002025
c[27] = 0.153869913608583546963794672743256
c[28] = 0.102806937966737030147096751318001
c[29] = 0.051471842555317695833025213166723
c[30] = 0.000000000000000000000000000000000
# Kronrod coefficients
xk[0] = 0.001389013698677007624551591226760
xk[1] = 0.003890461127099884051267201844516
xk[2] = 0.006630703915931292173319826369750
xk[3] = 0.009273279659517763428441146892024
xk[4] = 0.011823015253496341742232898853251
xk[5] = 0.014369729507045804812451432443580
xk[6] = 0.016920889189053272627572289420322
xk[7] = 0.019414141193942381173408951050128
xk[8] = 0.021828035821609192297167485738339
xk[9] = 0.024191162078080601365686370725232
xk[10] = 0.026509954882333101610601709335075
xk[11] = 0.028754048765041292843978785354334
xk[12] = 0.030907257562387762472884252943092
xk[13] = 0.032981447057483726031814191016854
xk[14] = 0.034979338028060024137499670731468
xk[15] = 0.036882364651821229223911065617136
xk[16] = 0.038678945624727592950348651532281
xk[17] = 0.040374538951535959111995279752468
xk[18] = 0.041969810215164246147147541285970
xk[19] = 0.043452539701356069316831728117073
xk[20] = 0.044814800133162663192355551616723
xk[21] = 0.046059238271006988116271735559374
xk[22] = 0.047185546569299153945261478181099
xk[23] = 0.048185861757087129140779492298305
xk[24] = 0.049055434555029778887528165367238
xk[25] = 0.049795683427074206357811569379942
xk[26] = 0.050405921402782346840893085653585
xk[27] = 0.050881795898749606492297473049805
xk[28] = 0.051221547849258772170656282604944
xk[29] = 0.051426128537459025933862879215781
xk[30] = 0.051494729429451567558340433647099
nn1=Int64(floor(nn/2))
for i=nn-1:-1:nn1
  c[i]=-c[nn-1-i]
end
for i=nn-1:-1:nn1
  xk[i]=xk[nn-1-i]
end
for i=ng-1:-1:0
  xg[nn-2-2*i]=xg[i]
  xg[1+2*i]=xg[i]
end
```

```
for i=0:nn1
  xg[2*i]=0
end
k1=0.5*(b-a)
k2=0.5*(b+a)
intg=0.0
intk=0.0
v=0.0
for i=0:nn-1
  v=ff(k1*c[i]+k2)
  intk=intk+v*xk[i]
  if i%2==1
    intg=intg+v*xg[i]
  end
end
intk=intk*(b-a)/2.0
intg=intg*(b-a)/2.0
A[0,0]=abs(intg-intk)
A[0,1]=intk
A[0,2]=a
A[0,3]=b
toplamhata=A[0,0]
integral=intk
if toplamhata<eps
   sonuc = true
   integral = intk
   Aused = 1
   return integral
end
Aused = 1
for h=1:n-1
    Aused = h+1
    gir(A, h, Aw)
    nh=h-1
    toplamhata = toplamhata-A[nh,0]
    ta = A[nh,2]
    tb = A[nh,3]
    A[nh,2] = ta
    A[nh,3] = 0.5*(ta+tb)
    A[h,2] = 0.5*(ta+tb)
    A[h,3] = tb
    for j=h-1:h
       k1 = 0.5*(A[j,3]-A[j,2])
       k2 = 0.5*(A[j,3]+A[j,2])
       intg = 0.0
       intk = 0.0
       for i=0:nn-1
         v = ff(k1*c[i]+k2)
          intk = intk+v*xk[i]
          if  i%2==1
           intg = intg+v*xg[i]
          end
       end # for i
       intk = intk*(A[j,3]-A[j,2])*0.5
       intg = intg*(A[j,3]-A[j,2])*0.5
       A[j,0] = abs(intg-intk)
       A[j,1] = intk
       toplamhata = toplamhata+A[j,0]
    end # for j
    cik(A,h-1,Aw)
    cik(A,h,Aw)
    if  toplamhata<eps
```

```
            break
          end
        end #for
        sonuc = toplamhata<eps
        integral = 0
        for j=0:Aused-1
          integral = integral+A[j,1]
        end # for j
return integral
end

function gir(A,n,Awidth)
     i = 0
     p = 0
     t = 0
     maxcp = 0
       if  n==1
         return
       end
       for i=0:Awidth-1
          t = A[n-1,i]
          A[n-1,i] = A[0,i]
          A[0,i] = t
       end # for
       p = 0
       while( 2*p+1<n-1 )
          maxcp = 2*p+1
          if (2*p+2)<(n-1)
             nx1= 2*p+2
             nx2=2*p+1
             if  A[nx1,0]>A[nx2,0]
               maxcp = 2*p+2
               end
          end
          if  A[p,0]<A[maxcp,0]
            for i=0:Awidth-1
                t = A[p,i]
                A[p,i] = A[maxcp,i]
                A[maxcp,i] = t
                end # for
            p = maxcp
          else
            break
          end # if
       end #of while
   end

function cik(A,n,Awidth)
       i = 0
       p = 0
       t = 0.0
       kk = 0
       if n==0
         return
       end
       p = n
       while( p!=0 )
          kk = Int64(floor((p-1)/2))
          if A[p,0]>A[kk,0]
          for i=0:Awidth-1
             t = A[p,i]
              A[p,i] = A[kk,i]
```

```
                A[kk,i] = t
              end # for
              p = kk
            else
              break
            end #if
          end # while
end

using OffsetArrays
f(x)=x^1.5
a=0.0
b=1.0
eps=1.0e-15
n=500
I=gauss_kronrad_integral(f,a,b,eps,n)
print("I = $I")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss_kronrad_integral.jl
I = 0.40000000000000024
> Terminated with exit code 0.
```

Julia Buit-in Gauss-Kronroad formula integration:

```
using Pkg; Pkg.add("QuadGK")
```

```
using QuadGK
f(x)=x^1.5
I1=quadgk(f,0.0,1.0)[1]
print("integral  = $I1 I_exact= 0.4")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_gk.jl
integral  = 0.39999999998326774 I_exact= 0.4
> Terminated with exit code 0.
```

Adaptive Newton-Cotes integration with Gander ve Gautschi[31] approach

In the previous adaptive integration approaches program follows a stopping criteria such as

if(Math.abs(I1-I2)< tolerance or

if(Math.abs((I1-I2)/I1) < tolerance

This type of stopping criteria might be unsuccessfull for some cases, and Maximum iteration error will be received. If we would like to further decrease the error and reach the machine error level a different type of error control can be used. If an error control such as

if(is+(I1-I2)==is)

is used, this control works only when I1 is equal to I2 with the machine accuracy. If this definition to be used with the halving approach

$$m = \frac{b-a}{2}$$

with an additional control definition can even give a better exit criteria

if((is+(I1-I2)==is) || (m<=a) || (b<=m))

Now this criteria will be applied to Newton-Cotes formulas. In this algorithm Newton Cotes Formula with 8th degree polynomial and quadratic polinomial (Simpson 1/3) will be used. 8th degree Newton cotes Formula :

$$I_{newton-cotes\_8} = \frac{b-a}{28350}\big[989f(x_0) + 5888f(x_1) - 928f(x_2) + 10496f(x_3) - 4540f(x_4) + 10496f(x_5) - 928f\big(x_6 + 5888f(x_7) + 988f(x_8)\big)\big]$$

Simpson 1/3integral in iterative form will be:

$$I_1 = \int_a^b f(x)dx = I[a,b] = \frac{b-a}{6}[f(x_0) + 4f(x_2) + f(x_4)]$$

$$I_2 = \int_a^b f(x)dx = \int_a^m f(x)dx + \int_m^b f(x)dx = I[a,b]$$

$$= \frac{b-a}{12}\big[\big(f(x_0) + 4f(x_1) + f(x_2)\big) + \big(f(x_2) + 4f(x_3) + f(x_4)\big)\big]$$

Program : Gander and Gautschi adaptive Simpson 1/3 integral formulation

```
 # ======================================================
# Numerical Analysis package in Julia
# example to show utilisation of integration (integral)
# and differentiation (derivative) functions
# Adaptive Gander and Gautschi simpson 1/3 integration
# Dr. M. Turhan Coban
# ======================================================
function  adaptive_simpson_integral2(ff,a,b)
   i=0
   h=(b-a)
   h2=h/4.0
   h3=h/8.0
   m=(a+b)/2.0
   x=OffsetArray{Float64}(undef,0:8)
   y=OffsetArray{Float64}(undef,0:8)
   for i=0:8
     x[i]=a+i*h3
     y[i]=ff(x[i])
   end #for
   is=h/28350.0*(989.0*y[0]+5888.0*y[1]-928.0*y[2]+10496.0*y[3]-4540.0*y[4]+
        10496.0*y[5]-928.0*y[6]+5888.0*y[7]+989.0*y[8])
   fa = y[0]
   fm = y[4]
   fb = y[8]
   if is==0
    is = b-a
   end #if
   Q = adaptsimstp(ff,a,b,fa,fm,fb,is)
   return Q
end

function adaptsimstp(ff,a,b,fa,fm,fb,is)
  m = (a + b)/2
  h = (b - a)/4
  x=OffsetArray{Float64}(undef,0:1)
  x[0]=a + h
  x[1]=b - h
  n=size(x)[1]
  y=OffsetArray{Float64}(undef,0:n-1)
  for i=0:n-1
    y[i]=ff(x[i])
  end
  fml = y[0]
  fmr = y[1]
  i1 = h/1.5 * (fa + 4*fm + fb)
  i2 = h/3 * (fa + 4*(fml + fmr) + 2*fm + fb)
```

```
  global Q=0
  i1 = (16.0*i2 - i1)/15.0 #romberg integration
  nx1=is + (i1-i2)
  if nx1==is || m<= a || b<=m
    if m <= a || b<=m
      print("required tolerance can not be achieved")
    end #if m
    Q = i1
  else
    Q = adaptsimstp(ff,a,m,fa,fml,fm,is)+adaptsimstp(ff,m,b,fm,fmr,fb,is)
  end #if nx1
  return Q
  end #function


using OffsetArrays
f(x)=sin(x)
a=0.0
b=pi
I=adaptive_simpson_integral2(f,a,b)
print("I = $I")
```

Gander and Gautschi adaptive Simpson 1/3 integral Formula with sample integral

$$I = \int_{0}^{\pi} sin(x)dx$$

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" adaptive_simpson_integral2.jl
I = 2.0
> Terminated with exit code 0.
```

$$I = \int_{0}^{1} x^{1.5}dx$$

```
using OffsetArrays
f(x)=x^1.5
a=0.0
b=1.0
I=adaptive_simpson_integral2(f,a,b)
print("I = $I")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" adaptive_simpson_integral2.jl
I = 0.4
> Terminated with exit code 0.
```

7.4-5 Adaptive Gauss-Lobatto integral with Gander ve Gautschi[31] approach

Gauss-Lobatto integration Formula is similar to Gauss integration Formula. The most important difference is that Lobatto Formula also covers end points

$$I_w \cong \int_{-1}^{1} f(x)dx = \sum_{k=0}^{n-1} c_k f(x_k)$$

For various n values Gauss-Lobatto coefficients and roots are given

Gauss-Lobatto integration equation coefficients and roots

N	Xk	Ck
4	-1.000000000000000000000000	0.166666666666667000000000
	1.000000000000000000000000	0.166666666666667000000000
	-0.447213595499958000000000	0.833333333333333000000000
	0.447213595499958000000000	0.833333333333333000000000
5	-1.000000000000000000000000	0.100000000000000000000000
	1.000000000000000000000000	0.100000000000000000000000

	-0.65465367070797700000000	0.5444444444444400000000
	0.65465367070797700000000	0.5444444444444400000000
	0.00000000000000000000000	0.7111111111111100000000
6	-1.00000000000000000000000	0.0666666666666660000000
	1.00000000000000000000000	0.0666666666666660000000
	-0.76505532392946500000000	0.3784749562978470000000
	0.76505532392946500000000	0.3784749562978470000000
	-0.28523151648063500000000	0.5548583770354860000000
	0.28523151648063500000000	0.5548583770354860000000
7	-1.00000000000000000000000	0.0523809523809524000000
	1.00000000000000000000000	0.0523809523809524000000
	-0.81649658092772600000000	0.2938775510204080000000
	0.81649658092772600000000	0.2938775510204080000000
	-0.44721359549995800000000	0.4251700680272110000000
	-0.44721359549995800000000	0.4251700680272110000000
	0.00000000000000000000000	0.4571428571428570000000
12	-1.00000000000000000000000	0.0158271919734801000000
	1.00000000000000000000000	0.0158271919734801000000
	-0.94288241569547900000000	0.0942738402188500000000
	0.94288241569547900000000	0.0942738402188500000000
	-0.81649658092772600000000	0.1550719873365850000000
	0.81649658092772600000000	0.1550719873365850000000
	-0.64185334234578100000000	0.1888215739601820000000
	0.64185334234578100000000	0.1888215739601820000000
	-0.44721359549995800000000	0.1997734052268580000000
	0.44721359549995800000000	0.1997734052268580000000
	-0.23638319966214900000000	0.2249264653333390000000
	0.23638319966214900000000	0.2249264653333390000000
	0.00000000000000000000000	0.2426110719014070000000

As the previous simpson 1/3 approach **Gander ve Gautschi[31]** approach will be used as a stopping criteria in this program also. For the interval [a,b]

if((is+(I1 - I2) == is) || (m<=a) || (b<=m))

In this condition m is given as

$m = \frac{(a-b)}{2}$ For the is first integration value degree 12 Lobatto integration will be used.

$$is \cong \int_a^b f(x)dx \cong (b-a)*\{0.0158271919734801*[f(a)+f(b)]+$$
$$0.094273840218850*[f(m-h*0.942882415695479)+f(m+h*0.942882415695479)]+$$
$$0.155071987336585*[f(m-h*0.816496580927726)+f(m+h*0.816496580927726)]+$$
$$0.188821573960182*[f(m-h*0.641853342345781)+f(m+h*0.641853342345781)]+$$
$$0.199773405226858*[f(m-h*0.447213595499958)+f(m+h*0.447213595499958)]+$$
$$0.224926465333339*[f(m-h*0.236383199662149)+f(m+h*0.236383199662149)]+$$
$$0.242611071901407*[f(0)]\}$$

As iteration Formula 4th and 7th degree Gauss-Lobatto formulations will be used.

$$I1 \cong \int_a^b f(x)dx \cong (b-a)*\{0.16666666\ 6666667\ *[f(a)+f(b)]+$$
$$0.83333333\ 3333333\ *[f(m-h*0.44721359\ 5499958)\ +f(m+h*0.44721359\ 5499958)]\ \}$$

$$I2 \cong \int_a^b f(x)dx \cong (b-a)*\{0.0523809523809524*[f(a)+f(b)]+$$
$$0.293877551020408*[f(m-h*0.816496580927726)+f(m+h*0.816496580927726)]+$$
$$0.425170068027211*[f(m-h*0.447213595499958)+f(m+h*0.447213595499958)]+$$
$$0.457142857142857*[f(0)]\}$$

Program code for this approach is given below.

Gander and Gautschi adaptive Gauss-Lobatto integral formula

```
# =====================================================
# Numerical Analysis package in Julia
# example to show utilisation of integration (integral)
# and differentiation (derivative) functions
# Gander and Gautschi adaptive Gauss-Lobatto integral
# Dr. M. Turhan Coban
# =====================================================
function adaptive_Lobatto_integral(ff,a,b)
   i=0
            m=(a+b)/2.0
            h=(b-a)/2.0
   alpha=sqrt(2.0/3.0)
   beta=1.0/sqrt(5.0)
   x1=0.942882415695479719056351758431857202 32
   x2=0.641853342345781305781235541329031883 54
   x3=0.236383199662149880282223773492052925 99
   A=0.015827191973480183087169986733305510591
   B=0.094273840218850045531282505077108171960
   C=0.155071987336585396253635979802102986 80
   D=0.188821573960182454420005339372971671 25
   E=0.199773405226858526792068022066488402 46
   F=0.224926465333339527016017687996395080 76
   G=0.242611071901407733799640957903256352 33
   n=13
   x=OffsetArray{Float64}(undef,0:n-1)
   y=OffsetArray{Float64}(undef,0:n-1)
   x[0]=a;x[1]=m-x1*h;x[2]=m-alpha*h;x[3]=m-x2*h;x[4]=m-beta*h;x[5]=m-
x3*h;x[6]=m;x[7]=m+x3*h;x[8]=m+beta*h;x[9]=m+x2*h;x[10]=m+alpha*h;x[11]=m+x1*h;;x[12]=b
   #x=Float64[a,(m-x1*h),(m-alpha*h),(m-x2*h),(m-beta*h),(m-
x3*h),m,(m+x3*h),(m+beta*h),(m+x2*h),(m+alpha*h),(m+x1*h),b]
            for i=0:n-1
              y[i]=ff(x[i])
            end
            fa=y[0]
            fb=y[n-1]
   i2=(h/6.0)*(y[0]+y[12]+5.0*(y[4]+y[8]))
   i1=(h/1470.0)*(77.0*(y[0]+y[12])+432.0*(y[2]+y[10])+625.0*(y[4]+y[8])+672.0*y[6])
   is=h*(A*(y[0]+y[12])+B*(y[1]+y[11])+C*(y[2]+y[10])+D*(y[3]+y[9])+E*(y[4]+y[8])+F*(y[5]+y[7])+G*y[6])
   s=Int64(is/abs(is))
   if(s==0)
     s=1
   end #if
   erri1=abs(i1-is)
   erri2=abs(i2-is)
   R=erri1/erri2
   if(is==0)
     is=b-a
   end #if
  global Q=0
  Q=Lobatto(ff,a,b,fa,fb,is)
  return Q
end

function Lobatto(ff,a,b,fa,fb,is)
  h=(b-a)/2.0
  m=(a+b)/2.0
  alpha=sqrt(2.0/3.0)
  beta=1.0/sqrt(5.0)
```

```
  mll=m-alpha*h
  ml=m-beta*h
  mr=m+beta*h
  mrr=m+alpha*h
  n=5
  x=OffsetArray{Float64}(undef,0:n-1)
  x[0]=mll;x[1]=ml;x[2]=m;x[3]=mr;x[4]=mrr
  #x=[mll,ml,m,mr,mrr]
  #n=size(x)[1]
  y=OffsetArray{Float64}(undef,0:n-1)
  for i=0:n-1
   y[i]=ff(x[i])
  end #for
  fmll=y[0]
  fml=y[1]
  fm=y[2]
  fmr=y[3]
  fmrr=y[4]
  i2=(h/6)*(fa+fb+5*(fml+fmr))
  i1=(h/1470)*(77*(fa+fb)+432*(fmll+fmrr)+625*(fml+fmr)+672*fm)
  w1=i1-i2
  w2=is
  w3=w2+w1
  c1=(w3==w2)
  c2=(mll<=a)
  c3=(b<=mrr)
  if  c1|| c2 ||c3
   if (m <= a) || (b<=m)
      print("required accuracy can nor be achieved")
   end #if
   Q=i1
  else
    Q=Lobatto(ff,a,mll,fa,fmll,is) + Lobatto(ff,mll,ml,fmll,fml,is)+
      Lobatto(ff,ml,m,fml,fm,is)   + Lobatto(ff,m,mr,fm,fmr,is)+
      Lobatto(ff,mr,mrr,fmr,fmrr,is)+Lobatto(ff,mrr,b,fmrr,fb,is)
  end # if
  return Q
end #function

using OffsetArrays
f(x)=sin(x)
a=0.0
b=pi
I=adaptive_Lobatto_integral(f,a,b)
print("I = $I")
```

Gander and Gautschi adaptive Gauss-Lobatto integral formula

$$I = \int_0^{\pi} sin(x)dx$$

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" adaptive_Lobatto_integral.jl
I = 2.0
> Terminated with exit code 0.
```

$$I = \int_0^{1} x^{1.5}dx$$

```
using OffsetArrays
f(x)=x^1.5
a=0.0
b=1.0
```

```
I=adaptive_Lobatto_integral(f,a,b)
print("I = $I")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" adaptive_Lobatto_integral.jl
I = 0.4
> Terminated with exit code 0.
```

In order to see how successfull the code will be a mathematical function test will be carried out as the next example. The definition of the error function :

$$erf(\eta) = \frac{1}{\sqrt{\pi}} \int_0^{\eta} e^{-\eta^2} d\eta$$

Because of the difficulty of calculating this equation with integration Formula, it is generally calculated by serial solutions. In our example both serial solution and Gauss-Lobatto adaptive integration solution will be carried out and compared.

Program 7.4-6 Calculation of error function with Gander and Gautschi adaptive Gauss-Lobatto integral formula

```
 using OffsetArrays
using SpecialFunctions
f(x)=2.0/sqrt(pi)*exp(-x*x)
a=0.0
b=[0.1,0.4,1.0,2.0]
n=size(b)[1]
for i=1:n
  xx=b[i]
  y1=erf(xx)
  y2=adaptive_Lobatto_integral(f,a,b[i])
  println("erf($xx) = $y1 GL integral erf($xx) = $y2")
end
```

Calculation of error function with Gander and Gautschi adaptive Gauss-Lobatto integral Formula output

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" adaptive_Lobatto_integral.jl
erf(0.1) = 0.1124629160182849 GL integral erf(0.1) = 0.1124629160182849
erf(0.4) = 0.42839235504666845 GL integral erf(0.4) = 0.42839235504666845
erf(1.0) = 0.8427007929497149 GL integral erf(1.0) = 0.8427007929497149
erf(2.0) = 0.9953222650189527 GL integral erf(2.0) = 0.9953222650189527

> Terminated with exit code 0.
```

ONE AND MULTI-DIMENSIONAL MONTE-CARLO INTEGRAL

Monte Carlo integral is based on a very simple principle

$$\int_a^b f(x)dx = (b-a) * f_{average}$$

$$x = a + (b-a) * random\_number(0,1)$$

$$f_{average} = \frac{\sum_{i=1}^{n} f(x)}{n} \qquad n \gg 1$$

In the interval [a,b] a random x number is selected and function is evalueated in this point. When this sequaence is repeated for a very large number of times avarage value of the function can be calculated and directly used in the calculatetions of the integral. Integral is the calculateion of the avarage function value with the interval width. Due to fact that this method requires a huge amount of

function evaluation, therefore it is not very practical for one variable integration, but in multivariable form it becomes a practical, sometimes only available approach.

One variable Monte Carlo integration

```
 # ====================================================
# Numerical Analysis package in Julia
# example to show utilisation of integration (integral)
# and differentiation (derivative) functions
# Monte Carlo integration with one variable
# Dr. Turhan Coban
# ====================================================
function monte_carlo(ff,a,b,n)
      # Monte-carlo integral
       h=(b-a)
       sum=0.0
       x=0.0
       f
       for i=1:n
         x=a+(b-a)*rand()
         sum+=ff(x)
       end
       return (b-a)*sum/n
end


f(x)=x
a=0.0
b=1.0
n=50000
I=monte_carlo(f,a,b,n)
print("I = $I")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" monte_carlo_integral.jl
I = 0.49994511866715496
> Terminated with exit code 0.
```

If Monte Carlo equation is written for two independent variables

$$\int_{ay}^{by}\int_{ax=\alpha(x)}^{bx=\beta(x)} f(x,y)dx\,dy = \mu(D) * f_{average}$$

$$x = ax + (bx - ax) * random\_number(0,1)$$

$$y = ay + (by - ay) * random\_number(0,1)$$

$$f_{average} = \frac{\sum_{i=1}^{n} f(x,y)}{n} \qquad n \gg 1$$

$$\mu(D) = \int_{ay}^{by}\int_{ax=\alpha(x)}^{bx=\beta(x)} dx\,dy$$

As it is seen from the equation process fro one variable integration and two variable integraton is almost the same. This is the main factor why it is frequently use as a high dimensional integration tool.

Program 7.6-2 Monte Carlo integration with two independent variables

```
 # ====================================================
# Numerical Analysis package in Julia
# example to show utilisation of integration (integral)
# and differentiation (derivative) functions
# Monte Carlo integration with one variable
# Dr. M. Turhan Coban
# ====================================================
function monte_carlo(ff,a,b,n)
      # Monte-carlo integral
```

```
    h=(b-a)
    sum=0.0
    x=0.0
    f
    for i=1:n
      x=a+(b-a)*rand()
      sum+=ff(x)
    end
    return (b-a)*sum/n
end


f(x)=x
a=0.0
b=1.0
n=50000
I=monte_carlo(f,a,b,n)
print("I = $I")
```

Output of Monte Carlo integration with two independent variables

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" monte_carlo_integral.jl
I = 0.49994511866715496
> Terminated with exit code 0.
```

MULTIDIMENSIONAL INTEGRATION BY PARAMETRIC METHODS

A two dimensional integration

$$\mathrm{I(D)} = \int\int_D f(x,y)dydx$$

Where D is a region in xy plane. If this region is defined as $a \le x \le b$ and $c(x) \le y \le d(x)$, one way two calculate two dimensional integration is to repeat one dimensional integration twice. In the first step intehration function g(x) is calculated as an integration in y dimension and then function g(x) is solved to get integration result.

$$I = \int_a^b \int_{c(x)}^{d(x)} f(x,y)dydx$$

$$\mathrm{g(x)} = \int_{c(x)}^{d(x)} f(x,y)dy, \quad a \le x \le b$$

$$\mathrm{I(D)} = \int_a^b g(x)dx$$

Any integration formulation can be used for this purpose. All it has to be done is to evaluate integration Formula twice onece for each independent variable. As our first example Gauss-Legendre formulation will be used to evaluate double integration. Two code is defined. In the first one Gauss-Legendre coefficients are defined as constants, in the second one they are calculated through Legendre polynomials. In the two dimensional integration x dimesion limits are still given as constant values, but y dimension limits c(x) and d(x) should be given as functions for proper evaluation of the integral.

$$I = \int_a^b \int_{c(x)}^{d(x)} f(x,y)dy = \int_a^b \frac{d(x)-c(x)}{2} \sum_{J=1}^{n} c_{n,j} f(x, \frac{(d(x)-c(x))r_{n,j} + d(x) + c(x)}{2})dx$$

In this equation $r_{n,j}$ root values and $c_{n,j}$ are equation coefficients. These values can be calculated from Legendre polynomials.

Program 7.7-1 60 constant points two variable Gauss-Legendre integration formula

```
# ==================================================
# Numerical Analysis package in Julia
# example to show utilisation of integration (egral)
# and differentiation (derivative) functions
# Gauss-Legendre integration with two variables
# constant number of coefficients (60 coefficients)
# and roots
# Dr. Turhan Coban
# ==================================================

function integral(f,a2,b2,c2,d2)
# integral f(x)dx
# integral of a function by using gauss-legendre quadrature
# coefficients are pre-calculated for 60 terms for [-1,1]
# band then utilises variable transform
  i=0
  j=0
  n=60
  global r=OffsetArray{Float64}(undef,0:n-1)
  global c=OffsetArray{Float64}(undef,0:n-1)
 r[0] = .1553257962675247000E-02
 r[1] = .8165938360126412000E-02
 r[2] = .1998906751584623000E-01
 r[3] = .3689997628536285000E-01
 r[4] = .5871973210397363000E-01
 r[5] = .8521711880861582000E-01
 r[6] = .1161112839475869000E+00
 r[7] = .1510747526033421000E+00
 r[8] = .1897369085053786000E+00
 r[9] = .2316879259289901000E+00
 r[10] = .2764831152309554000E+00
 r[11] = .3236476372345609000E+00
 r[12] = .3726815369160551000E+00
 r[13] = .4230650431957083000E+00
 r[14] = .4742640787223412000E+00
 r[15] = .5257359212776589000E+00
 r[16] = .5769349568042917000E+00
 r[17] = .6273184630839449000E+00
 r[18] = .6763523627654391000E+00
 r[19] = .7235168847690445000E+00
 r[20] = .7683120740710099000E+00
 r[21] = .8102630914946214000E+00
 r[22] = .8489252473966580000E+00
 r[23] = .8838887160524131000E+00
 r[24] = .9147828811913842000E+00
 r[25] = .9412802678960264000E+00
 r[26] = .9631000237146372000E+00
 r[27] = .9800109324841537000E+00
 r[28] = .9918340616398735000E+00
 r[29] = .9984467420373248000E+00
c[0] = .3984096248082779000E-02
c[1] = .9233234155545500000E-02
c[2] = .1439235394166167000E-01
c[3] = .1939959628481353000E-01
c[4] = .2420133641529259000E-01
c[5] = .2874657810880872000E-01
c[6] = .3298711494109008000E-01
```

```
c[7] = .36877987368852570000E-01
c[8] = .40377947614710090000E-01
c[9] = .43449893600541500000E-01
c[10] = .46061261118893050000E-01
c[11] = .48184368587322120000E-01
c[12] = .49796710293397640000E-01
c[13] = .50881194874202750000E-01
c[14] = .51426326446779420000E-01
c[15] = .51426326446779420000E-01
c[16] = .50881194874202750000E-01
c[17] = .49796710293397640000E-01
c[18] = .48184368587322120000E-01
c[19] = .46061261118893050000E-01
c[20] = .43449893600541500000E-01
c[21] = .40377947614710090000E-01
c[22] = .36877987368852570000E-01
c[23] = .32987114941090080000E-01
c[24] = .28746578108808720000E-01
c[25] = .24201336415292590000E-01
c[26] = .19399596284813530000E-01
c[27] = .14392353941661670000E-01
c[28] = .92332341555455000000E-02
c[29] = .39840962480827790000E-02
for i=0:29
  r[i+30]=-r[i]
  c[i+30]=c[i]
end
m=n
# 1=========================
a1=a2
b1=b2
h1=(b1-a1)/2.0
h2=(b1+a1)/2.0
J=0
# 2=========================
for i=0:m-1
# 3=========================
JX=0
x=OffsetArray{Float64}(undef,0:1)
x[0]=h1*r[i]+h2
d1=d2(x[0])
c1=c2(x[0])
k1=(d1-c1)/2.0
k2=(d1+c1)/2.0
# 4=========================
    for j=0:n-1
            x[1]=k1*r[j]+k2
            Q=f(x)
            JX+=c[j]*Q
            end
# =========================
J=J+c[i]*k1*JX
end
J=h1*J
return J
end

using OffsetArrays
f(x)=log(x[0]+2.0*x[1])
a2=1.4
b2=2.0
c2(x)=1.0
```

```
d2(x)=1.5
I=integral(f,a2,b2,c2,d2)
print("integral = $I")
```

constant points two variable Gauss-Legendre integration formula output

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_GL2.jl
integral = 0.429554527548266
> Terminated with exit code 0.
```

In the next example, instead of constant number of points Legendre transformation defined varible number of points are taken.

Program variable points two variable Gauss-Legendre integration formula

```
 # ==================================================
# Numerical Analysis package in Julia
# example to show utilisation of integration (integral)
# and differentiation (derivative) functions
# Gauss-Legendre integration with two variables
# variable number of coefficients and roots
# calculated by Legendre polynomials
# Dr. Turhan Coban
# ==================================================
function  gauss_legendre_coefficients(x1,x2,n)
        #calculates legendre gauss-coefficients as coefficients of the integral
        #for n terms
        EPS=3.0e-15
        m=0
        j=0
        i=0
        global a=OffsetArray{Float64}(undef,0:1,0:n-1)
        global pp=0.0
        m=Int64(floor((n+1)/2))
        xm=0.5*(x2+x1)
        xl=0.5*(x2-x1)
        for i=1:m
                z=cos(pi*((i-0.25)/(n+0.5)))
                while(true)
                        p1=1.0
                        p2=0.0
                        for j=1:n
                                p3=p2
                                p2=p1
                                p1=((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j
                        end # for
                        pp=n*(z*p1-p2)/(z*z-1.0)
                        z1=z
                        z=z1-p1/pp
                        if abs(z-z1) <= EPS
                          break
                        end #if
                end #while
                a[0,i-1]=xm-xl*z
                a[0,n-i]=xm+xl*z
                a[1,i-1]=2.0*xl/((1.0-z*z)*pp*pp)
                a[1,n-i]=a[1,i-1]
        end # for
return a
end # function
```

```
function integral(f,a2,b2,c2,d2,n,m)
# integral f(x,y)dxdy
# integral of a function by using gauss-legendre quadrature
# coefficients are pre-calculated for n,m terms for [-1,1]
# band then utilises variable transform
 i=0
 j=0
 global rn=OffsetArray{Float64}(undef,0:n-1)
 global rm=OffsetArray{Float64}(undef,0:m-1)
 global cn=OffsetArray{Float64}(undef,0:n-1)
 global cm=OffsetArray{Float64}(undef,0:m-1)
 global a=gauss_legendre_coefficients(-1.0,1.0,n)
 #print("a = $a")
 for i=0:n-1
   rn[i]=a[0,i]
   cn[i]=a[1,i]
 end #for
 for i=0:m-1
   rm[i]=a[0,i]
   cm[i]=a[1,i]
 end #for
 global J=0.0
# x[],h1,h2,J,a1,b1,c1,d1,k1,k2,JX,Q
x=OffsetArray{Float64}(undef,0:1)
a1=a2
b1=b2
h1=(b1-a1)/2.0
h2=(b1+a1)/2.0
J=0
for  i=0:m-1
JX=0
x[0]=h1*rm[i]+h2
d1=d2(x[0])
c1=c2(x[0])
k1=(d1-c1)/2.0
k2=(d1+c1)/2.0
#4=========================
    for j=0:n-1
            x[1]=k1*rn[j]+k2
            Q=f(x)
            JX+=cn[j]*Q
            end #for
#5=========================
J=J+cm[i]*k1*JX
end #for
J=h1*J
return J
end #function

using OffsetArrays
f(x)=log(x[0]+2.0*x[1])
a2=1.4
b2=2.0
c2(x)=1.0
d2(x)=1.5
I=integral(f,a2,b2,c2,d2,60,60)
print("integral = $I")
```

variable points two variable Gauss-Legendre integration Formula output

```
---------- Capture Output ----------
```

```
> "C:\co\Julia\bin\julia.exe" integral_GL_2A.jl
integral = 0.4295545275482747
> Terminated with exit code 0.
```

For the two variable integration Newton-Cotes formulas can also be used. As a example a Simpson 1/3 two dimensional integration code is prepared. Dor the quadratic polynomial Newton-Cotes equation (simpson 1/3 method) equations can be written as follows;
One dimensional form was

$$h = \frac{b-a}{n} = \frac{b-a}{2}$$

$$I = \int_a^b f(x)dx = \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)]$$

If this Formula can be written in composite form.

$$I = \int_a^b f(x)dx = \frac{k}{3}\left[f(x,y_0) + 2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 2\sum_{j=1}^{\frac{n}{2}} f(x_{2j-1}) + f(x_n)\right]$$

This Formula can be open to the second dimension similar to Gauss-Legendre integration
By defining

$$k = \frac{d(x)-c(x)}{m}$$

integration equation becomes

$$I = \int_{c(x)}^{d(x)} f(x,y)dx = \frac{k}{3}\left[f(x,y_0) + 2\sum_{j=1}^{\frac{m}{2}-1} f(x,y_{2j}) + 2\sum_{j=1}^{\frac{m}{2}} f(x,y_{2j-1}) + f(x,y_m)\right]$$

From here x integration can be carried out as:

$$I = \int_a^b \int_{c(x)}^{d(x)} f(x,y)dx = \int_a^b \frac{k}{3}\left[f(x,y_0) + 2\sum_{j=1}^{\frac{m}{2}-1} f(x,y_{2j}) + 2\sum_{j=1}^{\frac{m}{2}} f(x,y_{2j-1}) + f(x,y_m)\right]dx$$

The same integration equation is then applied to the outer integration. Complete code is given in the program 8.7.3 . In the example the same function as in previous Gauss-Legendre integration is taken to compare the results

Program 7.7-3 two dimensional Simpson 1/3 integration formula

```
 # =====================================================
# Numerical Analysis package in Julia
# simpson integration of two dimensional function
# and differentiation (derivative) functions
# two dimensional Simpson 1/3 integration
# (quadratic Newton-Cotes formula)
# Dr. Turhan Coban
# =====================================================

function integral(f,a,b,c,d,m,n)
#integral f(x)dx
#integral of a function by using simpson metod
h=(b-a)/n
```

```
global J1=0.0
global J2=0.0
global J3=0.0
global x=OffsetArray{Float64}(undef,0:1)
HX=0
K1=0.0
K2=0.0
K3=0.0
global  Q=0.0
global  L=0.0
global  J=0.0
for i=0:n-1
x[0]=a+i*h
d1=d(x[0])
c1=c(x[0])
HX=(d1-c1)/m
x[1]=c1
K1=f(x)
x[1]=d1
K1+=f(x)
K2=0.0
K3=0.0
  for j=1:m
    x[1]=c(x[0])+j*HX
    Q=f(x)
    m1=Int64(floor(j/2)*2)
    if m1==j
      K2=K2+Q
    else
      K3=K3+Q
    end #if
  end #for
L=(K1+2.0*K2+4.0*K3)*HX/3.0
n1=Int64(floor(i/2)*2)
if (i==0)||(i==n)
  J1=J1+L
elseif n1==i
   J2=J2+L
else J3=J3+L
end #if
end #for
J=h*(J1+2.0*J2+4.0*J3)/3.0
end #function

using OffsetArrays
f(x)=log(x[0]+2.0*x[1])
a2=1.4
b2=2.0
c2(x)=1.0
d2(x)=1.5
I=integral(f,a2,b2,c2,d2,20,20)
print("integral = $I")
```

two dimensional Simpson 1/3 integration formula

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_simpson1_3__2D.jl
integral = 0.4372450534360736
> Terminated with exit code 0.
```

DIRECT INTEGRATION OF DATA

If integration of a numerical data set is given two approches and be adapted for this process. The first one is to curve fit a function first and then integrate the fitted function. The second one is to apply an integration equation like Newton-Cotes equation directly to the data to achive integrated result without an intermediate process.

As an example to curve fitting integration a cubic spline integration will be described here. If a cubic polynomial of the following form is given :

$$s_k(x) = a_k(x - x_k) + b_k(x_{k+1} - x) + \frac{\left[(x - x_k)^3 c_{k+1} + (x_{k+1} - x)^3 c_k\right]}{6h_k} \quad 1 \le k \le n$$

Integration of the equation could be described with the equation

$$\int_a^b S_k(x)dx = \sum_{x=a}^{b} a_k \frac{(x - x_k)^2}{2} - b_k \frac{(x_{k+1} - x)^2}{2} + c_k \frac{(x - x_k)^4}{24h_k} - d_k \frac{(x_{k+1} - x)^4}{24h_k}$$

For a, x_0, x_1, x_2, ... x_n,. An example program using this approach is given as:

Program 7.8-1 Direct integration of data by using cubic polynomial fitting

```
 # ==================================================
# Numerical Analysis package in Julia
# example to show utilisation of integration (integral)
# and differentiation (derivative) functions
# Direct data integration by using cubic spline
# curve fitting
# Dr. Turhan Coban
# ==================================================
function thomas(a,r)
  n=size(a)[1]
  global f=OffsetArray{Float64}(undef,0:n-1)
  global e=OffsetArray{Float64}(undef,0:n-1)
  global g=OffsetArray{Float64}(undef,0:n-1)
  global x=OffsetArray{Float64}(undef,0:n-1)

  for i=0:n-1
    f[i]=a[i,i]
  end

  for i=0:n-1
    g[i]=a[i,i+1]
  end

  for i=0:n-1
    e[i]=a[i+1,i]
  end

  for k=1:n-1
    e[k]=e[k]/f[k-1]
    f[k]=f[k]-e[k]*g[k-1]
  end #for

  for k=1:n-1
    r[k]=r[k]-e[k]*r[k-1]
  end #for

  x[n-1]=r[n-1]/f[n-1]
  n1=n-2
  for k=n1:-1:0
    x[k]=(r[k]-g[k]*x[k+1])/f[k]
```

```
  end #for
  return x
  end #function

function thomas(f,e,g,r)
  n=size(f)[1]
  global x=OffsetArray{Float64}(undef,0:n-1)
  for k=1:n-1
    e[k]=e[k]/f[k-1]
    f[k]=f[k]-e[k]*g[k-1]
  end #for

  for k=1:n-1
    r[k]=r[k]-e[k]*r[k-1]
  end #for

  x[n-1]=r[n-1]/f[n-1]
  n1=n-2
  for k=n1:-1:0
    x[k]=(r[k]-g[k]*x[k+1])/f[k]
  end #for
  return x
end #function

function cubic_spline(xi,yi,c0,cn)
  # c0 second derivative of the first point
  # cn second derivative of the last point
  n=size(xi)[1]
  print("n = $n")
  Aw = 4
  global h=OffsetArray{Float64}(undef,0:n-1)
  global w=OffsetArray{Float64}(undef,0:n-1)
  global f=OffsetArray{Float64}(undef,0:n-1)
  global e=OffsetArray{Float64}(undef,0:n-1)
  global g=OffsetArray{Float64}(undef,0:n-1)
  global d=OffsetArray{Float64}(undef,0:n-1)
  global x=OffsetArray{Float64}(undef,0:n-1)
  global S=OffsetArray{Float64}(undef,0:Aw-1,0:n-1)
  k=0
  for k=0:n-2
    h[k]=xi[k+1]-xi[k]
    w[k]=(yi[k+1]-yi[k])/h[k]
       wk=w[k]
  end #for
  d[0]=c0
  d[n-1]=cn
  for k=1:n-2
     d[k]=6.0*(w[k]-w[k-1])
      dk=d[k]
  end #for
  f[0]=1.0
  f[n-1]=1.0
  g[0]=0.0
  g[n-1]=0.0
  e[0]=0.0
  e[n-1]=0.0
  for k=1:n-2
    f[k]=2.0*(h[k]+h[k-1])
    e[k]=h[k-1]
    g[k]=h[k]
  end #for
  xx1=thomas(f,e,g,d)
```

```
  for i=0:n-1
    S[2,i]=xx1[i]
    print
    S[3,i]=xi[i]
  end
  for k=0:n-2
    S[0,k]=(6.0*yi[k+1]-h[k]*h[k]*S[2,k+1])/(6.0*h[k])
    S[1,k]=(6.0*yi[k]-h[k]*h[k]*S[2,k])/(6.0*h[k])
  end #for
  return S
  end #for

function intSpline(S,a,b)
# cubic spline integration
  n=size(S)[2]
  xx1=0.0
  xx2=0.0
  hk=0.0
  toplam=0.0
  y1=0.0
  y2=0.0
  for k=0:n-2
     hk=(S[3,k+1]-S[3,k])
     if a>S[3,k+1] #condition 1
      toplam=0.0
     elseif  S[3,k]<=a && a<=S[3,k+1] && b>S[3,k+1]  #condition 2
       xx1=(a-S[3,k])
       xx2=(S[3,k+1]-a)
       y1=S[0,k]*xx1*xx1/2.0-S[1,k]*xx2*xx2/2.0+(xx1*xx1*xx1*xx1*S[2,k+1]-xx2*xx2*xx2*xx2*S[2,k])/(24.0*hk)
       xx1=hk
       xx2=0
       y2=S[0,k]*xx1*xx1/2.0-S[1,k]*xx2*xx2/2.0+(xx1*xx1*xx1*xx1*S[2,k+1]-xx2*xx2*xx2*xx2*S[2,k])/(24.0*hk)
       toplam+=(y2-y1)
    elseif S[3,k]<=a && a<=S[3,k+1] && S[3,k]<=b && b<=S[3,k+1] #condition 3
       xx1=(a-S[3,k])
       xx2=(S[3,k+1]-a)
       y1=S[0,k]*xx1*xx1/2.0-S[1,k]*xx2*xx2/2.0+(xx1*xx1*xx1*xx1*S[2,k+1]-xx2*xx2*xx2*xx2*S[2,k])/(24.0*hk)
       xx1=(b-S[3,k])
       xx2=(S[3,k+1]-b)
       y2=S[0,k]*xx1*xx1/2.0-S[1,k]*xx2*xx2/2.0+(xx1*xx1*xx1*xx1*S[2,k+1]-xx2*xx2*xx2*xx2*S[2,k])/(24.0*hk)
       toplam+=(y2-y1)
    elseif(a<S[3,k] && b>=S[3,k+1]) #condition 4
       xx1=0
       xx2=hk
       y1=S[0,k]*xx1*xx1/2.0-S[1,k]*xx2*xx2/2.0+(xx1*xx1*xx1*xx1*S[2,k+1]-xx2*xx2*xx2*xx2*S[2,k])/(24.0*hk)
       xx1=hk
       xx2=0
       y2=S[0,k]*xx1*xx1/2.0-S[1,k]*xx2*xx2/2.0+(xx1*xx1*xx1*xx1*S[2,k+1]-xx2*xx2*xx2*xx2*S[2,k])/(24.0*hk)
       toplam+=(y2-y1)
    elseif S[3,k]<=b && b<=S[3,k+1] #condition 5
       xx1=0
        xx2=hk
        y1=S[0,k]*xx1*xx1/2.0-S[1,k]*xx2*xx2/2.0+(xx1*xx1*xx1*xx1*S[2,k+1]-xx2*xx2*xx2*xx2*S[2,k])/(24.0*hk)
        xx1=(b-S[3,k])
        xx2=(S[3,k+1]-b)
        y2=S[0,k]*xx1*xx1/2.0-S[1,k]*xx2*xx2/2.0+(xx1*xx1*xx1*xx1*S[2,k+1]-xx2*xx2*xx2*xx2*S[2,k])/(24.0*hk)
        toplam+=(y2-y1)
    else
       break
    end #if
  end #for
  return toplam
```

```
end # function

function intSpline(xi,yi,c0,cn,a,b)
  S=cubic_spline(xi,yi,c0,cn)
  return intSpline(S,a,b)
  end

 function convert_to_float(column)
   new_column = map(x -> (
         x = ismissing(x) ? "" : x;  # no method matching tryparse(::Type{Float64}, ::Missing)
         x = tryparse(Float64, x);   # returns: Float64 or nothing
         isnothing(x) ? missing : x; # options: missing, or "", or 0.0, or nothing
         ), column)  # input
   # returns Array{Float64,1} OR Array{Union{Missing, Float64},1}
   return new_column
end

function readDouble(name)
 open(name) do f
 # read till end of file
 m=0
 while ! eof(f)
   m+=1
   line = readline(f)
   s=split(line," ")
   n=length(s)
   global m1=m
   global n1=n
 end  #while
 close(f)
 end  #open
 global x=Matrix{Float64}(undef,n1,m1)
 open(name) do f
 # read till end of file
 m=0
 global z=Array{Float64}(undef,n1,0)
 while ! eof(f)
   m+=1
   line = readline(f)
   s=split(line," ")
   n=length(s)
   x1=convert_to_float(s)
   z=hcat(z,x1)
   global x=z
 end  #while
 close(f)
 end  #open
 #transpose matrix
 global y=x'
 return y
 end

function readOffsetDouble(name)
 y=readDouble(name)
 n=size(y)[1]
 m=size(y)[2]
 global x=OffsetArray(y, 0:n-1,0:m-1)
 return x
 end

using OffsetArrays
name="c.txt"
```

```
z=readOffsetDouble(name)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), z)
n=size(x)[1]
m=size(x)[2]
print("\nn = $n  m = m")
x=OffsetArray{Float64}(undef,0:n-1)
y=OffsetArray{Float64}(undef,0:n-1)

for i=0:n-1
 x[i]=z[i,0]
 y[i]=z[i,1]
end
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), x)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), y)
S=cubic_spline(x,y,0.0,0.0)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), S)
a=0.0
b=10.0
I=intSpline(S,a,b)
print("I=$I")
```

Following data is taken to integrate

```
0 0
1 1
2 4
5 25
6 36
7 49
8 64
7    81
10 100
```

Sample data is created from function $f(x)=x^2$. Therefore integration between 0 and 10 will give 333.3333333 as result.

Cubic spline direct data integration

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_cubic_spline.jl
10X2 OffsetArray(adjoint(::
Matrix{Float64}), 0:9, 0:1) with eltype Float64 with indices 0:9X0:1:
  1.0   1.0
  2.0   4.0
  3.0   9.0
  4.0  16.0
  5.0  25.0
  6.0  36.0
  7.0  49.0
  8.0  64.0
  9.0  81.0
 10.0 100.0
n = 10  m = m10-element OffsetArray(::Vector{Float64}, 0:9) with eltype Float64 with indices 0:9:
  1.0
  2.0
  3.0
```

```
 4.0
 5.0
 6.0
 7.0
 8.0
 9.0
 10.010-element OffsetArray(::Vector{Float64}, 0:9) with eltype Float64 with indices 0:9:
  1.0
  4.0
  9.0
 16.0
 25.0
 36.0
 49.0
 64.0
 81.0
 100.0
n = 104X10 OffsetArray(::Matrix{Float64}, 0:3, 0:9) with eltype Float64 with indices 0:3X0:9:
 3.57736  8.69057  15.6604   24.6679   35.6679   48.6604   63.6906   80.5774  100.0      1.285e-321
 1.0      3.57736  8.69057  15.6604   24.6679   35.6679   48.6604   63.6906  80.5774    3.66e-322
 0.0      2.53585  1.8566   2.03774   1.99245   1.99245   2.03774   1.8566   2.53585    0.0
 1.0      2.0      3.0      4.0       5.0       6.0       7.0       8.0      9.0        10.0
I=333.09622641509435
> Terminated with exit code 0.
```

PROBLEMS (INTEGRATION & DERIVATION)

PROBLEM 1

of **y=f(x)=1/(1 + x^2)** fonction between limits –1 ile 1 with 3 points Gauss-Legendre integration Formula

Note: Required roots and coefficients :

coefficients	Roots
c_0 0.555555555555553	x_0 0.774596669241483
c_1 0.888888888888889	x_1 0.000000000000000
c_2 0.555555555555553	x_2 0.774596669241483

PROBLEM 2

$\int_{-3}^{3} \frac{2}{1+2x^2} dx$ solve with bole rule. Use n=2

Note: Bole rule for n=1
I=(b-a)*[7*f(x0)+32*f(x1)+12*f(x2)+32*f(x3)+7*f(x4)]/90
Step size h=(b-a)/n

PROBLEM 3

$$f_{0-\lambda} = \frac{c_1}{\sigma} \int_{x=0}^{\lambda T} \frac{dx}{x^5 \left[\exp(\frac{c_2}{x}) - 1 \right]}$$

Function is known as black body radiation shape function. The values of coefficients in the equation are:
c_1 =3.743 x10^8 W.μm^4/m^2, c_2 =1.4387x10^4 μm.K, σ=5.67x10^{-8} W/(m^2.K^4) dür.
Use Gauss-Legendre integration and for a giving λT calculate black body radiation shape factor. Create a table for the values of blackbody radiation shape factor from 5000 to infinity(?)

PROBLEM 4

$$I = \int_{x=0}^{1} \int_{y=0}^{1} \frac{dy\,dx}{(1+y)^2}$$

Solve the above integration by using analytical methods and Gauss-Legendre integration

PROBLEM 5

Solve the integration

$$I = \int_{x=0}^{1} \frac{1}{1+x^2} dx$$

a) by analithical methods
b) by Trapezoidal rule with n=2(divide region into 2 parts) and calculate % error
c) Trapezoidal rule with n=2(divide region into 2 parts) and calculate % error

PROBLEM 6

Solve the integration

$$I = \int_{x=0}^{2} \frac{1}{\sqrt{1+x^2}} dx$$

By using Gauss-Legendre formulation

PROBLEM 7

For x=1

$$f(x) = \ln\left(\frac{1}{1+x^2}\right)$$

a) Calculate analytical derivative
b) Calculate derivative with 3 points central difference formula with h=0.1
c) Calculate derivative with 3 points central difference formula with h=0.05
d) Calculate derivative with Richardson extrapolation formula with h1=0.5 and h2=0.25

PROBLEM 8

$$f(x) = \int_{-1}^{1} x.\exp(x)\ dx$$

Calculate the integration by
a) Trapezoidal rule integrali
b) Simpson 3/8 integration
c) Gauss-Legendre integration
d) Romberg integration

PROBLEM 9

$$f(x) = \int_0^1 \sin(1 - x + x^2)\, dx$$

Calculate the integration by

e) Trapezoidal rule integrali
f) Simpson 3/8 integration
g) Gauss-Legendre integration
h) Romberg integration

PROBLEM 10

$$f(x) = \int_{-1}^{1} \frac{\ln(1+x^2)}{\sqrt{1-x^2}}\, dx$$

Calculate the integration by

i) Trapezoidal rule integrali
j) Simpson 3/8 integration
k) Gauss-Legendre integration
l) Romberg integration

PROBLEM 11

X	**0.99**	**2.10**	**3.22**	**4.40**	**5.70**	**7.12**	**8.01**	**8.37**	**9.32**	**9.98**
f(x)	**4.90**	**5.70**	**4.20**	**7.04**	**8.31**	**7.82**	**5.97**	**7.01**	**6.68**	**4.79**

Data given in the above table. Find the integration by using trapezoidal rule.

PROBLEM 12

Find the derivative of data given in problem 11 at point x=5

PROBLEM 13

Depth(H) and velocity(U) profile of a channel is given in below table

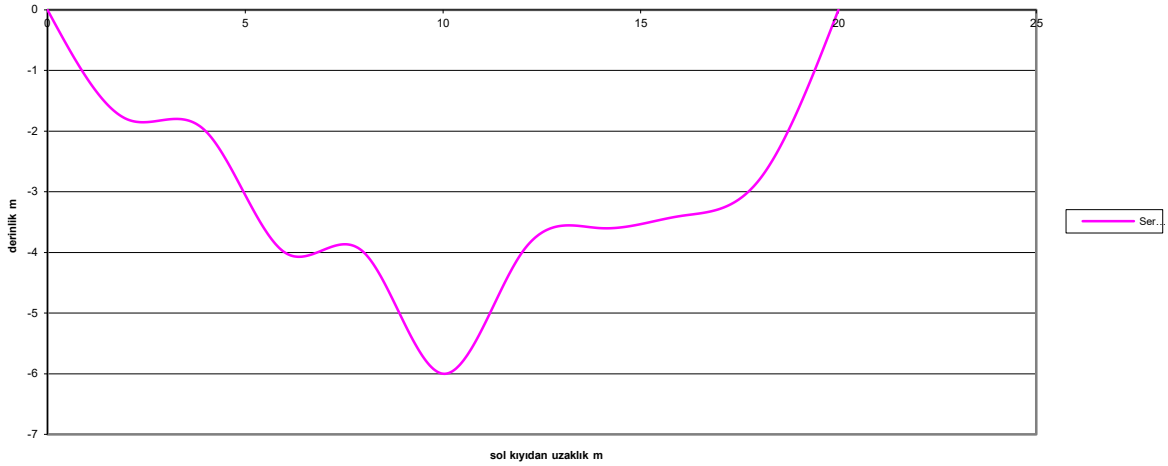
x, m	0	2	4	6	8	10	12	14	16	18	20
H, m	0	1.8	2	4	4	6	4	3.6	3.4	2.8	0
U m/s	0	0.03	0.045	0.055	0.065	0.12	0.07	0.06	0.05	0.04	0

Cross sectiaonal area of the channel can be calculated as :

$$A_c = \int_0^B H(x)dx$$

Flow rate of the channel will be

$Q = \int_0^B U(x)H(x)dx$ In this equations U is the velocity of the water measured at point x. Calculate cross sectional area and flow rate of the channel.



PROBLEM 14

Temperature profle of a solid body can be given as

$\frac{dT}{dt} = -k(T - T_a)$ In this equation T is the temperature, T_a is constant surface temperature.

A solid body with temperatures initially at T=90 C and, T_a =20 C put into a water container at temperature of T_a = 20 C. The temperature profile as a function of time is given at the table.

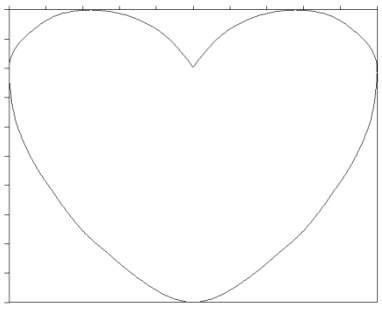
Zaman , t, s	0	5	10	15	20	25	30	35
Sicaklik T C	90	55	40	28	24	22	21.5	20.6

Calculate k thermal conductivity coefficient.

PROBLEM 15

Coordinate of a hard shape is given in the table. Find the area of the hard.

Xi	Yi	xi	yi	xi	yi	xi	Yi
241	127	331	157	215	246	149	120
258	107	323	178	199	232	160	106
283	96	312	197	184	217	175	97
307	97	298	217	170	197	199	96
322	106	283	232	159	178	224	107
333	120	267	246	151	157	241	127
334	138	241	258	148	138		



PROBLEM 16

$$I = \int_{0.1x^3}^{0.5x^2} \int e^{y/x} dydx$$ Solve the integral by Gauss-Legendre method.

PROBLEM 17

$$I = \int_{0}^{1} \int_{x}^{2x} (x^2 + y^3) dydx$$

Solve the equation with

a) 2 dimensional Gauss-Legendre integration
b) 2 dimensional simpson 1/3 integration
c) Develop an equation for 2 dimensional simpson. 3/8 integration and solve the integral

PROBLEM 18

$$I = \int_{x=0}^{1} \int_{y=0}^{1} \frac{dy\,dx}{(1+y)^2}$$

Solve the integral with Monte carlo method and check the results with Gauss-Legendre method.

PROBLEM 19

Show that equation below is correct

$$\pi = \int_{0}^{2} (4 - x^2)^{1/2} dx$$

PROBLEM 20

$$I = \int_{0}^{1} e^x dx$$

Solve the integral for m=25,50,100,500,1000,10000,100000 with monte-carlo method and compare the result with the analytical solution.

PROBLEM 21

An angine of a vehicle with mass of 53400 kg and a velocity of v=30 m/is suddenly stopped at t=0. In this instant equation of the movement can be given as

$$5400 * v * (dv/dx) = -8{,}276 * v^2 - 2000$$

In the equation v(m/sn), is the velocity of the vehicle at time t. When velocity is 15 m/s calculate the total distance vehicle moved by using 4 point Gauss integration Formula .

PROBLEM 22

A vehicle is at complete stop. When it is started velocity of the vehicle is recorded for the first 5 minutes. Calculate the total distance vehicle moved by using trapezoidal rule as a direct data integration method.

t(minutes)	0	0,5	1,0	1,5	2,0	2,5	3,0	3,5	4,0	4,5	5,0
v (km/h)	0	4	7	11	15	21	25	30	32	35	40

Note : $v = \frac{dx}{dt} \qquad x - x_0 = \int_{t0}^{t} v(t)dt$

PROBLEM 23

Upward velocity of a rocket can be calculated by using the formula

$$V = \frac{dz}{dt} = U * \ln\left[m_o / (m_o - q*t)\right] - g*t$$

In this equation V is velocity, U is the exit velocity of burning gases from the rocket nozzle, m_o is the mass of the rocket at time t = 0 sn, q is fuel spending rate , g is garavitational acceleration (assume constant as, 9,8 m^2/s). If U = 2000 m/s, m_o = 150 000 kg, q = 2600 kg/s, Calculate the height rocket will climb in 30 seconds by using Romberg inntegration with k=4 [$O(h^8)$] order of error.

Note : $V = \frac{dH}{dt} \qquad H = \int_{t0=0}^{t=30} V(t)dt$

PROBLEM 24

Calculate velocity and acceleration by using below data with

a) cental difference method [with $O(h^4)$ error level],
b) forward difference method [with $O(h^2)$ error level],
c) backward difference method [with $O(h^2)$ error level],

Time, t (s)	0	1	2	3	4	5	6	7	8
distance, x (m)	0	0.7	1.8	3.4	5.1	6.5	7.3	8.0	8.4

Note : Velocity $v = \frac{dx}{dt}$ Acceleration : $a = \frac{d^2x}{dt^2}$

PROBLEM 25

Integrate data by using trapezoidal rule

x	-3	-1	1	3	5	7	9	11
f(x)	1	-4	-9	2	4	2	6	-3

PROBLEM 26

In a river water height is measured in every 12.5 m distance. Water flow rate can be calculated by $Q = \int V*dA$ integration. In this equation Q: water flow rate (m^3/s), V: water flow speed (m/s), A: cross-sectional area (m^2). If avarage water velocity is 0.4 m/s. Calculate water flow rate by using Simpson 1/3 rule

X(m)	0	12,5	25	37,5	50	62,5	75	87,5	100
f(X) m	11	7,5	5,5	0	1,5	4	5,5	8	11

PROBLEM 27

a) $\int_0^{10} f(x)dx$ calculate the integral value from the data given below

b) calculate the derivative of the function at x=3,4,5,6,7,8

X	0.99	2.10	3.22	4.40	5.70	7.12	8.01	8.37	9.32	9.98
f(x)	4.90	5.70	4.20	7.04	8.31	7.82	5.97	7.01	6.68	4.79

PROBLEM 28

By using Gauss-Legendre integral formulation calculate $I = \int_{-4}^{4} \frac{dx}{1+x^2}$ integral for n=2,3,4,5 .

Compare the result with analytical solution.

PROBLEM 29

By using **Adaptive** simpson 1/3 solve $I = \int_{-4}^{4} \frac{dx}{1+x^2}$ integral with an error level of 1.0e-5

PROBLEM 30

By using Gauss-Chebychev integration equation solve the integral $I = \int_{-1}^{1} \frac{\cos(x)dx}{\sqrt{(1-x^2)}}$ = 2.40394 and

compare the result with the exact value.

PROBLEM 31

By using trapezoidal rule calculate the integral $I = \int_{0}^{1} \frac{2dx}{2+\sin(10\pi x)} = 1.15470054$ for n=2,4,8 ve 16

And then improve the result by using Richardson extrapolation Formula (Romberg integration).

PROBLEM 32

If velocity profile in a pipe is known, flow rate can be calculated as $Q = \int v\, dA$

In this equation v i the velocity of the fluid A is the cross-sectional area of the pipe. In a circular pipe A=(2 π r) and dA = (2 π r dr). If it is substituted into flow rate equation

Q =$\int$ v(2πr) dr is obtained. Where r is the radius. If the velocity distribution is given as a function of r as v = $3[1-(r/r_0)]^{(1/7)}$ and if the radius of the pipe r_0= 0.1 m calculate mass flow rate. Select your own method.

PROBLEM 33

In the integratls below exact solutions are given. By using an adaptive integration method. Chck out the results.

a) $I1 = \int_{0}^{1} \frac{4dx}{1+256(x-0.375)^2} = \frac{1}{4}[\tan^{-1}(10) + \tan^{-1}(6)] = 0.7191938309210011$

b) $I2 = \int_{0}^{1} x^2\sqrt{x}dx = \frac{2}{7}$

c) $I3 = \int_{0}^{1} \sqrt{x}dx = \frac{2}{3}$

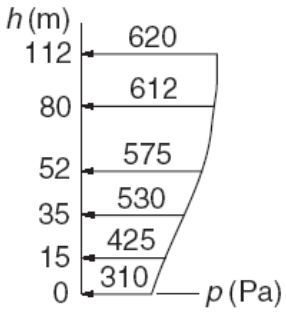
d) $I4 = \int_{0}^{1} \log(x)dx = -1$

e) $I5 = \int_{0}^{1} \log(|x-0.7|)dx = 0.3*\log(0.3) + 0.7*\log(0.7) - 1.0$

f) $I6 = \int_{0}^{2\pi} e^{-x}\sin(50x)dx = \frac{50}{2501}(1-e^{-2\pi})$

g) $I6 = \int_{-9}^{10000} \frac{dx}{\sqrt{|x|}} = 206$

PROBLEM 34



Wind pressure in a flat wall is measures as in the graphic. It is desired to find pressure effect center of the wall. It is calculated by using
:

$$\bar{h} = \frac{\int_0^{112} hp(h)dh}{\int_0^{112} p(h)dh}$$

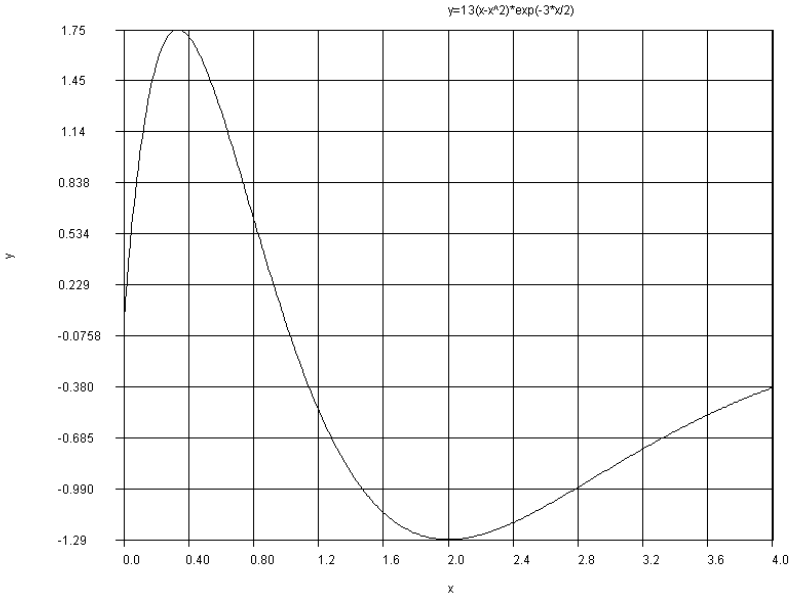
Calculate the value.

PROBLEM 35

For the analytically solved integration that result is given below:
Solve with with

a) Adaptive Simpson 1/3 method
b) Adaptive Gauss-Kronrod integral formula.

$$\int_0^4 13(x - x^2)e^{-3x/2}\,dx = \frac{4108e^{-6} - 52}{27}$$
$$= -1.5487883725279481333$$



PROBLEM 36

v(m/s)	0	1	1.8	2.5	3.5	4.4	5.1	6
P(kW)	0	4.7	12.2	19.0	31.8	40.1	43.8	43.2

Power transmitted to the wheels of a vehicle is given in the table as a function of velocty. If the mass of the ar is m=2000 kg, Calculate the required time to accelrate the car from
0 m/s^2 to 6 m/s^2

Note : $\Delta t = m\int_{0s}^{6s}(v/P)dv$

PROBLEM 37

$I = \int_0^1\int_1^2\int_0^y e^{x+y+z}dzdydx$ Calculate the integral by using 3 dimensional Gauss-Legendre integration

PROBLEM 38

$I = \int_0^{\pi}\int_0^{x}\int_0^{xy}\frac{1}{y}\sin(\frac{z}{y})dzdydx$ Calculate the integral by using 3 dimensional Gauss-Legendre integration

PROBLEM 39

Blackbody radiation function F(0- λT), λ(μm) T(degree Kelvin) can be calculated as

$x = \frac{C_2}{\lambda T} = \frac{14387.69\ \mu mK}{\lambda T}$

$$F(0-\lambda T) = 1 - \frac{15}{\pi^4}\int_0^x \frac{x^3}{e^x - 1}dx$$

It can also be calculated from the series solution

$$F(0-\lambda T) = \frac{15}{\pi^4}\sum_{n=1}^{\infty}\left[\frac{e^{-nx}}{n}(x^3 + \frac{3x^2}{n} + \frac{6x}{n^2} + \frac{6}{n^3})\right]$$

By creating a program that calculates the result with using series solution and integration. Compare the results. Resulting tables are given as a reference

λ T μ mK	F(0-λT)	λ T λ T	F(0-λT)	λ T μ mK	F(0-λT)	λ T μ mK	F(0-λT)	λ T μ mK	F(0-λT)	λ T μ mK	F(0-λT)
1000	0.00032	2900	0.25056	4800	0.60754	6900	0.8022	10700	0.92709	18000	0.98081
1050	0.00056	2950	0.26191	4850	0.61428	7000	0.80808	10800	0.92872	18200	0.98137
1100	0.00091	3000	0.27323	4900	0.62089	7100	0.81374	10900	0.93031	18400	0.98191
1150	0.00142	3050	0.28453	4950	0.62737	7200	0.81918	11000	0.93185	18600	0.98243
1200	0.00213	3100	0.29578	5000	0.63373	7300	0.82443	11200	0.9348	18800	0.98293
1250	0.00308	3150	0.30697	5050	0.63996	7400	0.82949	11400	0.93758	19000	0.98341
1300	0.00432	3200	0.3181	5100	0.64608	7500	0.83437	11600	0.94021	19200	0.98387
1350	0.00587	3250	0.32915	5150	0.65207	7600	0.83907	11800	0.9427	19400	0.98431
1400	0.00779	3300	0.34011	5200	0.65795	7700	0.8436	12000	0.94505	19600	0.98474
1450	0.01011	3350	0.35097	5250	0.66371	7800	0.84797	12200	0.94728	19800	0.98516
1500	0.01285	3400	0.36173	5300	0.66937	7900	0.85219	12400	0.94939	20000	0.98555
1550	0.01605	3450	0.37238	5350	0.67491	8000	0.85625	12600	0.95139	20500	0.98649
1600	0.01972	3500	0.38291	5400	0.68034	8100	0.86018	12800	0.95329	21000	0.98735
1650	0.02388	3550	0.39332	5450	0.68566	8200	0.86397	13000	0.95509	21500	0.98814
1700	0.02854	3600	0.4036	5500	0.69089	8300	0.86763	13200	0.95681	22000	0.98886
1750	0.03369	3650	0.41375	5550	0.696	8400	0.87116	13400	0.95843	22500	0.98952
1800	0.03934	3700	0.42377	5600	0.70102	8500	0.87457	13600	0.95998	23000	0.99014
1850	0.04549	3750	0.43364	5650	0.70594	8600	0.87787	13800	0.96145	23500	0.9907
1900	0.05211	3800	0.44338	5700	0.71077	8700	0.88105	14000	0.96285	24000	0.99123
1950	0.0592	3850	0.45297	5750	0.7155	8800	0.88413	14200	0.96419	24500	0.99172
2000	0.06673	3900	0.46241	5800	0.72013	8900	0.88711	14400	0.96546	25000	0.99217
2050	0.07469	3950	0.47172	5850	0.72468	9000	0.88999	14600	0.96667	26000	0.99297
2100	0.08306	4000	0.48087	5900	0.72914	9100	0.89278	14800	0.96783	27000	0.99368
2150	0.0918	4050	0.48987	5950	0.73351	9200	0.89547	15000	0.96893	28000	0.99429
2200	0.10089	4100	0.49873	6000	0.73779	9300	0.89808	15200	0.96999	29000	0.99482
2250	0.11031	4150	0.50744	6050	0.74199	9400	0.9006	15400	0.971	30000	0.99529
2300	0.12003	4200	0.516	6100	0.74611	9500	0.90305	15600	0.97196	32000	0.99607
2350	0.13002	4250	0.52442	6150	0.75015	9600	0.90541	15800	0.97289	34000	0.99669
2400	0.14026	4300	0.53269	6200	0.75411	9700	0.9077	16000	0.97377	36000	0.99719
2450	0.15071	4350	0.54081	6250	0.758	9800	0.90992	16200	0.97461	38000	0.99759

2500	0.16136	4400	0.54878	6300	0.76181	9900	0.91207	16400	0.97542	40000	0.99792
2550	0.17217	4450	0.55662	6350	0.76554	10000	0.91416	16600	0.9762	45000	0.99851
2600	0.18312	4500	0.56431	6400	0.76921	10100	0.91618	16800	0.97694	50000	0.9989
2650	0.19419	4550	0.57186	6450	0.7728	10200	0.91814	17000	0.97765	55000	0.99917
2700	0.20536	4600	0.57927	6500	0.77632	10300	0.92004	17200	0.97834	60000	0.99935
2750	0.2166	4650	0.58654	6600	0.78317	10400	0.92188	17400	0.97899		
2800	0.22789	4700	0.59367	6700	0.78976	10500	0.92367	17600	0.97962		
2850	0.23922	4750	0.60067	6800	0.7961	10600	0.9254	17800	0.98023		

PROBLEM 40

Integral of blackbody spectral radiation equation for all wavelengths will give

$$E_b = \left[\frac{C_1}{C_2^4}\int_{x=0}^{\infty}\frac{x^3}{e^x-1}dx\right]T^4 = \sigma T^4$$ In this equation C_1=3.74177489x10^8 Wmm^4/m^2 and C_2=1.438769x10^4 mm K . σ is called Stefan_Boltzman constant. Calculate this integral by using Gauss-Leguerre integral formula. The value of the Stephan Boltsman constant is given σ =5.670x10^{-8} W/m^{2}K^4

PROBLEM 41 Bessel function $J_n(x)$ can be defined as:

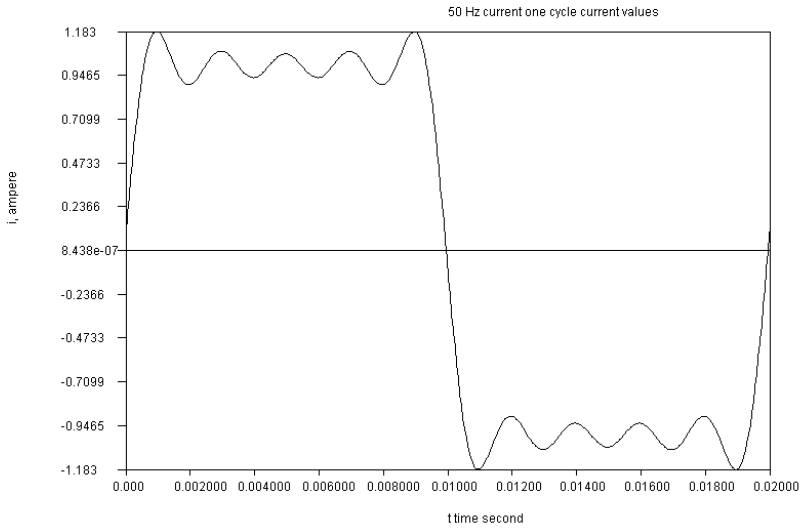
$$J_n(x) = \frac{1}{\pi}\int_0^{\pi}\cos(x\sin\theta - n\theta)d\theta$$

Write a program to define such a function.

PROBLEM 42 In an AC-DC converter the signal defined with the following function is obtained. Signal frequency is 50 hertz (period T=1.0/50=0.02 second). A true root mean square current value for an alternative current signal is defined as:

$$I_{RMS} = \sqrt{\frac{1}{T}\int_0^T i^2(t)dt}$$

Calculate True RMS current for the given signal.



PROBLEM 43 Work is defined as

Work =Force*Distance

In a real engineering problem force usually changes, Therefore Work for a variable force should be calculated as:

$$W = \int_{x0}^{xn} F(x)dx$$

In a system Force is measured as in the following table:

x(meter)	**0**	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**
F(Newton)	**0**	**9**	**12**	**15**	**18**	**23**	**16**	**13**	**11**

Calculate the work done by the system.

PROBLEM 44 Euler's constant is defined by the following equations:

$$\gamma = \int_{x=0}^{1} \left(\frac{1}{1+x^k} - e^{-x} \right) \frac{dx}{x} = \sum_{n=1}^{\infty} \left(\frac{1}{n} - \ln\left(\frac{n+1}{n} \right) \right) \text{ where } \quad k > 0$$

$\gamma =$ 0.57721566 4901532860 6065120900 8240243104 2159335939 92…

Calculate **euler's** constant by integration and compare it with the series solution. You chose different integration methods to see the accuracy of the integration formula.

PROBLEM 45 Catalan's constant is defined by the following equations:

$$K = \int_{x=0}^{1} \left(\frac{\ln(x)}{1+x^2} \right) dx$$

$$K = \frac{3}{64} \sum_{k=0}^{\infty} \frac{(-1)^k}{64^k} \left(\frac{32}{(12k+1)^2} - \frac{32}{(12k+2)^2} \frac{32}{(12k+3)^2} \frac{8}{(12k+5)^2} - \frac{16}{(12k+6)^2} - \frac{4}{(12k+7)^2} - \frac{4}{(12k+9)^2} - \frac{2}{(12k+10)^2} + \frac{1}{(12k+11)^2} \right)$$

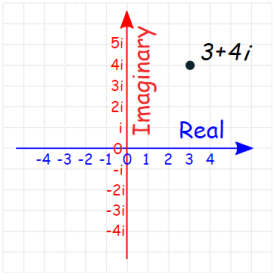
K=0.915965594177…

Calculate **Catalan's** constant by integration and compare it with the series solution. You chose different integration methods to see the accuracy of the integration formula.

4. COMPLEX NUMBERS

$z = a + bi$ where $i = \sqrt{-1}$

A complex number can now be shown as a point:



The complex number 3 + 4**i**

Arithmetic of the complex numbers

$z_1 = a_1 + b_1 i \qquad z_2 = a_2 + b_2 i$

$z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)i$

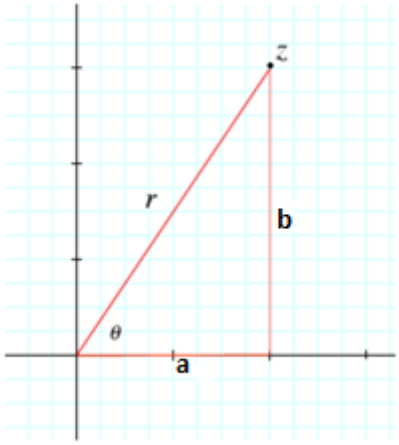
$z_1 - z_2 = (a_1 - a_2) + (b_1 - b_2)i$

Conjugate$(z) = a - bi$

$z_1 * z_2 = (a_1 * a_2 - b_1 * b_2) + (a_1 * b_2 + b_1 * a_2)i$

$$\frac{z_1}{z_2} = \frac{a_1 + b_1 i}{a_2 + b_2 i} = \frac{(a_1 + b_1 i)(a_2 - b_2 i)}{(a_2 + b_2 i)(a_2 - b_2 i)} = \frac{(a_1 * a_2 + b_1 * b_2) + (b_1 * a_2 - a_1 * b_2)i}{a_2^2 + b_2^2}$$

Complex numbers can also be expressed in polar coordinates



$z = a + bi = r(\cos(\theta) + i\sin(\theta))$

$r = \sqrt{a^2 + b^2} \quad \theta = \text{atan}\left(\frac{b}{a}\right)$

$a = r\cos(\theta) \quad b = \text{r}\sin(\theta)$

$e^{i\theta} = \cos(\theta) + i\sin(\theta)$

$e^{-i\theta} = \cos(\theta) - i\sin(\theta)$

$z = a + bi = r(\cos(\theta) + i\sin(\theta)) = re^{i\theta}$

This equation is called De Moivre formulation

Utilisation of polar coordinates makes power calculation of complex variables easier.

$z^n = (a + bi)^n = r^n(\cos(n\theta) + i\sin(n\theta)) = r^n e^{in\theta}$

$\sqrt{z} = (a + bi)^{0.5} = \sqrt{r}(\cos\left(\frac{\theta}{2}\right) + i\sin\left(\frac{\theta}{2}\right))$

Let us look at the roots of degree n

$z^{\frac{1}{n}} = (a + bi)^{\frac{1}{n}} = r^{\frac{1}{n}}(\cos\left(\frac{\theta}{n}\right) + i\sin\left(\frac{\theta}{n}\right)) = r^n e^{i\left(\frac{\theta}{n}\right)}$

But in this formula $\theta = \theta + 2k\pi$ If we substitude this into the equation

$z^{\frac{1}{n}} = (a+bi)^{\frac{1}{n}} = r^{\frac{1}{n}}(\cos\left(\frac{\theta+2k\pi}{n}\right) + i\sin\left(\frac{\theta+2k\pi}{n}\right)) = r^n e^{i\left(\frac{\theta+2k\pi}{n}\right)}$ k=0,1,2..(n-1)

So equation has n roots

Absulute value of complex numbers:

$|z| = r = \sqrt{a^2+b^2}$

Multiplication of complex numbers:

$z_1 * z_2 = \left(r_1 e^{i\theta_1}\right)\left(r_2 e^{i\theta_2}\right) = r_1 r_2 e^{i(\theta_1+\theta_2)}$

Division of complex numbers

$\frac{z_1}{z_2} = \frac{(r_1 e^{i\theta_1})}{(r_2 e^{i\theta_2})} = \frac{r_1}{r_2} e^{i(\theta_1-\theta_2)}$

Example:

$z_1 = 2+3i$

$z_2 = 4-5i$

$$z = \frac{z_1}{z_2} = \frac{2+3i}{4-5i} = \frac{2+3i}{4-5i}\frac{4+5i}{4+5i} = \frac{8+10i+12i+15i^2}{16+20i-20i-25i^2} = \frac{8+10i+12i-15}{16+20i-20i+25} = \frac{-7+22i}{41}$$

$$= \frac{-7}{41} + \frac{22}{41}i = -0.1707317 + 0.5365853i$$

In Julia complex numbers can directly be assigned and used.

```
z1=2+3im
z2=4-5im
z=z1/z2
print("z = $z")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" complex1.jl
z = -0.17073170731707318 + 0.5365853658536587im
> Terminated with exit code 0.
```

Example:

$z_1 = 2+3i$

$z_2 = z_1^2 = (2+3i\ \ )(2+3i\ \ ) = 4+6i+6i-9 = -5+12i$

Calculating by using polar coordinates:

$r = \sqrt{2^2+3^2} = \sqrt{13} = 3.605551 \quad \theta = \text{atan}\left(\frac{3}{2}\right) = 0.982$

$r^2 = 13 \quad \cos(2\,\theta) = -0.384615 \quad \sin(2\theta) = 0.923076$

$z_2 = z_1^2 = 13(-0.384616 + 0.923076i) = -5+12i$

```
z1=2+3im
z2=z1*z1
z3=z1^2
print("z2 = $z2 z3 = $z3")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" complex2.jl
z2 = -5 + 12im z3 = -5 + 12im
> Terminated with exit code 0.
```

Example:

$z_1 = 2+3i$

$z_2 = \sqrt{z_1} = z_1^{0.5}$

Calculating by using polar coordinates:

$r = \sqrt{2^2+3^2} = \sqrt{13} = 3.605551 \quad \theta = \text{atan}\left(\frac{3}{2}\right) = 0.982$

$\sqrt{r} = 13 = 1.898828 \quad \cos(0.5\,\theta) = 0.881674598 \quad \sin(0.5\theta) = 0.471857925$

$z_2 = z_1^2 = 1.898828 \quad (0.881674598 + 0.471857925i) = 1.674149 + 0.895977i$

```
z1=2+3im
z2=sqrt(z1)
z3=z1^0.5
print("z2 = $z2 z3 = $z3")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" complex3.jl
z2 = 1.6741492280355401 + 0.8959774761298381im z3 = 1.6741492280355401 + 0.895977476129838im
> Terminated with exit code 0.
```

Example:

$z_1 = 2 + 3i$

$z_2 = 4 - 5i$

$z_3 = {z_1}^{z_2}$

$\ln(z_3) = z_2 \ln(z_1)$

$z_3 = exp(z_2 \ln(z_1))$

```
z1=2+3im
z2=4-5im
z3=z1^z2
z4=exp(z2*log(z1))
print("z3 = $z3 z4 = $z4")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" complex3.jl
z3 = -18175.487698625275 - 14117.567839250678im z4 = -18175.487698625275 - 14117.56783925068im
> Terminated with exit code 0.
```

Example: Prove that following equation is correct

$$\frac{3i^{30} - i^{19}}{2i - 1} = 1 + i$$

$$\frac{3i^{30} - i^{19}}{2i - 1} = \frac{3 - i}{2i + 1} = \frac{(3 - i)}{(2i + 1)}\frac{(2i - 1)}{(2i - 1)} = \frac{(3 + 2) - i + 6i}{2^2 + 1^2} = \frac{5 + 5i}{5} = 1 + i$$

Example:

$$\sin(\theta) = \frac{1}{2i}\left(e^{i\theta} - e^{-i\theta}\right)$$

$$\cos(\theta) = \frac{1}{2}\left(e^{i\theta} + e^{-i\theta}\right)$$

$$\frac{d\sin(\theta)}{d\theta} = \frac{i}{2i}\left(e^{i\theta} + e^{-i\theta}\right) = \frac{1}{2}\left(e^{i\theta} + e^{-i\theta}\right) = \cos(\theta)$$

$$\frac{d\cos(\theta)}{d\theta} = \frac{i}{2}\left(e^{i\theta} - e^{-i\theta}\right) = \frac{i^2}{2i}\left(e^{i\theta} - e^{-i\theta}\right) = \frac{-1}{2i}\left(e^{i\theta} - e^{-i\theta}\right) = -\sin(\theta)$$

Numerical finite difference formula:

$\frac{df(x)}{dx} \cong \frac{f(x+dx)-f(x)}{dx}$ for $f(x) = \sin(x)$ and x=0, dx=0.0001

$$\frac{dsin(x)}{dx} \cong \frac{sin(x + dx) - \sin(x)}{dx} = \frac{sin(dx)}{dx} = \frac{9.9999999833333\text{e} - 5}{0.0001}$$

Sin(0.0001)=9.9999999833333e-5

$$\left.\frac{dsin(x)}{dx}\right|_{x=0} = 0.99999999833333 \cong \cos(0)$$

$$\int \sin(\theta)\, d\theta = \int \frac{1}{2i}(e^{i\theta} - e^{-i\theta})d\theta = \frac{1}{2i^2}(e^{i\theta} + e^{-i\theta}) = -\frac{1}{2}(e^{i\theta} + e^{-i\theta}) = -\cos(\theta)$$

So

$$\int_0^{\pi/2} \sin(\theta)\, d\theta = -\cos(\theta)|_0^{\pi/2} = -\cos\left(\frac{\pi}{2}\right) + \cos(0) = 1$$

If we use numerical integral (Gauss-Legendre) to calculate this:

```
#calculates legendre gauss-coefficients as coefficients of the integral
#for n terms
function gauss_legendre_coefficients(x1,x2,n)
        A = zeros(Float64,2,n)
        eps=3.0e-15
        mx=(n+1)/2
        m=floor(Int,mx)
        xm=0.5*(x2+x1)
        xl=0.5*(x2-x1)
        nmax=20
        pp=0.0
        for i=1:m
                z=cos(pi*((i-0.25)/(n+0.5)))
                for ii=1:nmax
                        p1=1.0
                        p2=0.0
                        for j=1:n
                          p3=p2
                          p2=p1
                          p1=(((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j)
                        end
                        pp=((z*p1-p2)*n/(z*z-1.0))
                        z1=z
                        z=z1-p1/pp
                        if abs(z-z1) < eps
                           break
                        end
                end
                y1=xm-xl*z
                y2=xm+xl*z
                i1=n+1-i
                A[1,i]=y1
                A[1,i1]=y2
                y3=2.0*xl/((1.0-z*z)*pp*pp)
                        A[2,i1]=y3
                 A[2,i]=y3

        end
        show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A)
        println(" ")
        return A
end
#Gauss-Legendre integration
function integral(f,x1,x2,n)
  #integral f1(x)dx
  #integral of a function by using gauss-legendre quadrature
  #between x1 and x2
  A=gauss_legendre_coefficients(x1,x2,n)
  z=0.0
  for i=1:n
    xx1=A[1,i]
    xx2=A[2,i]
    z=z+xx2*f(xx1)
  end
  return z
  end

f(x) = 1.0/2im*(exp(x*im)-exp(-x*im))
x1=0+0im
x2=pi/2.0+0im
r=integral(f,x1,x2,10)
print("integral  : $r")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" integral_gauss.jl
2X10 Matrix{Float64}:
 0.0204938  0.105979  0.251791  0.44501   0.668473  0.902324  1.12579   1.31901  1.46482   1.5503
 0.0523636  0.117379  0.17207   0.211482  0.232104  0.232104  0.211482  0.17207  0.117379  0.0523636
integral  : 0.9999999999999963 + 0.0im
> Terminated with exit code 0.
```

Same result is obtained

We will look at further utilisation of complex numbers in coming chapters.

LINEAR SYSTEM OF EQUATIONS

Lineear system of equations (or linear system) is a collection of one or more linear equations involving the same set of variables. For example

2x+3.5y=1.2
1.1x+1.5y=2
In this particular set, x and y is variables. System has two equations. It is a linear equations, becouse variables x and y linearly placed in the equation. The following set of equations is non-linear

$$2\sin(x) + 3.5e^{y} = 1.2$$

$$1.1\cos(x) + 1.5y = 2$$

In mathematics, linear system of equation is expressed as a matrix-vector format. For example above linear set can be written as

$$\begin{bmatrix} 2 & 3.5 \\ 1.1 & 1.5 \end{bmatrix} \begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} 1.2 \\ 2 \end{Bmatrix}$$

In some boooks the following formats of expressing the same system of linear equations can also be seen:

$$\left[\begin{array}{cc|c} 2 & 3.5 & 1.2 \\ 1.1 & 1.5 & 2 \end{array}\right]$$

Or simply

$$\left[\begin{array}{cc|c} 2 & 3.5 & 1.2 \\ 1.1 & 1.5 & 2 \end{array}\right]$$

In high school math, if it is only two equations and two unknown varibles variables, the following method of placement is shown to us to solve this system:
2x+3.5y=1.2
1.1x+1.5y=2
$x = (1.2 - 3.5y)/2$
$1.1\frac{(1.2-3.5y)}{2} + 1.5y = 2$
$0.66 - 1.925y + 1.5y = 2$
$-0.425y = 2 - 0.66$
y =-3.15294
Substituting to x gives
x =6.117647
In Julia system of equation solution ca directly be calculated

```
A=[2.0 3.5;1.1 1.5]
B=[1.2 2]'
X=A\B
print("x = $X")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" linearSE.jl
x = [6.117647058823525; -3.152941176470586]
```

> Terminated with exit code 0.

6.1 GAUSS ELIMINATION METHOD

2x+3.5y=1.2
1.1x+1.5y=2
Can be written as

$$\begin{bmatrix} 2 & 3.5 \\ 1.1 & 1.5 \end{bmatrix} \begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} 1.2 \\ 2 \end{Bmatrix}$$

In general

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \end{Bmatrix} = \begin{Bmatrix} b_0 \\ b_1 \end{Bmatrix}$$

2x+3.5y+3z=1.2
1.1x+1.5y-1.5z=2
x+5y-2z=3

$$\begin{bmatrix} 2 & 3.5 & 3 \\ 1.1 & 1.5 & -1.5 \\ 1 & 5 & -2 \end{bmatrix} \begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{Bmatrix} 1.2 \\ 2 \\ 3 \end{Bmatrix}$$

In general

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} b_0 \\ b_1 \\ b_2 \end{Bmatrix}$$

Upper triangle system

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ 0 & a_{11} & a_{12} \\ 0 & 0 & a_{22} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} b_0 \\ b_1 \\ b_2 \end{Bmatrix}$$

Gauss elimination method: Gauss elimnation method is based on two charecteristics of matrices (or system of linear equations as well):

1. If one equation (on eline of system is multiplied with a number the solution set remain the same
2. If one row is substracted or added to another row, solution set will still be the same.

By using this two characteristics, system of equation will be reduced to upper triabgle system. And then by starting from the last equation, sytem will be solved by backsubstitution process.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} b_0 \\ b_1 \\ b_2 \end{Bmatrix}$$

c1=a_{10}/a_{00} c2=a_{20}/a_{00}

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} - \mathrm{c1} * a_{00} & a_{11} - c1 * a_{01} & a_{12} - c1 * a_{02} \\ a_{20} - \mathrm{c2} * a_{00} & a_{21} - \mathrm{c2} * a_{01} & a_{22} - \mathrm{c2} ** a_{02} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} b_0 \\ b_1 - c1 * b_0 \\ b_2 - \mathrm{c2} * b_0 \end{Bmatrix}$$

Becomes

$$\begin{bmatrix} a'_{00} & a'_{01} & a'_{02} \\ 0 & a'_{11} & a'_{12} \\ 0 & a'_{21} & a'_{22} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} b'_0 \\ b'_1 \\ b'_2 \end{Bmatrix}$$

c3=a'_{21}/a'_{11}

$$\begin{bmatrix} a'_{00} & a'_{01} & a'_{02} \\ 0 & a'_{11} & a'_{12} \\ 0 & a'_{21} - c3 * a'_{11} & a'_{22} - c3 * a'_{11} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} b'_0 \\ b'_1 \\ b'_2 - c3 * b'_1 \end{Bmatrix}$$

Becomes

$$\begin{bmatrix} a"_{00} & a"_{01} & a"_{02} \\ 0 & a"_{11} & a"_{12} \\ 0 & 0 & a"_{22} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} b"_0 \\ b"_1 \\ b"_2 \end{Bmatrix}$$

Back substitution:

$$x_2 = b"_2 / a"_{22}$$
$$x_1 = (b"_1 - a"_{12} * x_2) / a"_{11}$$
$$x_0 = (b"_0 - a"_{01} * x_1 - a"_{02} * x_2) / a"_{00}$$

In the program following set of system of equation is taken :

$$\begin{bmatrix} 1 & 2 & 5 \\ 2 & 1 & 2 \\ 3 & 1 & 1 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

The solution set of this set of equation is 0,3 ve -1. As it is seen from the results, a small error is remained in the solution set. If the same elinination process is carried out by hand:

column 0(column indices starts from 0), row 1(raw indices starts from 0) multiplier $C_1=a_{10}/a_{00} = 2/1=2$
$a_{10} = a_{10} - C_1 * a_{00} = a_{10} - (a_{10}/a_{00}) * a_{00} = 2 - 2*1 = 0$
$a_{11} = a_{11} - C_1 * a_{01} = a_{11} - (a_{10}/a_{00}) * a_{01} = 1 - 2*2 = -3$
$a_{12} = a_{12} - C_1 * a_{02} = a_{12} - (a_{10}/a_{00}) * a_{02} = 2 - 2*5 = -8$
$b_2 = b_2 - C_1 * b_1 = b_2 - (a_{10}/a_{00}) * b_1 = 1 - 2*1 = -1$
column0 raw 1 elimination is completed
colun0 raw 2 elimination:
$C_1=a_{20}/a_{00} = 3/1=3$
$A_{20} = a_{20} - C_2 * a_{00} = a_{20} - (a_{20}/a_{00}) * a_{00} = 3 - 3*1 = 0$
$a_{21} = a_{21} - C_2 * a_{01} = a_{21} - (a_{20}/a_{00}) * a_{01} = 1 - 3*2 = -5$
$a_{22} = a_{22} - C_2 * a_{02} = a_{22} - (a_{20}/a_{00}) * a_{02} = 2 - 3*5 = -14$
$b_2 = b_2 - C_2 * b_1 = b_2 - (a_{10}/a_{00}) * b_1 = 1 - 3*1 = -1$
column 1, row 2 multiplier C
$C_3=a_{21}/a_{11} = -5/-3=(5/3)$
$a_{21} = a_{21} - C_3 * a_{11} = a_{21} - (a_{21}/a_{11}) * a_{11} = -5 - (5/3)*(-3) = 0$
$a_{22} = a_{22} - C_3 * a_{12} = a_{11} - (a_{21}/a_{11}) * a_{01} = -14 - (5/3)*(-8) = -0.6666666666666$
$b_2 = b_2 - C_3 * b_1 = b_2 - (a_{21}/a_{11}) * b_1 = -1 - (5/3)*(-1) = 0.6666666666666$
system of equation becomes:

$$\begin{bmatrix} 1 & 2 & 5 \\ 0 & -3 & -8 \\ 0 & 0 & -0.66666 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0.66666 \end{bmatrix}$$

Back-substitution process :
$X_2 = b_2/a_{22} = 0.6666666666/(-0.6666666666) = -1$
$X_1 = (b_1 - a_{12}*X_2)/a_{11} = (-1 - (-8)*(-1))/(-3) = (-9)/(-3)=3$
$X_0 = (b_0 - a_{01}*X_1 - a_{02}*X_2)/a_{00} = (1 - 2*3-5*(-1))/1=0$

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ -1 \end{bmatrix}$$

Julia direct solution:

```
A=[1.0 2 5;2 1 2;3 1 1]
B=[1.0 1 2]'
X=A\B
print("x = $X")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" linearSE.jl
x = [0.0; 3.0; -1.0]
> Terminated with exit code 0.
```

Another example for Gauss elimination with detailed steps :

$$\begin{bmatrix} 7 & 1 & 2 \\ -1 & 4 & -1 \\ 3 & 15 & 20 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 47 \\ 19 \\ 87 \end{bmatrix}$$

A B

	7	1	2	47
	-1	4	-1	19
	3	15	20	87

c					
	7	1	2	47
-0.14286	0	4.142857	-0.71429	25.71429
0.428571	0	14.57143	19.14286	66.85714

c					
	7	1	2	47
3.517241	0	4.142857	-0.71429	25.71429
	0	0	21.65517	-23.5862

X_0	**6.165605**
X_1	**6.019108**
X_2	**-1.08917**

```
A=[7.0 1 2;-1 4 -1;3 15 20]
B=[47.0 19 87]'
X=A\B
print("x = $X")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" linearSE.jl
x = [6.165605095541401; 6.019108280254777; -1.0891719745222923]
> Terminated with exit code 0.
```

As it is seen from the previous example some error existed in the solution set. It is called round off error. When gauss elimination process is used as it is defined above some error might be created by the process. In order to minimize the round off error, absolute value of the multiplier terms should be as big as possible. In the calculation process of the multiplier if divisor is zero term will go to infinity, if divisior is relatively small it will be a very big number, and this will cause an extra error in digital computers. In order to prevent this in a degree, elimination raw is interchanged with the raw with maximum absolute value for the multiplier term (in the diagonal). This process is called partial pivoting. Partial pivoting is shown in the example

$$\begin{bmatrix}1 & 2 & 5\\2 & 1 & 2\\3 & 1 & 1\end{bmatrix}\begin{bmatrix}X_0\\X_1\\X_2\end{bmatrix}=\begin{bmatrix}1\\1\\2\end{bmatrix}$$

Before starting the gauss elimination the column 0 elements are compared and raw with the largest absolute value of this column is replace with elimination raw (raw 0)

$$\begin{bmatrix}3 & 1 & 1\\2 & 1 & 2\\1 & 2 & 5\end{bmatrix}\begin{bmatrix}X_0\\X_1\\X_2\end{bmatrix}=\begin{bmatrix}2\\1\\1\end{bmatrix}$$

¶

```
function gauss_pivot(A1,b)
        n=size(A1)[1]
        carpan=0.0
        toplam=0.0
        global x=Array{Float64}(undef,n)
        global a=Array{Float64}(undef,n,n)
        for i=1:n
          for j=1:n
            a[i,j]=A1[i,j]
          end
        end
        for k=1:n
        #pivoting
           p=k
           buyuk=abs(a[k,k])
           for ii=k+1:n
             dummy=abs(a[ii,k])
             if dummy > buyuk
               buyuk=dummyp=ii
             end #if dummy
           end #for ii
           if p!=k
             for jj=k:n
               dummy=a[p,jj]
```

```
                    a[p,jj]=a[k,jj]
                    a[k,jj]=dummy
                    end #for jj

                  dummy=b[p]
                  b[p]=b[k]
                  b[k]=dummy
                 end #if p!
              # gauss elimination
   for i=k+1:n
                 carpan=a[i,k]/a[k,k]
                 a[i,k]=0.0
                 for j=k+1:n
                   a[i,j]=a[i,j]-carpan*a[k,j]
                 end #for j
                 b[i]=b[i]-carpan*b[k]
              end #for i
              end #k
                          # back substituting
               x[n]=b[n]/a[n,n]
               n1=n-1
               for i=n1:-1:1
                 toplam=0.0
                 for j=i+1:n
                   toplam=toplam+a[i,j]*x[j]
                 end #for j
                 x[i]=(b[i]-toplam)/a[i,i]
               end #for i
              return x
end #function

A=[7.0 1 2;-1 4 -1;3 15 20]
B=[47.0 19 87]
X=gauss_pivot(A,B)
print("X=$X")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss1.jl
X=[6.165605095541401, 6.019108280254777, -1.0891719745222925]
> Terminated with exit code 0.
```

6.2 GAUSS-JORDAN METHOD

Gauss Jordan elimination method is similar to Gauss elimination with exception of a normalisation process before elimination and application of the elimination for all raws (instead of only lower lows of diagonal reference row in gauss elimination) except the diagonal reference raw. The result is conversion of the right hand side into the solution without any back substitution process. Example program is given below

$$\begin{bmatrix}3 & 1 & -1\\ 1 & 4 & 1\\ 2 & 1 & 2\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}2\\ 12\\ 10\end{bmatrix}$$

$$\begin{bmatrix}1 & 0.3333 & -0.3333\\ 1 & 4 & 1\\ 2 & 1 & 2\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}0.6667\\ 12\\ 10\end{bmatrix}$$

$$\begin{bmatrix}1 & 0.3333 & -0.3333\\ 0 & 3.6667 & 1.3333\\ 2 & 1 & 2\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}0.6667\\ 11.333\\ 10\end{bmatrix}$$

$$\begin{bmatrix}1 & 0.3333 & -0.3333\\ 0 & 3.6667 & 1.3333\\ 0 & 0.3333 & 2.6667\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}0.6667\\ 11.333\\ 8.6667\end{bmatrix}$$

$$\begin{bmatrix}1 & 0.3333 & -0.3333\\ 0 & 1 & 0.3636\\ 0 & 0.3333 & 2.6667\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}0.6667\\ 3.0909\\ 8.6667\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & -0.4545\\ 0 & 1 & 0.3636\\ 0 & 0.3333 & 2.6667\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}-0.3636\\ 3.0909\\ 8.6667\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & -0.4545\\ 0 & 1 & 0.3636\\ 0 & 0 & 2.5455\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}-0.3636\\ 3.0909\\ 7.6365\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & -0.4545\\ 0 & 1 & 0.3636\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}-0.3636\\ 3.0909\\ 3.0000\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0.3636\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}1.0000\\ 3.0909\\ 3.0000\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}1.0000\\ 2.0001\\ 3.0000\end{bmatrix}$$

Solution vector

$$\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}1.0000\\ 2.0001\\ 3.0000\end{bmatrix}$$

Exact solution :

$$\begin{bmatrix}X_0\\ X_1\\ X_2\end{bmatrix}=\begin{bmatrix}1.0000\\ 2.0000\\ 3.0000\end{bmatrix}$$

```
function gauss_jordan(a,b)
n=size(b)[1]
np1=n+1
C=zeros(Float64,n,np1)
for i=1:n
  for j=1:n
    C[i,j]=a[i,j]
  end
  C[i,np1]=b[i]
end
for k=1:n
  for j=k+1:np1
    C[k,j]/=C[k,k]
  end
  C[k,k]=1.0
  for i=1:n
    if i!=k
      carpan=C[i,k]
      C[i,k]=0.0
      for j=k+1:np1
        C[i,j]-=carpan*C[k,j]
      end # for
    end # if
  end # for i
end # for k
for i=1:n
 x[i]=C[i,np1]
end # for i
return x
end # function
```

```
global a=[3.0 1.0 -1.0;1.0 4.0 1.0;2.0 1.0 2.0]
global b=[2.0 12.0 10.0]
b1=b'
x=a\b1
print("Julia x = $x")
x1=gauss_jordan(a,b1)
print("\nGauss-Jordan x = $x1")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss_jordan.jl
Julia x = [0.9999999999999999; 2.0000000000000004; 3.0]
Gauss-Jordan x = [0.9999999999999999; 2.0000000000000004; 3.0]
> Terminated with exit code 0.
```

Another example for Gauss-Jordan elimination with detailed steps :

$$\begin{bmatrix} 7 & 1 & 2 \\ -1 & 4 & -1 \\ 3 & 15 & 20 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 47 \\ 19 \\ 87 \end{bmatrix}$$

Gauss-Jordan

A				B
7	1	2	X_0	47
-1	4	-1	X_1	19
3	15	20	X_2	87
Normalisation				
1	0.142857	0.285714	X_0	6.714286
-1	4	-1	X_1	19
3	15	20	X_2	87
Elimination				
1	0.142857	0.285714	X_0	6.714286
0	4.142857	-0.71429	X_1	25.71429
0	14.57143	19.14286	X_2	66.85714
Normalisation				
1	0.142857	0.285714	X_0	6.714286
0	1	-0.17241	X_1	6.206897
0	14.57143	19.14286	X_2	66.85714
Elimination				
1	0	0.310345	X_0	5.827586
0	1	-0.17241	X_1	6.206897
0	0	21.65517	X_2	-23.5862
Normalisation				
1	0	0.310345	X_0	5.827586
0	1	-0.17241	X_1	6.206897
0	0	1	X_2	-1.08917
Elimination				
1	0	0	X_0	**6.165605**
0	1	0	X_1	**6.019108**
0	0	1	X_2	**-1.08917**

```
 function gauss_jordan(a,b)
n=size(b)[1]
np1=n+1
C=zeros(Float64,n,np1)
for i=1:n
  for j=1:n
    C[i,j]=a[i,j]
  end
  C[i,np1]=b[i]
end
for k=1:n
  for j=k+1:np1
    C[k,j]/=C[k,k]
  end
  C[k,k]=1.0
  for i=1:n
    if i!=k
      carpan=C[i,k]
      C[i,k]=0.0
      for j=k+1:np1
        C[i,j]-=carpan*C[k,j]
      end # for
    end # if
  end # for i
end # for k
for i=1:n
```

```
 x[i]=C[i,np1]
end # for i
return x
end # function

global a=[7 1 2;-1 4 -1;3 15 20]
global b=[47 19 87]
b1=b'
x=a\b1
print("Julia x = $x")
x1=gauss_jordan(a,b1)
print("\nGauss-Jordan x = $x1")
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss_jordan.jl
Julia x = [6.165605095541401; 6.019108280254777; -1.0891719745222923]
Gauss-Jordan x = [6.165605095541401; 6.019108280254777; -1.0891719745222925]
> Terminated with exit code 0.
```

LU-DECOMPOSITION METHODS

gauss(Dolittle) method

In Gauss or Gauss-Jordan elimination processes left hand matrix A and right hand vector b is eliminated together. Therefore if a new left hand side vector b1 will be considered to be used in as second set with the same A matrix, Matrix A is reprocessed. This will spend valuable calculation time for a repeated process. In order to avoid this repetation, Matrix A is decomposed into an upper triangle(U) and a lower triangle (L) matrices so that multiplication of this two matricess will give the original A matrix

For example for N=5

$$[L] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 & 0 \\ l_{41} & l_{42} & l_{43} & 1 & 0 \\ l_{51} & l_{52} & l_{53} & l_{54} & 1 \end{bmatrix}$$

$$[U] = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ 0 & u_{22} & u_{23} & u_{24} & u_{25} \\ 0 & 0 & u_{33} & u_{34} & u_{35} \\ 0 & 0 & 0 & u_{44} & u_{45} \\ 0 & 0 & 0 & 0 & u_{55} \end{bmatrix}$$

To solve system of equation with a B vector following process applies

[A]=[L][U]

[A]{X}={B}

[L][U]{X}={B}

[L]{D}={B}

[U]{X}={D}

In Gauss(Doolittle) decomposition process matrix U is the same as Gauss elimination upper triangualar matrix. L matrix is consist of the multipliers calculated in Gauss elimination process. Due to diagonal element of matrix L is 1, LU matrix can be represented in one matrix if space saving is desired. Example code is given below

```
function gaussLU(ai)
  # Gauss elimination with pivoting
  # Julia version
  n=size(ai)[1]
  x=zeros(Float64,n)
  a=zeros(Float64,n,n)
  carpan=0.0
  toplam=0.0
  for i=1:n
     for j=1:n
        a[i,j]=ai[i,j]
     end
  end
  for k=1:n-1
     for i=k+1:n
        carpan=a[i,k]/a[k,k];
        a[i,k]=carpan
        for j=k+1:n
```

```
            a[i,j]-=carpan*a[k,j]
          end # for j
       end # for i
    end # for k
    return a
end # function gaussLU

function I(n)
    # unit matrix of dimension n
    B =zeros(Float64,n,n)
    for i=1:n
        for j=1:n
          if i==j
            B[i,j]=1.0
          end # if
        end # for j
    end # for i
    return B
end # function I

function yerine_koyma(ai,bi)
    # gauss LU linear system of  equation solution
    n=size(ai)[1]
    x=zeros(Float64,n)
     a=zeros(Float64,n,n)
    b=zeros(Float64,n)
    for i=1:n
        for j=1:n
           a[i,j]=ai[i,j]
        end # for j
        b[i]=bi[i]
    end # for i

    for i=1:n
        toplam=b[i]
        for j=1:i-1
            toplam-=a[i,j]*b[j]
        end # for j
        b[i]=toplam
    end # for j
    print("\nd = $b")
    x[n]=b[n]/a[n,n]
    for i=(n-1):-1:1
        toplam=0.0
        for j=(i+1):n
          toplam+=a[i,j]*x[j]
        end #for j
        x[i]=(b[i]-toplam)/a[i,i]
    end # for i
    return x
end # funtion yerine koyma

function AXB(A,b)
    n=size(a)[1]
    c=gaussLU(a)
    s=yerine_koyma(c,b)
    return s
end # function AXB


a=[1 2 5;2 1 2;3 1 1]
b=[1 1 2]
c=gaussLU(a)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), c)
x=AXB(a,b)
print("\nx = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gaussLU.jl
3X3 Matrix{Float64}:
 1.0   2.0       5.0
 2.0  -3.0      -8.0
 3.0   1.66667  -0.666667
d = [1.0, -1.0, 0.6666666666666667]
x = [-8.881784197001252e-16, 3.000000000000003, -1.000000000000001]
> Terminated with exit code 0.
```

Gauss LU decompositionDolittle method) example in excel:

A

1	2	5
2	1	2
3	1	1

B

1
1
2

c

1	2	5
0	-3	-8
0	-5	-14

2
3

c

1	2	5
0	-3	-8
0	0	-0.6667

1.667

U

1	2	5
0	-3	-8
0	0	-0.6667

L

1		
2	1	
3	1.6667	1

d0 =
d1 =
d2 =

1
1
2

1
-1
0.6667

U

1	2	5
0	-3	-8
0	0	-0.6667

x0 =
x1 =
x2 =

1
-1
0.6667

0
3
-1

CHOLESKY SYMMETRIC MATRIX LU DECOMPOSITION METHOD

Cholesky method is similar to Dolittle(Gauss) LU method. This method is applied to symmetric matrix. Therefore, instead of LU, it decomposes to $[L][L]^T$ form.

$[A]=[L][L]^T$

For the decomposition process following steps are used

$l_{k,i} = \frac{a_{k,i} - \sum_{j=1}^{i-1} l_{i,j} l_{k,j}}{l_{i,i}}$ k=1,2,….,k-1

And

$$l_{k,k} = \sqrt{a_{k,k} - \sum_{j=0}^{n} l_{k,j} l_{k,j}}$$

Backsubstitution is similar to Dolittle method.

PROGRAM Cholesky symmetric matrix decomposition method Julia version

```
 function  choleskyLU(c,B)
# cholesky symmetric matrix LU
# symmetric matrices only
A=cholesky(c)
print("\n cholesky L\n")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A.L)
print("\n cholesky U\n")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A.U)
D=A.L\B'
x=A.U\D
return x
end # for function

using LinearAlgebra
a=[4.0 -1 1;-1 4.25 2.75;1 2.75 3.5]
B=[1.0 1 1]
```

```
x=choleskyLU(a,B)
print("\n cholesky x = $x\n")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" choleskyLU.jl

 cholesky L
3X3 LowerTriangular{Float64, Matrix{Float64}}:
  2.0   â‹…    â‹…
 -0.5  2.0   â‹…
  0.5  1.5  1.0
 cholesky U
3X3 UpperTriangular{Float64, Matrix{Float64}}:
 2.0  -0.5  0.5
  â‹…    2.0  1.5
  â‹…     â‹…   1.0
 cholesky x = [0.41015625; 0.453125; -0.1875]

> Terminated with exit code 0.
```

Let us solve the same problem step by step by hand:

$$\begin{bmatrix} 4 & -1 & 1 \\ -1 & 4.25 & 2.75 \\ 1 & 2.75 & 3.5 \end{bmatrix}$$

$$l_{11} = \sqrt{a_{11}} = \sqrt{4} = 2$$

$$l_{21} = \frac{a_{21}}{l_{11}} = \frac{-1}{2} = -0.5$$

$$l_{22} = \sqrt{a_{22} - l_{21}^2} = \sqrt{4.25 - (-0.5)^2} = 2$$

$$l_{32} = \frac{a_{32} - l_{21} - l_{31}}{l_{22}} = \frac{2.75 - (-0.5) * 0.5}{2} = 1.5$$

$$l_{33} = \sqrt{a_{33} - l_{31}^2 - l_{32}^2} = \sqrt{3.5 - (0.5)^2 - (1.5)^2} = 1$$

so cholesky decomposed L matrix :

$$[L] = \begin{bmatrix} 2 & 0 & 0 \\ -0.5 & 2 & 0 \\ 0.5 & 1.5 & 1 \end{bmatrix}$$

$$[U] = [L]^T = \begin{bmatrix} 2 & -0.5 & 0.5 \\ 0 & 2 & 1.5 \\ 0 & 0 & 1 \end{bmatrix}$$

$$[L]\{D\} = \{B\}$$

$$\begin{bmatrix} 2 & 0 & 0 \\ -0.5 & 2 & 0 \\ 0.5 & 1.5 & 1 \end{bmatrix} \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 1 \\ 1 \end{Bmatrix}$$

$d_1=1/2=0.5$

$d_2=(1-(-0.5)*0.5)/2=0.625$

$d_3=(1-0.5*0.5-1.5*0.625)/1=-0.1875$

$$[U]\{x\} = \{D\}$$

$$\begin{bmatrix} 2 & -0.5 & 0.5 \\ 0 & 2 & 1.5 \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 0.5 \\ 0.625 \\ -0.1875 \end{Bmatrix}$$

$x_3=-0.1875$

$x_2=(0.625-1.5*(-0.1875))/2=0.453125$

$x_1=(0.5-(-0.5)*0.453125-0.5*(-0.1875))/2=0.41015625$

JACOBI GAUSS -SEIDEL AND SUCCESSIVE RELAXATION METHODS

In this section, iterative solution methods will be described. Iterative methods could be important in solution of big matrices. Gauss elimination requires $n^3/3$, Gauss-Jordan method requires $n^3/2$ process. When n is big, the total number of process will climb up. In this case the total number of processes in iterative methods will be much smaller. The simplest of the iterative methods are Jacobi method. In this method, a set of x solution vector is selected and substituted into the system of equation in the following form :

$$x_i^{k+1} = \frac{1}{A_{ij}}\left(b_i - \sum_{j\neq i} A_{ij}x_j^k\right) \quad 1 \le i \le n \qquad (3.9.1)$$

When a new set of x values is found, it will be substituted in the the equation as the next iteration step. To ensure the convergence matrix should be diagonally dominant

$$\sum_{i=1}^{n}|A_{ii}| \ge \sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n}|A_{i,j}| \qquad (3.9.2)$$

Diagonal dominancy is the important condition for all iterative formulations. Unfortunately, very big matrices solved in engineering problems are in general fits to this definition.

In order to reduce number of iteration steps, every new calculated x value in the x vector set can be substited into the equation as soon as it is calculated and is not waited the complete set to be calculated as in Jacobi method. This form of iterative process is called Gauss-Seidel iteration.

$$x_i^{k+1} = \frac{1}{A_{ij}}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} A_{ij}x_j^k\right) \quad 1 \le i \le n \qquad (3.9.3)$$

Gauss-Seidel and Jacobi methods can be merged to define an intermediate method between two by defining a relaxation coefficient, α, in the equation.

$$x_i^{k+1} = (1-\alpha)x_i^{k+1} + \frac{\alpha}{A_{ij}}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} A_{ij}x_j^k\right) \quad 1 \le i \le n \quad (3.9.4)$$

When α is equal to one equation coverged to the Gauss-Seidel method.

```
function  gauss_seidel(ai,bi,xi,lambda)
  n=size(ai)[2]
  lamd=lambda
  imax=500#maximum number of iterations
  es=1.0e-20
  dummy=0.0
  ea=1.0e-1
  sum=0.0
  old=0.0
  sentinel=0.0
  i=0
  j=0
  n=size(ai)[1]
  x=zeros(Float64,n)
  b=zeros(Float64,n)
  a=zeros(Float64,n,n)

   for i=1:n
   for j=1:n
      a[i,j]=ai[i,j]
   end
      b[i]=bi[i]
      x[i]=xi[i]
   end

  for i=1:n
     dummy=a[i,i]
     for j=1:n
        a[i,j]/=dummy
     end # for j
     b[i]/=dummy
  end # for i
  for i=1:n
     sum=b[i]
     for j=1:n
         if i!=j
           sum=sum-a[i,j]*x[j]
         end # if
      end # for j
     x[i]=sum
  end #
  iter1=0
  while iter1<imax
   sentinel=1
   for i =1:n
      old=x[i]
```

```
        sum=b[i]
        for j =1:n
           if i!=j
              sum-=a[i,j]*x[j]
           end # if
        end # for j
        x[i]=lamd*sum+(1.0-lamd)*old
        if sentinel==1 && x[i]!=0.0
          ea=abs((x[i]-old)/x[i])*100.0
        end # if
        if ea<es
          return x
        end # if
     iter1+=1
     end # for i

   end # while
   if iter1>=imax
     print("Maximum number of iteration is exceed, result might not be valid iter = ",iter1)
   end # if
end #function



a=[3.0 -0.1 -0.2;0.1 7.0 -0.3;0.3 -0.2 10.0]
b=[7.85 -19.3 71.4]
x=[0,0,0]
lambda=0.5
x=gauss_seidel(a,b,x,lambda)
print("iterative relaxation  method $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss_seidel.jl
iterative relaxation  method [2.9999999999999982, -2.499999999999999, 7.0000000000000036]
> Terminated with exit code 0.
```

Jacobi and Gauss-Seidel methods in excel:

iterative method

3	-0.1	-0.2	x1	7.85
0.1	7	-0.3	x2	-19.3
0.3	-0.2	10	x3	71.4

Jacobi

Initial guess

x1	0
x2	0
x3	0

iter 1

x1	2.616667
x2	-2.75714
x3	7.14

iter 2

x1	3.000762
x2	-2.48852
x3	7.006357

iter 3

x1	3.000806
x2	-2.49974

Gauss-Seidel

Initial guess

x1	0
x2	0
x3	0

iter 1

x1	2.616667
x2	-2.79452
x3	7.00561

iter 2

x1	2.990557
x2	-2.49962
x3	7.000291

iter 3

x1	3.000032
x2	-2.49999

x3	7.000207
iter	4
x1	3.000022
x2	-2.5
x3	6.999981

x3	6.999999
iter	4
x1	3
x2	-2.5
x3	7

METHODS FOR BAND MATRICES

A three-band matrix can be shown as:

$$\begin{bmatrix} f_1 & g_1 & & & & & & \\ e_2 & f_2 & g_1 & & & & & \\ & e_3 & f_3 & g_1 & & & & \\ & & e_4 & f_4 & g_1 & & & \\ & & & e_5 & f_5 & g_1 & & \\ & & & & . & . & . & \\ & & & & & & f_{n-1} & g_{n-1} \\ & & & & & & e_n & f_n \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ . \\ x_{n-1} \\ x_n \end{Bmatrix} = \begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ . \\ r_{n-1} \\ r_n \end{Bmatrix}$$

This kind of matrix can be solved by using decomposition back and front substitution processes. Following form of basic solition process is called Thomas algorithm

Decomposition step :

```
for(int k=1;k<n;k++)
  {e[k]=e[k]/f[k-1];
   f[k]=f[k]-e[k]*g[k-1];
  }
```

Forward substitution :

```
for(int k=1;k<n;k++)
  {r[k]=r[k]-e[k]*r[k-1];
  }
```

Backward substitution:

```
x[n-1]=r[n-1]/f[n-1];
for(int k=(n-2);k>=0;k--)
  {x[k]=(r[k]-g[k]*x[k+1])/f[k];}
```

An example with detailed calculation steps:

$$\begin{bmatrix} f_1 & g_1 & 0 \\ e_2 & f_2 & g_2 \\ 0 & e_3 & f_3 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} r_1 \\ r_2 \\ r_3 \end{Bmatrix}$$

$$\begin{bmatrix} 0.8 & -0.4 & 0 \\ -0.4 & 0.8 & -0.4 \\ 0 & -0.4 & 0.8 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 41 \\ 25 \\ 105 \end{Bmatrix}$$

Decomposition:

$$e_k = \frac{e_k}{f_{k-1}}$$

$$f_k = f_k - e_k g_{k-1}$$

$$e_2 = \frac{e_2}{f_1} = \frac{-0.4}{0.8} = -0.5$$

$$f_2 = f_2 - e_2 g_1 = 0.8 - (-0.5)(-0.4) = 0.6$$

$$e_3 = \frac{e_3}{f_3} = \frac{-0.4}{0.6} = -0.666666666$$

$$f_3 = f_3 - e_3 g_3 = 0.8 - (-0.666666666)(-0.4) = 0.533333333$$

Matrix took the form of

$$\begin{bmatrix} 0.8 & -0.4 & 0 \\ -0.4 & 0.6 & -0.4 \\ 0 & -0.666666666 & 0.533333333 \end{bmatrix}$$

LU decomposed form of the matrix :

$$[A] = [L][U] = \begin{bmatrix} 1 & 0 & 0 \\ -0.5 & 1 & 0 \\ 0 & -0.66666 & 1 \end{bmatrix} \begin{bmatrix} 0.8 & -0.4 & 0 \\ 0 & 0.6 & -0.4 \\ 0 & 0 & 0.53333 \end{bmatrix}$$

Forward substitution

$$[L]\{D\} = \{B\}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ -0.5 & 1 & 0 \\ 0 & -0.66666 & 1 \end{bmatrix} \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \end{Bmatrix} = \begin{Bmatrix} 41 \\ 25 \\ 105 \end{Bmatrix}$$

d_1=41

d_2=25-(-0.5)*41=45.5

d_3=105-(-0.66666)*45.5=135.3333333

Backward substitution

$$[U]\{x\} = \{D\}$$

$$\begin{bmatrix} 0.8 & -0.4 & 0 \\ 0 & 0.6 & -0.4 \\ 0 & 0 & 0.53333 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 41 \\ 45.5 \\ 135.33333 \end{Bmatrix}$$

X_3=135.33333/0.53333=253.75

X_2=(45.5-(-0.4)* 253.75)/0.6=245

X_1=(41-(-0.4)*245)/0.8=173.75

Julia version of the Thomas algorithm is given below:

Program Thomas algorithm for band matrices (python version)

```
function thomas(a,b)
 n=size(a)[1]
 global f=zeros(Float64,n)
 global e=zeros(Float64,n)
 global g=zeros(Float64,n)
 global x=zeros(Float64,n)
 global r=zeros(Float64,n)
 i=0
 j=0
 k=0
   for i=1:n
   r[i]=b[i]
 end
 for i=1:n
   f[i]=a[i,i]
 end

 for i=1:n-1
   g[i]=a[i,i+1]
 end

 for i=1:n-1
   e[i+1]=a[i+1,i]
 end

 for k=2:n
   e[k]=e[k]/f[k-1]
   f[k]=f[k]-e[k]*g[k-1]
 end #for

 for k=2:n
   r[k]=r[k]-e[k]*r[k-1]
 end #for

 x[n]=r[n]/f[n]
 n1=n-1
 for k=n1:-1:1
   x[k]=(r[k]-g[k]*x[k+1])/f[k]
 end #for
 return x
 end #function
```

```
function thomas(f,e,g,r)
  n=size(f)[1]
  global x=Array{Float64}(undef,n)
for k=2:n
    e[k]=e[k]/f[k-1]
    f[k]=f[k]-e[k]*g[k-1]
  end #for

  for k=2:n
    r[k]=r[k]-e[k]*r[k-1]
  end #for

  x[n]=r[n]/f[n]
  n1=n-1
  for k=n1:-1:1
    x[k]=(r[k]-g[k]*x[k+1])/f[k]
  end #for
  return x
  end #function



a=[0.8 -0.4 0;-0.4 0.8 -0.4;0 -0.4 0.8]
b=[41 25 105]
x=thomas(a,b)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), x)
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" thomas.jl
3-element Vector{Float64}:
 173.75
 244.99999999999997
 253.74999999999997
> Terminated with exit code 0.
```

NORMS

Norms are an important tool to evaluate matrices: Several form of norms are defined. General definitions of norms are as follows:

Infinity norm : Maximum value of the sum of the columns

$$\|A\|_\infty = \sum_{i=1}^{n} max\left(\sum_{j=1}^{n}|a_{ij}|\right)$$

Norm: Maximum value of the sum of the raws

$$\|A\| = \sum_{j=1}^{n} max\left(\sum_{i=1}^{n}|a_{ij}|\right)$$

Euclidian norm: square root of the sum of the square of the elements

$$\|A\|_e = \|A\|_2 = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}^2}$$

m norm: mth root of the sum of the mth power of the elements

$$\|A\|_m = \sqrt[m]{\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}^m}$$

Of course the same norm definitions can be apply to the vectors as well, please be reminded that vectors are one dimensional matrices.

$$\|x\|_\infty = \sum_{i=1}^{n} max(|x_i|)$$

$$\|x\| = \sum_{i=1}^{n} max(|x_i|)$$

$$\|x\|_e = \|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

$$\|x\|_m = \sqrt[m]{\sum_{i=1}^{n} x_i^m}$$

Basic norm definitions are given in the norm class.

Matrix norm definitions : class norm

```
 # **************norm methods definition*****************
function norm(v)
# vector norm
total=0
n=length(v)
for i=1:n
  total+=v[i]*v[i]
end
return sqrt(total)
end

function norm(v,p)
   # p vector norm
  if(p==0.0)
    p=1.0
  end
  total=0.0
  n=length(v)
  for i=1:n
    x=abs(v[i])
    total+=x^p
  end
  y=total^(1.0/p)
  return y
  end


function norminf(v)
    #infinite vector norm
    max=0.0
    n=length(v)
    for i=1:n
      x=abs(v[i])
      if(x>max)
        max=x
      end
    end
    return max
    end


function norminf(a)
         #infinite matrix norm
         max = 0.0
         total=0.0
         n=size(a)[1]
         for i=1:n
           total=0.0
           for j=1:n
              total+=abs(a[i,j])
           end
         x=total
         if x>max
           max=x
         end
         end
       return max
```

```
    end

function norm(a)
    #matrix norm
        max = 0.0
        total=0.0
        n=size(a)[1]
        for i=1:n
          total=0
          for j=1:n
            total+=abs(a[i,j])
          end
          x=total
          if(x>max)
            max=x
          end
          end
        return max
    end

function normE(a)
        #Euclidian matrix norm
        total=0.0
        n=size(a)[1]
        for i=1:n
          for j=1:n
            total+=a[i,j]*a[i,j]
          end
        end
        return sqrt(total)
end

a=[1.0 2 3;4 5 6;7 8 9]
n1=norm(a)
println("norm = $n1")
n2=norminf(a)
println("norminf = $n2")
n3=normE(a)
println("normE = $n3")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" norm.jl
norm = 24.0
norminf = 24.0
normE = 16.881943016134134

> Terminated with exit code 0.
```

Norm is existed im LinearAlgebra package of Julia

```
using LinearAlgebra
a=[1.0 2 3;4 5 6;7 8 9]
n1=norm(a)
print("normE = $n1")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" norm1.jl
normE = 16.881943016134134
> Terminated with exit code 0.
```

UNIT MATRIX, TRANPOSE MATRIX, DETERMINANT AND MATRIX INVERSION

Unit matrix is a matrix with all diagonal elements are one and all the remaining elements are zero

$$a_{ii} = 1 \quad a_{ij, i \neq j} = 0$$

As an example for n=4

$$[I] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
using LinearAlgebra
n=4
I1=Matrix{Float64}(1I,n,n)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), I1)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" I.jl
4X4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0
> Terminated with exit code 0.
```

We can also create our own function for unit matrix

```
function unit_matrix(n)
  I=zeros(Float64,n,n)
  for i=1:n
    I[i,i]=1.0
  end
  return I
end

n=4
I1=unit_matrix(n)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), I1)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" unit_matrix.jl
4X4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0
> Terminated with exit code 0.
```

Transpose matrix is the matrix with rows and colums are interchanged. Mathematically, it can be shown with a T sign at the top of the matrix. Transpose of a matrix can be taken as in the following method:

```
 function Transpose(A)
    n=size(A)[1]
    m=size(A)[2]
    B=zeros(Float64,m,n)
    for i=1:n
        for j=1:m
            B[j,i]=A[i,j]
        end # for j
    end # for i
    return B
end # function

A=[1.0 2 3;4 5 6;7 8 9]
B=Transpose(A)
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A)
print("\n")
```

```
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), B)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" Transpose.jl
3X3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0
3X3 Matrix{Float64}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
> Terminated with exit code 0.
```

In Julia Transpose of a matrix will simply be calculated as transpose(A)=A’

```
A=[1.0 2 3;4 5 6;7 8 9]
B=A'
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A)
print("\n")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), B)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" Transpose1.jl
3X3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0
3X3 adjoint(::Matrix{Float64}) with eltype Float64:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
> Terminated with exit code 0.
```

Inverse matrix is the matrix when multiplied with the original matrix will give the unit matrix

$$[A][A]^{-1} = [I]$$

```
 function inverse_matrix(A)
  n=size(A)[1]
  Im=Matrix{Float64}(1I,n,n)
  B=zeros(Float64,n,n)
  I1=zeros(Float64,n)
  B=A\Im
  return B
end # function



using LinearAlgebra
A=[1.0 2 5;2 1 2;3 1 1]
print("A \n")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A)
B=inverse_matrix(A)
print("\n inverse A B\n")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), B)
D1=A*B
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), D1)
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" inverse_matrix1.jl
A
3X3 Matrix{Float64}:
 1.0  2.0  5.0
 2.0  1.0  2.0
 3.0  1.0  1.0
 inverse A B
3X3 Matrix{Float64}:
```

```
 -0.5   1.5  -0.5
  2.0  -7.0   4.0
 -0.5   2.5  -1.53X3 Matrix{Float64}:
  1.0         -1.77636e-15  8.88178e-16
  0.0          1.0          0.0
 -1.11022e-16   4.44089e-16  1.0
> Terminated with exit code 0.
```

inv(A) is directly available in LinearAlgebra package

```
using LinearAlgebra
A=[1.0 2 5;2 1 2;3 1 1]
print("A \n")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A)
B=inv(A)
print("\n inverse A B\n")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), B)
D1=A*B
print("\n A*inv(A)\n")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), D1)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" inverse_matrix2.jl
A
3X3 Matrix{Float64}:
 1.0  2.0  5.0
 2.0  1.0  2.0
 3.0  1.0  1.0
 inverse A B
3X3 Matrix{Float64}:
 -0.5   1.5  -0.5
  2.0  -7.0   4.0
 -0.5   2.5  -1.5
 A*inv(A)
3X3 Matrix{Float64}:
  1.0         -1.77636e-15  8.88178e-16
  0.0          1.0          0.0
 -1.11022e-16   4.44089e-16  1.0
> Terminated with exit code 0.
```

Determinant of a matrix can be defined as follows: if a matrix A is given as

$$[A] = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

Determinat of this matrix determined as:

$$D[A] = \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix} = a_{00} \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} - a_{01} \begin{vmatrix} a_{10} & a_{12} \\ a_{20} & a_{22} \end{vmatrix} + a_{02} \begin{vmatrix} 0 & a_{11} \\ a_{20} & a_{21} \end{vmatrix}$$

$$D[A] = a_{00}(a_{11}a_{22} - a_{12}a_{21}) - a_{01}(a_{10}a_{22} - a_{12}a_{20}) + a_{02}(a_{10}a_{22} - a_{12}a_{20})$$

If matrix is converted to a upper triangular form Determinant will simply be the product of diagonals

$$[A] = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ 0 & a_{11} & a_{12} \\ 0 & 0 & a_{22} \end{bmatrix}$$

$$D[A] = a_{00}a_{11}a_{11}$$

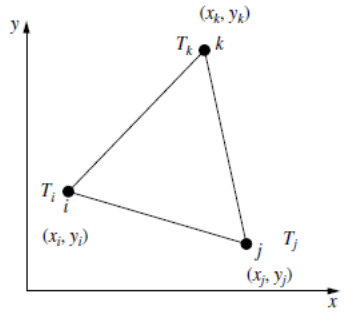
$$D[A] = \prod_{i=1}^{n} a_{ii}$$

```
function determinant(A1)
        n=size(A1)[1]
        carpan=0.0
        toplam=0.0
        global a=Array{Float64}(undef,n,n)
        for i=1:n
```

```
                for j=1:n
                  a[i,j]=A1[i,j]
                end
              end
              for k=1:n
              #pivoting
                 p=k
                 buyuk=abs(a[k,k])
                 for ii=k+1:n
                   dummy=abs(a[ii,k])
                   if dummy > buyuk
                     buyuk=dummyp=ii
                   end #if dummy
                 end #for ii
                 if p!=k
                   for jj=k:n
                     dummy=a[p,jj]
                     a[p,jj]=a[k,jj]
                     a[k,jj]=dummy
                     end #for jj
                 end #if p!
              end # for i
              det=1.0
              for i=1:n
    det*=a[i,i]
              end #for i
              return det
end #function

A=[1.0 0 0;1 1 0;1 0 1]
X=determinant(A)
print("Determinant = $X")
```

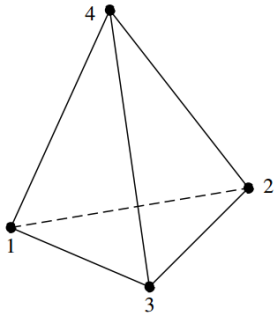
```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss1.jl
Determinant = 1.0
> Terminated with exit code 0.
```

Additional note: Area of a triangle



$$2A = det\begin{bmatrix}1 & x_i & y_i\\ 1 & x_j & y_j\\ 1 & x_k & y_k\end{bmatrix} = (x_i y_j - x_j y_i) + (x_k y_j - x_i y_k) + (x_j y_k - x_k y_j)$$

Volume of a tetrahedron



$$6V = det\begin{bmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{bmatrix}$$

```
function determinant(A1)
            n=size(A1)[1]
            carpan=0.0
            toplam=0.0
            global a=Array{Float64}(undef,n,n)
            for i=1:n
              for j=1:n
                a[i,j]=A1[i,j]
              end
            end
            for k=1:n
            #pivoting
               p=k
               buyuk=abs(a[k,k])
               for ii=k+1:n
                 dummy=abs(a[ii,k])
                 if dummy > buyuk
                    buyuk=dummyp=ii
                 end #if dummy
               end #for ii
               if p!=k
                 for jj=k:n
                   dummy=a[p,jj]
                   a[p,jj]=a[k,jj]
                   a[k,jj]=dummy
                   end #for jj
               end #if p!
            end # for i
            det=1.0
            for i=1:n
    det*=a[i,i]
            end #for i
            return det
end #function

A=[1.0 0 0;1 1 0;1 0 1]
Area=0.5*determinant(A)
print("Area = $Area")
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" gauss1.jl
Area = 0.5
> Terminated with exit code 0.
```

By using LinearAlgebra package mehod LowerTriangular, you can alco create determinant

```
function determinant(A1)
   A=LowerTriangular(A1)
   n=size(A1)[1]
            det=1.0
            for i=1:n
     det*=A[i,i]
            end #for i
            return det
end #function

using LinearAlgebra
A=[1.0 0 0;1 1 0;1 0 1]
Area=0.5*determinant(A)
print("Area = $Area")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" determinant.jl
Area = 0.5
```

```
> Terminated with exit code 0.
```

Determinant is also directly suppoted in julia by using LinearAlgebra package

```
using LinearAlgebra
A=[1.0 0 0;1 1 0;1 0 1]
Area=0.5*det(A)
print("Area = $Area")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" determinant2.jl
Area = 0.5
> Terminated with exit code 0.
```

Multiplication of matrices:

Vectoral multiplication can be applied by using *

A3=A1*A2

Scalar multiplication can be applied by using .*

A3=A1.*A2

```
A1=[1.1 2.2 3.3;4.4 5.5 6.6;7.7 8.8 9.9]
A2=[1.7 2.7 3.7;4.7 5.7 6.7;7.7 8.7 9.7]
A3=A1*A2
A4=A1.*A2
println("vector multiplication of two matrices")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A3)
println("\nscalar multiplication of two matrices")
show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A4)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" matrix_multiply.jl
vector multiplication of two matrices
3X3 Matrix{Float64}:
  37.62   44.22   50.82
  84.15  100.65  117.15
 130.68  157.08  183.48
scalar multiplication of two matrices
3X3 Matrix{Float64}:
  1.87   5.94  12.21
 20.68  31.35  44.22
 59.29  76.56  96.03
> Terminated with exit code 0.
```

CRAMER'S RULE

Another method to solve linear system of equations is Cramer's rule. The rule based on solutions of determinants. This solution method is not preferred much as a computer solution method, but it is practical for hand solutions of small systems. If a system of equation given as:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{Bmatrix}$$

then solution can be given as:

$$D = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} \quad x_1 = \begin{vmatrix} b_1 & a_{12} & \dots & a_{1n} \\ b_2 & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ b_n & a_{n2} & \dots & a_{nn} \end{vmatrix} \frac{1}{D} \quad x_1 = \begin{vmatrix} a_{11} & b_1 & \dots & a_{1n} \\ a_{21} & b_2 & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & b_n & \dots & a_{nn} \end{vmatrix} \frac{1}{D} \quad \dots.$$

Due to fact that solution of the determinat requires solution of the system of eqautions in computer environment. Cramer's rule is not very practical numerical method to solve system of equations, but could be an easy solution method for hand calculations.

PROBLEMS

PROBLEM 1

Solve system of equation by using Gauss elimination method.

$$\begin{bmatrix} 7 & 1 & 2 \\ -1 & 4 & -1 \\ 3 & 15 & 20 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 47 \\ 19 \\ 87 \end{Bmatrix}$$

PROBLEM 2

Solve system of equation by using

a) By using Gauss elimination
b) By using Gauss-Jordan elimination

$$\begin{bmatrix} 1.1348 & 3.8226 & 1.1651 & 3.4017 \\ 0.5301 & 1.7875 & 2.5330 & 1.5435 \\ 3.4129 & 4.9317 & 8.7643 & 1.3142 \\ 1.2371 & 4.9998 & 10.6721 & 0.0147 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 9.5342 \\ 6.3941 \\ 18.4231 \\ 16.9237 \end{Bmatrix}$$

Exact solution: $\begin{Bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{Bmatrix}$

PROBLEM 3

Solve system of equation by using

a) By using Gauss elimination
b) By using Gauss-Jordan elimination

$$\begin{bmatrix} 4 & 3 & -1 \\ 7 & -2 & 3 \\ 5 & -18 & 13 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 9 \\ 3 \end{Bmatrix}$$

PROBLEM 4

Solve system of equation by using

a) By using Gauss elimination
b) By using Gauss-Jordan elimination
c) LU decomposition (Dolittle)
d) LU Craut Decomposition

$$\begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/4 & 1/4 & 1/6 & 1/7 & 1/8 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} = \begin{bmatrix} 137/60 \\ 87/60 \\ 459/60 \\ 743/840 \\ 1879/2520 \end{bmatrix}$$

PROBLEM 5

Solve system of equation by using

a) By using Gauss elimination
b) Gauss-Seidel iteration
c) Conjugate gradient method

$$\begin{bmatrix} 8 & 2 & 3 \\ 3 & -9 & 2 \\ 2 & 3 & 6 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 30 \\ 1 \\ 31 \end{Bmatrix}$$

PROBLEM 6 Solve system of equation by using

a) By using Gauss elimination
b) Gauss-Seidel with relaxation
c) Conjugate gradient method

$$\begin{bmatrix} 3 & -5 & 47 & 20 \\ 11 & 16 & 17 & 10 \\ 56 & 22 & 11 & -18 \\ 17 & 66 & -12 & 7 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 18 \\ 26 \\ 34 \\ 82 \end{Bmatrix}$$

PROBLEM 7 Solve system of equation by using Thomas algorithm (3 band matrix method)

$$\begin{bmatrix} -2 & 1 & & & & & & & \\ 1 & -2 & 1 & & & & & & \\ & 1 & -2 & 1 & & & & & \\ & & & \ldots & & & & & \\ & & & 1 & -2 & 1 & & & \\ & & & & 1 & -2 & 1 & & \\ & & & & & 1 & -2 & & \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \\ X_7 \\ X_8 \\ X_9 \end{bmatrix} = \begin{bmatrix} -0.5 \\ -1.5 \\ -1.5 \\ \ldots \\ -1.5 \\ -1.5 \\ 0.5 \end{bmatrix}$$

PROBLEM 8 Solve system of equation by using Thomass algorithm (3 band matrix method)

$$\begin{bmatrix} -4 & 1 & & & & & & & \\ 1 & -4 & 1 & & & & & & \\ & 1 & -4 & 1 & & & & & \\ & & & \ldots & & & & & \\ & & & 1 & -4 & 1 & & & \\ & & & & 1 & -4 & 1 & & \\ & & & & & 1 & -4 & & \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \\ X_7 \\ X_8 \\ X_9 \end{bmatrix} = \begin{bmatrix} -27 \\ -15 \\ -15 \\ \ldots \\ -15 \\ -15 \\ -15 \end{bmatrix}$$

PROBLEM 9

Solve the following system of equation

$$\begin{bmatrix} 3 & -5 & 6 & 4 & -2 & -3 & 8 \\ 1 & 1 & -9 & 15 & 1 & -9 & 2 \\ 2 & -1 & 7 & 5 & -1 & 6 & 11 \\ -1 & 1 & 3 & 2 & 7 & -1 & -2 \\ 4 & 3 & 1 & -7 & 2 & 1 & 1 \\ 2 & 9 & -8 & 11 & -1 & -4 & -1 \\ 7 & 2 & -1 & 2 & 7 & -1 & 9 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \end{bmatrix} = \begin{bmatrix} 47 \\ 17 \\ 24 \\ 8 \\ 13 \\ -10 \\ 34 \end{bmatrix}$$

PROBLEM 10

Solve the following system of equation

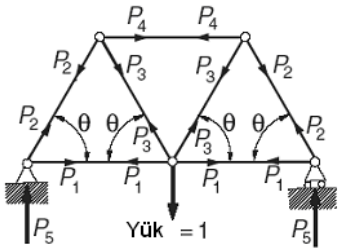
$$\begin{bmatrix} 1 & -1 & 2 & 5 & -7 & -8 \\ 3 & -9 & 1 & -1 & 8 & 1 \\ -1 & 1 & 9 & -9 & 2 & 3 \\ 1 & 7 & 2 & -3 & -1 & 4 \\ 7 & 1 & -2 & 4 & 1 & -1 \\ 2 & 3 & -9 & 12 & -2 & 7 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{bmatrix} = \begin{bmatrix} -12 \\ 8 \\ 22 \\ 41 \\ 15 \\ 50 \end{bmatrix}$$

PROBLEM 11

Solve the following system of equation

$$\begin{bmatrix} 10 & -2 & -1 & 2 & 3 & 1 & -4 & 7 \\ 5 & 11 & 3 & 10 & -3 & 3 & 3 & -4 \\ 7 & 12 & 1 & 5 & 3 & -12 & 2 & 3 \\ 8 & 7 & -2 & 1 & 3 & 2 & 2 & 4 \\ 2 & -15 & -1 & 1 & 4 & -1 & 8 & 3 \\ 4 & 2 & 9 & 1 & 12 & -1 & 4 & 1 \\ -1 & 4 & -7 & -1 & 1 & 1 & -1 & -3 \\ -1 & 3 & 4 & 1 & 3 & -4 & 7 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} 0 \\ 12 \\ -5 \\ 3 \\ -25 \\ -26 \\ 9 \\ -7 \end{bmatrix}$$

PROBLEM 12

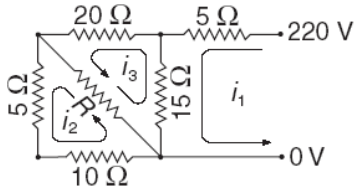


Load Matrix of the figure can be given with the following system of eqautions

$$\begin{bmatrix} c & 1 & 0 & 0 & 0 \\ 0 & s & 0 & 0 & 1 \\ 0 & 0 & 2s & 0 & 0 \\ 0 & -c & c & 1 & 0 \\ 0 & s & s & 0 & 0 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Where c=cos(θ),s=sin(θ) . For θ =60º find the load distribution profile

PROBLEM 13



For the given electrical circuit current density-Voltage eqautions can be given as:

$$5i_1 + 15(i_1 - i_3) = 220\ V$$
$$5i_1 + 10i_2 + R(i_2 - i_3) = 0$$
$$20i_3 + R(i_3 - i_2) + 15(i_3 - i_1) = 0$$

Calculate current densities for R=10 Ω.

PROBLEM 14

$-x_1 + 3x_2 + 5x_3 + x_4 = 8$
$x_1 + 9x_2 + 3x_3 + 4x_4 = 10$
$x_2 + x_4 = 2$
$3x_1 + x_2 + x_3 - x_4 = -4$

Solve the given system of equations. Determine the best method to solve it.

PROBLEM 15 Solve the following system of equation by using Doolittle LU method

$$\begin{bmatrix} 1 & 3 & 5 & 7 \\ 11 & 16 & 17 & 10 \\ 0 & 0 & 2 & 5 \\ -2 & -6 & -3 & 1 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{Bmatrix}$$

PROBLEM 16 Solve system of equation by using

a) By using Gauss elimination
b) By using Gauss-Jordan elimination
c) LU decomposition (Dolittle)
d) LU Craut Decomposition

$3x_1 + 5x_2 + 4x_3 + 2x_4 = 11$
$6x_1 + 14x_2 + 11x_3 + 6x_4 = 26$
$9x_1 + 11x_2 + 16x_3 + 5x_4 = 68$
$3x_1 + 13x_2 + 17x_3 + 12x_4 = 43$

PROBLEM 17 Solve system of equation by using

a) By using Gauss elimination
b) By using Gauss-Jordan elimination
c) LU decomposition (Dolittle)
d) LU Craut Decomposition
e) Gauss Seidel iteration

f) Conjugate-Gradian method

$$x_1 + 3x_2 + x_3 + 5x_4 = 4$$
$$2x_1 + x_2 + 3x_4 = 5$$
$$4x_1 + 2x_2 + 2x_3 + x_4 = 11$$
$$-3x_1 + x_2 + 3x_3 + 2x_4 = 3$$

PROBLEM 18 Solve system of equation by using

a) By using Gauss elimination
b) By using Gauss-Jordan elimination
c) LU decomposition (Dolittle)
d) LU Craut Decomposition

$x_1 + x_2 - x_3 = -3$
$6x_1 + 2x_2 + 2x_3 = 2$
$-3x_1 + 4x_2 + x_3 = 1$

PROBLEM 19 Find infinite norm two norm and m=3 norm of the matrix

$$\begin{bmatrix} 7 & 1 & 2 \\ -1 & 4 & -1 \\ 3 & 15 & 20 \end{bmatrix}$$

$$\begin{bmatrix} 7 & 1 & 2 \\ -1 & 4 & -1 \\ 3 & 15 & 20 \end{bmatrix}$$

PROBLEM 20 Find the inverse matrix

$$\begin{bmatrix} 1.1348 & 3.8326 & 1.1651 & 3.4017 \\ 0.5301 & 1.7875 & 2.5330 & 1.5435 \\ 3.4129 & 4.9317 & 8.7643 & 1.3142 \\ 1.2371 & 4.9998 & 10.6721 & 0.0147 \end{bmatrix}$$

PROBLEM 21 Find the inverse matrix

$$\begin{bmatrix} 4 & 3 & -1 \\ 7 & -2 & 3 \\ 5 & -18 & 13 \end{bmatrix}$$

PROBLEM 22

Find two norm and inverse matrix

$$\begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/4 & 1/4 & 1/6 & 1/7 & 1/8 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \end{bmatrix}$$

PROBLEM 23

Find two norm, infinite norm and and inverse matrix

$$\begin{bmatrix} 8 & 2 & 3 \\ 1 & -9 & 2 \\ 2 & 3 & 6 \end{bmatrix}$$

PROBLEM 24

Find two norm, infinite norm and and inverse matrix

$$\begin{bmatrix} 3 & -5 & 47 & 20 \\ 11 & 16 & 17 & 10 \\ 56 & 22 & 11 & -18 \\ 17 & 66 & -12 & 7 \end{bmatrix}$$

PROBLEM 25

Find two norm, infinite norm and and inverse matrix

$$\begin{bmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & & \ldots & & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 & 1 \\ & & & & & 1 & -2 \end{bmatrix}$$

FINDING THE ROOTS OF EQUATION NON-LINEAR FUNCTIONS WITH ONE VARIABLE

Root finding is one of the most used numerical methods. For the one independent variable functions, it requires to find x value where the function is zero f(x)=0. If multidimensional (has more than one independent value) functions are involved, it requires to find $x_1,x_2,x_3,\ldots,x_n$ roots of n equation $f_i(x_1,x_2,x_3,\ldots,x_n)=0$, i=1..n

Some of the root finding methods will be investigated. As a starting point, we will start a simpler form of a function y=f(x). In this equation you have one root value..

Julia language has its own root finding package to be utilised for this purpose

```
import Pkg;Pkg.add("Roots")
```

```
using Roots
f(x)=x^2-2
y=find_zero(f,0.6)
print("y = $y")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" roots2.jl
y = 1.4142135623730951
> Terminated with exit code 0.
```

BISECTION METHOD

In the bisection method root of the functions is searched in a given region $a \leq x \leq b$ If a single root is existed in the given region equation $f(a)f(b) < 0$ (2.1.1) will be satisfied. If this condition is not satisfied, it can be assumed that there is no root in the given region. Of course the reason will be existance of double root in the given region. If the root existed region is divide to two equal parts as

$$p = \frac{a+b}{2}$$

and function is evaluated in the middle. If

$f(a) * f(p) < 0$

root should be located in a-p else it is located in p-b region. Rarely it is also possible that

$f(a) * f(p) = 0$

can happen, that will indicate the root is located at point p. The new root region is used in further iterations. In order to use iterative process an exceptible criteria for error limit can be assumed

$\left|\frac{(b-a)}{(b+a)}\right| < \epsilon$ or $|f(p)| < \epsilon$

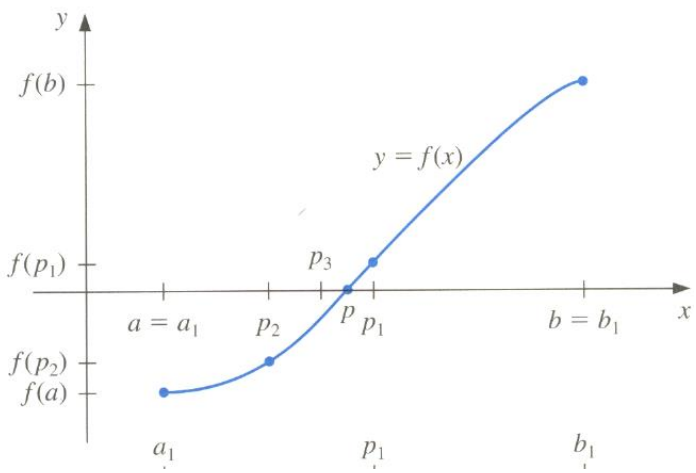


Figure 2.1 Convergence of bisection Method

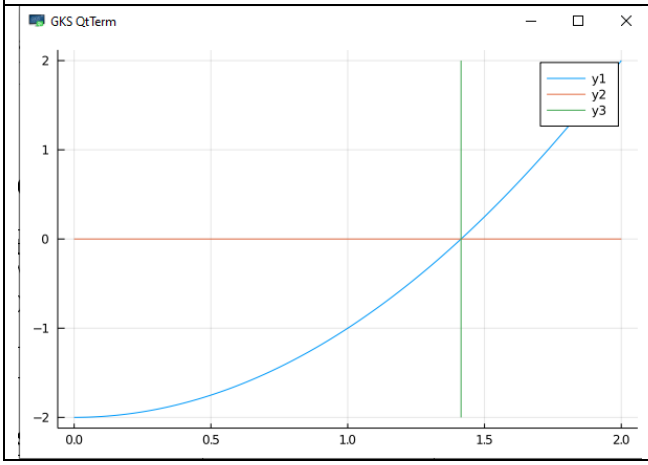
An example code is created for the Bisection method. In order to define function as a general definition abstract class f_x defined in the following code. There are also defined methods for the derivatives of to function in this class. The details of the numerical derivatives will be defined in related chapter.

Bisection method bisection.jl

```
 function bisection(f,xl,xu)
test=1.0
xr=0.0
fx=0.0
fxr=0.0
fxu=0.0
xold=0.0
maxit=100
iter=0
es=0.00001
global ea=1.5*es
s=""
fxl= f(xl)
fxu= f(xu)
while ea>es && iter<maxit
   xold=xr
   xr=(xl+xu)/2.0
   fxr= f(xr)
   iter=iter+1
   ea=abs(fxr)
   test= fxl*fxr
   if(test==0.0)
     ea=0.0
   elseif(test<0.0)
     xu=xr
     fxu=fxr
   elseif(test>0)
     xl=xr
     fxl=fxr
   else
      ea=0.0
   end
end # while
if(iter>=maxit)
  print("Maximum number of iteration is exceeded \nresult might not be valid")
end
   return xr
end

using Plots
f(x)=x*x-2.0
```

```
a=0.0
b=2.0
x=bisection(f,a,b)
print("x = $x")
f1(x)=0.0
x1=[x,x]
y1=[-2.0,2.0]
p=plot([f, f1],0.0:0.01:2.0)
plot!(p,x1,y1)
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" bisection.jl
x = 1.414215087890625
> Terminated with exit code 0.
```



Program Bisection method in excel environment $f(x) = x^2 - 2$

xl	xu	xr	fxl	fxu	fxr	fxr*fxl
1	2	1.5	-1	2	0.25	-0.25
1	1.5	1.25	-1	0.25	-0.4375	0.4375
1.25	1.5	1.375	-0.4375	0.25	-0.10938	0.047852
1.375	1.5	1.4375	-0.10938	0.25	0.066406	-0.00726
1.375	1.4375	1.40625	-0.10938	0.066406	-0.02246	0.002457
1.40625	1.4375	1.421875	-0.02246	0.066406	0.021729	-0.00049
1.40625	1.421875	1.4140625	-0.02246	0.021729	-0.00043	9.6E-06
1.4140625	1.421875	1.41796875	-0.00043	0.021729	0.010635	-4.5E-06
1.4140625	1.41796875	1.416015625	-0.00043	0.010635	0.0051	-2.2E-06
1.4140625	1.416015625	1.415039063	-0.00043	0.0051	0.002336	-1E-06
1.4140625	1.415039063	1.414550781	-0.00043	0.002336	0.000954	-4.1E-07
1.4140625	1.414550781	1.414306641	-0.00043	0.000954	0.000263	-1.1E-07
1.4140625	1.414306641	1.41418457	-0.00043	0.000263	-8.2E-05	3.5E-08
1.41418457	1.414306641	1.414245605	-8.2E-05	0.000263	9.06E-05	-7.4E-09

PROBLEM: Following equation is given:

$$f(x) = -xJ_1(x) - BiJ_0(x) = 0$$

Where $J_0(x)$ and $J_1(x)$ first and second order Bessel functions

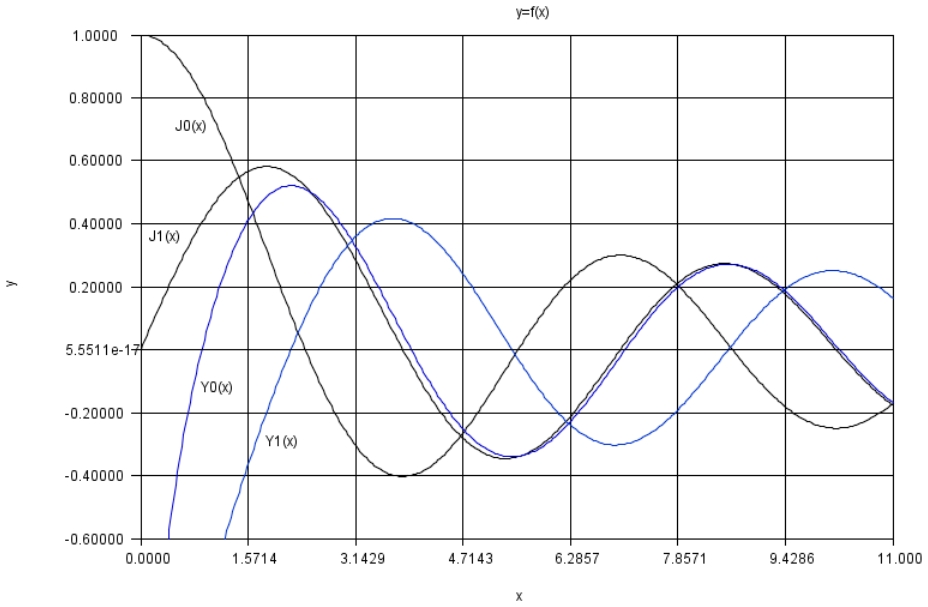
Bessel function $J_\nu(z)$ can be defined by using the following series:

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} (-1)^k \frac{\left(\frac{z}{2}\right)^{2\kappa}}{k!\,\Gamma(\nu+k+1)}$$

Where gamma function defined as

$$\Gamma(z) = \int_{z=0}^{\infty} t^{z-1}e^{-t}dt = \frac{e^{-\gamma z}}{z}\prod_{n=1}^{\infty}\left(1+\frac{z}{n}\right)^{-1} e^{z/n}$$

Where γ=0.57721566490153286... is the Euler-Mascheroni constant. The Bessel function $Y_\nu(z)$ is given by:



For Bi=0.1 Find the multiple roots of the given equation

```
function bisection(f,a,b)
   b1=1.1*b
   r=(a+b)/2.0
   fr=f(r)
   fa=f(a)
   eps=1.0e-6
   nmax=100
   i=1
   while abs(fr)>eps && i<nmax
      if fa*fr<0
         b=r
      else
         a=r
         fa=fr
      end
      r=(a+b)/2.0
      fr=f(r)
      i=i+1
   end
   if i>=nmax
     r=bisection(f,a,b1)
   end
   return r
end


function enlarge(f,x0,dx)
   # enlarge region untill a root is existed
   x1=x0
   x2=x1+dx
   NTRY=200
   FACTOR=1.001
   j=0
   if x1 == x2
    print("input variables are wrong")
   end
   f1=f(x1)
   f2=f(x2)
   for j=1:NTRY+1
      if f1*f2 < 0.0
         break
      else
         x2=x2+dx
         f2=f(x2)
      end
   end
   return x2
end
```

```
using SpecialFunctions
Bi=0.1
f(x)=x*besselj1(x)-Bi*besselj0(x)
global x0=0.0
global dx=0.001
for n=1:20
   x1=enlarge(f,x0,dx)
   #print("x1 = $x1")
   ksi=bisection(f,x0,x1)
   global x0=ksi+dx
   ff=f(ksi)
   println("Bi = $Bi n = $n x = $ksi f = $ff")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" bisection1.jl
Bi = 0.1 n = 1 x = 0.44168142409737154 f = -1.5856229017285894e-7
Bi = 0.1 n = 2 x = 3.85771028940425 f = -5.972990407415724e-7
Bi = 0.1 n = 3 x = 7.029825369592585 f = 2.8627071236089763e-7
Bi = 0.1 n = 4 x = 10.183292774584878 f = -5.330633232672466e-7
Bi = 0.1 n = 5 x = 13.331195186460318 f = 2.2249596180398679e-7
Bi = 0.1 n = 6 x = 16.476700564557774 f = -9.5064303932621e-7
Bi = 0.1 n = 7 x = 19.620955665797776 f = -1.9191641553728545e-7
Bi = 0.1 n = 8 x = 22.7644775014558 f = 3.185612201840604e-7
Bi = 0.1 n = 9 x = 25.907532057621463 f = -7.29784770465286e-7
Bi = 0.1 n = 10 x = 29.050270873930266 f = 6.881174789922367e-7
Bi = 0.1 n = 11 x = 32.19278651300452 f = 7.975333959336223e-7
Bi = 0.1 n = 12 x = 35.335137694139746 f = 2.7911729158849874e-8
Bi = 0.1 n = 13 x = 38.47736544533629 f = 9.803157654426908e-7
Bi = 0.1 n = 14 x = 41.61949703048528 f = -1.6925118888518054e-7
Bi = 0.1 n = 15 x = 44.76155293117235 f = -9.534498347195386e-7
Bi = 0.1 n = 16 x = 47.90354857359345 f = -6.401504185019752e-7
Bi = 0.1 n = 17 x = 51.04549425511897 f = -1.7453986554671896e-9
Bi = 0.1 n = 18 x = 54.18739917737421 f = -3.4937047258161436e-7
Bi = 0.1 n = 19 x = 57.32926974134999 f = -1.8724345816900634e-7
Bi = 0.1 n = 20 x = 60.47111154822296 f = 2.581444502602892e-9

> Terminated with exit code 0.
```

The same program is given with user defined J bessel function:

```
# Bessel function J
function J(v,x)
JJ=0
a=1.0
b=0.0
fact=1
if  x<=20
            x1=0.0
            x2=0.0
            x3=0.0
            x4=0.0
            x5=1
            lna=0.0
            x1=v*log(0.5*x);
            lna1=log(0.25*x*x);
            for k=1:100
                x2=lna
                x3=loggamma(k)
                x4=loggamma(v+k)
                        b=x1+x2-x3-x4;
                        b=x5*exp(b);
                        lna+=lna1;
                        x5*=-1;
                        JJ+=b;
            end
elseif v==0 && x==0
  JJ=1.0
else #x>30
  JJ=sqrt(2.0/pi/x)*cos(x-0.5*v*pi-0.25*pi)
end
return JJ;
end # function
```

```
function bisection(f,a,b)
    b1=1.1*b
    r=(a+b)/2.0
    fr=f(r)
    fa=f(a)
    eps=1.0e-6
    nmax=100
    i=1
    while abs(fr)>eps && i<nmax
        if fa*fr<0
            b=r
        else
            a=r
            fa=fr
        end
        r=(a+b)/2.0
        fr=f(r)
        i=i+1
    end
    if i>=nmax
      r=bisection(f,a,b1)
    end
    return r
end


function enlarge(f,x0,dx)
    # enlarge region untill a root is existed
    x1=x0
    x2=x1+dx
    NTRY=200
    FACTOR=1.001
    j=0
    if x1 == x2
      print("input variables are wrong")
    end
    f1=f(x1)
    f2=f(x2)
    for j=1:NTRY+1
        if f1*f2 < 0.0
            break
        else
            x2=x2+dx
            f2=f(x2)
        end
    end
    return x2
end


using SpecialFunctions
Bi=0.1
f(x)=x*J(1,x)-Bi*J(0,x)
global x0=0.0
global dx=0.001
for n=1:20
    x1=enlarge(f,x0,dx)
    #print("x1 = $x1")
    ksi=bisection(f,x0,x1)
    global x0=ksi+dx
    ff=f(ksi)
    println("Bi = $Bi n = $n x = $ksi f = $ff")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" bisection3.jl
Bi = 0.1 n = 1 x = 0.44168142409737154 f = -1.5856229017285894e-7
Bi = 0.1 n = 2 x = 3.85771028940425 f = -5.972990452032811e-7
Bi = 0.1 n = 3 x = 7.029825369592585 f = 2.862704970921226e-7
Bi = 0.1 n = 4 x = 10.183292774584878 f = -5.330583249078813e-7
Bi = 0.1 n = 5 x = 13.331195186460318 f = 2.22557060280959e-7
Bi = 0.1 n = 6 x = 16.476700564557774 f = -9.280136813206996e-7
Bi = 0.1 n = 7 x = 19.620955665797776 f = 5.308665928481504e-7
Bi = 0.1 n = 8 x = 22.780936100092504 f = 9.348724384315721e-7
```

```
Bi = 0.1 n = 9 x = 25.921997117309406 f = 6.854885873860961e-8
Bi = 0.1 n = 10 x = 29.0631727331714 f = 3.4206956301199853e-7
Bi = 0.1 n = 11 x = 32.204429688792445 f = -7.399280716711976e-7
Bi = 0.1 n = 12 x = 35.34574648462637 f = 2.630303112139548e-7
Bi = 0.1 n = 13 x = 38.4871082430234 f = -1.496648539909412e-7
Bi = 0.1 n = 14 x = 41.628504916998224 f = -3.162006329150008e-7
Bi = 0.1 n = 15 x = 44.76992896512047 f = 6.961250890134829e-8
Bi = 0.1 n = 16 x = 47.91137518640969 f = -1.943174872936354e-7
Bi = 0.1 n = 17 x = 51.052839435342094 f = 3.540394542290992e-7
Bi = 0.1 n = 18 x = 54.19431840018331 f = 4.93537519649298e-7
Bi = 0.1 n = 19 x = 57.33580996009236 f = -4.643536838065676e-7
Bi = 0.1 n = 20 x = 60.4773121109086 f = -1.1275709955842206e-7

> Terminated with exit code 0.
```

FALSE POSITION METHOD (REGULA FALSI)

In bisection method search region was devided by two in each time. It is caused a relatively long interpolation period. In order to improve interpolation time, two points can be joined with a line and root of the line can be taken as the next iteration point .

$$p = b - f(b)\frac{(a-b)}{f(a)-f(b)} = \frac{bf(a)-af(b)}{f(a)-f(b)} \quad (2.2.1)$$

Region selection process is same as bisection process. If f(a)*f(p) < 0 root should be located in a-p else it is located in p-b region. The new root region is used in further iterations. In order to use iterative process an exceptible criteria for error limit can be assumed

$$\left|\frac{(b-a)}{(b+a)}\right| < \epsilon \quad \text{or} \quad |f(p)| < \epsilon \quad (2.2.2)$$

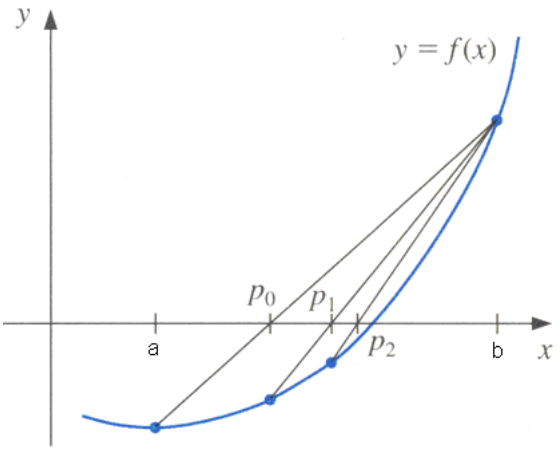


Figure 2.2 Convergence of the False position Method

Program 2.2-1 False position (Regula falsi) NA2

```
 function false_position(f,xl,xu)
test=1.0
xr=0.0
fx=0.0
fxr=0.0
fxu=0.0
xold=0.0
maxit=100
iter=0
es=0.00001
global ea=1.5*es
s=""
fxl= f(xl)
fxu= f(xu)
while ea>es && iter<maxit
   xold=xr
   xr=xu-fxu*(xl-xu)/(fxl-fxu)
```

```
    fxr= f(xr)
    iter=iter+1
    ea=abs(fxr)
    test= fxl*fxr
    if(test==0.0)
      ea=0.0
    elseif(test<0.0)
      xu=xr
      fxu=fxr
    elseif(test>0)
      xl=xr
      fxl=fxr
    else
       ea=0.0
    end
end # while
if(iter>=maxit)
  print("Maximum number of iteration is exceeded \nresult might not be valid")
end
   return xr
end

using Plots
f(x)=x*x-2.0
a=0.0
b=2.0
x=false_position(f,a,b)
print("x = $x")
f1(x)=0.0
x1=[x,x]
y1=[-2.0,2.0]
p=plot([f, f1],0.0:0.01:2.0)
plot!(p,x1,y1)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" bisection.jl
x = 1.41421143847487
> Terminated with exit code 0.
```

Plot of the iterations through this example is shown in the plot.

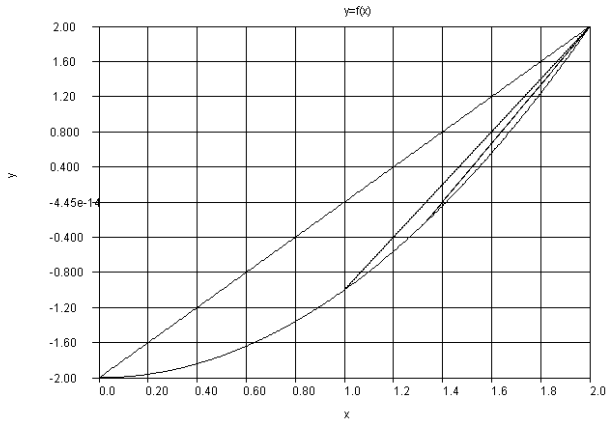


Figure Convergence of the False position Method example program function $f(x)=x^2-2$

Steps of False position method is shown in detail in the following open Office calc (excel) program for function **$f(x)=x^2-2$**

Program 2.2-2 False position (Regula falsi) open Office calc (excel)

xl	xu	xr	fxl	fxu	fxr	fxr*fxl
1	2	1.333333333	-1	2	-0.22222	0.222222
1.333333333	2	1.4	-0.22222	2	-0.04	0.008889
1.4	2	1.411764706	-0.04	2	-0.00692	0.000277
1.411764706	2	1.413793103	-0.00692	2	-0.00119	8.23E-06
1.413793103	2	1.414141414	-0.00119	2	-0.0002	2.43E-07
1.414141414	2	1.414201183	-0.0002	2	-3.5E-05	7.14E-09

1.414201183	2	1.414211438	-3.5E-05	2	-6E-06	2.1E-10
1.414211438	2	1.414213198	-6E-06	2	-1E-06	6.19E-12
1.414213198	2	1.4142135	-1E-06	2	-1.8E-07	1.82E-13
1.4142135	2	1.414213552	-1.8E-07	2	-3E-08	5.37E-15
1.414213552	2	1.414213561	-3E-08	2	-5.2E-09	1.58E-16
1.414213561	2	1.414213562	-5.2E-09	2	-8.9E-10	4.65E-18
1.414213562	2	1.414213562	-8.9E-10	2	-1.5E-10	1.37E-19
1.414213562	2	1.414213562	-1.5E-10	2	-2.6E-11	4.03E-21
1.414213562	2	1.414213562	-2.6E-11	2	-4.5E-12	1.19E-22
1.414213562	2	1.414213562	-4.5E-12	2	-7.7E-13	3.49E-24
1.414213562	2	1.414213562	-7.7E-13	2	-1.3E-13	1.03E-25
1.414213562	2	1.414213562	-1.3E-13	2	-2.3E-14	3.04E-27
1.414213562	2	1.414213562	-2.3E-14	2	-4.2E-15	9.65E-29

As it is seen from the plot and table for iteration steps 2.0 is always stays and root search goes on between this two points. Even if solution is close to first point iterations did not cut from the left side of the search area. It may cause a long number of iteration steps.
In order to overcome the difficulty of one sided cut of the region a bisection step can be put in with some regular intervals. For example once in every 4 false position steps. This is called **modified false position** method. The next example looks the root of $f(x)=x^2-2$ with modified false position method.

Program Modified False position (Regula falsi)

```
function false_position_bisection(f,xl,xu)
test=1.0
xr=0.0
fx=0.0
fxr=0.0
fxu=0.0
xold=0.0
maxit=100
iter=0
es=0.00001
global ea=1.5*es
s=""
fxl= f(xl)
fxu= f(xu)
while ea>es && iter<maxit
   xold=xr
   if iter%4!=0
     xr=xu-fxu*(xl-xu)/(fxl-fxu)
   else
    xr=(xl+xu)/2.0
   end
   fxr= f(xr)
   iter=iter+1
   ea=abs(fxr)
   test= fxl*fxr
   if(test==0.0)
     ea=0.0
   elseif(test<0.0)
     xu=xr
     fxu=fxr
   elseif(test>0)
     xl=xr
     fxl=fxr
   else
      ea=0.0
   end
end # while
if(iter>=maxit)
  print("Maximum number of iteration is exceeded \nresult might not be valid")
end
   return xr
end

using Plots
f(x)=x*x-2.0
a=0.0
```

```
b=2.0
x=false_position_bisection(f,a,b)
print("x = $x")
f1(x)=0.0
x1=[x,x]
y1=[-2.0,2.0]
p=plot([f, f1],0.0:0.01:2.0)
plot!(p,x1,y1)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" bisection.jl
x = 1.4142115601832654
> Terminated with exit code 0.
```

FIXED POSITION METHOD WITH AITKEN EXTRAPOLATION (STEPHENSEN METHOD)

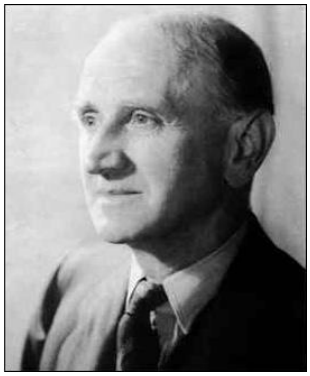


Figure Alexander Craig Aitken

Fixed position method is relatively simple. When a function f(x) is given to find the root, the function f(x) converted with algebraic manupulations and converted to f(x)=g(x) – x form. Since this equation can be written in as g(x)=x, starting from an initial guess, iteration continued till x and g(x) gives the same x value. Even though the relative simplicity, this method is not used much due to high failure possibility. The equation can diverge from the root sometimes. In order to overcome this problem aitken extrapolation process will be combined with fixed point , iteration. Method is called Stephensen method.
Aitken extrapolation process is a error reduction method. For a very big k number

$$\frac{x_{k+2}-a}{x_{k+1}-a} \approx \frac{x_{k+1}-a}{x_k-a} \quad k \gg 1$$

Relation can be valid. If the value of fourth point desired to be calculated from this equation

$$x = \frac{x_k x_{k+2} - x_{k+1}^2}{2x_k + 2x_{k+1} + 2x_{k+2}}$$

When 3 points are previously known, a third one can be evaluated

$$x_k = x_{k+1} - x_k$$

$$\Delta^2 x_k = x_{k+2} - 2x_{k+1} + x_k$$

$$x_e = x_{k+2} - \frac{(\Delta x_k)^2}{\Delta^2 x_k}$$

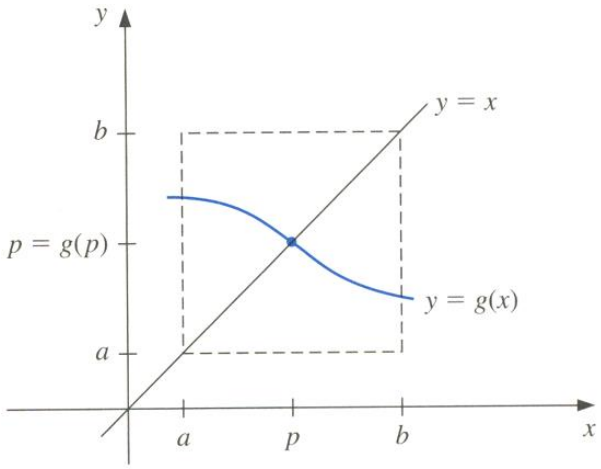


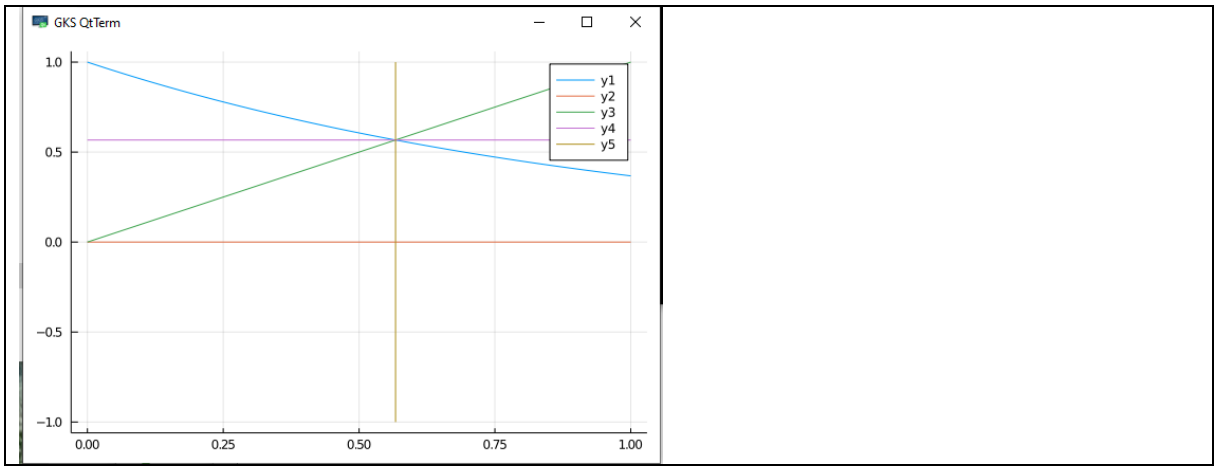
Figure Convergence of the Fixed iteration Method

Stephensen method evaluates equation by using an aitken extrapolation step after two step of fixed point iteration.

Program: Fixed point iteration (Stephensen method) :

```
 # Stephensen method x=g(x)
function  stephensen(f,x0)
maxit=100
iter=0
es=0.000001
ea=1.1*es
x=0.0
x1=0.5
x2=1.0
while iter<maxit && ea>es
            x1=f(x0)
            x2=f(x1)
            x=x0-(x1-x0)*(x1-x0)/(x2-2*x1+x0)
            ea=abs(x-x0)
            x0=x
            iter=iter+1
            if iter>=maxit
              print("Maximum number of iteration is exceeded \nresult might not be valid")
            end
end
return x
end

using Plots
f(x)=exp(-x)
x0=1.0
x2=stephensen(f,x0)
print("x = $x2")
f1(x)=0.0
f2(x)=x
f3(x)=x2
x1=[x2,x2]
y1=[-1.0,1.0]
p=plot([f, f1,f2,f3],0.0:0.01:1.0)
plot!(p,x1,y1)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" stephensen.jl
x = 0.5671432904097838
> Terminated with exit code 0.
```



Stephensen Method is simple enough to calculate in excel format: The example problem is :

$$f(x) = x - \sqrt{\frac{5}{x+1}} = x - g(x) \qquad g(x) = \sqrt{\frac{5}{x+1}}$$

Program 2.3-2 : Fixed point iteration (Stephensen method) with excel:

Fixediteration

f(x)=x-(5/(x+1)^0.5

Aitken		dxk	d2xk	xe=xk-(dxk)^2/d2xk	g(x)=(5/(x+1))^0.5
	x				
0	0.5				1.825741858
0	1.825741858	1.325741858			1.330205563
0	1.330205563	-0.495536296	-1.821278154	1.465031877	1.464832223
1	1.465031877	0.134826314	0.63036261	1.436194292	1.424209021
0	1.424209021	-0.040822856	-0.17564917	1.433696715	1.436150556
0	1.436150556	0.011941535	0.052764391	1.433447971	1.432626367
0	1.432626367	-0.003524189	-0.015465724	1.433429427	1.433663727
1	1.433429427	0.00080306	0.004327249	1.433280394	1.433427145
0	1.433427145	-2.28264E-06	-0.000805343	1.433427151	1.433427817
0	1.433427817	6.72303E-07	2.95494E-06	1.433427664	1.433427619
0	1.433427619	-1.98012E-07	-8.70315E-07	1.433427664	1.433427677
1	1.433427664	4.50514E-08	2.43064E-07	1.433427656	1.433427664
0	1.433427664	-7.32747E-15	-4.50514E-08	1.433427664	1.433427664
0	1.433427664	1.9984E-15	9.32587E-15	1.433427664	1.433427664
0	1.433427664	0	-2.44249E-15	1.433427664	1.433427664

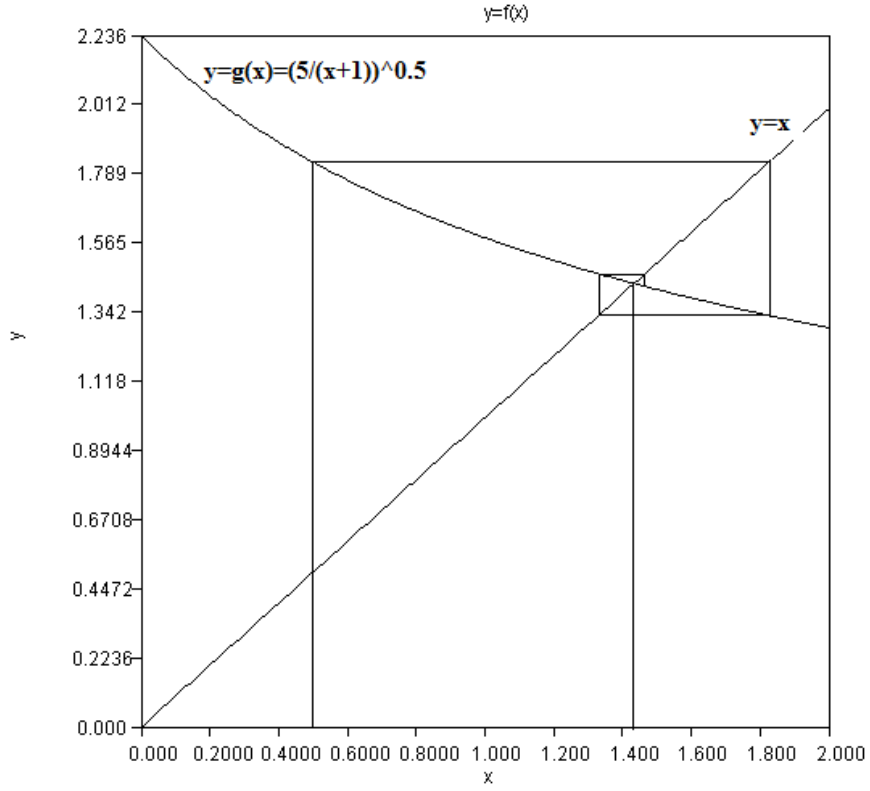


Figure Convergence of the Fixed position Method

NEWTON-RAPHSON METHOD

Isaac Newton

Joseph Raphson

Taylor Formula can be written as

$$f(x_{n+1}) = f(x_n) + \frac{f'(x_n)}{1!}(x_{n+1} - x_n) + \frac{f''(x_n)}{2!}(x_{n+1} - x_n)^2 + \frac{f^{(3)}(x_n)}{3!}(x_{n+1} - x_n)^3 + \cdots \quad (2.4.1)$$

We would like to find $f(x_{n+1}) = 0$. If We substitude it into the Taylor Formula, and also cancel out terms above the first degree , The equation becomes

$$0 = f(x_n) + f'(x_n)(x_{n+1} - x_n) \quad (2.4.2)$$

If x_{n+1} taken to left hand side Newton-Raphson Formula is obtained

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (2.4.3)$$

We can find the root iteratively starting from a first guess. Newton Formula aproached to the root quickly if the first guess is close to the actual root. But if the first gues is far away from the root, it might take longer or it

might fail to reach to the root. Another difficulty in Newton-Raphson Formula is the requirement of knowing the actual derivative of the function

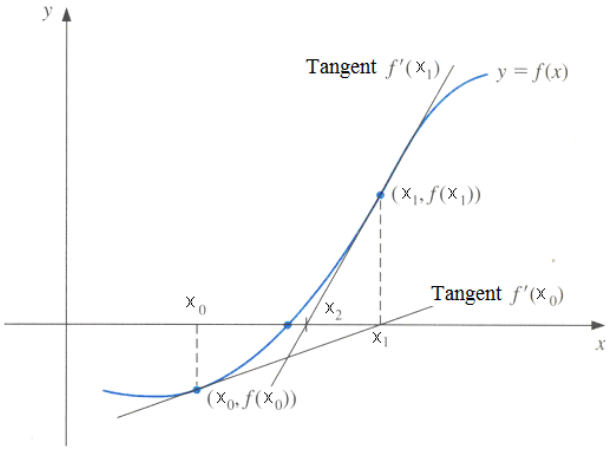
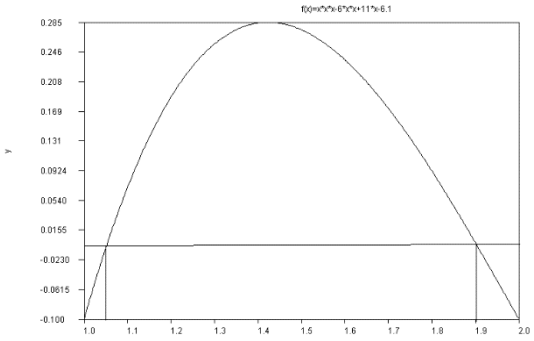


Figure Convergence of the Newton-Raphson Method

PROGRAM Newton-Raphson method (java code)

```
 function newton(f,df,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(x)
  x-=fx/dfx
  println("i = $i x = $x fx = $fx dfx = $dfx")
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

f(x)=x*x-2.0
df(x)=2.0*x
x=1.0
x=newton(f,df,x)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton.jl
i = 0 x = 1.5 fx = -1.0 dfx = 2.0
i = 1 x = 1.4166666666666667 fx = 0.25 dfx = 3.0
i = 2 x = 1.4142156862745099 fx = 0.006944444444444642 dfx = 2.8333333333333335
i = 3 x = 1.4142135623746899 fx = 6.007304882871267e-6 dfx = 2.8284313725490198
i = 4 x = 1.4142135623730951 fx = 4.510614104447086e-12 dfx = 2.8284271247493797
x = 1.4142135623730951
> Terminated with exit code 0.
```



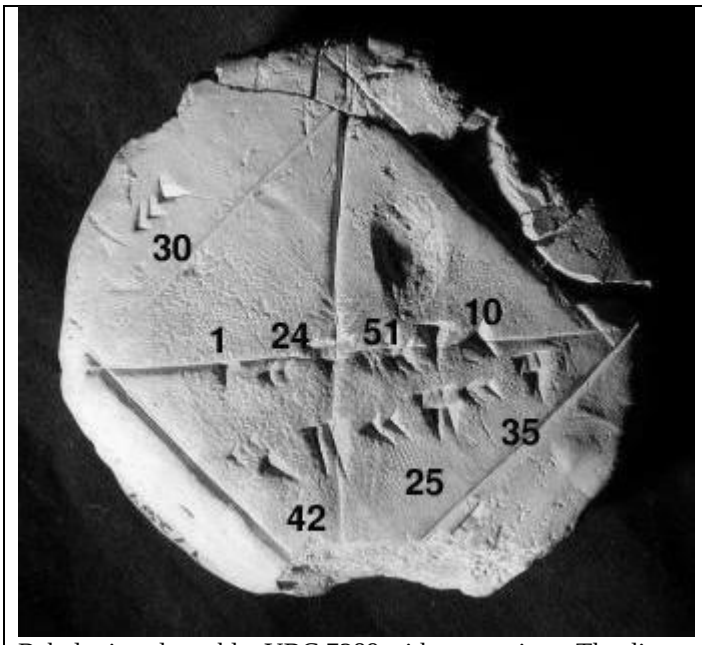
In order to see step calculations of Newton-Raphson method, root of $f(x)=x^2-2$ is calculated in the spreadsheet environment as well

PROGRAM Newton-Raphson method in excel

Newton-Raphson root finding $f(x)=x^2-a$

a	2	
x	f(x)=x*x-a	fx/dx=2*x
1.00000000000000000	-1	2
1.50000000000000000	0.25	3
1.41666666666667000	0.006944444	2.833333333
1.41421568627451000	6.0073E-06	2.828431373
1.41421356237469000	4.51061E-12	2.828427125
1.41421356237310000	0	2.828427125
1.41421356237310000	0	2.828427125
1.41421356237310000	0	2.828427125

As it is seen from the results, convergence is quite fast for this problem.



Babylonian clay tablet YBC 7289 with annotations. The diagonal displays an approximation of the square root of 2 in four sexagesimal(60 based) figures, 1 24 51 10, which is good to about six decimal digits.
$1 + 24/60 + 51/60^2 + 10/60^3 = 1.41421296...$

Babylonian algorithm is actually newton-raphson method.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

$$f(x_n) = x^2 - 2$$

$$f'(x_n) = 2x$$

$$x = x - \frac{x^2 - 2}{2x} = \frac{x}{2} + \frac{1}{x}$$

Babylonian algorithm in julia format:

```
function sqrt(x)
 y =x*x-2
 e = 1.0e-10
 while abs(y) > e
   x =x /2.0+ 1.0/x
   y =x*x-2.0
   println("x = $x y = $y")
 end #for
 return x
end #function
```

```
x= sqrt(2.0)
print("x= $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" sqrt1.jl
x = 1.5 y = 0.25
x = 1.4166666666666665 y = 0.006944444444444198
x = 1.4142156862745097 y = 6.007304882427178e-6
x = 1.4142135623746899 y = 4.510614104447086e-12
x= 1.4142135623746899
> Terminated with exit code 0.
```

SECANT METHOD

One of the difficulty of the Newton-Raphson method is the requirement to find the derivative. If difference equation (Numerical derivative) is used instead of derivative (tangent to the function), a secant to the function is drawn to the function, so the method is called secant method. If derivative is approximated with a first degree difference equation

$$f'(x_n) \cong \frac{f(x_n)-f(x_{n-1})}{(x_n-x_{n-1})}$$

Newton –Raphson Formula is converted to

$$x_{n+1} = x_n - \frac{f(x_n)(x_n-x_{n-1})}{f(x_n)-f(x_{n-1})}$$

Two first estimation is needed to start iteration by using this formula $f(x_0)$ and $f(x_{-1})$

If we would like to establish a one estimation secant Formula, the previous difference equation can be written for a small Δx as (central difference Formula)

$$f'(x_n) \cong \frac{f(x_n+\Delta x)-f(x_n-\Delta x)}{2\Delta x}$$

Second equation becomes:

$$x_{n+1} = x_n - \frac{2f(x_n)\Delta x}{f(x_n+\Delta x)-f(x_n-\Delta x)}$$ In order to minimize error Δx should be relatively small, to have a further approximation a second or higher derivative (difference) formulas can also be used

$$f'(x_n) \cong \frac{-f(x_n+2\Delta x)+8f(x_n+\Delta x)-8f(x_n-\Delta x)+f(x_n-2\Delta x)}{12\Delta x}$$

$$f'(x_n) \cong \frac{f(x_n+3\Delta x)-9f(x_n+2\Delta x)+45f(x_n+\Delta x)-45f(x_n-\Delta x)+9f(x_n-2\Delta x)-f(x_n-3\Delta x)}{60\Delta x}$$

$$f'(x_n) \cong \frac{-3f(x_n+4\Delta x)+32f(x_n+3\Delta x)-168f(x_n+2\Delta x)+672f(x_n+\Delta x)-672f(x_n-\Delta x)+168f(x_n-2\Delta x)-32f(x_n-3\Delta x)+3f(x_n-4\Delta x)}{840\Delta x}$$

(2.5.7)

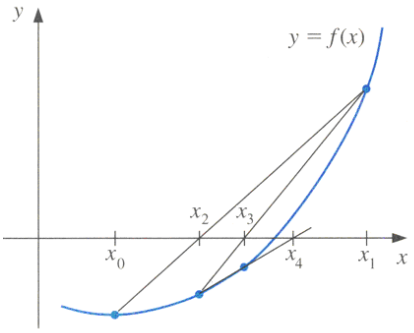


Figure 2.6 : Second method approximation with linear difference

PROBLEM : Secant method with two initial root estimation (juliacode)

```
 function secant(f,df,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
```

```
  dfx=df(f,x)
  x-=fx/dfx
  println("i = $i x = $x fx = $fx dfx = $dfx")
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function df(f,x)
 dx=0.001
 ff=(-3.0f(x+3dx)+32.0f(x+2dx)-168.0f(x+2dx)+672.0f(x+dx)-672.0f(x-dx)+168.0f(x-2dx)-32f(x-3dx)+3f(x-4dx))/(840.0dx)
 return ff
end

f(x)=x*x-2.0
x=1.0
x=secant(f,df,x)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton.jl
i = 0 x = 1.5179235458526263 fx = -1.0 dfx = 1.9307869047617066
i = 1 x = 1.4141688176043983 fx = 0.3040918910538104 dfx = 2.930872608776783
i = 2 x = 1.4142151659128144 fx = -0.0001265553153781962 dfx = 2.7305271692308506
i = 3 x = 1.4142135049334956 fx = 4.535497808877409e-6 dfx = 2.7306166656073354
i = 4 x = 1.4142135644306453 fx = -1.6246371781214464e-7 dfx = 2.730613458335259
i = 5 x = 1.4142135622993912 fx = 5.819630999326364e-9 dfx = 2.730613573221527
i = 6 x = 1.4142135623757353 fx = -2.0846591120005087e-10 dfx = 2.730613569106061
i = 7 x = 1.4142135623730003 fx = 7.467804152838653e-12 dfx = 2.7306135692534137
x = 1.4142135623730003
> Terminated with exit code 0.
```

PROBLEM Secant method with one initial root estimation, quadratic difference Formula in excel (or open ofice calc) spreadsheet

Secant 3 $f(x)=x^2-a$

a	2									
Δx	0.1									
x	f(x)=x*x-a	x+2Δx	x+Δx	x-Δx	x-2Δx	f(x+2Δx)	f(x+Δx)	f(x-Δx)	f(x-2Δx)	f'(x)
1	-1	1.2	1.1	0.9	0.8	-0.56	-0.79	-1.19	-1.36	2.99167
1.334261838	-0.21975	1.53426	1.43426	1.23426	1.13426	0.35396	0.05711	-0.4766	-0.71345	2.08121
1.439846995	0.07316	1.63985	1.53985	1.33985	1.23985	0.6891	0.37113	-0.20481	-0.46278	1.71622
1.397218859	-0.04778	1.59722	1.49722	1.29722	1.19722	0.55111	0.24166	-0.31722	-0.56667	1.86805
1.422795983	0.02435	1.6228	1.5228	1.3228	1.2228	0.63347	0.31891	-0.25021	-0.50477	1.77768
1.409099259	-0.01444	1.6091	1.5091	1.3091	1.2091	0.5892	0.27738	-0.28626	-0.53808	1.82635
1.417005353	0.0079	1.61701	1.51701	1.31701	1.21701	0.61471	0.30131	-0.2655	-0.5189	1.79833
1.412610076	-0.00453	1.61261	1.51261	1.31261	1.21261	0.60051	0.28799	-0.27705	-0.52958	1.81393
1.415108941	0.00253	1.61511	1.51511	1.31511	1.21511	0.60858	0.29556	-0.27049	-0.52351	1.80507
1.413705498	-0.00144	1.61371	1.51371	1.31371	1.21371	0.60405	0.2913	-0.27418	-0.52692	1.81005
1.414499268	0.00081	1.6145	1.5145	1.3145	1.2145	0.60661	0.29371	-0.27209	-0.52499	1.80724
1.414052077	-0.00046	1.61405	1.51405	1.31405	1.21405	0.60516	0.29235	-0.27327	-0.52608	1.80882
1.414304575	0.00026	1.6143	1.5143	1.3143	1.2143	0.60598	0.29312	-0.2726	-0.52546	1.80793
1.414162185	-0.00015	1.61416	1.51416	1.31416	1.21416	0.60552	0.29269	-0.27298	-0.52581	1.80843
1.414242539	8.2E-05	1.61424	1.51424	1.31424	1.21424	0.60578	0.29293	-0.27277	-0.52562	1.80815
1.414197211	-4.6E-05	1.6142	1.5142	1.3142	1.2142	0.60563	0.29279	-0.27289	-0.52573	1.80831
1.414222786	2.6E-05	1.61422	1.51422	1.31422	1.21422	0.60572	0.29287	-0.27282	-0.52566	1.80822
1.414208358	-1.5E-05	1.61421	1.51421	1.31421	1.21421	0.60567	0.29283	-0.27286	-0.5257	1.80827
1.414216498	8.3E-06	1.61422	1.51422	1.31422	1.21422	0.60569	0.29285	-0.27283	-0.52568	1.80824

Let us try a complex root now:

```
 function secant(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  x-=fx/dfx
  println("i = $i x = $x fx = $fx dfx = $dfx")
  if abs(fx)<tolerance
```

```
    return x
  end #f
end #for
return x
end #function

function df(f,x)
 dx=0.001
 ff=(-3.0f(x+3dx)+32.0f(x+2dx)-168.0f(x+2dx)+672.0f(x+dx)-672.0f(x-dx)+168.0f(x-2dx)-32f(x-3dx)+3f(x-4dx))/(840.0dx)
 return ff
end

f(x)=x*x+2.0
x=1.0+1im
x=secant(f,x)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton.jl
i = 0 x = -0.035802703577893835 + 1.0000443843233529im fx = 2.0 + 2.0im dfx = 1.9307869047618762 + 1.9309523809520521im
i = 1 x = 0.0198149936939985 + 1.5165224545909592im fx = 1.0011930629668127 - 0.07160858531333268im dfx = -
0.06929879190867586 + 1.931038084967228im
i = 2 x = 0.0006250391365980393 + 1.4140140224387467im fx = -0.29944772130349495 + 0.06009976574905397im dfx =
0.03809633306177515 + 2.9283326444601654im
i = 3 x = -2.2429148695638293e-5 + 1.4142206082908915im fx = 0.0005647350205177926 + 0.0017676282074452696im dfx =
0.0010414446185980615 + 2.730393743328152im
i = 4 x = 8.01470657986998e-7 + 1.414213308849167im fx = -1.9928411592040618e-5 - 6.343952862358489e-5im dfx = -
0.0002087858083918558 + 2.7307926507712215im
i = 5 x = -2.86434804586515e-8 + 1.414213571488776im fx = 7.170745328632933e-7 + 2.266900942354623e-6im dfx = -
0.0001639285887472794 5 + 2.7307785558968445im
i = 6 x = 1.0236703951864 48e-9 + 1.4142135620453358im fx = -2.578303837097451e-8 - 8.101599759859701e-8im dfx = -
0.0001655314996 7005586 + 2.7307790630414224im
i = 7 x = -3.6584161715999224e-11 + 1.4142135623848795im fx = 9.270433309893633e-10 + 2.8953771118739662e-9im dfx = -
0.0001654742137577986 + 2.7307790448065887im
i = 8 x = 1.3074484980161337e-12 + 1.4142135623726713im fx = -3.33315597345063e-11 - 1.0347563533449558e-10im dfx = -
0.0001654762610751 4074 + 2.7307790454622314im
i = 9 x = -4.672556362748515e-14 + 1.4142135623731102im fx = 1.198596777385319e-12 + 3.69802279599639e-12im dfx = -
0.0001654761880454632 + 2.730779045438658im
x = -4.672556362748515e-14 + 1.4142135623731102im
> Terminated with exit code 0.
```

MULLER'S METHOD

Figure Werner Muller, German Mathematician

In false position method roots were found by using the root of the linear line passing through the two interval points. The same concept can be use to fit a quadratic curve and find the roots of the quadratic function. But quadratic function fitting is required 3 estimation point. The method consist of deriving the coefficients of a quadratic curve(parabola) that goes through three points. If function evaluation of these 3 points are $f(x_0)$, $f(x_1)$ and $f(x_2)$, enough information will be available to make a quadratic curve root estimation. Assuming quadratic function defined b the following equation :

$f(x) = a(x - x_2)^2 + b(x - x_2) + c$ (2.6.1)

Evaluation of the function in 3 given points will be

$f(x_0) = a(x_0 - x_2)^2 + b(x_0 - x_2) + c$ (2.6.2)

$f(x_1) = a(x_1 - x_2)^2 + b(x_1 - x_2) + c$ (2.6.3)

$f(x_2) = a(x_2 - x_2)^2 + b(x_2 - x_2) + c = c$ (2.6.4)

Considering the difference of the functions:

$f(x_0) - f(x_2) = a(x_0 - x_2)^2 + b(x_0 - x_2)$ (2.6.5)

$f(x_1) - f(x_2) = a(x_1 - x_2)^2 + b(x_1 - x_2)$ (2.6.6)

$h_0 = (x_1 - x_0)$ (2.6.7)

$h_1 = (x_2 - x_0)$ (2.6.8)

$d_0 = \frac{f(x_1) - f(x_0)}{h_0}$ (2.6.9)

$d_1 = \frac{f(x_2) - f(x_0)}{h_1}$ (2.6.10)

When these are substituted back into the main equation

$(h_0 + h_1)b - (h_0 + h_1)^2 a = h_0 d_0 + h_1 d_1$ (2.6.11)

$h_1 b - (h_1)^2 a = h_1 d_1$ (2.6.12)

is obtained. a and b can be solved from here.

$a = \frac{d_1 - d_0}{h_1 - h_0}$ (2.6.13)

$b = a h_1 + d_1$ (2.6.14)

$c = f(x_2)$ (2.6.16)

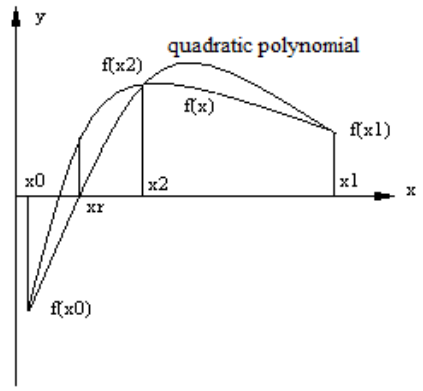


Figure graphic representation of Muller root finding method.

to find root root :

$\Delta = \sqrt{b^2 - 4ac}$

if

$|b + \Delta| > |b + \Delta| \quad then\ e = b + \Delta$

Else

$e = b + \Delta$

$x_r = x_2 - \frac{2c}{e}$

$x_0 = x_1 \quad x_1 = x_2 \quad x_2 = x_r$

x_2 will be substituted with x_r and iteration continues. It should be noted that in a quadratic formula complex roots can be located as well as the real roots, therefore this method can be used to find complex roots as well. The program NA10 is developed to calculate real roots by

Program Muller method root finding with quadratic formula

```
 function  muller(f,x0,x1,x2)
 # Finding real roots by using Muller method
maxit=5
xr=x0
iter=0
es1=0.001
es=es1
ea=1.1*es
h0=0.0
h1=0.0
d0=0.0
d1=0.0
a=0.0
b=0.0
c=0.0
fx0=0.0
fx1=0.0
```

```
fx2=0.0
determinant=0.0
den=0.0
dxr=0.0
while iter<maxit && ea>es
   fx0=f(x0)
   fx1=f(x1)
   fx2=f(x2)
   h0=x1-x0
   h1=x2-x1
   d0=(fx1-fx0)/h0
   d1=(fx2-fx1)/h1
   a=(d1-d0)/(h1+h0)
   b=a*h1+d1
   c=fx2
   determinant=b*b-4.0*a*c
   if determinant<=0.0
     determinant=0
   else
     determinant=sqrt(determinant)
   end
   if abs(b+determinant)>abs(b-determinant)
     den=b+determinant
   else
     den=b-determinant
   end
   dxr=-2*c/den
   xr=x2+dxr
   println("a= $a b= $b c= $c disc= $determinant xr= $xr iter= $iter")
   ea=abs((xr-x2)/xr)*100
   iter+=1
   x0=x1
   x1=x2
   x2=xr
end
if iter>=maxit
  print("Maximum number of iteration is exceeded \nresult might not be valid")
end
return xr
end

f(x)=x*x-2.0
x0=0.0
x1=1.0
x2=2.0
x=muller(f,x0,x1,x2)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" muller.jl
a= 1.0 b= 4.0 c= 2.0 disc= 2.8284271247461903 xr= 1.414213562373095 iter= 0
a= 1.0 b= 2.82842712474619 c= -4.440892098500626e-16 disc= 2.8284271247461903 xr= 1.4142135623730951 iter= 1
x = 1.4142135623730951
> Terminated with exit code 0.
```

Complex roots:

```
function  muller(f,x0,x1,x2)
 # Finding real roots by using Muller method
maxit=5
xr=x0
iter=0
es1=0.001
es=es1
ea=1.1*es
h0=0.0
h1=0.0
d0=0.0
d1=0.0
a=0.0
b=0.0
c=0.0
```

```
fx0=0.0
fx1=0.0
fx2=0.0
determinant=0.0
den=0.0
dxr=0.0
while iter<maxit && ea>es
   fx0=f(x0)
   fx1=f(x1)
   fx2=f(x2)
   h0=x1-x0
   h1=x2-x1
   d0=(fx1-fx0)/h0
   d1=(fx2-fx1)/h1
   a=(d1-d0)/(h1+h0)
   b=a*h1+d1
   c=fx2
   determinant=sqrt(b*b-4.0*a*c)
   if abs(b+determinant)>abs(b-determinant)
     den=b+determinant
   else
     den=b-determinant
   end
   dxr=-2*c/den
   xr=x2+dxr
   println("a= $a b= $b c= $c disc= $determinant xr= $xr iter= $iter")
   ea=abs((xr-x2)/xr)*100
   iter+=1
   x0=x1
   x1=x2
   x2=xr
end
if iter>=maxit
  print("Maximum number of iteration is exceeded \nresult might not be valid")
end
return xr
end

f(x)=x*x+2.0
x0=0.0+1im
x1=1.0+1im
x2=2.0+1im
x=muller(f,x0,x1,x2)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" muller.jl
a= 1.0 + 0.0im b= 4.0 + 2.0im c= 5.0 + 4.0im disc= 0.0 + 2.8284271247461903im xr= 0.0 + 1.414213562373095im iter= 0
a= 1.0 - 0.0im b= 0.0 + 2.82842712474619im c= 4.440892098500626e-16 + 0.0im disc= 0.0 + 2.8284271247461903im xr= 0.0 +
1.4142135623730951im iter= 1
x = 0.0 + 1.4142135623730951im
> Terminated with exit code 0.
```

NEWTON-RAPHSON (SECANT) – BISECTION COMBINED METHOD

Bisection method is good in getting closer to the root. But when it get close to root it takes a long time to achieve the final accuracy. On the other hand, Newton-Raphson method is very efficient when the estimation is close to the actual root but rather poor performer when the root estimation is not very close to the actual root. By combining two methods advantages of both methods will be maximized while inefficiencies will be eliminated. Therefore a combine method is a good candidate for an efficient algorithm.

Program Newton-Raphson (Secant)-Bisection combined method

```
 function newton_bisection(f,x1,x2)
xacc=1.0e-10 #accuracy
h=0.0001    #derivative function step
MAXIT=100   #Maximum number of iterations
j=0
dfr=0.0
dx=0.0
dxold=0.0
fr=0.0
```

```
fh=0.0
fl=0.0
temp=.0
b=0.0
a=0.0
r=0.0
fl=f(x1)
fh=f(x2)
#Check if a root is existed in the given region
if ((fl > 0.0 && fh > 0.0) || (fl < 0.0 && fh < 0.0))
  print("There are no root in the given region or root?")
end
if fl == 0.0
  return x1
elseif fh == 0.0
  return x2
elseif  fl < 0.0
  a=x1
  b=x2
else
  b=x1a=x2
end
r=0.5*(x1+x2) #midpovalue
dxold=abs(x2-x1)
dx=dxold
fr=f(r)   #function value at midpoint
dfr=df(f,r) #derivative of function value at midpo
for j=1:MAXIT
if ((r-b)*dfr-fr)*((r-a)*dfr-fr) > 0.0 || (abs(2.0*fr) > abs(dxold*dfr))
    #bisection step
                dxold=dx
    dx=0.5*(b-a)
    r=a+dx
    if a == r
      return r
    end
else
    #Newton-Raphson (Secant) step
                dxold=dx
    dx=fr/dfr
    temp=r
    r -= dx
    if (temp == r)
       return r  #solution!!!!!
    end
end
if (abs(dx) < xacc)
   return r #solution!!!!!
end
fr=f(r)
dfr=df(f,r)
if (fr < 0.0)
a=r
else
b=r
end
end
return 0.0
end

function df(f,x)
 dx=0.001
 ff=(-3.0f(x+3dx)+32.0f(x+2dx)-168.0f(x+2dx)+672.0f(x+dx)-672.0f(x-dx)+168.0f(x-2dx)-32f(x-3dx)+3f(x-4dx))/(840.0dx)
 return ff
end

f(x)=x*x-2.0
x1=1.0
x2=3.0
x=newton_bisection(f,x1,x2)
```

```
print("x = $x")
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" secant.jl
x = 1.4142135623713463
> Terminated with exit code 0.
```

INVERSE QUADRATIC LAGRANGE INTERPOLATION

Lagrange interpolation will be investigated further in the interpolation section. In here, the Formula for the quadratic inverse lagrange polinomial will be introduced to utilise in finding roots of a function.

$$x = \frac{[y-f(x_1)][y-f(x_2)]x_3}{[f(x_3)-f(x_1)][f(x_3)-f(x_2)]} + \frac{[y-f(x_2)][y-f(x_3)]x_1}{[f(x_1)-f(x_2)][f(x_1)-f(x_3)]} + \frac{[y-f(x_3)][y-f(x_1)]x_2}{[f(x_2)-f(x_3)][f(x_2)-f(x_1)]}$$

In this equation if y=0 is taken, the value x will give us a quadratic approximation to the root of the function

$$x = \frac{f(x_1)f(x_2)x_3}{[f(x_3)-f(x_1)][f(x_3)-f(x_2)]} + \frac{f(x_2)f(x_3)x_1}{[f(x_1)-f(x_2)][f(x_1)-f(x_3)]} + \frac{f(x_3)f(x_1)x_2}{[f(x_2)-f(x_3)][f(x_2)-f(x_1)]}$$

In order to make the equation simpler further definations will apply as:

$R = \frac{f(x_2)}{f(x_3)}$ (2.8.4) $S = \frac{f(x_2)}{f(x_1)}$ (2.8.5) $T = \frac{f(x_1)}{f(x_3)}$

$$P = S[T(R-T)(x_3-x_2) - (1-R)(x_2-x_1)]$$

$$Q = (T-1)(R-1)(S-1)$$

Now the root can be written as:

$$x = x_2 + \frac{P}{Q} \qquad (2.8.9)$$

When a new root is found the neigbour region of the root is left for the next iteration and the region further away to the root is eliminated.

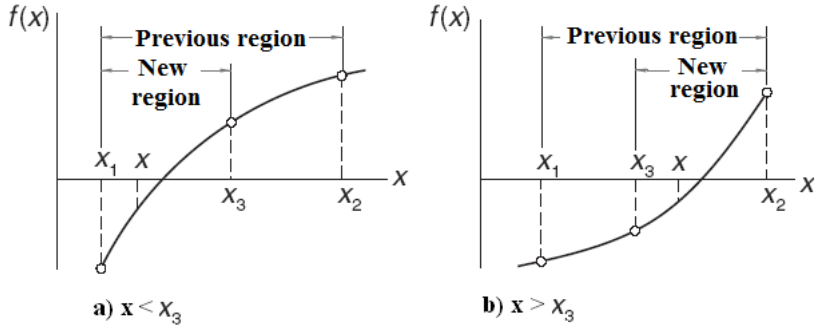


Figure 2.10 Quadratic inverse Lagrange interpolation region selection

Program quadratic inverse lagrange interpolation formula method (java code)

```
 function inverseinter2Droot(f,x1,x3)
# Inverse quadratic Lagrange interpolation
x2=(x1+x3)/2.0154254
test=0.0
global p=0.0
f1=0.0
f2=0.0
f3=0.0
fp=0.0
maxit=100
iter=0
```

```
tol=1.0e-10
es=0.00000001
ea=0.001
f1=f(x1)
f2=f(x2)
f3=f(x3)
if f1==0
 return x1
elseif f2==0
 return x2
elseif f3==0
 return x3
end
if f1*f3>0
 print("Root is not existed in the given region")
end
while ea>es && iter<maxit
            p=-(f2*f3*x1*(f2-f3)+f3*f1*x2*(f3-f1)+f1*f2*x3*(f1-f2))/((f1-f2)*(f2-f3)*(f3-f1))
            println("i = $iter p = $p")
   fp=f(p)
   ea=abs(x3-p)
            if abs(f3)<tol
              return x3
            end
   if p>x3
      x1=x3
      f1=f3
      x3=p
      f3=fp
   elseif p<x3
      x2=x3
      f2=f3
      x3=p
      f3=fp
   end
   iter+=1
end
if iter>=maxit
  print("Maximum number of iteration is exceeded \nresult might not be valid")
end
return p
end

f(x)=x*x-2.0
x0=0.0
x2=2.0
x=inverseinter2Droot(f,x0,x2)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" inverse_quadratic_lagrange_int_root.jl
i = 0 p = 1.6735267838105297
i = 1 p = 1.3253225124221881
i = 2 p = 1.4358421880222398
i = 3 p = 1.4141003048715692
i = 4 p = 1.4142136188326897
i = 5 p = 1.414213562373061
i = 6 p = 1.4142135623730951
x = 1.414213562373061
> Terminated with exit code 0.
```

BRENT METHOD (INVERSE QUADRATIC LAGRANGE INTERPOLATION – BISECTION AND SECANT COMBINED METHOD)

Figure 2.9.1 Richard P Brent

Being a quadratic method, Inverse quadratic Lagrange interpolation is an efficient method by combining it with bi-section and secant method one of the most used root finding method is created. Two versions of the program code and algorithm of the code is given below.

BRENT METHOD (julia language version)

```
 function brent(f,a,b)
test=0.0
p=0.0
f1=0.0
f2=0.0
f3=0.0
fp=0.0
maxit=500
iter=0
tol=1.0e-15
es=0.0000001
x1=a
f1=f(x1)
x2=b
f2=f(x2)
x3=(x1+x2)/2.0
f3=f(x3)
if(f1==0)
  return x1
elseif(f2==0)
  return x2
elseif(f3==0)
  return x3
end
if(f1*f2>0)
  print("No root is existed in the given region")
end
p=-(f2*f3*x1*(f2-f3)+f3*f1*x2*(f3-f1)+f1*f2*x3*(f1-f2))/((f1-f2)*(f2-f3)*(f3-f1))
fp=f(p)
ea=abs(x3-p)
while (ea>es)&&(iter<maxit)
                if abs(f3)<tol
                   return x3
                end
   if (p<x1) && (p>x2)
     p=(x1+x2)/2.0
      if p>x3
        x1=x3
        f1=f3
        x3=p
        f3=fp
      elseif p<x3
        x2=x3
        f2=f3
        x3=p
        f3=fp
      end
   else  #(p<x2) && (p>x3)
      if p>x3
```

```
        x1=x3
        f1=f3
        x3=p
        f3=fp
      elseif p<x3
        x2=x3
        f2=f3
        x3=p
        f3=fp
      end # if p>x3
    end #if else
    p=-(f2*f3*x1*(f2-f3)+f3*f1*x2*(f3-f1)+f1*f2*x3*(f1-f2))/((f1-f2)*(f2-f3)*(f3-f1))
    fp=f(p)
    ea=abs(x3-p)
    iter+=1
end  # while
if iter>=maxit
  print("Maximum number of iteration is exceeded \nresult might not be valid")
end #if
return p
end # function

f(x)=x*x-2.0
a=0.0
b=2.0
x=brent(f,a,b)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" brent.jl
x = 1.4142135623730954
> Terminated with exit code 0.
```

Algorithm
Input a,b and a function f
Calculate f(a) and f(b)
if(f(a)*f(b)<0 then give no root error
if |f(a)| < |f(b)| then swap(a,b)
c=a
repeat untill f(b)=0 or s=0 or |b-a| is small enough (convergence)
if(f(a)≠f(c) and f(b)≠f(c) then

$$s = \frac{af(b)f(c)}{(f(a)-f(b))(f(a)-f(c))} + \frac{bf(a)f(c)}{(f(b)-f(a))(f(b)-f(c))} + \frac{cf(a)f(b)}{(f(c)-f(a))(f(c)-f(b))}$$

(inverse quadratic interpolation)
else

$$s = b - f(b)\frac{b-a}{(f(b)-f(a))}$$ (secant rule)

end if
if *(condition 1) s* is not between (3*a* + *b*)/4 and *b*
or *(condition 2)* (mflag is set and |*s*−*b*| ≥ |*b*−*c*| / 2)
or *(condition 3)* (mflag is cleared and |*s*−*b*| ≥ |*c*−*d*| / 2)
or *(condition 4)* (mflag is set and |*b*−*c*| < |δ|)
or *(condition 5)* (mflag is cleared and |*c*−*d*| < |δ|)
then

$$s := \frac{a+b}{2}$$ *(bisection method)*

set mflag
else
clear mflag
end if
calculate *f*(*s*)
d := *c* *(d is assigned for the first time here; it won't be used above on the first iteration because mflag is set)*
c := *b*
if *f*(*a*) *f*(*s*) < 0 then *b* := *s* else *a* := *s* end if
if |*f*(*a*)| < |*f*(*b*)| then swap (*a*,*b*) end if
end repeat
output *b or s (return the root*

HIGHER DERIVATIVE ORDER METHODS

Consider the Taylor Formula, but this time assume that two term is remaining

$$f(x_{n+1}) = 0 = f(x_n) + \frac{f'(x_n)}{1!}(x_{n+1} - x_n) + \frac{f''(x_n)}{2!}(x_{n+1} - x_n)^2 + \frac{f^{(3)}(x_n)}{3!}(x_{n+1} - x_n)^3 + \cdots$$

When second term of the Taylor Formula is also taken into acount the Newton-Raphson Formula takes the form:

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2f'(x_n)f'(x_n) - f(x_n)f''(x_n)}$$

This method is called **Halley's method**.

Figure Edmond Halley, British Mathematician and Astronomer

The iterative methods with higher-order convergence are presented in some literature . **Olver's method** is also cubically convergent second order method. It is faster than the Newton-Raphson method.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{1}{2}\frac{f(x_n)f(x_n)f''(x_n)}{f'(x_n)f'(x_n)f'(x_n)}$$

In [48], Abbasbandy gives a third order iterative method, called **Abbasbandy's method**(AM) provisionally, which is expressed as

Figure Daeid Abbasbandy, Iranian Mathematician

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{1}{2}\frac{f(x_n)f(x_n)f"(x_n)}{f'(x_n)f'(x_n)f'(x_n)} - \frac{1}{6}\frac{f(x_n)f(x_n)f(x_n)f'''(x_n)}{f'(x_n)f'(x_n)f'(x_n)f'(x_n)}$$

Li and Wang[49] suggested two fourth order methods and called them **Thiele Method 1** (TM1) and **Thiele Method 2** (TM2) due to fact that equations derived using Thiele's continued fraction method.

Thorvald N. Thiele, Danish atronomer & mathematician

Equations has a general form of

$$x_{n+1} = x_n - \frac{P_1(x_n)}{Q_1(x_n)} \quad P_1(x) = 4f(x)f'(x)[3f'(x)f'(x)f"(x)\text{-}3f(x)f"(x)f"(x) + f(x)f'(x)f'''(x)] \quad (2.11.6)$$

$$Q_1(x) = 12f'(x)f'(x)f'(x)f'(x)f(x)\text{-}18f(x)f'(x)f'(x)f"(x)f"(x) + 3f(x)f(x)\,f"(x)f"(x)f"(x) + 4f(x)f'(x)f'(x)f'(x)f'''(x) \qquad x_{n+1} = x_n - \frac{P_2(x_n)}{Q_2(x_n)}$$

$$P_2(x) = f(x)[6f'(x)f'(x)\text{-}3f(x)f"(x)f"(x) + 2f(x)f'(x)f'''(x)]$$

$$Q_2(x) = 2f'(x)\left[3f'(x)f'(x)f"(x) - 3f(x)f"(x)f"(x) + f(x)f'(x)f'''(x)\right]$$

Program Higher order methods, Newton, Halley, Abbasbandy Thiele1 and Thiele2

```
 function newton(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  x-=fx/dfx
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function halley(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
```

```
  d2fx=d2f(f,x)
  x-=2.0*fx*dfx/(2.0*dfx*dfx-fx*d2fx)
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function abbasbandy(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  d3fx=d2f(f,x)
     x-=(fx/dfx+fx*fx*d2fx/(2.0*dfx*dfx*dfx)+fx*fx*fx*d3fx/(6.0*dfx*dfx*dfx*dfx))
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function TM1(f,x)
# Thiele 1
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  d3fx=d2f(f,x)
  P1=4.0*fx*dfx*(3.0*dfx*dfx*d2fx-3.0*fx*d2fx*d2fx+fx*dfx*d3fx)
  Q1=12.0*dfx*dfx*dfx*dfx*d2fx-18.0*fx*dfx*dfx*d2fx*d2fx+3.0*fx*fx*d2fx*d2fx*d2fx+4.0*fx*dfx*dfx*dfx*d3fx
  x-=P1/Q1;
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function TM2(f,x)
# Thiele 2
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  d3fx=d2f(f,x)
  P2=fx*(6.0*dfx*dfx*d2fx-3.0*fx*d2fx*d2fx+2*fx*dfx*d3fx)
  Q2=2.0*dfx*(3.0*dfx*dfx*d2fx-3.0*fx*d2fx*d2fx+fx*dfx*d3fx)
  x-=P2/Q2;
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function
```

```
#first derivative
function df(f,x)
 dx=0.001
 c=[1.0/280.0 -4.0/105.0  1.0/5.0  -4.0/5.0  0  4.0/5.0  -1.0/5.0  4.0/105.0 -1.0/280.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/dx
end
#second derivative
function d2f(f,x)
 dx=0.001
 c=[-1.0/560.0  8.0/315.0 -1.0/5.0 8.0/5.0 -205.0/72.0 8.0/5.0 -1.0/5.0 8.0/315.0 -1.0/560.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx)
end
#third derivative
function d3f(f,x)
 dx=0.001
 c=[-7.0/240.0  3.0/10.0  -169.0/120.0  61.0/30.0 0.0  -61.0/30.0  169.0/120.0  -3.0/10.0
      7.0/240.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx*dx)
end

f(x)=x*x*x+3.6*x-36.4
x=1.0
s=["Newton" "Halley" "Abbasbandy" "Thiele1" "Thiele2"]
r=zeros(Float64,5)
r[1]=newton(f,x)
r[2]=halley(f,x)
r[3]=abbasbandy(f,x)
r[4]=TM1(f,x)
r[5]=TM2(f,x)
for i=1:5
  s1=s[i]
  r1=r[i]
  f1=f(r1)
  println("$s1 x = $r1 y = $f1")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton2.jl
Newton x = 2.953618919393856 y = 7.105427357601002e-15
Halley x = 2.9536189193938553 y = -7.105427357601002e-15
Abbasbandy x = 2.953618919393856 y = 7.105427357601002e-15
Thiele1 x = 2.9536189193938553 y = -7.105427357601002e-15
Thiele2 x = 2.9536189193938553 y = -7.105427357601002e-15

> Terminated with exit code 0.
```

This methods can also be used for complex roots

```
function newton(f,x)
nmax=100
tolerance=1.0e-10
```

```
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  x-=fx/dfx
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function halley(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  x-=2.0*fx*dfx/(2.0*dfx*dfx-fx*d2fx)
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function abbasbandy(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  d3fx=d2f(f,x)
     x-=(fx/dfx+fx*fx*d2fx/(2.0*dfx*dfx*dfx)+fx*fx*fx*d3fx/(6.0*dfx*dfx*dfx*dfx))
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function TM1(f,x)
# Thiele 1
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  d3fx=d2f(f,x)
  P1=4.0*fx*dfx*(3.0*dfx*dfx*d2fx-3.0*fx*d2fx*d2fx+fx*dfx*d3fx)
  Q1=12.0*dfx*dfx*dfx*dfx*d2fx-18.0*fx*dfx*dfx*d2fx*d2fx+3.0*fx*fx*d2fx*d2fx*d2fx+4.0*fx*dfx*dfx*dfx*d3fx
  x-=P1/Q1;
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function TM2(f,x)
# Thiele 2
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  d3fx=d2f(f,x)
  P2=fx*(6.0*dfx*dfx*d2fx-3.0*fx*d2fx*d2fx+2*fx*dfx*d3fx)
  Q2=2.0*dfx*(3.0*dfx*dfx*d2fx-3.0*fx*d2fx*d2fx+fx*dfx*d3fx)
  x-=P2/Q2;
  if abs(fx)<tolerance
    return x
  end #f
```

```
end #for
return x
end #function

#first derivative
function df(f,x)
 dx=0.001
 c=[1.0/280.0 -4.0/105.0 1.0/5.0 -4.0/5.0 0 4.0/5.0 -1.0/5.0 4.0/105.0 -1.0/280.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/dx
end
#second derivative
function d2f(f,x)
 dx=0.001
 c=[-1.0/560.0 8.0/315.0 -1.0/5.0 8.0/5.0 -205.0/72.0 8.0/5.0 -1.0/5.0 8.0/315.0 -1.0/560.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx)
end
#third derivative
function d3f(f,x)
 dx=0.001
 c=[-7.0/240.0 3.0/10.0 -169.0/120.0 61.0/30.0 0.0 -61.0/30.0 169.0/120.0 -3.0/10.0 7.0/240.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx*dx)
end

f(x)=x*x*-3.6*x-36.4
x=1.0+1im
s=["Newton" "Halley" "Abbasbandy" "Thiele1" "Thiele2"]
r=zeros(ComplexF64,5)
r[1]=newton(f,x)
r[2]=halley(f,x)
r[3]=abbasbandy(f,x)
r[4]=TM1(f,x)
r[5]=TM2(f,x)
for i=1:5
  s1=s[i]
  r1=r[i]
  f1=f(r1)
  println("$s1 x = $r1 y = $f1")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" newton3.jl
Newton x = 1.0811923527860583 + 1.8726800877803869im y = 0.0 + 3.552713678800501e-15im
Halley x = 1.0811923527860583 + 1.8726800877803869im y = 0.0 + 3.552713678800501e-15im
Abbasbandy x = 1.0811923527860583 + 1.8726800877803869im y = 0.0 + 3.552713678800501e-15im
Thiele1 x = 1.0811923527860583 + 1.8726800877803869im y = 0.0 + 3.552713678800501e-15im
Thiele2 x = 1.0811923527860585 + 1.872680087780387im y = 1.4210854715202004e-14 + 0.0im

> Terminated with exit code 0.
```

Hasan[50] proposed several higher order methods that is relatively more economical to use (relatively less function calculations) Some of them are listed here.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'\left(x_n - \frac{f(x_n)}{2f'(x_n)}\right)}$$

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'(x_n) + f'\left(x_n - \frac{f(x_n)}{2f'(x_n)}\right)}$$

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'\left(x_n - \alpha\frac{f(x_n)}{f'(x_n)}\right) + f'\left(x_n - \beta\frac{f(x_n)}{f'(x_n)}\right)} \qquad where\ \alpha + \beta = 1$$

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'(x_n)+\frac{1}{r}f'(x_n)log\left(\frac{f'(x_n)f'(x_n)-f(x_n)f"(x_n)}{f'(x_n)f'(x_n)}\right)}$$

Program Higher order methods, proposed by Hasan[50] java code

```
 function newton(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  x-=fx/dfx
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function hasan47a(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  x1=x-fx/(2.0*dfx)
  dfx1=df(f,x1)
  x-=fx/dfx1
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function hasan47b(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  x1=x-fx/dfx
  dfx1=df(f,x1)
  x-=2.0*fx/(dfx+dfx1)
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function hasan47c(f,x,alfa)
nmax=100
tolerance=1.0e-10
beta=1.0-alfa
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  x1=x-alfa*fx/dfx
  dfx1=df(f,x1)
  x2=x-beta*fx/dfx
```

```
  dfx2=d2f(f,x2)
  x-=2.0*fx/(dfx1+dfx2)
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function hasan43(f,x)
nmax=100
tolerance=1.0e-10
r=3.0 #fourth order
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  yx=(dfx*dfx-fx*d2fx)/(dfx*dfx)
  dyx=dfx+1.0/r*dfx*log(yx)
  x-=fx/dyx
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

#first derivative
function df(f,x)
 dx=0.001
 c=[1.0/280.0 -4.0/105.0  1.0/5.0  -4.0/5.0  0  4.0/5.0  -1.0/5.0  4.0/105.0 -1.0/280.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/dx
end
#second derivative
function d2f(f,x)
 dx=0.001
 c=[-1.0/560.0  8.0/315.0 -1.0/5.0 8.0/5.0 -205.0/72.0 8.0/5.0 -1.0/5.0 8.0/315.0 -1.0/560.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx)
end
#third derivative
function d3f(f,x)
 dx=0.001
 c=[-7.0/240.0  3.0/10.0  -169.0/120.0  61.0/30.0 0.0  -61.0/30.0  169.0/120.0  -3.0/10.0
     7.0/240.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx*dx)
end

f(x)=x*x*x+3.6*x-36.4
x=1.0
s=["Newton" "Hasan47a" "Hasan47b" "Hasan47c" "Hasan43" ]
```

```
r=zeros(Float64,5)
r[1]=newton(f,x)
r[2]=hasan47a(f,x)
r[3]=hasan47b(f,x)
r[4]=hasan47c(f,x,0.5)
r[5]=hasan43(f,x)
for i=1:5
  s1=s[i]
  r1=r[i]
  f1=f(r1)
  println("$s1 x = $r1 y = $f1")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" hasan.jl
Newton x = 2.953618919393856 y = 7.105427357601002e-15
Hasan47a x = 2.953618919393856 y = 7.105427357601002e-15
Hasan47b x = 2.9536189193938553 y = -7.105427357601002e-15
Hasan47c x = 2.9536189193934064 y = -1.3372414287005085e-11
Hasan43 x = 2.9536189193938553 y = -7.105427357601002e-15

> Terminated with exit code 0.
```

A complex version of Hasan formula is also given here. In order to change julia code to complex version, changing input variable to complex is enough

Program Complex Hasan method[50] (julia version) hasan1.jl

```
 function newton(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  x-=fx/dfx
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function hasan47a(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  x1=x-fx/(2.0*dfx)
  dfx1=df(f,x1)
  x-=fx/dfx1
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function hasan47b(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  x1=x-fx/dfx
  dfx1=df(f,x1)
  x-=2.0*fx/(dfx+dfx1)
  if abs(fx)<tolerance
    return x
```

```
  end #f
end #for
return x
end #function

function hasan47c(f,x,alfa)
nmax=100
tolerance=1.0e-10
beta=1.0-alfa
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  x1=x-alfa*fx/dfx
  dfx1=df(f,x1)
  x2=x-beta*fx/dfx
  dfx2=d2f(f,x2)
  x-=2.0*fx/(dfx1+dfx2)
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

function hasan43(f,x)
nmax=100
tolerance=1.0e-10
r=3.0 #fourth order
for i=0:nmax
  fx=f(x)
  dfx=df(f,x)
  d2fx=d2f(f,x)
  yx=(dfx*dfx-fx*d2fx)/(dfx*dfx)
  dyx=dfx+1.0/r*dfx*log(yx)
  x-=fx/dyx
  if abs(fx)<tolerance
    return x
  end #f
end #for
return x
end #function

#first derivative
function df(f,x)
 dx=0.001
 c=[1.0/280.0 -4.0/105.0  1.0/5.0  -4.0/5.0  0  4.0/5.0  -1.0/5.0  4.0/105.0  -1.0/280.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/dx
end
#second derivative
function d2f(f,x)
 dx=0.001
 c=[-1.0/560.0  8.0/315.0 -1.0/5.0 8.0/5.0 -205.0/72.0 8.0/5.0 -1.0/5.0 8.0/315.0 -1.0/560.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx)
end
#third derivative
function d3f(f,x)
 dx=0.001
 c=[-7.0/240.0  3.0/10.0  -169.0/120.0  61.0/30.0  0.0  -61.0/30.0 169.0/120.0  -3.0/10.0  7.0/240.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx*dx)
end

f(x)=x*x*x+3.6*x-36.4
x=1.0+1im
s=["Newton" "Hasan47a" "Hasan47b" "Hasan47c" "Hasan43" ]
```

```
r=zeros(ComplexF64,5)
r[1]=newton(f,x)
r[2]=hasan47a(f,x)
r[3]=hasan47b(f,x)
r[4]=hasan47c(f,x,0.5)
r[5]=hasan43(f,x)
for i=1:5
  s1=s[i]
  r1=r[i]
  f1=f(r1)
  println("$s1 x = $r1 y = $f1")
end
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" hasan1.jl
Newton x = 2.9536189193938553 - 8.519697776386927e-29im y = -7.105427357601002e-15 - 2.53644984590039e-27im
Hasan47a x = 2.953618919393856 - 4.353526120688459e-29im y = 7.105427357601002e-15 - 1.296114128431739e-27im
Hasan47b x = -1.4768094596969277 - 3.1847917578314275im y = -7.105427357601002e-15 - 1.7763568394002505e-15im
Hasan47c x = 2.9536189193942644 + 5.003957697275287e-13im y = 1.2164491636212915e-11 + 1.48975797772156e-11im
Hasan43 x = 2.9536189193938553 + 1.0349855076499675e-27im y = -7.105427357601002e-15 + 3.081316849834548e-26im

> Terminated with exit code 0.
```

RIDDER METHOD

False position (Ragula Falsi) method was previously investigated. Ridder method is a modified form False position method. If a root existed between x1 and x2 Ridder method first evaluate the midpoint

$$x_3 = \frac{x_1+x_2}{2} \qquad (2.13.1)$$

and than solve a special conversion conversion factor e^Q. This convertion linearize the equation. It is basically find the root of the quadratic function

$$f(x_1) - 2\,f(x_3)e^Q + f(x_1) - 2\,f(x_2)e^Q = 0 \qquad (2.13.2)$$

The root of this function can be calculated as
of the quadratic function

$$e^Q = \frac{f(x_3)+sign[f(x_2)]\sqrt{f(x_3)^2-f(x_1)f(x_2)}}{f(x_2)} \qquad (2.13.3)$$

Function sign(x) will be given the sign of x. If x is smaller than 0 it will be -1, if x is bigger than 0 it will be +1. Now the false position method is applied, not to the values f(x1), f(x3), f(x2), but to the values $f(x_1)$, $f(x_3)e^Q$, $f(x_2)e^{2Q}$, yielding a new guess for the root, x_4. The overall formula is:

$$x_4 = x_3 + (x_3 - x_1)\frac{f(x_3)+sign[f(x_1)-f(x_2)]f(x_3)}{\sqrt{f(x_3)^2-f(x_1)f(x_2)}} \qquad (2.13.4)$$

Because the equation is quadratic, it supplies a quadratic approximation. In addition the root x_4 should be in between x_1 and x_2.

Program : Root finding Ridder Method

```
 unction SIGN(a,b)
  return abs(a)*b/abs(b)
end

function ridder(f,x1,x2)
   xacc=1.0e-9
   NR=0
            MAXITER=200
            ABC=-1.0e30
            j=0
            answer=0.0
            f_high=0.0
            f_low=0.0
            f_mid=0.0
```

```
        fnew=0.0
        s=0.0
        x_high=0.0
        x_low=0.0
        x_mid=0.0
        xnew=0.0
        f_low=f(x1)
        f_high=f(x2)
        if (f_low > 0.0 && f_high < 0.0) || (f_low < 0.0 && f_high > 0.0)
                x_low=x1
                x_high=x2
                answer=ABC
                for j=1:MAXITER
                        x_mid=0.5*(x_low+x_high)
                        f_mid=f(x_mid)
                        s=sqrt(f_mid*f_mid-f_low*f_high)
                        if s == 0.0
                          return answer
                        end
                        xnew=x_mid+(x_mid-x_low)*((f_low >= f_high ? 1.0 : -1.0)*f_mid/s)
                        if abs(xnew-answer) <= xacc
                           return answer
                        end
                        answer=xnew
                        fnew=f(answer)
                        if fnew == 0.0
                          return answer
                        end
                        if SIGN(f_mid,fnew) != f_mid
                                x_low=x_mid
                                f_low=f_mid
                                x_high=answer
                                f_high=fnew
                        elseif SIGN(f_low,fnew) != f_low
                                x_high=answer
                                f_high=fnew
                        elseif SIGN(f_high,fnew) != f_high
                                x_low=answer
                                f_low=fnew
                        else
                           print("we should not reach to this point")
                        end # if SIGN
                        if (abs(x_high-x_low) <= xacc)
                          return answer
                        end
                end #for MAXITER
    print("Maximum number of iteration is exceeded \nresult might not be valid")
        else
                if  f_low == 0.0
                  return x1
                elseif f_high == 0.0
                  return x2
                end
  end #if(f_low >
  print("no roots existed between given limits")
        return 0.0
end # function

f(x)=x*x-2.0
a=0.0
b=2.0
x=ridder(f,a,b)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" ridder.jl
x = 1.4142135624398362
> Terminated with exit code 0.
```

2.14 ROOTS OF THE SECOND AND THIRD DEGREE POLYNOMIALS

The roots of quadratic polynomials are well known.

$f(x) = ax^2 + bx + c = 0$

$\Delta = b^2 - 4ac$

$x_0 = \frac{-b}{2a} + \frac{\sqrt{\Delta}}{2a}$

$x_1 = \frac{-b}{2a} - \frac{\sqrt{\Delta}}{2a}$

Of course if $\Delta = b^2 - 4ac$ is negative, roots will be complex numbers.

Program Root of quadratic polynomial (analytical formula)

```
function square_roots(d)
  x=zeros(ComplexF64,2)
  a=d[3]
  b=d[2];
  c=d[1];
  delta=b*b-4*a*c
  delta1=Complex(delta)^0.5
  x[1]=(-b+delta1)/(2*a)
  x[2]=(-b-delta1)/(2*a);
  return x
end

d=[1 -2 1]
x=square_roots(d)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" square_roots.jl
x = ComplexF64[1.0 + 0.0im, 1.0 - 0.0im]
> Terminated with exit code 0.
> Terminated with exit code 0.
```

```
d=[2 0 1]
x=square_roots(d)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" square_roots.jl
x = ComplexF64[0.0 + 1.4142135623730951im, 0.0 - 1.4142135623730951im]
> Terminated with exit code 0.
```

Similar to quadratic polynomials, third degree polynomials can also be solved by using analytical formulations. The basic difference is that the roots should be calculated as complex variables. Assuming a third degree polynomial of

Figure NiccoloTartaglia

$f(x) = d_3x^3 + d_2x^2 + d_1x + d_0 = 0$

$a = \frac{d_2}{d_3} \quad b = \frac{d_1}{d_3} \quad c = \frac{d_0}{d_3}$

$f(x) = x^3 + ax^2 + bx + c = 0$

$Q = \frac{a^2 - 3b}{9}$ (2.14.4) $\quad y_1 = 2a^3$ (2.14.5) $y_2 = -9ab$ (2.14.6) $y_3 = 27c$

$y_4 = y_1 + y_2 + y_3$

$R = \frac{y_4}{54} = \frac{2a^3 - 9ab + 27c}{54}$ $\theta = cos^{-1}\left(\frac{R}{\sqrt{Q^3}}\right)$

If $R^2 < Q^3$ three real roots are existed

$x_0 = \left(-2\sqrt{Q}\right)cos\left(\frac{\theta}{3}\right) - \frac{a}{3}$

$x_1 = \left(-2\sqrt{Q}\right)cos\left(\frac{\theta - 2\pi}{3}\right) - \frac{a}{3}$

$x_2 = \left(-2\sqrt{Q}\right)cos\left(\frac{\theta + 2\pi}{3}\right) - \frac{a}{3}$

(This equation first appears in Chapter VI of François Viete's treatise "De emendatione," published in 1615!, Original equation created by Niccolo Tartaglia in 1539)

If $R^2 \geq Q^3$

$A = (R + \sqrt{R^2 - Q^3})^{1/3}$

If A=0 $\Longrightarrow$B=0

If A< >0 $\Longrightarrow B = \frac{Q}{A}$

$x_0 = (A + B) - \frac{a}{3}$ real root

$x_1 = \left[-\frac{(A+B)}{2} - \frac{a}{3}\right] + \left[-\frac{\sqrt{3}(A-B)}{2}\right]i$ complex root

$x_2 = \left[-\frac{(A+B)}{2} - \frac{a}{3}\right] - \left[-\frac{\sqrt{3}(A-B)}{2}\right]i$ complex root

Program Root of third degree polynomial (analytical formula-julia version)

```
function cubic_roots(d)
        # roots of a degree 3 polynomial
        # P(x)=a[3]*x^3+a[2]*x^2+a[1]*x+a[0]=0
        # assuming all a is all real
        # if more than 4 values are entered the remaining ones will be ignored
        x=zeros(Float64,2,3)
        x1=zeros(Float64,3)
        x2=zeros(ComplexF64,3)
        a=d[3]/d[4];
        b=d[2]/d[4];
        c=d[1]/d[4];
        R=0.0
        theta=0.0
        Q=(a*a-3.0*b)/9.0
        y1=2.0*a*a*a
        y2=-9*a*b
        y3=27*c
        y4=y1+y2+y3
        R=y4/54.0
        Q3=Q*Q*Q
        R2=R*R
        qq=0.0
        A=0.0
        B=0.0
if(R2<=Q3)
          qq=-2.0*sqrt(Q)
          if(R==0)
            theta=acos(R)
          else
            theta=acos(R/sqrt(Q3))
          end
          x1[1]=qq*cos(theta/3.0)-a/3.0
          x1[2]=qq*cos((theta-2.0*pi)/3.0)-a/3.0
          x1[3]=qq*cos((theta+2.0*pi)/3.0)-a/3.0
          return x1
else
          A=-(R+sqrt(R2-Q3))^(1.0/3.0)
          if(A==0)
                  B=0.0
    else
                  B=Q/A
```

```
        end
                x[1,1]=(A+B)-a/3.0
                x[1,2]=-0.5*(A+B)-a/3
                x[2,2]=sqrt(3.0)/2.0*(A-B)
                x[1,3]=-0.5*(A+B)-a/3
                x[2,3]=-sqrt(3.0)/2.0*(A-B)
                x2[1]=complex(x[1,1],0.0)
                x2[2]=complex(x[1,2],x[2,2])
                x2[3]=complex(x[1,3],x[2,3])
                return x2
end
end


d=[1.0 -3.0 3.0 -2.0]
x=cubic_roots(d)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" cubic_roots.jl
x = ComplexF64[0.5 + 0.0im, 0.5 - 0.8660254037844386im, 0.5 + 0.8660254037844386im]
> Terminated with exit code 0.
```

```
d=[1.0 -3.0 3.0 -1.0]
x=cubic_roots(d)
print("x = $x")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" cubic_roots.jl
x = [1.0, 1.0, 1.0]
> Terminated with exit code 0.
```

Excel version of the program

Program 2.14-4 Root of third degree polynomial (analytical formula) excel version

Root of third degree polynomial (analytical formula) Excel version

$f(x)=d_3x^3+d_2x^2+d_1x+d_0=0$ $\quad f(x)=x^3+ax^2+bx+c=0$

Enter poynomial coefficients

d3	d2	d1	d0
1	3	3	-1

	a	b	c
1	3	3	-1

Q	0
y1	54
y2	-81
y3	-27
y4	-54
R	-1
θ	#SAYI/0!

true/false
0

true if R^2<Q^3

x0	x1	x2
0	0	0

true if R^2>=Q^3

A	0	
B	0	

true/false
1

xo real	xo imaginary
-1	0

x1 real	x1 imaginary
-1	0

x2 real	x2 imaginary
-1	0

Python version of the program:

Tartaglia's Poem

Italian:	English Translation:

Quando chel cubo con le cose appresso *Se agguaglia à qualche numero discreto* *Trouan dui altri differenti in esso.* *Dapoi terrai questo per consueto* *Che'llor produtto sempre sia eguale* *Alterzo cubo delle cose neto,* *El residuo poi suo generale* *Delli lor lati cubi ben sottratti* *Varra la tua cosa principale.* *In el secondo de cotestiatti* *Quando che'l cubo restasse lui solo* *Tu osseruarai quest'altri contratti,* *Del numer farai due tal part'à uolo* *Che l'una in l'altra si produca schietto* *El terzo cubo delle cose in stolo* *Delle qual poi, per communprecetto* *Torrai li lati cubi insieme gionti* *Et cotal somma sara il tuo concetto.* *El terzo poi de questi nostri conti* *Se solue col secondo se ben guardi* *Che per natura son quasi congionti.* *Questi trouai, & non con paβi tardi* *Nel mille cinquecentè, quatroe trenta* *Con fondamenti ben sald'è gagliardi* *Nella citta dal mar'intorno centa.*	01) *When the cube with the cose beside it <x^3+px >* 02) *Equates itself to some other whole number, <=q>* 03) *Find two other, of which it is the difference. <u–v=q>* 04) *Hereafter you will consider this customarily* 05) *That their product always will be equal <uv=>* 06) *To the third of the cube of the cose net.* 3/3, instead of $(p/3)^3$>* 07) *Its general remainder then* 08) *Of their cube sides , well subtracted, <u−−√3−v√3u3−v3>* 09) *Will be the value of your principal unknown. <=x>* 10) *In the second of these acts,* 11) *When the cube remains solo , <$x^3=px+q$>* 12) *You will observe these other arrangements:* 13) *Of the number <q> you will quickly make two such parts, <q=u+v>* 14) *That the one times the other will produce straightforward <uv=>* 15) *The third of the cube of the cose in a multitude, <p^3/3, instead of $(p/3)^3$>* 16) *Of which then, per common precept,* 17) *You will take the cube sides joined together. <u−−√3+v√3u3+v3* 18) *And this sum will be your concept. <=x>* 19) *The third then of these our calculations <$x^3+q=px$>* 20) *Solves itself with the second, if you look well after,* 21) *That by nature they are quasi conjoined.* 22) *I found these, & not with slow steps,* 23) *In thousand five hundred, four and thirty* 24) *With very firm and strong foundations* 25) *In the city girded around by the sea.*

COMPLEX ROOTS OF THE POLYNOMIALS BY UTILISING SYNTHETIC DIVISION PROCESS

Horners method utilises synthetic division process to solve roots of polynomials. Synthetic division process use recursive process to obtain a division for a polynomial. If

$P(x) = f_n(x) = a_0 + a_1x + a_1x^2 + \cdots + a_nx^n$ (2.15.1)

and if this polynomial divides with $(x - x_0)$ to form polynomial

$Q(x) = f_{n-1}(x) = b_1 + b_2x + b_3x^2 + \cdots + b_{n-1}x^{n-1}$ (2.15.2)

Where $P(x) = f_n(x) = b_0 + (x - x_0)f_{n-1}(x)$ or $P(x) = b_0 + (x - x_0)Q(x)$ (2.15.3)

$R = b_0$ is the remainder of the synthetic division process. Of course if the remainder is equal to zero x_0 would be the root of this polynomial. Synthetic division can be calculated as following iterative equation:

$b_n = a_n$ (2.15.4)

$b_i = a_i + b_{i+1}x_0$ (2.15.5)

For example if a polynomial

$f_n(x) = 2x^4 - 3x^2 + 3x - 4$ is given, and x_0 is -2 {divide by (x+2)} synthetic division is applied as

x_0	a_4	a_3	a_2	a_1	a_0
-2	**2**	**0**	**-3**	**3**	**-4**
	b_4	b_3	b_2	b_1	b_0
	2	**-4**	**5**	**-7**	**10**

$Q(x) = f_{n-1}(x) = 2x^3 - 4x^2 + 5x - 7$

$P(x) = f_n(x) = 10 + (x + 2)Q(x)$

Differentiating of $f_n(x)$with respect to x gives

$f'_n(x) = f_{n-1}(x) + (x - x_0)f'_{n-1}(x)$ (2.15.6)

and at x=x_0

$f_n(x) = b_0$

$f'_n(x) = f_{n-1}(x) = Q(x)$ (2.15.7)

Therefore Newton - Raphson iterative method can be applied to find the roots by using iterative process

$x_n = x_{n-1} - \frac{f_n(x_{n-1})}{f'_n(x_{n-1})} = x_{n-1} - \frac{b_0}{f'_n(x_{n-1})}$ (2.15.8)

$x_n = x_{n-1} - \frac{b_0}{Q(x_{n-1})}$ (2.15.9)

xo	a4	a3	a2	a1	a0
-2	2	0	-3	3	-4
	2	-4	5	-7	10

Q	-49
x1	-1.7959184

x1	a4	a3	a2	a1	a0
-1.7959184	2	0	-3	3	-4
	2	-3.5918367	3.4506456	-3.1970777	1.7416906

Q	-32.563821
x2	-1.7424329

x2	a4	a3	a2	a1	a0
-1.7424329	2	0	-3	3	-4
	2	-3.4848658	3.0721449	-2.3530065	0.0999559

Q	-28.866623
x3	-1.7389702

x3	a4	a3	a2	a1	a0
-1.7389702	2	0	-3	3	-4
	2	-3.4779405	3.048035	-2.3004421	0.0004003

Q	-28.63559
x4	-1.7389563

x4	a4	a3	a2	a1	a0
-1.7389563	2	0	-3	3	-4
	2	-3.4779125	3.0479377	-2.3002304	6.505E-09

Q	-28.634659
x5	-1.7389563

x5	a4	a3	a2	a1	a0
-1.7389563	2	0	-3	3	-4
	2	-3.4779125	3.0479377	-2.3002304	0

Q	-28.634659
x5	-1.7389563

Note that roots of a polynomial can be complex numbers, therefore it is better to calculate the roots as complex numbers. It should be also note that after reducing polynomial to a third, second or first degree remaining roots can be calculated by using standard root finding equations. An example program for Horner's method is listed below (complex class in java language was defined previously) :

Program Complex Horner method example to find the roots of a polynomial horner.jl

```
function f_poly(c1,x)
    # this function calculates the value of
    # c least square curve fitting function
    n=size(c1)[1]
    if n!=0.0
      ff=c1[n]
      for i=n-1:-1:1
        ff=ff*x+c1[i]
      end
    else
      ff=0.0
    end
    return ff
end

function horner(a)
  # roots of a polynomial by using synthetic division process
    n=size(a)[2]
    x1=-2.0
    n1=n
    nmax=20
    tolerance=1.0e-10
    n2=n-1
    ref=3.1
    ii=0
    global y=zeros(ComplexF64,n2)
    for k=1:n2
          x=x1
          xold=3.4878
          ref=abs(xold-x)
          b=zeros(ComplexF64,n1)
          global c=zeros(ComplexF64,n2)
          b[n]=a[n]
          xx=b[n]
          ii=0
          # find the root in while
          while ref>tolerance && ii<nmax
            for i=n2:-1:1
              b[i]=a[i]+b[i+1]*x
              xx=b[i]
            end # for
            for i=n2:-1:1
              c[i]=b[i+1]
            end # for
            Q=f_poly(c,x)
            P=b[1]
            x=x-P/Q
            ref=abs(xold-x)
            ii=ii+1
            xold=x
          end # while
          n-=1
          a=c
          y[k]=x
    end # for
return y
end  # function

a=[-4 3 -3 0 2]
print(horner(a))
```

$$f(x) = -4x^4 + 3x^3 - 3x^2 + 2$$

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" horner.jl
ComplexF64[-1.738956256451892 + 0.0im, 1.254881884834291 - 0.0im, -0.6819503221875776 + 0.0im, 2.474337965776697 - 0.0im]
> Terminated with exit code 0.
```

$$f(x) = x^3 - 3x^2 + 3x - 1$$

```
a=[1 -3 3 -1]
print(horner(a))
```

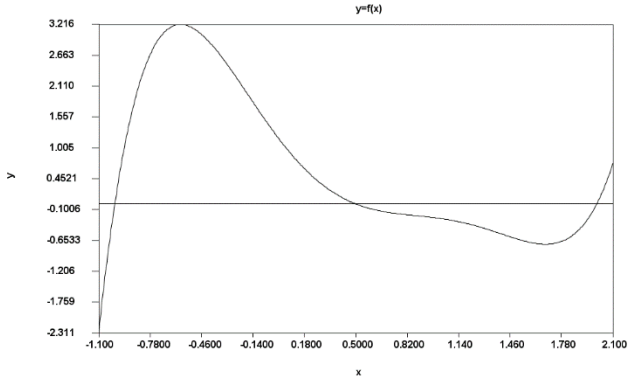
```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" horner.jl
```

```
ComplexF64[1.0000030176099868 + 0.0im, 0.9999963232971536 + 0.0im, 1.0000040546033393 + 0.0im]
> Terminated with exit code 0.
```

$$f(x) = x^5 - 3.5x^4 + 2.75x^3 + 2.125x^2 - 3.875x + 1.25$$

```
a=[1.25 -3.875 2.125 2.75 -3.5 1]
print(horner(a))
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" horner.jl
ComplexF64[-1.0 - 0.0im, 0.5000000000002416 + 0.0im, 1.999999999999984 + 0.0im, 1.37720103480294 + 0.0im,
1.2491220612554996 - 0.0im]
> Terminated with exit code 0.
```



Now let us find roots of $f(x) = x^3 + 9813.18x^2 + 8571.08x + 0.781736 = 0$ by using horner method

```
a=[0.781736 8571.08 9813.18 1]
print(horner(a))
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" horner.jl
ComplexF64[
-0.8734118615170947 + 0.0im, -9.121576846334152e-5 - 0.0im, -9812.30649692273 - 0.0im]
> Terminated with exit code 0.
```

ROOT FINDING BY USING SOFTWARE PACKAGES

In order to find roots, some built in functions already existed in package math programs. In Matlab-Octave function **fzero** is existed to find roots of a function and **roots** to find roots of a polynomial

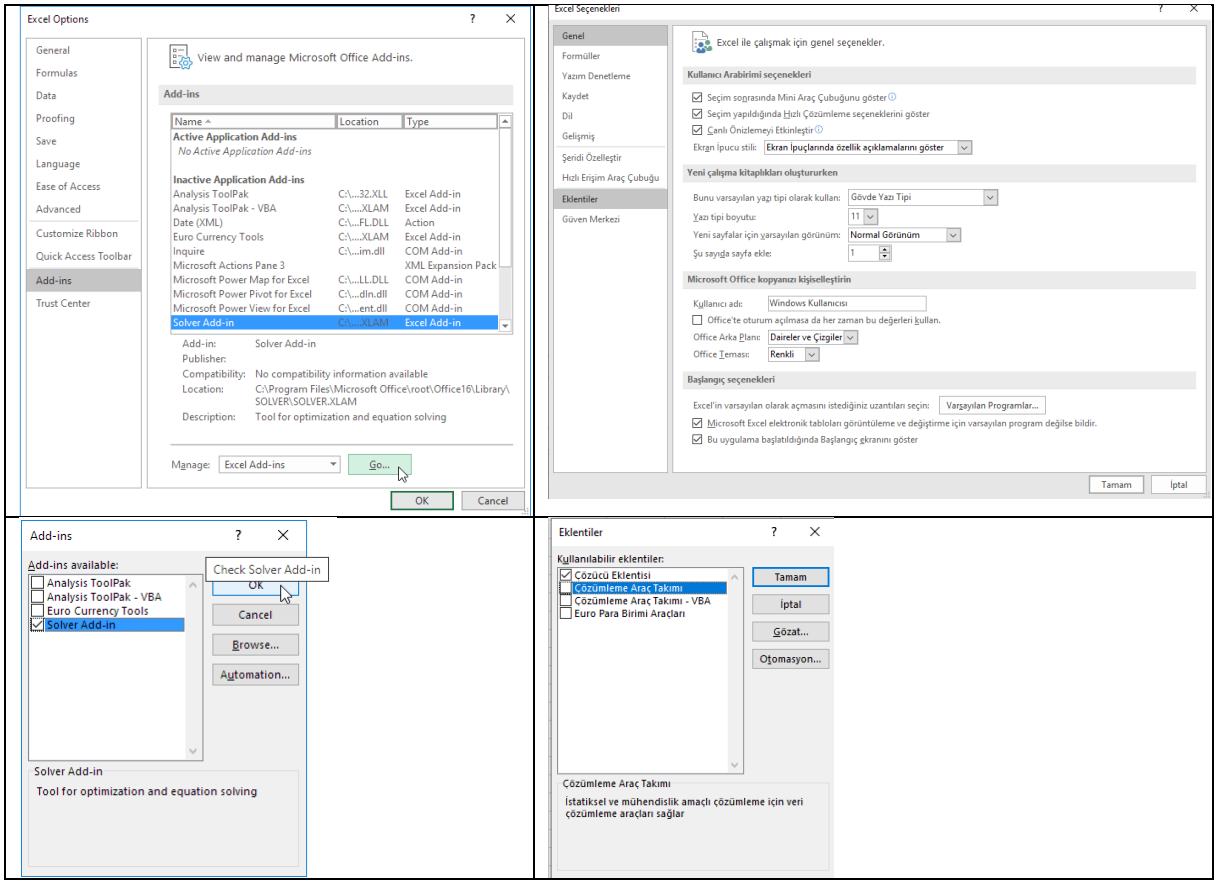
```
>> format long
>> x=2;
>> y=@(x)x*x-2;
>> fzero(y,x)
ans =  1.41421356237310
>>
```

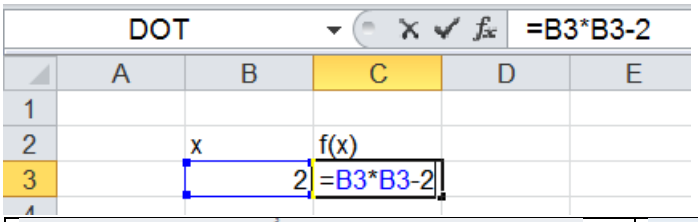
```
>> x=1;
>> y=@(x)x^5-3.5*x^4+2.75*x^3+2.125*x^2-3.875*x+1.25;
>> fzero(y,x)
ans =  0.500000000000001
```

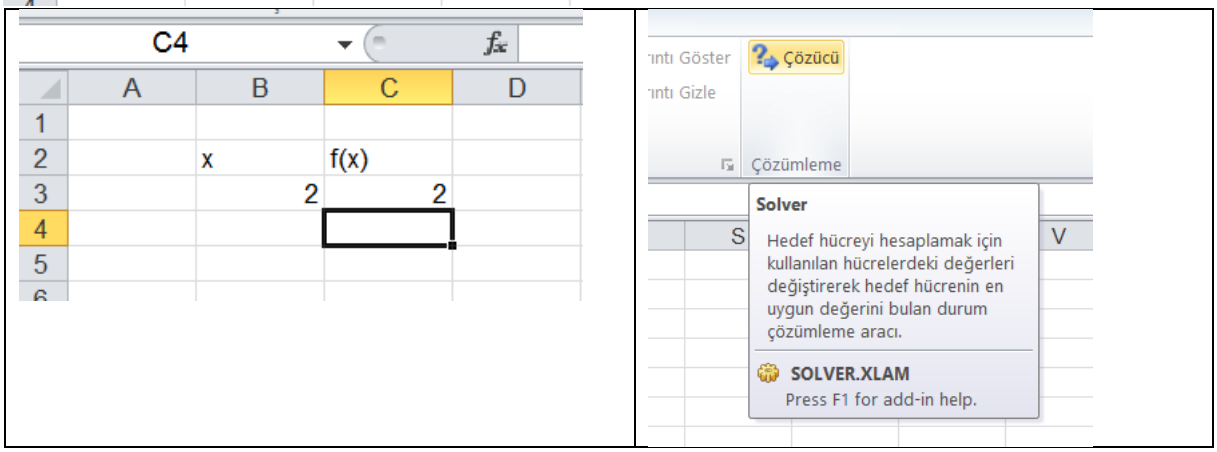
```
>> a=[1 -3.5 2.75 2.125 -3.875 1.25];
>> roots(a)
ans =
   2.000000000000003 + 0.000000000000000i
  -1.000000000000002 + 0.000000000000000i
   1.000000000000000 + 0.500000000000000i
   1.000000000000000 - 0.500000000000000i
   0.499999999999999 + 0.000000000000000i
```

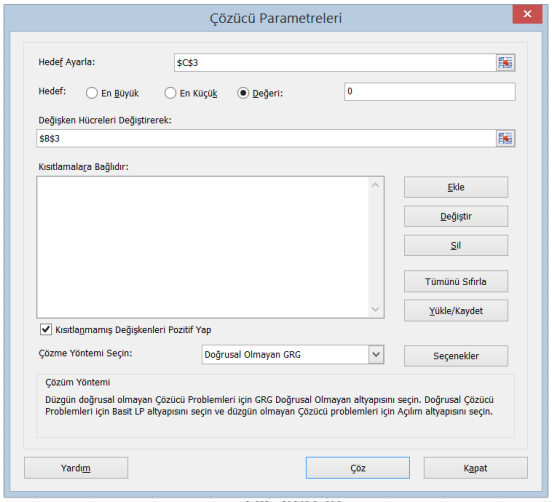
```
>> y=@(x)x*x+2;
>> x=1+1i;
>> fsolve(y,x)
ans =
 -0.000000000000016 + 1.414213562374690i
```

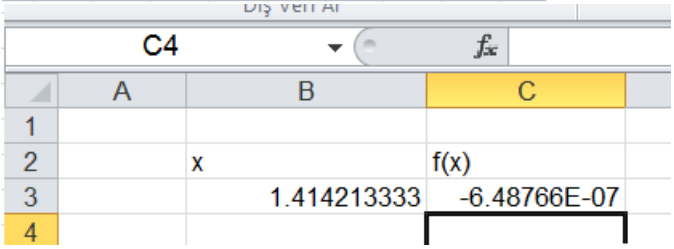
In excel, solver is existed to find real roots: We can use solver add in. In order to open solver:
On the File tab, click Options. (in turkish version dosya, seçenekler)
Under Add-ins(in turkish version eklentiler), select Solver Add-in (in turkish version Çözücü eklentisi), and click on the Go(Git..) button. You can find the Solver on the Data tab(Veri), in the Analyze group.









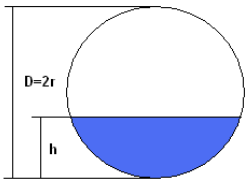


2.18 PROBLEMS

PROBLEM 1 Find the roots of $\boldsymbol{f(x) = x^2 - 0.9x - 8.5}$ by using bisection method

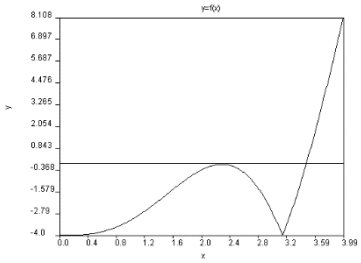
PROBLEM 2 Calculate $\sqrt{7}$ by using one of the root finding method

PROBLEM 3

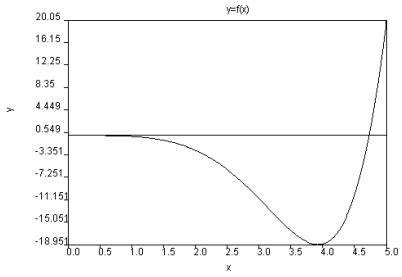


A spherical tank of radius r is filled with a liquid of height h. The Volume of liquid (V) in the tank is given by he following eqaution $\boldsymbol{V = \pi h^2(3r - h)/3}$
If r = 1 m and V= 0.5 m^3 find the liquid height
Note : due to geometry of a sphere h can not be less than 0 and can not be bigger than D=2r. The maximum amount of liquid ths spherical container can be take is 4.18879 m^3 (you do not need this data to solve problem)

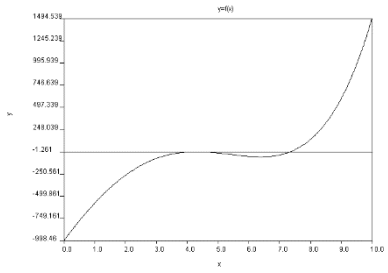
PROBLEM 4 Find the roots of $f(x) = x^2|sin(x)| - 4.$ by using one of the root finding method



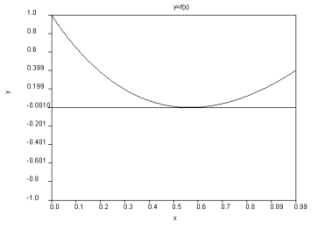
PROBLEM 5 Find the roots of $f(x) = \cos(x)\cosh(x) - 1$ by using one of the root finding method



PROBLEM 6 Find the roots of $f(\boldsymbol{x}) = x^4 - 8.6x^3 - 35.5x^2 + 464.4x - 998.46$ by using one of the root finding method Try x_0=4 and $\boldsymbol{7 \leq x \leq 8}$

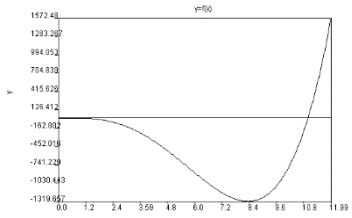


PROBLEM 7 Function $\boldsymbol{f(x)} = x^2 - 2e^{-x} + e^{-2x}$ has only one real root. Find the root bu using Newton-Raphson method. Use as starting value x_0=1

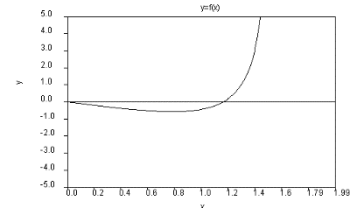


PROBLEM 8

Find all real roots of function $f(\boldsymbol{x}) = x^4 - 7.223x^3 + 13.4475x^2 - 0.672x - 10.223$



PROBLEM 9 Find roots of functionf $(\boldsymbol{x}) = \tan(\mathrm{x}) - 2x$.



PROBLEM 10 For Turbulent flow inside of a pipe (Re>2300) friction coefficient inside of a pipe can be calculated by using Colebrook equation.

$$\frac{\mathbf{1}}{\sqrt{\boldsymbol{f}}} = -2.0log_{10}\left[\frac{\varepsilon/D}{3.7} + \frac{2.51}{Re\sqrt{f}}\right]$$

Re is Reynolds number, ε is surface rougness. Write a program to calculate total pressure drop in the pipe for a given Re number, pipe diameter, pipe length and local pressure drop coefficient.

f : friction coefficient

D pipe diameter

Hint : Change the colebrook equation into the form

$$F(x) = X + 2.0log_{10}\left[\frac{\varepsilon/D}{3.7} + \frac{2.51X}{Re}\right] = 0$$

Then find the root of the equation where $X = \frac{1}{\sqrt{f}}$

Use Haaland equation to obtaing the first estimate of the solution which has the form of

$$\frac{\mathbf{1}}{\sqrt{\boldsymbol{f}}} = -1.8log_{10}\left[\frac{(\varepsilon/D)^{1.11}}{3.7} + \frac{6.9}{Re}\right]$$

PROBLEM 11

In round smooth tubes for turbulent flow region Fanning friction factor is given as:

$$\frac{\mathbf{1}}{\sqrt{\boldsymbol{f}}} = 1.737\ln\left[Re\sqrt{f}\right] - 0.4$$

This equation is valid for $4 * 10^3 < Re < 3 * 10^6$ By using one of the root finding methods, calculate friction factor for Re=5000. You can utilize any root finding method you learned so far.

Note : You can assume that f is changing between 0,001 and 0.01

PROBLEM 12

van der Waals equation of state for the gases :

$P = \frac{RT}{v-b} - \frac{a}{v^2}$ In this equation

$a = \frac{27}{64}\frac{R^2T_c^2}{P_c}$

$b = \frac{RT_c}{8P_c}$

Redlich ve Kwong equation of states

$P = \frac{RT}{v-b} - \frac{a}{v(v+b)T^{1/2}}$ In this equation

$a = 0.42748\frac{R^2T_c^{2.5}}{P_c}$

$b = 0.08664\frac{RT_c}{P_c}$

O2, for oxygen P_c=5.043 MPa T_c=154.58 K M=31.999 kg/kmol.

N2, for nitrogen P_c=3.39 MPa T_c=126.2 K M=28.013 kg/kmol.

R=8314.5 J/kmolK

Benedict -Webb-Rubin (BWR) equation of state

$$P = \frac{RT}{v} + \frac{RTB_0 - A_0 - C_0/T^2}{v^2} + \frac{RTb - a}{v^3} + \frac{a\alpha}{v^6} + \frac{c}{v^3}\left(1 + \frac{\gamma}{v^2}\right)e^{-\frac{\gamma}{v^2}}$$

Gas	A_0	B_0	$C_0.10^{-6}$	A	B	$c.10^{-6}$	$\alpha.10^3$	$\gamma.10^3$
O_2	1.49880	0.046524	0.0038617	-0.040507	-0.00027963	-0.00020376	0.008641	0.359
N_2	1.1925	0.0458	0.0058891	0.01490	0.00198154	0.000548064	0.291545	0.75

Develop a program to calculate v when T and P is given. By using

A) van der Waals equation of state : gas : oxygen and nitrogen
B) Redlich Kwong equation of state : gas : oxygen and nitrogen
C) Benedict-webb rubin equation of state : gas : oxygen
D) Benedict-webb equation of state : gas : nitrogen
E) Compare the result with ideal gas equation $P = \frac{RT}{v}$

PROBLEM 13

Find the root of function $f(x)=x\,e^{-x/2} + x^3 + 1$ with the limits of x_1=-1, x_2=0, x_3=1 ny using muller method.

PROBLEM 14

Find the root of function $f(x)=x^4-20x^3-25x^2+100x+130$ by using Leguerre method.

PROBLEM 15

Find the root of fuction $f(x)=x^2+1$.

PROBLEM 16

For a paralel flow heat exchanger Number of transfer unit(NTU) and effectiveness (ε) equation given as follows

$$\varepsilon = \frac{1 - exp[-NTU(1 + C^*)]}{(1 + C^*)}$$

After the experiments for a specific heat exchanger NTU=1.989 and ε = 0.574 is found from the experiment

a) Determine the range C^* . For this start C^*=0.1 and increase 0.1 in each step
b) Calculate C^* by using false position method.

PROBLEM 17

Two end of 5 cm long plate with crossection of 0.1cm x 20 cm plate is held at 80 °C ve 60 °C. The environmental temperature is T_∞ = 10 °C, convective heat transfer coefficient h=15 W/m^2 K. Bar cross sectional area A=2×10^{-4} m^2, and perimeter P=0.402 m.In the experiment mid point(x=0.025 m) temperature is found as T=67.5 °C. In this conditions, temperature distribution in the bar is given as

$$T - T_\infty = 68.7e^{-0.025m} + 11.3e^{0.025m}$$

In this equation m is defined as: $m = (\frac{hP}{kA})^{0.5}$

a) Find the range of m, stating from m=5, and increasing 5 at each step.
b) Solve m by using Newton-Raphson method
c) Determine thermal conductivity coefficient k from this value.

PROBLEM 18

The cost of the heat exchanger(M), is defined as a function of the length of heat excahger, L, diameter of the heat exchanger, D as**:** ***M=900 + 1100 D^{2,5} L + 320 D L*** In the optimization study the length of the heat exchanger is found as L=1.3 m and the cost as M=1900 TL. Find the diameter of the heat exchanger.

PROBLEM 19

An instrument 20000 TL worth is purcahased with no first payment and 4000 TL annual payment for 6 years. Calculate the interest rate applied by using secant method.

$$A = P\frac{i(i+1)^n}{(i+1)^n - 1}$$

In the equation P is the present worth, A is the annual payment, i is the interestrate, n is the number of payment years.

PROBLEM 20

In order to heat a substance to a required temperature the energy equation is given as
$Q = \int mC_p dT$ eşitliği ile bulunmaktadir.

A substance with an initial temperature of 30 °C, with a mass m=0.2 kg, is heated to some temperature and total energy spend is calculated as 3069.7 kJ . The specific heat of the substance is given with the equation C_p = 41.87+ 1.6748* T + 0.007537* T^2 (kJ/kg K). Find the final temperature by using bi-section method.

PROBLEM 21

Find the roots of f(x) = x^3-6x^2+11x-6.1 by using Legurre method

PROBLEM 22 Find the root(s) of the following equation in the range of [-3.5,3.5]

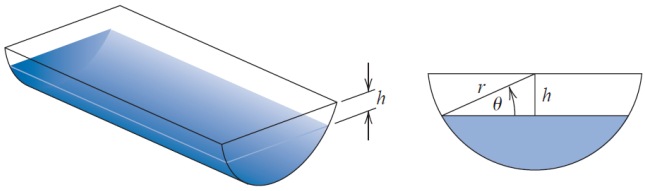
$$f(x) = \frac{1}{\sqrt{2\pi}}e^{-(\frac{1}{2})x^2} + \frac{1}{10}\sin(\pi x)$$

PROBLEM 23 Find the root(s) of the following equation

$$f(x) = \left[10x\left(1 - e^{-\frac{0.0004}{2000x}}\right) - 0.00001\right]$$

PROBLEM 24 In celestial mechanics, **Kepler's equation** is important. It reads $\boldsymbol{x = y - \varepsilon \sin y}$, in which x is a planet's mean anomaly, y its eccentric anomaly, and ε the eccentricity of its orbit. Taking ε = 0.9, construct a table of y for 30 equally spaced values of x in the interval 0 to π. Use Newton's method to obtain each value of y. The y corresponding to an x can be used as the starting point for the iteration when x is changed slightly.

PROBLEM 25 A trough of length L has a cross section in the shape of a semicircle with radius r. When filled with water to within a distance h of the top, the volume V of water is



$$V = L\left[0.5\pi r^2 - r^2 \arcsin\left(\frac{h}{r}\right) - h(r^2 - h^2)^{1/2}\right]$$

Suppose L=1 m, r=0.25 m and V=0.05 m^3 Find the depth of water**,** h

PROBLEM 26 Following functions and root values are given. Investigate number of iteration they are taken when Newton, Halley, Abbasbandy, Thiele 1 and Thiele 2 methods are used.

$f(x) = x^3 + 4x^2 - 25 \qquad x^* = 2.0352684811\ 82$

$f(x) = x^3 - e^x - 3x + 2 \qquad x^* = 0.2575302854\ 39$

$f(x) = x^3 - 10 \qquad x^* = 2.1544346900\ 32$

$f(x) = \cos(x) - x \qquad x^* = 0.7390851332\ 15$

$f(x) = \sin^2(x) - x^2 + 1 \qquad x^* = 1.4044916482\ 15$

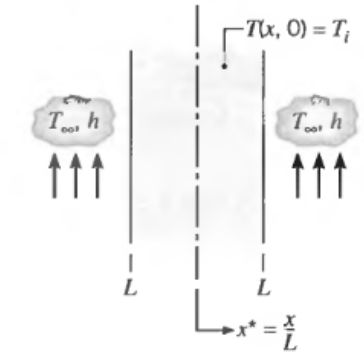
$f(x) = x^2 + \sin(x/5) - 1/4 \qquad x^* = 0.4099920179\ 89$

PROBLEM 27 (a) Use bisection to find the smallest positive zero of the function $f(x) = \cot(3x) - \dfrac{x^2 - 1}{2x}$

(b) Did you remember to plot the function before using bisection?

c) repeat using the root by Newton-Raphson, Illinois, Halley, Abbasbandy , Thiele1 and Thiele 2 methods. Find the number of iteration it takes in each case.

PROBLEM 28 In time dependent heat transfer, heat transfer of a Wall where both sides are exposed to convective heat transfer with convective heat transfer coefficient h and temperature T_∞ is shown in the figure (reference Fundamentals of heat and mass transfer Incropera, DeWitt, Bergman**)**



Time dependent temperature difference for such a wall is given by the equation

$$\theta_0^* = \frac{T_0 - T_\infty}{T_i - T_\infty} = \sum_{n=1}^{\infty} C_n \exp(-\xi_n^2 Fo)\cos(\xi_n x^*)$$

Where $Fo = \dfrac{\alpha t}{L^2}$ is Fourier number t is time(second), $\alpha = \dfrac{k}{\rho C_p}$ is thermal diffisuvity coefficient, L is the half length of the Wall. In order to solve coefficients ξ_n (eigenvalues) the following equation should be solved.

$\tan\xi_n - \dfrac{Bi}{\xi_n} = 0$ where $Bi = \dfrac{hL}{k}$ is Biot number, h is thermal covection coefficient and k is thermal conductivity of the Wall material. Assume Bi number is given and developed an algorithm to solve first 4 roots

of the ξ_n. In order to help you The first 4 roots of the equation for various Bi number are given in the following table.

$Bi = \frac{hL}{k}$	ξ_1	ξ_2	ξ_3	ξ_4
0	0	3.1416	6.2832	9.4248
0.001	0.0316	3.1419	6.2833	9.4249
0.002	0.0447	3.1422	6.2835	9.4250
0.004	0.0632	3.1429	6.2838	9.4252
0.006	0.0774	3.1435	6.2841	9.4254
0.008	0.0893	3.1441	6.2845	9.4256
0.01	0.0998	3.1448	6.2848	9.4258
0.02	0.1410	3.1479	6.2864	9.4269
0.04	0.1987	3.1543	6.2895	9.4290
0.06	0.2425	3.1606	6.2927	9.4311
0.08	0.2791	3.1668	6.2959	9.4333
0.1	0.3111	3.1731	6.2991	9.4354
0.2	0.4328	3.2039	6.3148	9.4459
0.3	0.5218	3.2341	6.3305	9.4565
0.4	0.5932	3.2636	6.3461	9.4670
0.5	0.6533	3.2923	6.3616	9.4775
0.6	0.7051	3.3204	6.3770	9.4879
0.7	0.7506	3.3477	6.3923	9.4983
0.8	0.7910	3.3744	6.4074	9.5087
0.9	0.8274	3.4003	6.4224	9.5190
1.0	0.8603	3.4256	6.4373	9.5293
1.5	0.9882	3.5422	6.5097	9.5801
2.0	1.0769	3.6436	6.5783	9.6296
3.0	1.1925	3.8088	6.7040	9.7240
4.0	1.2646	3.9352	6.8140	9.8119
5.0	1.3138	4.0336	6.9096	9.8928

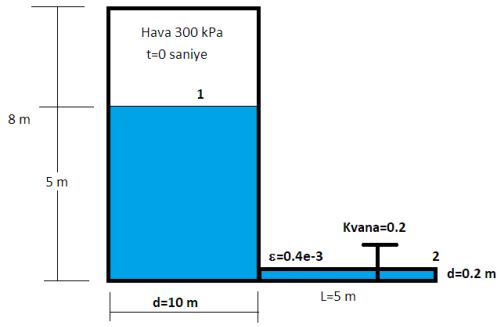
PROBLEM 29 Solve equation $3^x x^3 = 1$

PROBLEM 30 Calculate all roots of the following polynomials

$$f(x) = x^3 - x^2 + 2x - 2$$

$$f(x) = 3.704x^3 + 16.3x^2 - 21.97x + 9.34$$

PROBLEM 31A water container filled to H=5 m level. Total height of the container is HT=8 meters. At the top of water air with 300 kPa and 27 C existed. The top of the storage tank is closed. Diameter of the container is 10 m. Water is discharged through a pipe of d=0.2 m in diameter and L=5 m in length. There is a globe valve at this pipe (KL=0.2). Calculate velocity of the discharge water as a function of time



NONLINEAR SYSTEM OF EQUATIONS (ROOTS OF SYSTEM OF EQUATIONS)

NEWTON-RAPHSON METHOD

One dimensional Newton-Raphson Formula was:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad (4.1.1)$$

If the equation format changed a little, it can be written in the form of

$$\Delta x_n = x_{n+1} - x_n = -\frac{f(x_n)}{f'(x_n)} \qquad (4.1.2)$$

$$f'(x_n)\Delta x_n = -f(x_n) \qquad (4.1.3)$$

Now the same equation can be considered for a system of non-linear equation

$$[\nabla f] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \quad (4.1.4) \qquad \{\Delta x\} = \begin{Bmatrix} \Delta x_1 \\ \Delta x_2 \\ \cdots \\ \Delta x_n \end{Bmatrix} \quad \{f\} = \begin{Bmatrix} f_1 \\ f_2 \\ \cdots \\ f_n \end{Bmatrix} \quad (4.1.5)$$

So multidimensional Newton-Raphson equation becomes

$$[\nabla f]\{\Delta x\} = -\{f\} \qquad (4.1.6)$$

The equation can be solved by using tecqniques such as gauss elimination. An initial estimate for all the x values are required to start iterative solution

First example function

$f_0(x_0,x_1)=x_0^2+ x_0\ x_1-10$

$f_1(x_0,x_1)=x_1+3\ x_0\ x_1^2-57$

$$[\nabla f] = \begin{bmatrix} 2x_0 + x_1 & x_0 \\ 3x_1^2 & 1 + 6x_0x_1 \end{bmatrix}$$

PROGRAM 4.1-1 NA29 : Newton Raphson method both function and derivatives are defined

```
 #function set
function func(x)
# system of non-linear equation
ff=zeros(Float64,2)
ff[1]=x[1]*x[1]+x[1]*x[2]-10.0
ff[2]=x[2]+3.0*x[1]*x[2]*x[2]-57.0
return ff
end

#single function with reference number i
function fi(f,x,i)
# system of non-linear equation
y=f(x)
return y[i]
end

#first derivative
function dfunc(func,x)
 global  dx=0.001
 global c=[1.0/280.0 -4.0/105.0          1.0/5.0    -4.0/5.0  0          4.0/5.0    -1.0/5.0   4.0/105.0 -1.0/280.0]
 n=length(x)
 ff=zeros(Float64,n,n)
 global yy=ff[1]
 x1=zeros(Float64,n)
 for ieqn=1:n
  for ix=1:n
    for i=-4:4
     # create x1set for derivative input
     # ================================
     for k=1:n
       xx=x[k]
       if k!=ix
         x1[k]=xx
       else
         delta=Float64(i)*dx
         x1[k]=xx+delta
       end
      end # for k
      # ================================
      ff1=fi(func,x1,ieqn)
```

```
        ff[ieqn,ix]+=c[i+5]* ff1
        yx=ff[ieqn,ix]
      end # for i
      ff[ieqn,ix]/=dx
      yx=ff[ieqn,ix]
   end # for ix
 end # for ieqn
 return ff
end # for function

function gauss_pivot(A1,b)
            n=size(A1)[1]
            carpan=0.0
            toplam=0.0
            global x=Array{Float64}(undef,n)
            global a=Array{Float64}(undef,n,n)
            for i=1:n
              for j=1:n
                a[i,j]=A1[i,j]
              end
            end
            for k=1:n
            #pivoting
               p=k
               buyuk=abs(a[k,k])
               for ii=k+1:n
                 dummy=abs(a[ii,k])
                 if dummy > buyuk
                    buyuk=dummyp=ii
                 end #if dummy
               end #for ii
               if p!=k
                 for jj=k:n
                   dummy=a[p,jj]
                   a[p,jj]=a[k,jj]
                   a[k,jj]=dummy
                   end #for jj

                 dummy=b[p]
                 b[p]=b[k]
                 b[k]=dummy
               end #if p!
            # gauss elimination
   for i=k+1:n
               carpan=a[i,k]/a[k,k]
               a[i,k]=0.0
               for j=k+1:n
                 a[i,j]=a[i,j]-carpan*a[k,j]
               end #for j
               b[i]=b[i]-carpan*b[k]
            end #for i
            end #k
                        # back substituting
              x[n]=b[n]/a[n,n]
              n1=n-1
              for i=n1:-1:1
                toplam=0.0
                for j=i+1:n
                  toplam=toplam+a[i,j]*x[j]
                end #for j
                x[i]=(b[i]-toplam)/a[i,i]
              end #for i
```

```
            return x
end #function

function newton(func,x)
  # Newton-Raphson root finding method
  k=length(x)
  nmax=100
  tolerance=1.0e-10
  for i=1:nmax
    fx=func(x)
    dfx=dfunc(func,x)
    dx=gauss_pivot(dfx,fx)
    for j=1:k
      x[j]=x[j]-dx[j]
    end # for j
    total=0.0;
    for j=1:k
      total+=fx[j]
    end # for j
    if abs(total)<tolerance
     return x
    end
  end #for i
  return x
end # function


x=[1.0 1.0]
y=func(x)
dy=dfunc(func,x)
println("x = $x \ny = $y dy = $dy")
r=newton(func,x)
println("r = $r")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" NL_newton.jl
x = [1.0 1.0]
y = [-8.0, -53.0] dy = [2.999999999998608 1.0000000000000218; 2.9999999999981983 6.999999999992484]
r = [2.0 3.0]

> Terminated with exit code 0.
```

If the same problem is solved by hand starting from the initial guess x={1,2}

Newton-Raphson Non-linear equation solution

Initial guess

x0	1
x1	2

	4	1	dx0	7
	12	13	dx1	43

3	4	1	dx0	7
	0	10	dx1	22

dx0	1.2
dx1	2.2

First iteration result

x0	2.2
x1	4.2

	8.6	2.2	dx0	-4.08
	52.92	56.44	dx1	-63.624

6.153488	8.6	2.2	dx0	-4.08
	0	42.90233	dx1	-38.5178

dx0	-0.24475
dx1	-0.8978

Second iteration results

x0	1.955252
x1	3.302199

	7.212702	1.955252	dx0	-0.27964
	32.71355	39.73977	dx1	-10.2654

4.535547	7.212702	1.955252	dx0	-0.27964
	0	27.70816	dx1	-8.54467

dx0	0.044827
dx1	-0.30838

Third iteration results

x0	2.000079
x1	2.993818

	6.993975	2.000079	dx0	0.011814
	26.88883	36.92723	dx1	0.2264

3.844571	6.993975	2.000079	dx0	0.011814
	0	29.23778	dx1	0.180979

dx0	-8.1E-05
dx1	0.00619

x0	1.999998
x1	3.000008

	7.000003	1.999998	dx0	4.94E-07
	27.00014	37.00005	dx1	-0.00022

3.857161	7.000003	1.999998	dx0	4.94E-07
	0	29.28574	dx1	-0.00022

dx0	2.24E-06
dx1	-7.6E-06

Fourth iteration results

x0	2
x1	3

	7	2	dx0	1.2E-11
	27	37	dx1	-4E-11

3.857143	7	2	dx0	1.2E-11
	0	29.28571	dx1	-8.6E-11

dx0	2.56E-12
dx1	-2.9E-12

Fifth iteration results

x0	2
x1	3

The problem steps including numerical derivative is evaluated step by step as :

Newton-Raphson Non-linear equation solution

Initial guess

x0	1
x1	2

	4.0000	1.0000	dx0	7.0000
	12.0000	13.0000	dx1	43.0000

3	4	1	dx0	7
	0	10	dx1	22

dx0	1.2
dx1	2.2

dx0	0.01					
x0+dx0	1.01	f1(x0+dx0,x1)	-6.9599	f2(x0+dx0,x1)	-42.88	
x0+2dx0	1.02	f1(x0+2dx0,x1)	-6.9196	f2(x0+2dx0,x1)	-42.76	
x0-dx0	0.99	f1(x0-dx0,x1)	-7.0399	f2(x0-dx0,x1)	-43.12	
x0-2dx0	0.98	f1(x0-2dx0,x1)	-7.0796	f2(x0-2dx0,x1)	-43.24	
x1+dx1	2.01	f1(x0,x1+dx1)	-6.99	f2(x0,x1+dx1)	-42.8697	
x1+2dx1	2.02	f1(x0,x1+2dx1)	-6.98	f2(x0,x1+2dx1)	-42.7388	
x1-dx1	1.99	f1(x0,x1-dx1)	-7.01	f2(x0,x1-dx1)	-43.1297	
x1-2dx1	1.98	f1(x0,x1-2dx1)	-7.02	f2(x0,x1-2dx1)	-43.2588	

First iteration result

x0	2.2
x1	4.2

	8.6000	2.2000	dx0	-4.0800
	52.9200	56.4400	dx1	-63.6240
6.153488	8.6000	2.2000	dx0	-4.0800
	0	42.90233	dx1	-38.51777

dx0	-0.244748
dx1	-0.897801

dx0	0.01									
x0+dx0	2.21	f1(x0+dx0,x1)	4.1661	f2(x0+dx0,x1)	64.1532					
x0+2dx0	2.22	f1(x0+2dx0,x1)	4.2524	f2(x0+2dx0,x1)	64.6824					
x0-dx0	2.19	f1(x0-dx0,x1)	3.9941	f2(x0-dx0,x1)	63.0948					
x0-2dx0	2.18	f1(x0-2dx0,x1)	3.9084	f2(x0-2dx0,x1)	62.5656					
x1+dx1	4.21	f1(x0,x1+dx1)	4.102	f2(x0,x1+dx1)	64.18906					
x1+2dx1	4.22	f1(x0,x1+2dx1)	4.124	f2(x0,x1+2dx1)	64.75544					
x1-dx1	4.19	f1(x0,x1-dx1)	4.058	f2(x0,x1-dx1)	63.06026					
x1-2dx1	4.18	f1(x0,x1-2dx1)	4.036	f2(x0,x1-2dx1)	62.49784					

Second iteration results

x0	1.955252
x1	3.302199

	7.2127	1.9553	dx0	-0.279637
	32.7135	39.7398	dx1	-10.26541
4.535547	7.212702	1.955252	dx0	-0.279637
	0	27.70816	dx1	-8.544666

dx0	0.044827
dx1	-0.308381

dx0	0.01									
x0+dx0	2.0101	f1(x0+dx0,x1)	0.058225	f2(x0+dx0,x1)	0.042488					
x0+2dx0	2.0201	f1(x0+2dx0,x1)	0.128465	f2(x0+2dx0,x1)	0.311377					
x0-dx0	1.9901	f1(x0-dx0,x1)	-0.081654	f2(x0-dx0,x1)	-0.495288					
x0-2dx0	1.9801	f1(x0-2dx0,x1)	-0.151294	f2(x0-2dx0,x1)	-0.764177					
x1+dx1	3.0038	f1(x0,x1+dx1)	0.008186	f2(x0,x1+dx1)	0.143472					
x1+2dx1	3.0138	f1(x0,x1+2dx1)	0.028187	f2(x0,x1+2dx1)	0.514545					
x1-dx1	2.9838	f1(x0,x1-dx1)	-0.031815	f2(x0,x1-dx1)	-0.595072					
x1-2dx1	2.9738	f1(x0,x1-2dx1)	-0.051816	f2(x0,x1-2dx1)	-0.962544					

Third iteration results

x0	2.000079
x1	2.993818

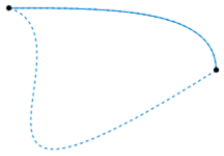
	6.9940	2.0001	dx0	0.011814
	26.8888	36.9272	dx1	0.2264
3.844571	6.993975	2.000079	dx0	0.011814
	0	29.23778	dx1	0.180979

dx0	-8.09E-05
dx1	0.00619

x0	1.999998
x1	3.000008

HOMOTOPY OR CONTINUATION METHODS

In topology, two continuous functions from one topological space to another are called **homotopic**(from Greek ὁμός homós "same, similar" and τόπος tópos "place") if one can be "continuously deformed" into the other, such a deformation being called a **homotopy** between the two functions.



When a problem of system of nonlinear equations of the form F(x)=0 desired to be solved, another interesting solution method is Homotopy or Continuation method. Assume that solution set to be found is x*. Consider a parametric function G(λ,x) in the form of

$G(\lambda,x) = \lambda F(x) + (1-\lambda)[\, F(x) - F(x(0)) \,]$ (4.3.1)

Where λ=0 coresponds to initial guess of the solution , x(0) ,and where λ=1 value correspods the actual solution set x(1)= x* (4.3.2)

It is desired to be found G(λ,x) = 0 therefore for λ=0 equation becomes

$G(\lambda,x) = G(0,x) = F(x) - F(x(0))$ and for λ=1 (4.3.4)

$0 = G(1,x) = F(x)$ (4.3.4)

Therefore x(1)=x* solution set will be obtained. If a function G(λ,x) satisfies the above equation can be found, it will also give us the solution. Function G is called a homotopy between the function

G(0,x) and G(1,x)=F(x) (4.3.5)

In order to find such a function, it is assumed to have a function

G(λ,x)=0 is existed and partial derivative of this function with respect to λ and x will also be zero

$$\frac{\partial G(\lambda,x)}{\partial \lambda} + \frac{\partial G(\lambda,x)}{\partial x} x'(\lambda) = 0 \qquad (4.3.6)$$

if x'(λ) is isolated form this equation, it becomes:

$$x'(\lambda) = -\left[\frac{\partial G(\lambda,x)}{\partial x}\right]^{-1}\left[\frac{\partial G(\lambda,x)}{\partial \lambda}\right] \quad (4.3.7)$$

If $G(\lambda,x)=\lambda F(x)+(1-\lambda)[F(x)-F(x(0))]$ the equation is substituted into the differential equation

$$\left[\frac{\partial G(\lambda,x)}{\partial \lambda}\right] = \begin{bmatrix} \frac{\partial f_1(x(\lambda))}{\partial x_1} & \frac{\partial f_1(x(\lambda))}{\partial x_2} & \frac{\partial f_1(x(\lambda))}{\partial x_{33}} \\ \frac{\partial f_2(x(\lambda))}{\partial x_1} & \frac{\partial f_2(x(\lambda))}{\partial x_2} & \frac{\partial f_2(x(\lambda))}{\partial x_3} \\ \frac{\partial f_3(x(\lambda))}{\partial x_1} & \frac{\partial f_3(x(\lambda))}{\partial x_2} & \frac{\partial f_3(x(\lambda))}{\partial x_3} \end{bmatrix} = J(x(\lambda)) \quad (4.3.8)$$

Forms a Jacobian matrix. and

$$\left[\frac{\partial G(\lambda,x)}{\partial x}\right] = F(x(0)) \quad (4.3.9)$$

Differential equation becomes

$$x'(\lambda) = \frac{dx(\lambda)}{d\lambda} = -[J(x(\lambda)]^{-1}F\big(x(0)\big) \qquad 0 \le \lambda \le 1 \quad (4.3.10)$$

It is possible to solve such a differential equation by using initial value problem approaches, solution at x(1) will be given us the roots of the system of equation. Solutions of initial value problems will be given latter chapters in details, but A fourth order Runge-Kutta solution will be defined here to solve our homotopy problem. If equation

$\frac{dx(\lambda)}{d\lambda} = f(\lambda,x(\lambda))$ (4.3.11) is given the 4th order Runge-Kutta method to numerically solve this differential equation is defined as:

$x_{i+1} = x_i + (1/6)*(k_1 + 2k_2 +2k_3+k_4)h$ (4.3.12)

$k_1=f(\lambda_i,x_i)$

$k_2=f(\lambda_i+0.5h,x_i+0.5k_1h)$

$k_3=f(\lambda_i+0.5h,x_i+0.5k_2h)$

$k_4=f(\lambda_i+h,x_i+k_3h)$

In these equations h is finite difference step size. Solution starts by using the initial value λ=0 , x0(0) and adds h into λ in each iteration step. The example program uses 4th degree and 6th degree Runge-Kutta method to solve homotopy(Continuation problem). It should be note that Homotophy method is less dependent to initial value compare to methods such as Newton-Raphson therefore one possibility is to approachsolution with a relatively rough estimate with homotophy following with a Newton-Raphson type of method. In the following code methods of Homotophy with 4th and 6th order Runge-Kutta methods and combination of Homotophy with 4th order Runge-Kutta and Newton-Raphson method is given.

Program 4.3-1 Homotophy (Contiuation) method (java version)

```
function func(x)
    ff=zeros(Float64,3)
    ff[1]=3.0*x[1]-cos(x[2]*x[3])-0.5
    ff[2]=x[1]*x[1]-81.0*(x[2]+0.1)*(x[2]+0.1)+sin(x[3])+1.06
    ff[3]=exp(-x[1]*x[2])+20.0*x[3]+(10.0*pi-3.0)/3.0
    return ff
end

#single function with reference number i
function fi(f,x,i)
# system of non-linear equation
y=f(x)
return y[i]
end

#first derivative
```

```
function dfunc(func,x)
 global  dx=0.001
 global c=[1.0/280.0 -4.0/105.0          1.0/5.0    -4.0/5.0  0          4.0/5.0    -1.0/5.0   4.0/105.0 -1.0/280.0]
 n=length(x)
 ff=zeros(Float64,n,n)
 global yy=ff[1]
 x1=zeros(Float64,n)
 for ieqn=1:n
  for ix=1:n
    for i=-4:4
      # create x1set for derivative input
      # ================================
      for k=1:n
        xx=x[k]
        if k!=ix
          x1[k]=xx
        else
          delta=Float64(i)*dx
          x1[k]=xx+delta
        end
       end # for k
       # ================================
       ff1=fi(func,x1,ieqn)

       ff[ieqn,ix]+=c[i+5]* ff1
       yx=ff[ieqn,ix]
     end # for i
     ff[ieqn,ix]/=dx
     yx=ff[ieqn,ix]
   end # for ix
 end # for ieqn
 return ff
end # for function


function continuationRK4(f,x,N)
   # ================================================
   # Roots of nonlinear system of equations Homotophy RK6
   # 4.th order Runge-Kutta
   # yi+1 = yi + (1/6)*( k1 + 2k2 +2k3+k4)h
   # k1=f(xi,yi)
   # k2=f(xi+0.5h,yi+0.5k1h)
   # k3=f(xi+0.5h,yi+0.5k2h)
   # k4=f(xi+h,yi +k3h)
   # ================================================
   # x vector of independent variables
   # y vector of dependent variables
   # dy derivative vector of dependent variables
   nmax=400
   tolerance=1.0e-20
   n=length(x)
   h=1.0/Float64(N)
   global b=zeros(Float64,n)
   x1=zeros(Float64,n)
   A= zeros(Float64,n,n)
   b=-h*f(x)
   for i=1:N
      x1=x
      A=dfunc(f,x1)
      k1=A\b
      kk=0.5*k1
      x1=x+kk'
      A=dfunc(f,x1)
      k2=A\b
      kk=0.5*k2
      x1=x+kk'
      A=dfunc(f,x1)
```

```
        k3=A\b
        kk=k3
        x1=x+kk'
        A=dfunc(f,x1)
        k4=A\b
        c=[1.0/6.0 2.0/6.0 2.0/6.0 1.0/6.0]
        kk=c[1]*k1
        kk+=c[2]*k2
        kk+=c[3]*k3
        kk+=c[4]*k4
        x+=kk'
    end
    return x
end

function newton(func,x)
  # Newton-Raphson root finding method
  k=length(x)
  nmax=100
  tolerance=1.0e-10
  for i=1:nmax
    fx=func(x)
    dfx=dfunc(func,x)
    dx=dfx\fx    #gauss_pivot(dfx,fx)
    for j=1:k
       x[j]=x[j]-dx[j]
    end # for j
    total=0.0;
    for j=1:k
       total+=fx[j]
    end # for j
    if abs(total)<tolerance
      return x
    end
  end #for i
  return x
end # function

x=[1 1 1]
x= continuationRK4(func,x,10)
print("x continuation : $x \n")
x=newton(func,x)
print("x newton : $x \n")
print("x : $x \n")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" continuity.jl
x continuation : [0.4999682669519078 -3.9723398572444335e-5 -0.5235991652361665]
x newton : [0.5 -1.81723061024014e-17 -0.5235987755982989]
x : [0.5 -1.81723061024014e-17 -0.5235987755982989]

> Terminated with exit code 0.
```

JACOBİ, GAUSS-SEİDEL AND RELAXATION METHODS FOR NON-LINEAR SYSTEM OF EQUATIONS

Iterative solution methods of linear system of eqaution were investigated in the previous chapter. The same methods can also be applied to solve non-linear system of equations. There are no guarantee that these type of soutions will be converged. In this method the basic nonlinear set of eqaution first converged as

$f_j(x_i) = 0 \quad i = 0..n \;\; j = 0..n$ (4.4.1) from the original form to

$x_j = g_j(x_i) \quad i = 0..n \;\; j = 0..n$ or in an open form:

$x_0 = g_0(x_0, x_1, x_2, .., x_n)$

$x_1 = g_1(x_0, x_1, x_2, .., x_n)$

…..

$x_n = g_n(x_0, x_1, x_2, .., x_n)$

So out iterative set of eqaution will be

$x_0^{k+1} = (1-\alpha)x_0^k + \alpha g_0(x_0, x_1, x_2, .., x_n)$
$x_1^{k+1} = (1-\alpha)x_1^k + \alpha g_1(x_0, x_1, x_2, .., x_n)$
$x_2^{k+1} = (1-\alpha)x_2^k + \alpha g_2(x_0, x_1, x_2, .., x_n)$
…..
$x_j^{k+1} = (1-\alpha)x_j^k + \alpha g_j(x_0, x_1, x_2, .., x_n)$
.....
$x_n^{k+1} = (1-\alpha)x_n^k + \alpha g_n(x_0, x_1, x_2, .., x_n)$

The value could be between 0 and 1 or bigger than 1. Let us give an example problem as:
$3x_0 - \cos(x_1 x_2) - 0.5 = 0$
$x_0^2 - 81(x_1 + 0.01) + \sin(x_2) + 1.06 = 0$
-exp(-$x_0 x_1$) + $20x_2 + \frac{10\pi - 3}{3} = 0$
When we converged the system of equation, it will become:
$x_0 = [\cos(x_1 x_2) + 0.5]/3$
$x_1 = \frac{[x_0^2 + \sin(x_2) + 1.06]}{81} + 0.01$
$x_2 = (-\exp(-x_0 x_1) - \frac{10\pi - 3}{3})/20$

Program list is given as follows:

Program Jacobi-Gauss_seidel and relaxation method

```
 #Non-linear gauss_seidel
#function set
function func(x)
# system of non-linear equation
g=zeros(Float64,3)
g[1]=(cos(x[2]*x[3])+0.5)/3.0;
g[2]=sqrt((x[1]*x[1]+sin(x[3]+1.06))/81.0)-0.1;
g[3]=(-exp(-x[1]*x[2])-(10.0*pi-3)/3.0)/20.0;
return g
end

function gauss_seidel(f,x,alpha)
global n=length(x)
global xx=zeros(Float64,n)
global xx1=zeros(Float64,n)
global g=zeros(Float64,n)
global delta=0.0
for i=1:n
  xx[i]=x[i]
end

println("Initial estimation xx= $xx alpha = $alpha")
g=zeros(Float64,n)
iter=500
for k=1:iter
  for i=1:n
    xx1[i]=xx[i]
  end #for i
  delta=0.0
  for j=1:n
      g=f(xx)
      zx1=alpha*g[j]
      zx2=(1.0-alpha)*xx[j]
      zx3=zx1.+zx2
                xx[j]=zx3
                xj1=      xx[j]-xx1[j]
                delta=delta+abs(xj1)
                z1=xx[j]
  end #for j
  if delta<1e-15
    println("iteration number = $k delta = $delta")
    break
```

```
  end # if
end # for k
return xx
end # function

x=[1.01 1.012 1.0121]
alpha=0.1
x=gauss_seidel(func,x,alpha)
print("iterative non-linear relaxation  method $x")
```

Gauss_seidel relaxation method

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" NL_gauss_seidel.jl
Initial estimation xx= [1.01, 1.012, 1.0121] alpha = 0.1
iteration number = 331 delta = 9.32847549206528e-16
iterative non-linear relaxation  method [0.4999995683317303, -0.0030731824419162226, -0.523675664150972]
> Terminated with exit code 0.
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" NL_gauss_seidel.jl
Initial estimation xx= [1.01, 1.012, 1.0121] alpha = 0.2
iteration number = 160 delta = 9.514958265732787e-16
iterative non-linear relaxation  method [0.49999956833173037, -0.0030731824419189787, -0.523675664150974]
> Terminated with exit code 0.
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" NL_gauss_seidel.jl
Initial estimation xx= [1.01, 1.012, 1.0121] alpha = 0.5
iteration number = 54 delta = 6.951904329977054e-16
iterative non-linear relaxation  method [0.4999995683317301, -0.0030731824419208396, -0.523675664150975]
> Terminated with exit code 0.
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" NL_gauss_seidel.jl
Initial estimation xx= [1.01, 1.012, 1.0121] alpha = 0.8
iteration number = 25 delta = 2.8796409701215e-16
iterative non-linear relaxation  method [0.49999956833173004, -0.0030731824419212824, -0.5236756641509752]
> Terminated with exit code 0.
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" NL_gauss_seidel.jl
Initial estimation xx= [1.01, 1.012, 1.0121] alpha = 1.0
iteration number = 7 delta = 4.1633363423443337e-17
iterative non-linear relaxation  method [0.49999956833173004, -0.0030731824419213266, -0.5236756641509752]
> Terminated with exit code 0.
```

PROBLEMS

PROBLEM 1

Solve the following system of non-linear equations

$4x_1^2-20x_1+0.25x_2^2+8=0$

$0.5x_1x_2^2+2x_1+8=0$

By using

a) Newton raphson method

b) Continuity(Homotopy) method

PROBLEM 2

Solve the following system of non-linear equations

$3x_1^2-\cos(x_2x_2) - 0.5=0$

$4x_1^2-6.25x_2^2+2x_3-1=0$

$e^{x1\ x2} +20x_3+(10\pi-3)/3=0$

By using
a) Newton raphson method
b) Continuity(Homotopy) method

PROBLEM 3

Solve te following system of non-linear equations
$x_1(1- x_1) + 4x_3 – 12 =0$
$(x_1-2)^2+(2x_2-3)^2-25 =0$
By using
a) Newton raphson method
b) Continuity(Homotopy) method

PROBLEM 4

Solve the following system of non-linear equations
$\sin(4\pi x_1x_2)-2x_2-x_1=0$
$\left(\frac{4\pi-1}{4\pi}\right)(e^{2x_1}-e)+4ex_2^2-2ex_1=0$
By using
a) Newton raphson method
b) Continuity(Homotopy) method

PROBLEM 5

Solve the following system of non-linear equations
$10x_1-2x_2^2+x_2-2x_3-5=0$
$8x_2^2+4x_3^2-9=0$
$8x_2x_3+4=0$
By using
a) Newton raphson method
b) Continuity(Homotopy) method

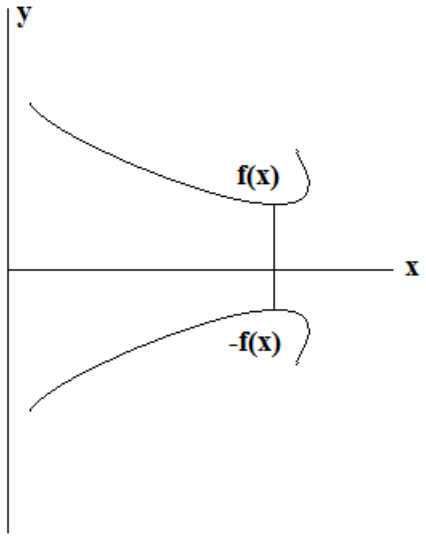
OPTIMIZATION

DEFINITION OF OPTIMIZATION AND GRAPHIC METHODS

Optimization is the name given to maximum or minimum finding process. When it is considered in mathematically, the root of the derivative of the function defines the optimum point.

$$\frac{df(x)}{dx}=0$$

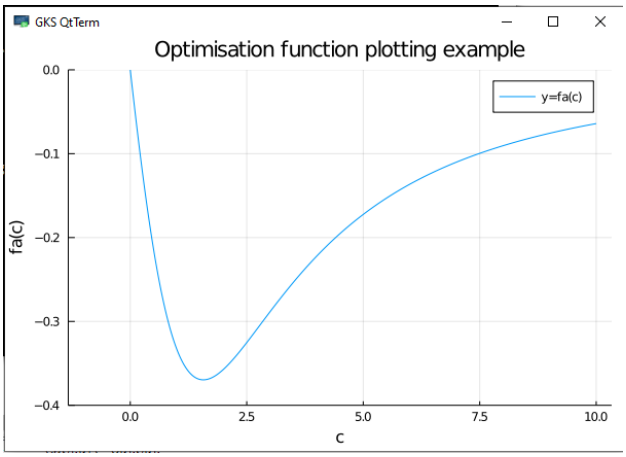
As the mathematical process, maximum and minimum is similar processes because
maximum (f(x)) = minimum(-f(x))



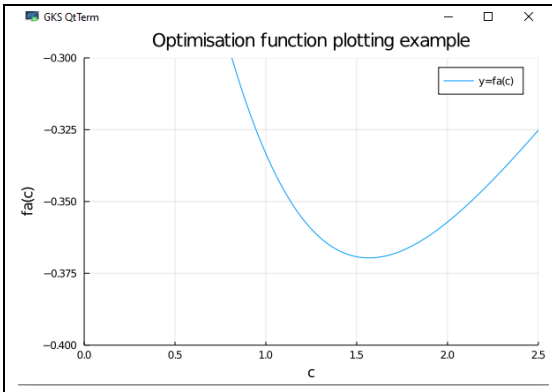
As a first example to one dimensional non-linear optimization, Plotting will be utilised. Mathematically plotting means to subdivide the region into n equal parts and investigate the region of the minimum.

PROGRAM plots as optimization example

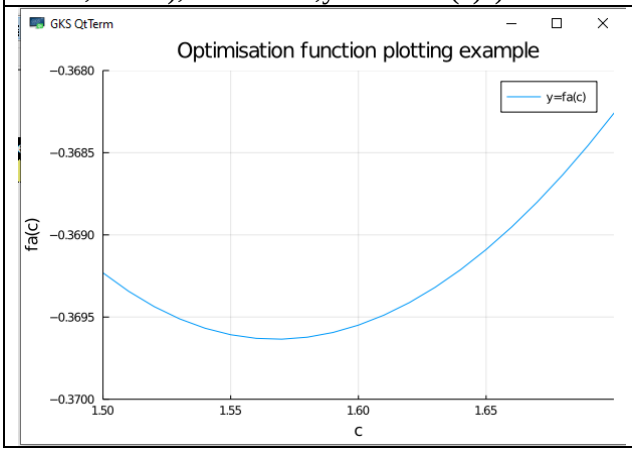
```
 using Plots
fa(c)=-2c/(4.0+0.8c+c^2+0.2*c^3)
x1=-1:0.01:10
plot(fa,x1,title="Optimisation function plotting example",label="y=fa(c)",ylims = (-0.4,0.0),xaxis = "c",yaxis = "fa(c)")
```




```
using Plots
fa(c)=-2c/(4.0+0.8c+c^2+0.2*c^3)
x1=-1:0.01:10
plot(fa,x1,title="Optimisation function plotting
example",label="y=fa(c)",xlims=(0.0,2.5),ylims = (-0.4,-0.3),xaxis = "c",yaxis = "fa(c)")
```




```
using Plots
fa(c)=-2c/(4.0+0.8c+c^2+0.2*c^3)
x1=-1:0.01:10
plot(fa,x1,title="Optimisation function plotting example",label="y=fa(c)",xlims=(1.5,1.7),ylims = (-0.370,-0.368),xaxis = "c",yaxis = "fa(c)")
```



As it is seen from the plot windows, a guess of 1.5634 can be obtained for the optimum point.

ONE DIMENSIONAL NON-LINEAR FIBONNACHI SEARCH (GOLDEN SEARCH) METHOD

Series 0,1,1,2,3,5,8,13,21,34,… is called fibonnachi series. General definition is in the form of F(n)=F(n-1)+F(n-2) F(0)=0, F(1)=1

For the big fibonnaci terms the ration of two following fibonnachi number becomes constant. This constant is known as golden ratio,

$$R = \lim_{n\to\infty} \frac{F(n-1)}{F(n)} = \frac{\sqrt{5}-1}{2} = 0.618033989$$

If a minimum being searched in a given region (a,c). In order to determine if a minimum is existed, region should be divided up into three sub-regions. This requires avaluation of two intermediate data points. Neigbors of the minimum point remains and the third region is eliminated. In the new avaluation a new two points will be required. Golden ratio is given us a fine propery so that one of the selected points will always be one of the new search point. So that only one new point will be evaluated in each step of the evaluation.

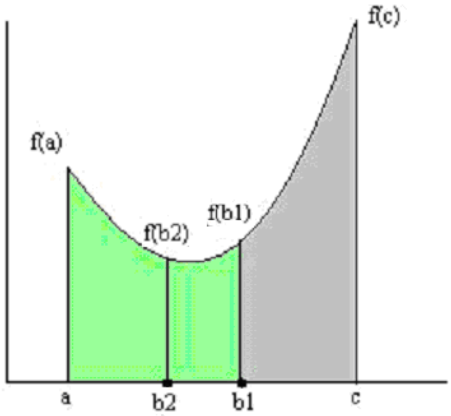


Figure 5.3-1 Golden ratio search

If minimumin in region [a,c] is required
The length of the region will be d0=(c – a) .
Then b1 is taken as b1=a+d0*R
And b2 is taken as b2 = c – d0*R = a + d0*R^2
And if it is assumed that minimum is in point b2, region (b1-c) will be thrown. Our new region will be d1=d0*R olacağindan
new b1 is taken as b1=a+d1*R=a+d0*R^2 which is the same as previous b2.
New b2 is taken n as b2 = c – d1*R = a + d1*R^2 = a + d0*R^3 .

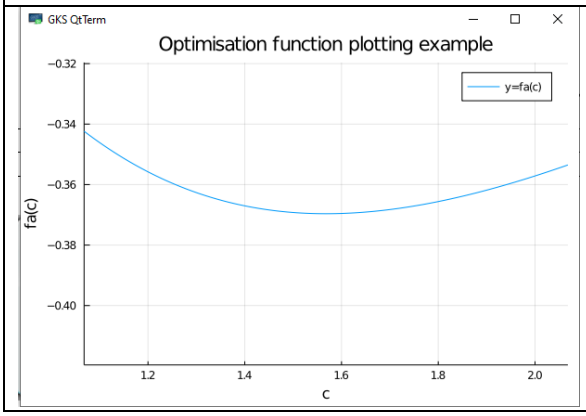
Since one of the point is the same, evaluation of only one new point is required for each iteration step. The Algorithm of Golden search is given in Program 5.3-1

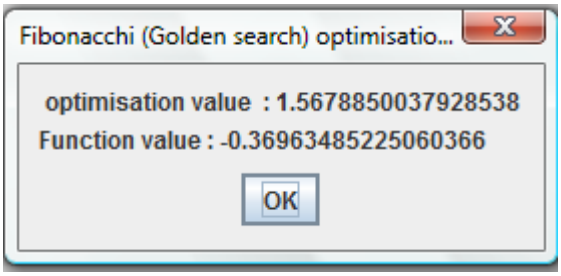
PROGRAM 5.2-1 Golden Ratio or Fibonnachi search

```
 # OPTIMIZATION Golden search or fibonnachi method
function golden(f,a,b,epsilon,delta,print)

#  find the minimum of the function
# note  maximum f(x) = minimum (-f(x))
   r1 = (sqrt(5.0)-1.0)/2.0 # golden ratio
   r2 = r1*r1
   h = b - a
   ya = f(a)
   yb = f(b)
   c = a + r2*h
   d = a + r1*h
   yc = f(c)
   yd = f(d)
   k = 1
   dp=0.0
   dy=0.0
   p=0.0
   yp=0.0
   while abs(yb-ya)>epsilon || h>delta
     k+=1
     if yc<yd
       b = d
       yb = yd
       d = c
       yd = yc
       h = b - a
       c = a + r2 * h
```

```
        yc = f(c)
      else
        a = c
        ya = yc
        c = d
        yc = yd
        h = b - a
        d = a + r1 * h
        yd = f(d)
      end #end of if
    end #end of while
    dp = abs(b-a)
    dy = abs(yb-ya)
    p = a
    yp = ya
    if  yb<ya
      p = b
      yp = yb
    end # if
    if print==1
    print("x min = $p ymin = $yp error of x = $dp error of y $dy")
    end # if
    return p
end # function

using Plots
a=0.5
b=1.7
fa(c)=-2c/(4.0+0.8c+c^2+0.2*c^3)
r= golden(fa,a,b,1.0e-10,1.0e-5,0)
fr=fa(r)
print("optimisation value = $r function value = $fr")
x1=r-0.5:0.01:r+0.5
plot(fa,x1,title="Optimisation function plotting example",label="y=fa(c)",xlims=(r-0.5,r+0.5),ylims = (fr-
0.05,fr+0.05),xaxis = "c",yaxis = "fa(c)")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_fibonnachi.jl
optimisation value = 1.5678909109047903 function value = -0.3696348522520682
> Terminated with exit code 0.
```



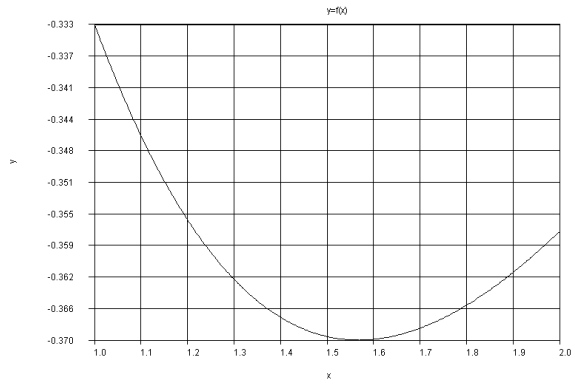


If search method is investigated step by step :

PROGRAM 5.2-1A Golden Ratio or Fibonnachi search by excel

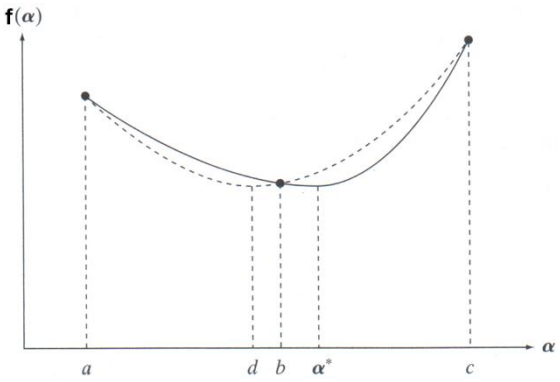
a	b	h	c	d	yc	yd
1	2	1	1.381966	1.618034	-0.366410476	-0.369427868
1.381966011	2	0.618034	1.618034	1.763932	-0.369427868	-0.366711114
1.381966011	1.763932	0.381966	1.527864	1.618034	-0.36949646	-0.369427868
1.381966011	1.618034	0.236068	1.472136	1.527864	-0.368819066	-0.36949646
1.472135955	1.618034	0.145898	1.527864	1.562306	-0.36949646	-0.369632208
1.527864045	1.618034	0.09017	1.562306	1.583592	-0.369632208	-0.369614176
1.527864045	1.583592	0.055728	1.54915	1.562306	-0.36960486	-0.369632208
1.549150281	1.583592	0.034442	1.562306	1.570437	-0.369632208	-0.369634304
1.562305899	1.583592	0.021286	1.570437	1.575462	-0.369634304	-0.369630023
1.562305899	1.575462	0.013156	1.567331	1.570437	-0.369634826	-0.369634304
1.562305899	1.570437	0.008131	1.565412	1.567331	-0.369634332	-0.369634826
1.565411519	1.570437	0.005025	1.567331	1.568517	-0.369634826	-0.369634819
1.565411519	1.568517	0.003106	1.566598	1.567331	-0.369634711	-0.369634826
1.56659776	1.568517	0.001919	1.567331	1.567784	-0.369634826	-0.369634851
1.567330897	1.568517	0.001186	1.567784	1.568064	-0.369634851	-0.36963485
1.567330897	1.568064	0.000733	1.567611	1.567784	-0.369634846	-0.369634851

Graphic exit of the function



ONE DIMENSIONAL NON-LINEAR QUADRATIC POLYNOMIAL METHOD

Golden search is a linear search method, it will always Works if a minimum(Maximum) is existed in the given region. But convergence speed is relatively slow. A quadratic search should theoretically be given values closer to the actual optimum value.



Minimum value of a quadratic equation is given with the Formula

$$d = b + 0.5\frac{(c-b)^2[f(a)-f(b)]-(a-b)^2[f(c)-f(a)]}{(c-b)[f(a)-f(b)]-(a-b)[f(c)-f(a)]}$$

After finding point d, point b is replaced with this point and iteration continues.

Quadratic polynomial search

```
 # OPTIMIZATION Golden search or fibonnachi method
function  OPT_quadratic_poly(f,x0,x1,x2,epsilon,delta,print1)

maxit=100
   global f0 = f(x0)
   global f1 = f(x1)
   global f2 = f(x2)
   global f3 = f2
   global x3 = 0.0
   h = x1 - x0
   k=1
   dd=abs(f1-f0)
   while dd>epsilon || h>delta
     k+=1
     x3=(f0*(x1*x1-x2*x2)+f1*(x2*x2-x0*x0)+f2*(x0*x0-x1*x1))/(2*f0*(x1-x2)+2.0*f1*(x2-x0)+2.0*f2*(x0-x1))
     f3=f(x3)
     println("x3 = $x3 f3 = $f3")
     if x3 >=x0 && x3<x1
       x2=x1
       f2=f1
       x1=x3
       f1=f3
     elseif x3>=x1 && x3<x2
       x0=x1
       f0=f1
       x1=x3
       f1=f3
     elseif x3>x2
       x0=x1
       f0=f1
       x1=x2
       f1=f2
       x2=x3
       f2=f3
     elseif x3<x0
       x0=x3
       f0=f3
       x1=x0
       f1=f0
       x2=x1
       f2=f1
     end #if
     dd=abs(f1-f0)
     h=abs(x1-x0)
     if(k>maxit)
        break
     end # if
   end  #end of while
   if(print1==1)
      println("x  = $x3 f = $f3 ")
   end
   return x3
end # function
```

```
using Plots
x0=1.0
x1=1.5
x2=2.0
epsilon=1.e-10
delta=1.0e-10
print1=0
fa(c)=-2.0c/(4.0+0.8c+c^2+0.2*c^3)
r= OPT_quadratic_poly(fa,x0,x1,x2,epsilon,delta,print1)
fr=fa(r)
print("optimisation value = $r function value = $fr")
x1=r-0.5:0.01:r+0.5
plot(fa,x1,title="Optimisation function plotting example",label="y=fa(c)",xlims=(r-0.5,r+0.5),ylims = (fr-
0.05,fr+0.05),xaxis = "c",yaxis = "fa(c)")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" OPT_quadratic_poly.jl
x3 = 1.624045801526718 f3 = -0.3693761003047835
x3 = 1.5707114578848842 f3 = -0.3696341798401909
x3 = 1.5689145983133213 f3 = -0.36963476342950463
x3 = 1.5679570580017286 f3 = -0.36963485186545303
x3 = 1.5679088123360234 f3 = -0.36963485222046666
x3 = 1.5678909680728423 f3 = -0.3696348522520535
x3 = 1.5678897945555532 f3 = -0.36963485225224363
x3 = 1.5678894677852748 f3 = -0.3696348522522551
x3 = 1.5678894305561077 f3 = -0.36963485225225534
x3 = 1.5678894698582673 f3 = -0.36963485225225506
x3 = 2.0 f3 = -0.35714285714285715
x3 = 1.5678906524892626 f3 = -0.3696348522521276
x3 = 1.5678892563835765 f3 = -0.36963485225225295
optimisation value = 1.5678892563835765 function value = -0.36963485225225295
> Terminated with exit code 0.
```

ONE DIMENSIONAL NON-LINEAR NEWTON-RAPHSON METHOD

Newton- Raphson is investigated as one of the root finding methods. Due to the fact that roots of the derivatives of a function is also given the optimum point makes Newton method a cendidate to find optimum through the roots of the derivative of the function.

Minimum of f(x)

Roots of $g(x) = \frac{df(x)}{dx} = 0$

Newton raphson formula

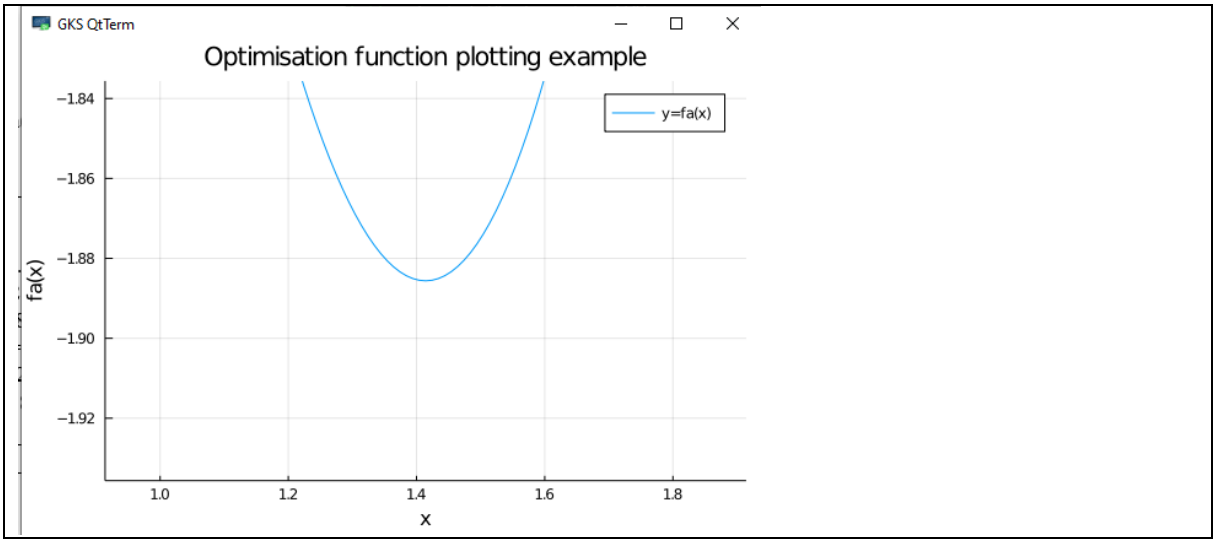
$$x_{n+1} = x_n - \frac{g(x)}{\frac{dg(x)}{dx}} = x_n - \frac{\frac{df(x)}{dx}}{\frac{d^2f(x)}{dx^2}}$$

Program 5.5-1 Newton-Raphson method

```
 #first derivative
function df(f,x)
 dx=0.001
 c=[1.0/280.0 -4.0/105.0    1.0/5.0    -4.0/5.0   0          4.0/5.0    -1.0/5.0   4.0/105.0 -1.0/280.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/dx
end
#second derivative
function d2f(f,x)
 dx=0.001
```

```
 c=[-1.0/560.0        8.0/315.0 -1.0/5.0 8.0/5.0 -205.0/72.0 8.0/5.0 -1.0/5.0 8.0/315.0 -1.0/560.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx)
end
#third derivative
function d3f(f,x)
 dx=0.001
 c=[-7.0/240.0        3.0/10.0   -169.0/120.0          61.0/30.0 0.0          -61.0/30.0              169.0/120.0          -3.0/10.0
             7.0/240.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx*dx)
end

function newton(f,x)
nmax=100
tolerance=1.0e-10
for i=0:nmax
  dfx=df(f,x)
  d2fx=d2f(f,x)
  x-=dfx/d2fx
  println("i = $i x = $x dfx = $dfx d2fx = $d2fx")
  if abs(dfx)<tolerance
    return x
  end #f
end #for
return x
end #function

using Plots
x0=1.001
epsilon=1.e-10
delta=1.0e-10
print1=0
fa(x)=1.0/3.0*x*x*x-2*x
x= newton(fa,x0)
fx=fa(x)
print("optimisation value = $x function value = $fx")
x1=x-0.5:0.01:x+0.5
plot(fa,x1,title="Optimisation function plotting example",label="y=fa(x)",xlims=(x-0.5,x+0.5),ylims = (fx-
0.05,fx+0.05),xaxis = "x",yaxis = "fa(x)")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_newton.jl
i = 0 x = 1.4995009989832748 dfx = -0.9979990000000115 d2fx = 2.0020000000712037
i = 1 x = 1.4166390181909343 dfx = 0.24850324595151324 d2fx = 2.9990019979643514
i = 2 x = 1.41421563870902 dfx = 0.006866107861130087 d2fx = 2.8332780368785055
i = 3 x = 1.4142135623746328 dfx = 5.8727691217008715e-6 d2fx = 2.8284312770122253
i = 4 x = 1.4142135623731422 dfx = 4.21624540836163e-12 d2fx = 2.8284271249763053
optimisation value = 1.4142135623731422 function value = -1.8856180831641267
> Terminated with exit code 0.
```



BISECTION METHOD

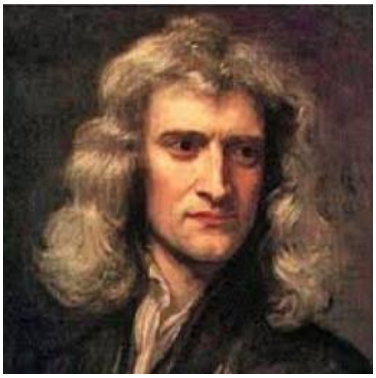
Bisection method is an important root finding method. By definition of optimisation which optimum point is the root of the derivative of the function, optimum van be obtained by finding root of derivative function

```
#first derivative
function df(f,x)
 dx=0.001
 c=[1.0/280.0 -4.0/105.0          1.0/5.0     -4.0/5.0    0          4.0/5.0     -1.0/5.0    4.0/105.0  -1.0/280.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/dx
end

function opt_bisection(f,a,b)
    b1=1.1*b
    r=(a+b)/2.0
    fr=f(r)
    fa=df(f,a)
    eps=1.0e-6
    nmax=100
    i=1
    while abs(fr)>eps && i<nmax
        if fa*fr<0
            b=r
        else
            a=r
            fa=fr
        end
        r=(a+b)/2.0
        fr=df(f,r)
        i=i+1
    end
    if i>=nmax
       r=bisection(f,a,b1)
    end
    return r
end


function enlarge(f,x0,dx)
    # enlarge region untill a root is existed
    x1=x0
    x2=x1+dx
    NTRY=200
    FACTOR=1.001
    j=0
```

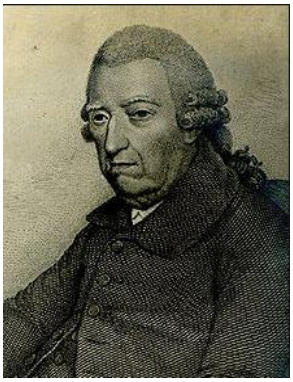
```
    if x1 == x2
      print("input variables are wrong")
    end
    f1=f(x1)
    f2=f(x2)
    for j=1:NTRY+1
       if f1*f2 < 0.0
          break
       else
          x2=x2+dx
          f2=f(x2)
       end
    end
    return x2
end
a=0.0
b=2.0
fa(c)=-2.0c/(4.0+0.8c+c^2+0.2*c^3)
r= opt_bisection(fa,a,b)
fr=fa(r)
print("optimum value : $r f($r) = $fr")
fr=fa(r)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_bisection.jl
optimum value : 1.5678863525390625 f(1.5678863525390625) = -0.3696348522514579
> Terminated with exit code 0.
```

NEWTON-RAPHSON METHOD FOR NONLINEAR MULTIVARIABLE OPTIMIZATION



Sir Isaac Newton



Joseph Raphson

Root finding of multivariable functions by using Newton-Raphson method was prevously investigated. By remembering that optimum point is the root of the derivative of the function, the method can be utilise also to find optimum point.

If function $f(x_1,x_2,x_3,\ldots,x_n)$ First and second derivative and independent difference vector of the function can be written as:

$$\nabla f = \begin{Bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \\ \cdots\cdots \\ \frac{\partial f}{\partial x_n} \end{Bmatrix} \text{ and } \delta^{m+1} = \begin{Bmatrix} (x_1^{m+1} - x_1^m) \\ (x_2^{m+1} - x_2^m) \\ (x_3^{m+1} - x_3^m) \\ \cdots\cdots \\ (x_n^{m+1} - x_n^m) \end{Bmatrix} \text{ and } [H] = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_1 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \frac{\partial^2 f}{\partial x_2 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \frac{\partial^2 f}{\partial x_3 \partial x_1} & \frac{\partial^2 f}{\partial x_3 \partial x_2} & \frac{\partial^2 f}{\partial x_3^2} & \cdots & \frac{\partial^2 f}{\partial x_3 \partial x_n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \frac{\partial^2 f}{\partial x_2 \partial x_n} & \frac{\partial^2 f}{\partial x_3 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Newton-Raphson iteration for (m+1) th iteration

$$[H^m]\{\delta^{m+1}\} = -\{\nabla f^m\}$$

As it is seen from the equation a linear system of equation is formed. The equation can be solved by using methods such as gauss elimination. An initial estimate for all the x values are required to start iterative solution. Second derivartive matrix is also called a Hessian matrix. First and second

derivatives can also be calculated numerically, but that will increase the errors orthe number of iteration steps. The system of equation can also be written as:

$\{\delta^{m+1}\} = [H^m]^{-1}[-\{\nabla f^m\}]$

In this format$[H^m]^{-1}$ is the inverse Hessian matrix. If vector δ is written in open form, it becomes

$\{x^{m+1}\} = \{x^m\} + [H^m]^{-1}[-\{\nabla f^m\}]$

As a more general definition of this equation a step function α can be added into the term. It becomes

$\{x^{m+1}\} = \{x^m\} + \alpha^m[H^m]^{-1}[-\{\nabla f^m\}]$

In Newton-Raphson method step function α is equal to unity. In some other methods different utilisation of this general description will be invetigated.

Newton optimisation method

```
 #function set
function func(x)
ff=3.0*x[1]*x[1]-4.0*x[1]*x[2]+2.0*x[2]*x[2]-x[1]-x[2]
return ff
end

#single function with reference number i
function fi(f,x,i)
# system of non-linear equation
y=dfunc(f,x)
return y[i]
end

#first derivative
function dfunc(func,x)
 global  dx=0.001
 global c=[1.0/280.0 -4.0/105.0   1.0/5.0      -4.0/5.0  0 4.0/5.0      -1.0/5.0     4.0/105.0   -1.0/280.0]
 n=length(x)
 ff=zeros(Float64,n)
 global yy=ff
 x1=zeros(Float64,n)
   for ix=1:n
      for i=-4:4
        # create x1set for derivative input
        # ================================
        for k=1:n
          xx=x[k]
          if k!=ix
            x1[k]=xx
          else
            delta=Float64(i)*dx
            x1[k]=xx+delta
          end
        end # for k
        # ================================
        ff1=func(x1)
        ff[ix]+=c[i+5]*ff1
        yx=ff[ix]
      end # for i
      ff[ix]/=dx
      yx=ff[ix]
    end # for ix
 return ff
end # for function

#second derivative
function d2func(func,x)
 global  dx=0.001
 global c=[1.0/280.0 -4.0/105.0   1.0/5.0      -4.0/5.0  0 4.0/5.0      -1.0/5.0     4.0/105.0   -1.0/280.0]
 n=length(x)
 ff=zeros(Float64,n,n)
 global yy=ff[1]
 x1=zeros(Float64,n)
 for ieqn=1:n
   for ix=1:n
      for i=-4:4
        # create x1set for derivative input
        # ================================
        for k=1:n
```

```
          xx=x[k]
          if k!=ix
            x1[k]=xx
          else
            delta=Float64(i)*dx
            x1[k]=xx+delta
          end
        end # for k
        # ================================
        ff1=fi(func,x1,ieqn)
        ff[ieqn,ix]+=c[i+5]*ff1
        yx=ff[ieqn,ix]
      end # for i
      ff[ieqn,ix]/=dx
      yx=ff[ieqn,ix]
    end # for ix
  end # for ieqn
  return ff
end # for function

function gauss_pivot(A1,b)
              n=size(A1)[1]
              carpan=0.0
              toplam=0.0
              global x=Array{Float64}(undef,n)
              global a=Array{Float64}(undef,n,n)
              for i=1:n
                for j=1:n
                  a[i,j]=A1[i,j]
                end
              end
              for k=1:n
              #pivoting
                 p=k
                 buyuk=abs(a[k,k])
                 for ii=k+1:n
                   dummy=abs(a[ii,k])
                   if dummy > buyuk
                     buyuk=dummyp=ii
                   end #if dummy
                 end #for ii
                 if p!=k
                   for jj=k:n
                     dummy=a[p,jj]
                     a[p,jj]=a[k,jj]
                     a[k,jj]=dummy
                     end #for jj

                   dummy=b[p]
                   b[p]=b[k]
                   b[k]=dummy
                 end #if p!
              # gauss elimination
    for i=k+1:n
                 carpan=a[i,k]/a[k,k]
                 a[i,k]=0.0
                 for j=k+1:n
                   a[i,j]=a[i,j]-carpan*a[k,j]
                 end #for j
                 b[i]=b[i]-carpan*b[k]
              end #for i
              end #k
                            # back substituting
                x[n]=b[n]/a[n,n]
                n1=n-1
                for i=n1:-1:1
                  toplam=0.0
                  for j=i+1:n
                    toplam=toplam+a[i,j]*x[j]
                  end #for j
                  x[i]=(b[i]-toplam)/a[i,i]
                end #for i
              return x
end #function

function newton(func,x)
```

```
  # Newton-Raphson root finding method
  k=length(x)
  nmax=100
  tolerance=1.0e-10
  for i=1:nmax
    fx=dfunc(func,x)
    dfx=d2func(func,x)
    dx=gauss_pivot(dfx,fx)
    for j=1:k
       x[j]=x[j]-dx[j]
    end # for j
    total=0.0;
    for j=1:k
       total+=fx[j]
    end # for j
    if abs(total)<tolerance
      return x
    end
  end #for i
  return x
end # function


x=[1.0 1.0]
y=func(x)
dy=dfunc(func,x)
d2y=d2func(func,x)
println("x = $x \ny = $y dy = $dy d2y = $d2y")
r=newton(func,x)
println("r = $r")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" NL_newton.jl
x = [1.0 1.0]
y = -1.0 dy = [1.0000000000000988, -1.0000000000000855]
d2y = [6.000000000161124 -4.000000000009855; -3.9999999999164455 3.9999999999791522]
r = [1.0000000000000777 1.2500000000001292]

> Terminated with exit code 0.
```

Julia has optimisation library as a package, This package can be use to carry out Newton optimisation

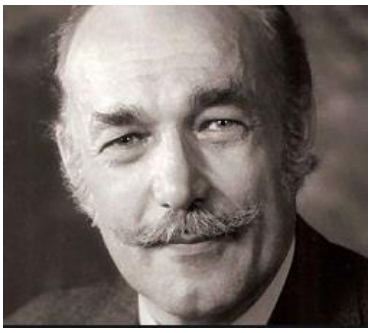
```
using Pkg; Pkg.add("Optim")
```

```
#Newton Optimisation by using Optim LBFGS

using Optim
func(x)=3.0*x[1]*x[1]-4.0*x[1]*x[2]+2.0*x[2]*x[2]-x[1]-x[2]
x=[1.0 1.0]
z=optimize(func,x, LBFGS())
print(z)
x=Optim.minimizer(z)
print("x = $x")
#y=func(x)
y=Optim.minimum(z)
print("\nminimum  = $y")
```

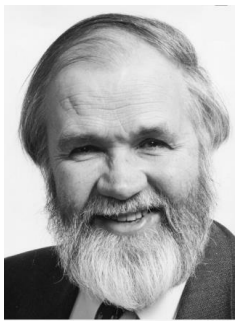
```
julia> include("opt_newton1.jl")
 * Status: success

 * Candidate solution
    Final objective value:     -1.125000e+00

 * Found with
    Algorithm:     L-BFGS

 * Convergence measures
    |x - x'|               = 1.39e-01 ≰ 0.0e+00
    |x - x'|/|x'|          = 1.11e-01 ≰ 0.0e+00
    |f(x) - f(x')|         = 1.39e-02 ≰ 0.0e+00
    |f(x) - f(x')|/|f(x')| = 1.23e-02 ≰ 0.0e+00
    |g(x)|                 = 1.47e-10 ≤ 1.0e-08

 * Work counters
```

```
   Seconds run:   1  (vs limit Inf)
   Iterations:    2
   f(x) calls:    7
   ∇f(x) calls:   7
x = [0.9999999999994034 1.2500000000357743]
minimum  = -1.1249999999999991
julia>
```

NELDER – MEAD SIMPLEX NONLINEAR MULTIVARIABLE OPTIMIZATION METHOD

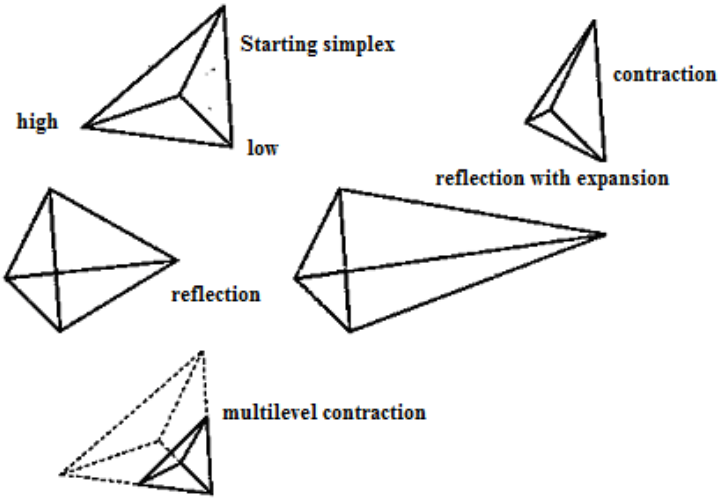


John Nelder



Roger Mead

The simplex method developed by Nelder and Mead method is a multidimensional search method . Simplex is an n dimensional geometric entity. It contains (n+1) points in n dimensional space. The method uses the concept of a simplex, which is a special polytop of $N + 1$ vertices in N dimensions. Examples of simplices include a line segment on a line, a triangle on a plane, a tetrahedron in three-dimensional space and so forth. Required processes for the simplex amobea's movements are shown in the figure



Process can be given as follows

- **1. Order** according to the values of the vertices:
  $\boldsymbol{f(x_1) \leq f(x_2) \leq \cdots \leq f(x_{n+1})}$
- **2. Calculate** x_0, the center of gravity off all points except x_{n+1}
- **3. Reflection:** Compute reflected point $x_r = x_0 + \alpha(x_0 - x_{n+1})$

If the reflected point is better than the second worst, but not better than the best, i.e.:
$\boldsymbol{f(x_1) \leq f(x_2) \leq f(x_n)}$
then obtain a new simplex by replacing the worst point x_{n+1} with the reflected point x_r, and go to step1

- **4. Expansion:**

If the reflected point is the best point so far, $\boldsymbol{f(x_r) < f(x_1)}$ **Then** compute the expanded point
$x_e = x_0 + \gamma(x_0 - x_{n+1})$
If the expanded point is better than the reflected point, $\boldsymbol{f(x_e) < f(x_r)}$ **then** obtain a new simplex

by replacing the worst point x_{n+1} with the expanded point x_e, and go to step 1.

Else obtain a new simplex by replacing the worst point x_{n+1} with the reflected point x_r, and go to step 1.

Else (i.e. reflected point is worse than second worst) continue at step 5.

- **5. Contraction:**

Here, it is certain that , $\boldsymbol{f(x_r) < f(x_n)}$

Compute contracted point $x_c = x_{n+1} + \rho(x_0 - x_{n+1})$

If the contracted point is better than the worst point, i.e. $\boldsymbol{f(x_c) < f(x_{n+1})}$

then obtain a new simplex by replacing the worst point x_{n+1} with the contracted point x_c, and go to step 1.

Else go to step 6.

- **6. Reduction:**

For all but the best point with

$x_i = x_1 + \sigma(x_i - x_1)$ for all $i \in \{2, \dots, n+1\}$ goto step 1

Note: $\boldsymbol{\alpha, \gamma, \rho}$ and σ are respectively the reflection, the expansion, the contraction and the shrink coefficient. Standard values are $\boldsymbol{\alpha = 1, \gamma = 2, \rho = 1/2}$ and $\boldsymbol{\sigma = 1/2}$

For the **reflection**, since x_{n+1} is the vertex with the higher associated value among the vertices, we can expect to find a lower value at the reflection of x_{n+1} in the opposite face formed by all vertices point x_i except x_{n+1}.

For the **expansion**, if the reflection point x_r is the new minimum along the vertices we can expect to find interesting values along the direction from x_o to x_r.

Concerning the **contraction**: If $f(x_r) > f(x_n)$ we can expect that a better value will be inside the simplex formed by all the vertices x_i.

The initial simplex is important, indeed, a too small initial simplex can lead to a local search, consequently the NM can get more easily stuck. So this simplex should depend on the nature of the problem. Instead of given n+1 point same process can be done by giving one point and a change vector by defining

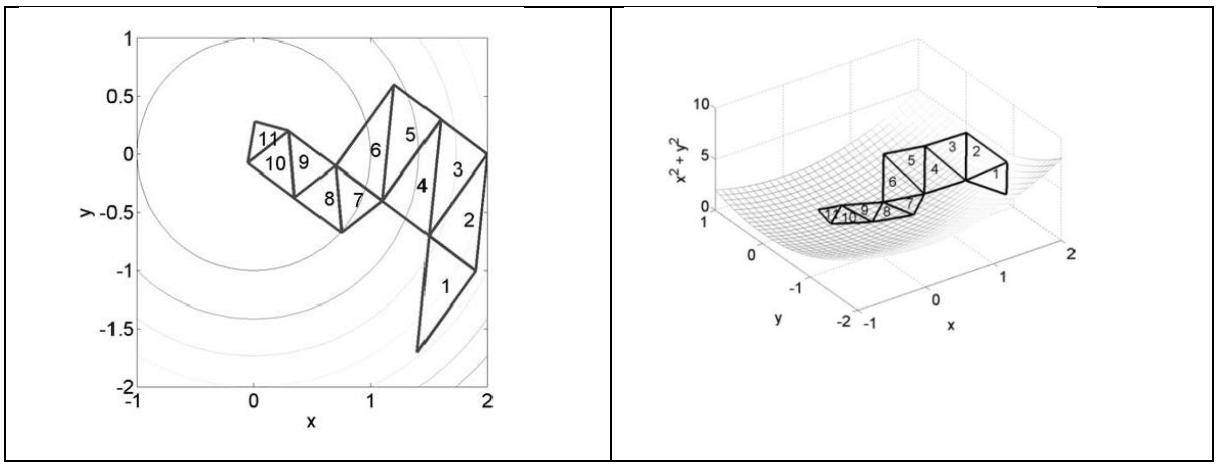
$x_{i-1} = P_i + \lambda\, dx_j$ where if i=j λ=1 else if $i \neq j$ λ=0 .

For example:

if $P = \begin{Bmatrix} a \\ b \\ c \end{Bmatrix}$ and $dx = \begin{Bmatrix} da \\ db \\ dc \end{Bmatrix}$

$$x_0 = \begin{Bmatrix} a \\ b \\ c \end{Bmatrix} \quad x_1 = \begin{Bmatrix} a + da \\ b \\ c \end{Bmatrix} \quad x_2 = \begin{Bmatrix} a \\ b + db \\ c \end{Bmatrix} \quad x_3 = \begin{Bmatrix} a \\ b \\ c + dc \end{Bmatrix}$$

Figure 6.2-1 Movements of the nelder-Mead simplex element to find the optimum point



Nelder and Mead simplex method

```
 # 6-2. Nelder and Mead Simplex multivariable nonlinear optimization method
#  Nelder & Mead 1965 Computer J, v.7, 308-313.

function  nelder(fnelder,a,da)
            maxiteration=100
            tolerance=1e-6
            global pavg=0.0
   i=1
            j=1
            flo=0.0
            il0=1
            fhi=0.0
            ihi=1
            fnhi=0.0
            inhi=1
            fe=0.0
            fr=0.0
            fc=0.0
            fcc=0.0
            n=length(a)
            n1=n+1
x=zeros(Float64,n1,n)
x1=zeros(Float64,n)
global p=zeros(Float64,n1,n1)
f1=zeros(Float64,n)
for i=1:n1
  for j=1:n
     if i==j
       x[i,j]=a[j]+da[j]
       x1[j]=x[i,j]
       p[i,j]=x[i,j]
     else
       x[i,j]=a[j]
       x1[j]=x[i,j]
       p[i,j]=x[i,j]
     end # if i==j
  end # for j=1:n
  p[i,n1]=fnelder(x1)
end # for i=1:n1
   # Inlet variable definitions
   # fnelder : abstract multivariable function f(x)
   # x : independent variable set of n+1 simplex elements
   # maxiteration : maximum iteration number
   # tolerance :
            NDIMS = n
            NPTS = n1
            FUNC = n1
            ncalls = 0
            ave = zeros(Float64,NDIMS)
            # reflect
            r =zeros(Float64,NDIMS)
            # shrink:
    c=zeros(Float64,NDIMS)
    # shrink all dimensions
    cc=zeros(Float64,NDIMS)
    # expand
    e = zeros(Float64,NDIMS)
        # construct the starting simplex #########
        # p  [NPTS,NPTS] # [row,col] = [whichvx,coord,FUNC]
        z=zeros(Float64,NDIMS)
        best = 1e99
        global pavg=0.0
        # ####### calculate the first function values for the simplex ########
        iter=1
        for iter=1:maxiteration
                #  ####/  define lo, nhi, hi (low high next_to_high #######
```

```
            #println("Block 1 : entry to iter loop iter= $iter \n")
            #show(IOContext(stdout, :limit=>false), MIME"text/plain"(), p)
            #println("\n flo= $flo fhi $fhi fnhi = $fnhi fr = $fr fe= $fe fc = $fc fcc=$fcc\n")
  ihi=1
  fnhi=0.0
  inhi=1
  fe=0.0
  fr=0.0
  fc=0.0
  fcc=0.0
ilo=1
ihi=1
inhi = -1 # -1 means missing
flo = p[1,FUNC]
fhi = flo
sterr=0.0
for i=2:NPTS
   if p[i,FUNC] < flo
     flo=p[i,FUNC]
     ilo=i
   end  # if p[i,FUNC] < flo
   if p[i,FUNC] > fhi
     fhi=p[i,FUNC]
     ihi=i
   end # p[i,FUNC] > fhi
end # for i=1:NPTS
fnhi = flo
inhi = ilo
for i=1:NPTS
  if (i != ihi) && (p[i,FUNC] > fnhi)
    fnhi=p[i,FUNC]
    inhi=i
  end # if (i != ihi) && (p[i,FUNC] > fnhi)
end # for i=1:NPTS
##### exit criteria #######
ix=iter % (4*NDIMS);
if ix == 1
    # calculate the average (including maximum value)
    pavg=0.0
    for i=1:NPTS
      pavg+=p[i,FUNC]
    end # for i=1:NPTS
    pavg/=NPTS
    tot=0.0
    for i=1:NPTS
      tot=(p[i,FUNC]-pavg)*(p[i,FUNC]-pavg)
    end #  for i=1:NPTS
    sterr=sqrt(tot/NPTS)
    for j=1:NDIMS
       z[j]=p[ilo,j]
    end # for j=1:NDIMS
    best = p[ilo,FUNC]
 end # if (iter % 4*NDIMS) == 0
##/ calculate avarage without maximum value ###
ave = zeros(Float64,NDIMS)
for i=1:NPTS
  if i != ihi
    for j=1:NDIMS
       ave[j] += p[i,j]
    end # for j=1:NDIMS
  end # if i != ihi
end # for i=1:NPTS
for j=1:NDIMS
    ave[j] /= (NPTS-1.0)
end #  for j=1:NDIMS
####/ reflect ########
for j=1:NDIMS
  r[j] = 2.0*ave[j] - p[ihi,j]
  r1=r[j]
  r2=ave[j]
```

```
  r3=p[ihi,j]
  #print("j = $j  r = $r1 ave = $r2 pihi,j = $r3\n")
end # for j=1:NDIMS
fr = fnelder(r)
if (flo <= fr) && (fr < fnhi)  # in zone: accept
   for j=1:NDIMS
     p[ihi,j] = r[j]
   end # for j=1:NDIMS
   p[ihi,FUNC] = fr
   continue
end # if (flo <= fr) && (fr < fnhi)
if  fr < flo  ## expand
   for j=1:NDIMS
     e[j] = 3.0*ave[j] - 2.0*p[ihi,j]
   end # for j=1:NDIMS
   fe = fnelder(e)
   if fe < fr
      for j=1:NDIMS
        p[ihi,j] = e[j]
      end # for j=1:NDIMS
      p[ihi,FUNC] = fe
      continue
   else
      for j=1:NDIMS
        p[ihi,j] = r[j]
      end # for j=1:NDIMS
      p[ihi,FUNC] = fr
      continue
   end # if fe < fr
 end # if fr < flo

######/ shrink:
if fr < fhi
   for  j=1:NDIMS
     c[j] = 1.5*ave[j] - 0.5*p[ihi,j]
   end # for  j=1:NDIMS
   fc = fnelder(c)
   if fc <= fr
      for  j=1:NDIMS
        p[ihi,j] = c[j]
      end # for  j=1:NDIMS
      p[ihi,FUNC] = fc
      continue
   else   ###/ shrink
      for i=1:NPTS
        if  i != ilo
           for  j=1:NDIMS
             p[i,j] = 0.5*p[ilo,j] + 0.5*p[i,j]
             x1[j]=p[i,j]
           end # for  j=1:NDIMS
           p[i,FUNC] = fnelder(x1)
        end # if  i != ilo
      end # for i=1:NPTS
      continue
  end # if fc <= fr
end # if fr < fhi
if  fr >= fhi   #/
   for  j=1:NDIMS
     cc[j] = 0.5*ave[j] + 0.5*p[ihi,j]
   end # for  j=1:NDIMS
   fcc = fnelder(cc)
   if (fcc < fhi)
      for  j=1:NDIMS
        p[ihi,j] = cc[j]
      end # for  j=1:NDIMS
      p[ihi,FUNC] = fcc
      continue
   else    ####/
      for i=1:NPTS
        if (i != ilo)
```

```
                        for  j=1:NDIMS
                          p[i,j] = 0.5*p[ilo,j] + 0.5*p[i,j]
                          x1[j]=p[i,j]
                        end # for  j=1:NDIMS
                        p[i,FUNC] = fnelder(x1)
                      end # if (i != ilo)
                    end # for i=1:NPTS
                  end # if (fcc < fhi)
              end #  if  fr >= fhi
              end  # for iter=1:maxiteration
      return z
 end

function  nelder(fnelder,a)
 n=length(a)
 da=zeros(Float64,n)
 for i=1:n
   dx=0.1*rand(Float64)
   da[i]=dx*a[i]
 end # for
return nelder(fnelder,a,da)
end # function

func(x)=(x[1]-2.0)*(x[1]-2.0)*(x[1]-2.0)*(x[1]-2.0)+(x[2]-1.0)*(x[2]-1.0)+
3.0*(x[3]-1.0)*(x[3]-1.0)*(x[3]-1.0)*(x[3]-1.0)*(x[3]-1.0)*(x[3]-1.0)
a=[1.0 2 3]
z=nelder(func,a)
print("\nz = $z")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_nelder_mead.jl

z = [1.9998420725538413, 0.9999993740596262, 1.0116906510607162]
> Terminated with exit code 0.
```

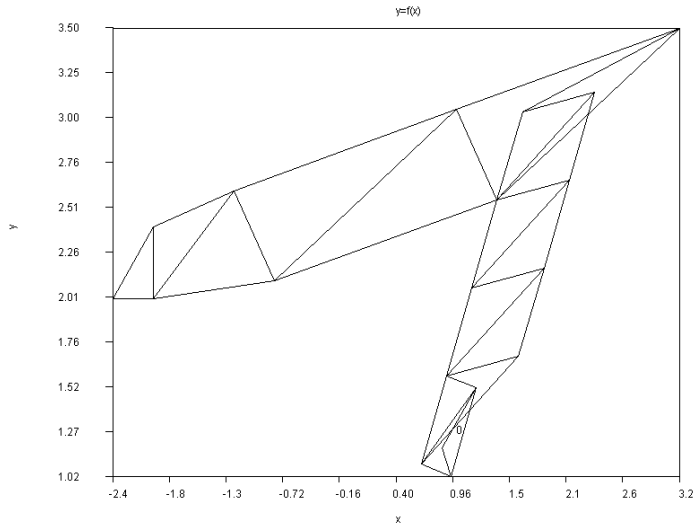
EXAMPLE PROBLEM

It is desired to find the minimum of function $\mathbf{f(x_1,x_2)=3x_1^2-4x_1x_2+2x_2^2-x_1-x_2}$ with the initial value of {x}=[-2,2]
Function written in java :

```
func(x)=3.0*x[1]*x[1]-4.0*x[1]*x[2]+2.0*x[2]*x[2]-x[1]-x[2]
a=[-2.0 2]
z=nelder(func,a)
print("\nz = $z")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_nelder_mead.jl

z = [0.9999999887469639, 1.249999982589196]
> Terminated with exit code 0.
```

Nelder-Mead actual amobea movement for the example function



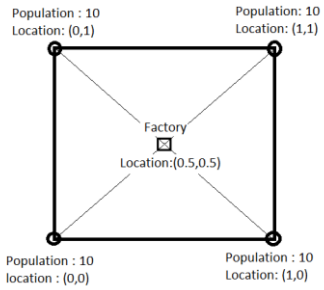
EXAMPLE PROBLEM

As a second example factory space selection problem is taken. In this problem, every city has a number of potential customer. In order to minimize transportation cost of the product new factory should be located in a place minimizing the following function

$$f = \sum_{i=1}^{n} N_i[(x - x_i)^2 + (y - y_i)^2]$$

Where N_i population of the city i and , x_i and y_i are location of the city. x,y are the location of the factory. Following city location function is given as a test function in file city.txt

Population	X coordinate of the city	Y coordinate of the city
10	0	0
10	0	1
10	1	0
10	1	1



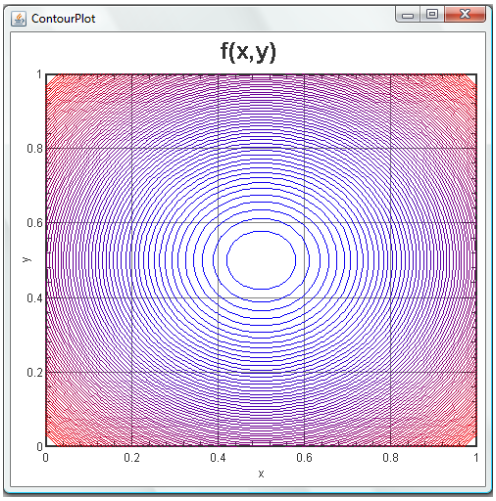
It is clear that optimum point for such a data will be located at [0.5,0.5]. In the example program nelder mead method is taken from class NA41

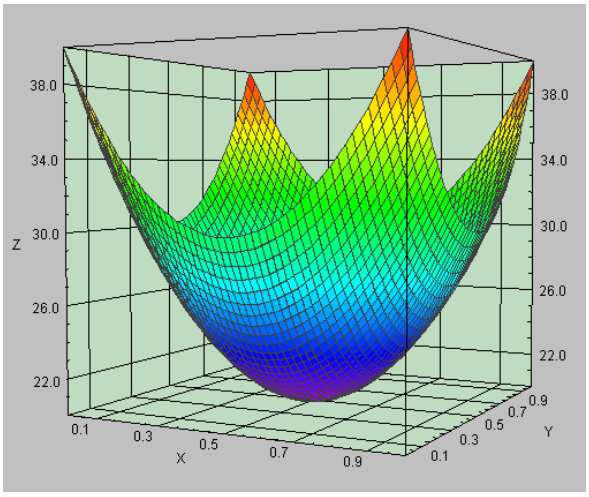
factory placement optimization

```
 function func(x)
n=4
a=[10.0 10 10 10;0.0 0 1 1;0.0 1 0 1]
ff=0.0
for i=1:n
  ff+=a[1,i]*((x[1]-a[2,i])*(x[1]-a[2,i])+(x[2]-a[3,i])*(x[2]-a[3,i]))
end
return ff
end # function
```

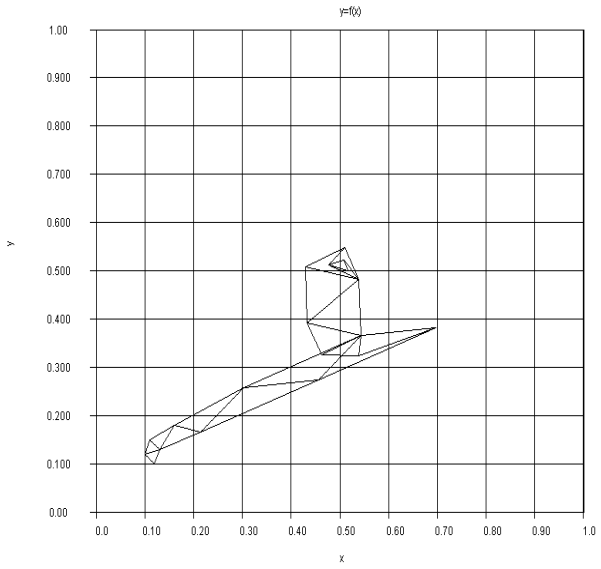
```
a=[1.0 1]
z=nelder(func,a)
print("\nz = $z")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_nelder_mead.jl

z = [0.5000000019636436, 0.5000000043279331]
> Terminated with exit code 0.
```



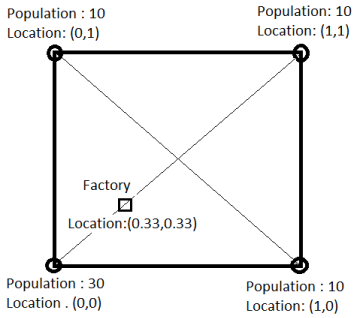



```
 function  nelder(fnelder,a)
 n=length(a)
 da=zeros(Float64,n)
 for i=1:n
   dx=0.1*rand(Float64)
   da[i]=dx*a[i]
 end # for
return nelder(fnelder,a,da)
end # function

function func(x)
n=4
a=[10.0 10 10 10;0.0 0 1 1;0.0 1 0 1]
ff=0.0
for i=1:n
  ff+=a[1,i]*((x[1]-a[2,i])*(x[1]-a[2,i])+(x[2]-a[3,i])*(x[2]-a[3,i]))
end
return ff
end # function


a=[0.1 0.1]
z=nelder(func,a)
print("\nz = $z")
```



What will happen if the population of one of the city is different, for example

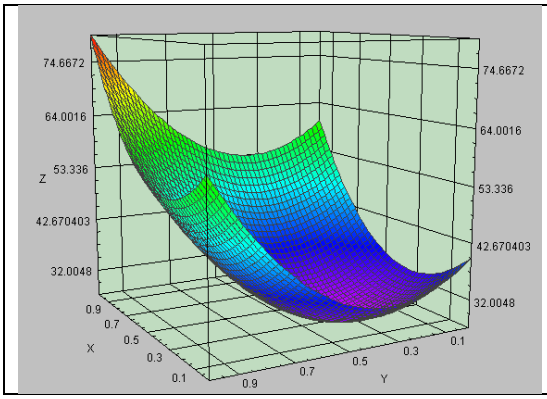
Population	X coordinate of the city	Y coordinate of the city
30	0	0
10	0	1
10	1	0
10	1	1



Then optimum point will move to:

```
function func(x)
n=4
a=[30.0 10 10 10;0.0 0 1 1;0.0 1 0 1]
ff=0.0
for i=1:n
  ff+=a[1,i]*((x[1]-a[2,i])*(x[1]-a[2,i])+(x[2]-a[3,i])*(x[2]-a[3,i]))
end
return ff
end # function


a=[0.1 0.1]
z=nelder(func,a)
print("\nz = $z")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_nelder_mead.jl

z = [0.33333333193655074, 0.33333333362547546]
> Terminated with exit code 0.
```



Let us look at a realistic version of the example problem. Cities Latitude and longitudes of Turkish cities. What will be ideal place to locate the factory for minimising the transportation cost. It is assumed that each person has the same purchasing potential

The optimum factory location in Turkey:

Place Name	2021 Population	Latitude	Longitude
Adana, Turkey	1248900	37	35.321335
Afyonkarahisar, Turkey	146136	38.756886	30.538704
Amasya, Turkey	82896	40.652382	35.828819
Ankara, Turkey	3517182	39.925533	32.866287
Antalya, Turkey	758188	36.884804	30.704044
Artvin, Turkey	24266	41.183224	41.830982
Bolu, Turkey	96629	40.731647	31.589813
Bursa, Turkey	1412701	40.193298	29.074202
Denizli, Turkey	313238	37.783333	29.094715
Diyarbakir, Turkey	644763	37.91	40.240002
Edirne, Turkey	126470	41.674965	26.583481
Elazığ, Turkey	298004	38.680969	39.226398
Eskişehir, Turkey	514869	39.766193	30.526714
Gaziantep, Turkey	1065975	37.066666	37.383331
Giresun, Turkey	98864	40.917534	38.392654
Hatay, Turkey	210000	36.200001	36.166668
Isparta, Turkey	172334	37.768002	30.561905
Istanbul, Turkey	14804116	41.015137	28.97953
Izmir, Turkey	2500603	38.423733	27.142826
Izmit, Kocaeli, Turkey	196571	40.766666	29.916668
Kahramanmaras, Turkey	376045	37.575275	36.922821
Kayseri, Turkey	592840	38.734802	35.467987
Konya, Turkey	875530	37.874641	32.493156
Malatya, Turkey	441805	38.356869	38.309669
Manisa, Turkey	243971	38.630554	27.422222
Mersin, Turkey	537847	36.812103	34.641479
Nevşehir, Turkey	75527	38.626995	34.719975
Rize, Turkey	92772	41.025513	40.517666
Sakarya, Turkey	1042649	40.783333	30.4
Sanliurfa, Turkey	449549	37.158333	38.791668
Sivas, Turkey	264022	39.750359	37.015598
Tekirdag, Turkey	122287	40.977779	27.515278
Trabzon, Turkey	255083	41.002697	39.716763
Van, Turkey	371713	38.499817	43.378143
Yozgat, Turkey	87881	39.82206	34.808132
Zonguldak, Turkey	100229	41.451733	31.791344

```
function func(x)
n=36
a=[1248900   146136   82896   3517182   758188   24266   96629   1412701   313238   644763   126470
     298004   514869   1065975   98864   210000   172334   14804116   2500603   196571   376045   592840
```

```
          875530   441805   243971   537847   75527    92772    1042649  449549   264022   122287   255083
          371713   87881    100229;
37        38.756886 40.652382 39.925533 36.884804 41.183224 40.731647 40.193298 37.783333 37.91    41.674965 38.680969
          39.766193 37.066666 40.917534 36.200001 37.768002 41.015137 38.423733 40.766666 37.575275 38.734802 37.874641
          38.356869 38.630554 36.812103 38.626995 41.025513 40.783333 37.158333 39.750359 40.977779 41.002697 38.499817
          39.82206  41.451733;
35.321335 30.538704 35.828819 32.866287 30.704044 41.830982 31.589813 29.074202 29.094715 40.240002 26.583481 39.226398
          30.526714 37.383331 38.392654 36.166668 30.561905 28.97953  27.142826 29.916668 36.922821 35.467987 32.493156
          38.309669 27.422222 34.641479 34.719975 40.517666 30.4     38.791668 37.015598 27.515278 39.716763 43.378143
          34.808132 31.791344]
ff=0.0
for i=1:n
  ff+=a[1,i]*((x[1]-a[2,i])*(x[1]-a[2,i])+(x[2]-a[3,i])*(x[2]-a[3,i]))
end
return ff
end # function


a=[0.1 0.1]
z=nelder(func,a)
print("\n[Latitude,Longitude] = ",z)
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_nelder_mead.jl
[Latitude,Longitude] = [39.75600091923573, 31.245720445008125]
> Terminated with exit code 0.
```

Sakarya(40.70,30.4) could be the ideal place to locate the factory. Other alternatives will be Eskisehir(38.68,30.52) or Ankara(39.92,32.87)

Julia has optimisation library as a package, This package can be use to carry out Nelder-Mead optimisation
Let us utilise the program now, to use much more cities anddifferent populations for each cities

```
using Pkg; Pkg.add("Optim")
```

```
# 6-2. Nelder and Mead Simplex multivariable nonlinear optimization method
#  Nelder & Mead 1965 Computer J, v.7, 308-313.
using Optim
func(x)=(x[1]-2.0)*(x[1]-2.0)*(x[1]-2.0)*(x[1]-2.0)+(x[2]-1.0)*(x[2]-1.0)+
3.0*(x[3]-1.0)*(x[3]-1.0)*(x[3]-1.0)*(x[3]-1.0)*(x[3]-1.0)*(x[3]-1.0)
a=[1.0,2.0,3.0]
z=optimize(func,a, NelderMead())
print(z)
x=Optim.minimizer(z)
print("x = $x")
#y=func(x)
y=Optim.minimum(z)
print("\nminimum  = $y")
```

```
---------- Capture Output ----------
```

```
> "C:\co\Julia\bin\julia.exe" opt_nelder_mead3.jl
x = [1.991433188031417, 0.9999753150134735, 0.9683777585648278]
minimum  = 8.9951752856918e-9
> Terminated with exit code 0.
```

```
include("opt_nelder_mead3.jl")
 * Status: success

 * Candidate solution
   Final objective value:     8.995175e-09

 * Found with
   Algorithm:     Nelder-Mead

 * Convergence measures
   √(Σ(yᵢ-ȳ)²)/n ≤ 1.0e-08

 * Work counters
   Seconds run:   0  (vs limit Inf)
   Iterations:    65
   f(x) calls:    122
x = [1.991433188031417, 0.9999753150134735, 0.9683777585648278]
minimum  = 8.9951752856918e-9
```

NON-LINEAR MULTIDIMENSIONAL STEEPEST DESCENT METHOD

For the n dimensional non lineear f(x_0,x_1,x_2,…,x_n) function , we would like to find minimum value of the function. The General definition of the Newton –Raphson approach can be written as:

$$\{x^{m+1}\} = \{x^m\} + \alpha^m [H^m]^{-1}[-\{\nabla f^m\}]$$

where

$$\nabla f(x_1, x_2, x_3, .., x_n) = \begin{Bmatrix} \frac{\partial f(x_1,x_2,x_3,..,x_n)}{\partial x_1} \\ \frac{\partial f(x_1,x_2,x_3,..,x_n)}{\partial x_2} \\ \frac{\partial f(x_1,x_2,x_3,..,x_n)}{\partial x_3} \\ \cdots\cdots \\ \frac{\partial f(x_1,x_2,x_3,..,x_n)}{\partial x_n} \end{Bmatrix} \text{ and } [H] = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_1 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \frac{\partial^2 f}{\partial x_2 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \frac{\partial^2 f}{\partial x_3 \partial x_1} & \frac{\partial^2 f}{\partial x_3 \partial x_2} & \frac{\partial^2 f}{\partial x_3^2} & \cdots & \frac{\partial^2 f}{\partial x_3 \partial x_n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \frac{\partial^2 f}{\partial x_2 \partial x_n} & \frac{\partial^2 f}{\partial x_3 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

is Hessian Matrix, α is a step function. In steepest descent method the value of inverse hessian, $[H]^{-1}$ is taken as unit vector [I] Then the definition of the equation becomes:

$$\{x^{m+1}\} = \{x^m\} + \alpha^m [-\{\nabla f^m\}]$$

In this method, As a first action a starting point for the iteration will be selected. If the starting function is P_0

$P^0 = f(x_1^0, x_2^0, x_3^0, .., x_n^0)$

In each iteration step values of the function and derivatives are calculated.

$d^k = -\nabla f(x_1^k, x_2^k, x_3^k, .., x_n^k)$ k=0..n

Then for each x value is increased by d^k with an unknown multiplier α. In the new equation as only unkown remains is unknown multiplier α (so equation converted to a one dimensional equation as a function of α)

$f(\alpha^k) = f(x^k + \alpha^k d^k) = f\left(\left[x_1^k - \alpha^k \frac{\partial f(x_1^k,x_2^k,x_3^k,..,x_n^k)}{\partial x_1}\right], \left[x_2^k - \alpha^k \frac{\partial f(x_1^k,x_2^k,x_3^k,..,x_n^k)}{\partial x_2}\right], \ldots, \left[x_n^k - \alpha^k \frac{\partial f(x_1^k,x_2^k,x_3^k,..,x_n^k)}{\partial x_n}\right]\right)$

α^k value minimizing function $f(\alpha^k)$ is found by using one of the one dimensional minimisation methods. The new x^{k+1} value calculated by using

$$x^{k+1} = x^k + \alpha^k d^k$$

Process continues until the error is small.

EXAMPLE PROBLEM

Find minimum value of function $f(x_0, x_1) = 3x_0^2 - 4x_0x_1 + 2x_1^2 - x_0 - x_1$with the initial value of

$\{x^0\} = \begin{Bmatrix} x_0^0 \\ x_1^0 \end{Bmatrix} = \begin{Bmatrix} -2 \\ 2 \end{Bmatrix}$ by using steepest descent method.

$$\nabla f(x_0, x_1) = \begin{Bmatrix} 6x_0^0 - 4x_1^0 - 1 \\ -4x_0^0 + 4x_1^0 - 1 \end{Bmatrix} = \begin{Bmatrix} -21 \\ 15 \end{Bmatrix}$$

$$\begin{Bmatrix} x_0^1 \\ x_1^1 \end{Bmatrix} = \begin{Bmatrix} x_0^0 \\ x_1^0 \end{Bmatrix} + \alpha^0 \begin{Bmatrix} 6x_0^0 - 4x_1^0 - 1 \\ -4x_0^0 + 4x_1^0 - 1 \end{Bmatrix} = \begin{Bmatrix} -2 \\ 2 \end{Bmatrix} + \alpha^0 \begin{Bmatrix} -21 \\ 15 \end{Bmatrix} = \begin{Bmatrix} -2 - 21\alpha^0 \\ 2 + 15\alpha^0 \end{Bmatrix}$$

If these values are substituted into the function

$f(x_0, x_1) = 3x_0^2 - 4x_0x_1 + 2x_1^2 - x_0 - x_1$ it becomes

$f(\alpha^0) = 3(-2 - 21\alpha^0)^2 - 4(-2 - 21\alpha^0)(2 + 15\alpha^0) + 2(-2 - 21\alpha^0)^2 - (-2 - 21\alpha^0) - (2 + 15\alpha^0)$

$f(\alpha^0) = 36 + 666\alpha^0 + 3033\alpha_0^2$

In order to find minimum of this function, analitical solution will be used in this example

$f'^{(\alpha^0)} = 666 + 6066\alpha^0 = 0 \;\; \alpha^0 = -0.109792284$

If this value is substituted into x_0^1 and x_1^1

For α^0=-0.109792284 $\begin{Bmatrix} x_0^1 \\ x_1^1 \end{Bmatrix} = \begin{Bmatrix} 0.3056379 \\ 0.35311572 \end{Bmatrix}$

For α^1=-1.121212 $\begin{Bmatrix} x_0^2 \\ x_1^2 \end{Bmatrix} = \begin{Bmatrix} 0.9544105 \\ 1.2613973 \end{Bmatrix}$

For α^2=-0.109792284 $\begin{Bmatrix} x_0^3 \\ x_1^3 \end{Bmatrix} = \begin{Bmatrix} 0.9894 \\ 1.2363 \end{Bmatrix}$

For α^3=-1.121212 $\begin{Bmatrix} x_0^4 \\ x_1^4 \end{Bmatrix} = \begin{Bmatrix} 0.9993 \\ 1.250173 \end{Bmatrix}$

The same problem is also solved in the algorithm

Program steepest descent algorithm

```
 # Multidimensional non-linear steepest descent algorithm

# one dimensional first derivative
#first derivative
function df(f,x)
 dx=0.001
 c=[1.0/280.0 -4.0/105.0  1.0/5.0  -4.0/5.0  0  4.0/5.0  -1.0/5.0  4.0/105.0 -1.0/280.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/dx
end

#one dimensional second derivative
function d2f(f,x)
 dx=0.001
 c=[-1.0/560.0  8.0/315.0 -1.0/5.0 8.0/5.0 -205.0/72.0 8.0/5.0 -1.0/5.0 8.0/315.0 -1.0/560.0]
 ff=0.0
 for i=-4:4
   ff+=c[i+5]*f(x+dx*i)
 end
 return ff/(dx*dx)
end

# multideimensional first derivative
# first derivative
function dfunc(func,x)
 global  dx=0.001
```

```
 global c=[1.0/280.0 -4.0/105.0          1.0/5.0    -4.0/5.0  0          4.0/5.0    -1.0/5.0   4.0/105.0 -1.0/280.0]
 n=length(x)
 ff=zeros(Float64,n)
 global yy=ff
 x1=zeros(Float64,n)
  for ix=1:n
    for i=-4:4
     # create x1set for derivative input
     # ================================
     for k=1:n
       xx=x[k]
       if k!=ix
         x1[k]=xx
       else
         delta=Float64(i)*dx
         x1[k]=xx+delta
       end
      end # for k
      # ================================
      ff1=func(x1)
      ff[ix]+=c[i+5]*ff1
      yx=ff[ix]
    end # for i
    ff[ix]/=dx
    yx=ff[ix]
  end # for ix
 return ff
end # for function

# one dimensional reduced function
# to be utilised in one dimensional optimisation routine
function f1dim(ff1,pc,xc,x)
   n=length(pc)
   xt =zeros(Float64,n)
   for j=1:n
      xt[j]=pc[j]+x*xc[j]
   end # for
   yy=ff1(xt)
   return yy
end # function

# minimisation function
function linmin(ff1,pc,xc)
   f1(x)=f1dim(f,pc,xc,x)
   xi=0.0
   xmin=newton(f1,xi)
   return xmin
end # function


# one dimensional newton-raphson (secant) optimisation
function newton(f,x)
# Newton-Raphson(secant) optimisation method
# single variable
nmax=100
tolerance=1.0e-10
for i=0:nmax
  dfx=df(f,x)
  d2fx=d2f(f,x)
  x-=dfx/d2fx
  if abs(dfx)<tolerance
    return x
```

```
  end #f
end #for
return x
end # function

# adding vectors
function add(left,right)
   #addition of two vectors
   n1=length(left)
   n2=length(right)
   if n1>=n2
     nMax=n1
   else
     nMax=n2
   end # if n1>=n2
   b =zeros(Float64,nMax)
   for i=1:n1
      b[i]=b[i]+left[i]
   end #  for i=1:n1
   for i=1:n2
      b[i]=b[i]+right[i]
   end # for i=1:n2
   return b
end # function add(left,right)

# multiplying vectors
function multiply_c_v(left,right)
   # multiplying a vector with a constant
   n=length(right)
   b = zeros(Float64,n)
   for i=1:n
      b[i]=left*right[i]
   end # for
   return b
end # function

# steepest descent optimisation
function steepestdescent(f,x)
   # steepestdescent method
   ti=1.0
   nmax=100
   tolerance=1.0e-10
   n=length(x)
   b =ones(Float64,n)
   xc =zeros(Float64,n)
   i=0
   total=0.0
   for i=1:n
     total+=b[i]
   end # for
   i=0
   while i<nmax && total>tolerance
      xc=dfunc(f,x)
      pc=x
      lm=linmin(f,pc,xc)
      b= multiply_c_v(lm,xc)
      x=add(x,b)
      total=0
      for j=1:n
         total+=abs(b[j])
      end
      i=i+1
```

```
    end # while
    return x
end # function

x=[-2,2]
n=length(x)
f(x)=3.0*x[1]*x[1]-4.0*x[1]*x[2]+2.0*x[2]*x[2]-x[1]-x[2]
r=steepestdescent(f,x)
print("r= $r")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_steepest_descent.jl
r= [0.9999995947658935, 1.2499992827564628]
> Terminated with exit code 0.
```

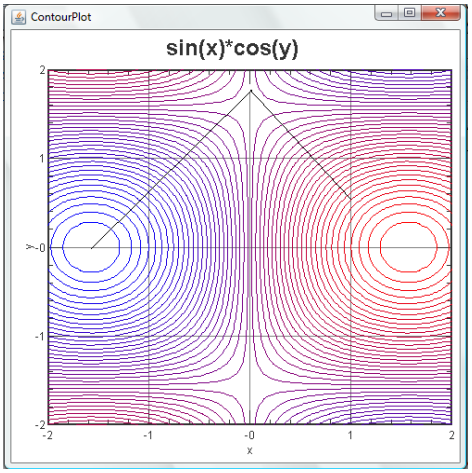
EXAMPLE PROBLEM

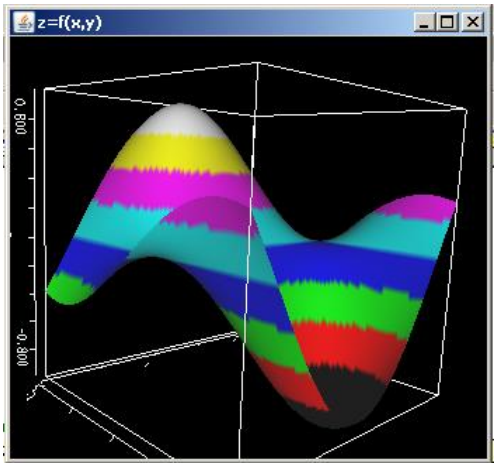
Find the minimum of the function $f(x_1,x_2)=\sin(x_1)\cos(x_2)$ with the initial starting point $\{x\}=[1,0.5]$

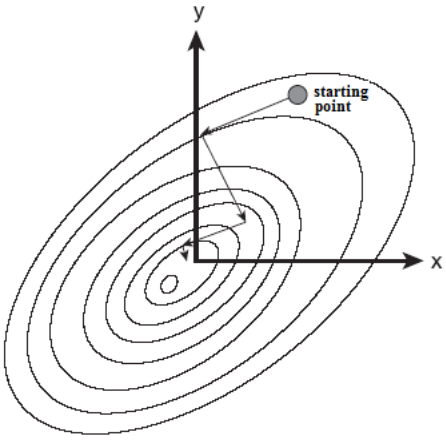
Sample function used in the problem

```
 x=[1.0,0.5]
n=length(x)
f(x)=sin(x[1])*cos(x[2])
r=steepestdescent(f,x)
print("r= $r")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_steepest_descent.jl
r= [1.5707963530335516, -2.2306434420770096e-8]
> Terminated with exit code 0.
```







STOCHASTIC METHODS: MONTE-CARLO METHOD

Monte carlo optimization as a method is very simple. Points between the given limits are selected randomly and evaluated. When number of points increase to very high numbers, statistically there will be a homogeneous coverege of all the search region. Record the best optimum found so far.What is achived is similar to plotting data, but in monte carlo method taken constant steps and walk through the steps are not needed because randomness takes care of uniformity of search when the total number is big enough.

```
 #function set
function func(x)
ff=1.0+cos(pi*(x[1]-3.0))*cos(2.0*pi*(x[2]-2.0))/(1+(x[1]-3.0)*(x[1]-3.0)+(x[2]-2.0)*(x[2]-2.0))
return ff
end

function monte_carlo_opt(func,imin,imax,N)
  # monte-Carlo optimisation method
  min=imin
  max=imax
  n=length(min)
  global xmax=zeros(Float64,n)
  global x=zeros(Float64,n)
  max_number=-1.0e99
  for k=1:N
    for i=1:n
      x[i]=min[i]+(max[i]-min[i])*rand()
    end #for i=1:n
    f=func(x)
    if f>max_number
       max_number=f
       for i=1:n
         xmax[i]=x[i]
       end #for i=1:n
    end # if f>max_number
 end # for k
 return xmax
    end # function

imin=[-10.0 -10.0]
imax=[10.0 10.0]
N=10000
r=monte_carlo_opt(func,imin,imax,N)
y=func(r)
println("r = $r y = $y")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_newton.jl
r = [3.112835283143717, 1.9680878153402759] y = 1.9065714544579773

> Terminated with exit code 0.
```

Sample test program:

```
 #function set
function func(x)
ff=x[1]*sin(10.0*pi*x[1])+3.0
return ff
end

function monte_carlo_opt(func,imin,imax,N)
  # monte-Carlo optimisation method
  min=imin
  max=imax
  n=length(min)
  global xmax=zeros(Float64,n)
  global x=zeros(Float64,n)
  max_number=-1.0e99
  for k=1:N
    for i=1:n
      x[i]=min[i]+(max[i]-min[i])*rand()
    end #for i=1:n
```

```
    f=func(x)
    if f>max_number
       max_number=f
       for i=1:n
         xmax[i]=x[i]
       end #for i=1:n
    end # if f>max_number
  end # for k
  return xmax
    end # function

imin=[-1.0]
imax=[2]
N=10000
r=monte_carlo_opt(func,imin,imax,N)
y=func(r)
println("r = $r y = $y")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" opt_monte_carlo.jl
r = [1.8504975256833687] y = 4.8502714882191444

> Terminated with exit code 0.
```

STOCHASTIC METHODS: PARTICLE SWARM ALGORITHM

Particle swarm optimization (PSO) was developed by Kennedy and Eberhart in 1995, based on swarm behaviour such as fish and bird schooling in nature. PSO searches the space of an objective function by adjusting the trajectories of individual agents, called particles. The movement of the particles has both stochastic and deterministic elements. Each particle is attracted toward the best location of the given time and the best location of the near past (a current global). It has also tendency to move randomly. When a particle finds a location that is better than any previously found locations, then it updates it as the new current best for particle *i*.It also remembers all the currents best values to update a current global value. The aim is to find the global best among all the current best values available until no more improvements are available.

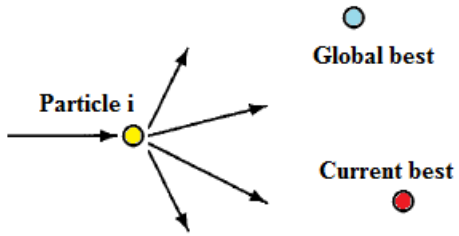


Figure 5.23.1 Particle will attract toward the global best, current best and also move randomly

The method is given in Table 5.23.1 as pseodocode. If x_i and v_i position vector and velocity for particle i, the new velocity vector is determined as:

$v_i^{t+1} = v_i^t + \alpha^* \varepsilon_1^* (g^* - x_i^t) + \beta^* \varepsilon_2^* (x_i^* - x_i^t)$

Where ε_1 and ε_1 are two random vectors between 0 and 1, g^* is the current global best x_i^* is local best. The parameters α and β are the learning parameters or acceleration constants. The initial velocity of the particles can be taken as zero, $v_i^{t=0} = 0$. The new position can then be updated by $x_i^{t+1} = x_i^t + v_i^{t+1}$. Although v_i can be any values, it is bounded in some range $[0, v_{max}]$.

There are many variants of the standart PSO algorithm. The most noticiable one is the version in which previous velocity function multiplied with an inertia function. This is called an accelerated PSO.

$v_i^{t+1} = \theta v_i^t + \alpha^* \varepsilon_1^* (g^* - x_i^t) + \beta^* \varepsilon_2^* (x_i^* - x_i^t)$

This is equivalent to introduce a virtual mass to stabilise the motion of the particles.

Table 5.23.1 Particle swarm algorithm

Objective function $f(P_i)$, $P_i = (x_0, ..., x_m)_i$ for particle i and m space dimension variable
Initialize locations P_i and velocity V_i of n particles by random selection Use as velocity vector: $V_i^{t+1} = \theta V_i^t + \alpha^* random()[GPmin - x_i^t] + \beta^* random()[Pmin - x_i^t]$

Where θ is inertia function(an iterative variable between 0 and 1)
α and β are learning parameters or acceleration constants
$\alpha = \alpha_0 \gamma^t$ where $0 \leq \gamma \leq 1$ $\alpha_0 \cong 0.5..1$ for t is the virtual time
Find Global best GPmin from min{ $f(P_0)\ldots f(P_n)$} (at t = 0)

while (criterion)

t = t + 1 (pseudo time or iteration counter)
for loop over all n particles and all m dimensions

Generate new velocity V_i^{t+1}
Calculate new locations $P_i^{t+1} = P_i^t + V_i^{t+1}$
Evaluate objective functions at new locations
Find the current best for each particle Pmin
Find the global best for each particle GPmin

end for
end while
Output the final results Pmin and GPmin

```
#function set
function func(x)
ff=(1.0-x[1])*(1.0-x[1])+100.0*(x[2]-x[1]*x[1])*(x[2]-x[1]*x[1])
return ff
end

function particle_swarm_opt(ff,mi,  xxmin, xxmax, alphai, betai,ngenerationi)
teta=0.5
gamma=0.5
#initial population set
n=length(xxmin)
m=mi
alpha=alphai
beta0=betai
beta=beta0
ngeneration=ngenerationi
x=zeros(Float64,m,n)
v=zeros(Float64,m,n)
y=zeros(Float64,m)
xmin=zeros(Float64,n)
gmin=zeros(Float64,n)
x1=zeros(Float64,n)
ygmin=0
x2=0
ngeneration=ngenerationi
yxmin=1.0e50
# ======================
for i=1:m
 for j=1:n
   x2=xxmin[j]+rand()*(xxmax[j]-xxmin[j])
   x[i,j]=x2
   x1[j]=x2
   v[i,j]=rand()
  end # for j=1:n
  y[i]=ff(x1)
  y1=y[i]
  if y1<yxmin
     yxmin=y1
     ygmin=y1
    for k=1:n
     xmin[k]=x[i,k]
     gmin[k]=x[i,k]
    end #end k=1:n
  end #end if y[i]<yxmin
end #i=1:m
# end of init
# ======================
t=0
while(t<ngeneration)
yxmin=1.0e50
x2=0
```

```
for i=1:m
  for j=1:n
    x2=x[i,j]+v[i,j]
    x1[j]=x2
    x[i,j]=x2
    if x[i,j]<xxmin[j]
      x[i,j]=xxmin[j]
      x1[j]=xxmin[j]
    end # if
    if x[i,j]>xxmax[j]
      x[i,j]=xxmax[j]
      x1[j]=xxmax[j]
    end # if
  end # for j
  y[i]=ff(x1)
  if y[i]<yxmin
   yxmin=y[i]
   for k=1:n
     xmin[k]=x[i,k]
   end       # for k=1:n
  end # if y[i]<yxmin
end           # for i=1:m
for i=1:m
  if y[i]<ygmin
    ygmin=y[i]
    for k=1:n
      gmin[k]=x[i,k]
    end # for k=1:n
  end # if y[i]<ygmin
end # for i=1:m
for i=1:m
  for j=1:n
    v[i,j]=teta*v[i,j]+alpha*rand()*(gmin[j]-xmin[j])+beta*rand()*(xmin[j]-x[i,j])
  end # for j=1:n
end # end for i=1:m
f1=ff(xmin)
t=t+1
beta=beta0*gamma^(0.01*t)
end # while
return gmin
end # function

xmin=[0.0 0.0]
xmax=[2.0 2.0]
m=100
alpha=2.04865767
beta=0.78742864
ngeneration=500
r1=particle_swarm_opt(func,m, xmin,xmax,alpha,beta,ngeneration)
y=func(r1)
println("r = $r1 y = $y")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" particle_swarm_opt.jl
r = [1.0, 1.0] y = 0.0

> Terminated with exit code 0.
```

STOCHASTIC METHODS: BAT SEARCH ALGORITHM

Xin-She Yang (2010) [51] propose the Bat Algorithm (BA), BA is inspired by the research on the social behavior of bats. The BA is based on the echolocation behaviour of bats. Microbats use a type of sonar (echolocation) to detect prey, avoid obstacles, and locate their roosting crevices in the dark. These bats emit a very loud sound pulse and listen for the echo that bounces back from the surrounding objects. Their pulses vary in properties and can be correlated with their hunting strategies, depending on the species. Based on the above description of bat process, Xin-She Yang proposes the Bat algorithm. The structure of the pseudo code of the Bat Algorithm is as follows :

Table Bat Search algorithm

Objective function $f(x_1, x_2, \ldots, x_n)$
Initialize the bat population x_i (i = 1, 2, …, n) and v_i
Define pulse frequency f_i at x_i
Initialize pulse rates r_i and the loudness A_i

while (t < Max Number of iterations)
{
Generate new solutions by adjusting frequency
And updating velocities and locations/solutions:

$$f_i = f_{min} + (f_{max} - f_{min})\beta$$

$$v_i^t = v_i^{t-1} + (x_i^t - x_*)f_i$$

$$x_i^t = x_i^{t-1} + v_i^t$$

If (rand >*ri*)
{ Select a solution among the best solutions
Generate a local solution around the selected best solution
}
End if
Generate a new solution by flying randomly
If (rand < A_i and $f(x_i) < f(x_*)$)
{ Accept the new solutions
Increase r_i and reduce A_i
}
End if
Rank the bats and find the current best *x**
} **End while**
Postprocess results

Bats fly randomly with velocity *vi* at position *xi* with a fixed frequency f_{min}, varying wavelength λ and loudness *A0* to search the prey, they can automatically adjust the wavelength (or frequency) of their emitted pulses and adjust the rate of pulse emission $r \in [0,1]$, depending on the proximity of their target. Although the loudness can vary from a large positive *A*0 to a minimum constant value *Amin*. $\beta \in [0,1]$is a random vector drawn from a uniform random number generator. A program using above algorithm is developed. Code are listed below:

```
function func(x)
ff1=(1.0-x[1])*(1.0-x[1])+100.0*(x[2]-x[1]*x[1])*(x[2]-x[1]*x[1])
return ff1
end # function
function dist(x,i,j,n)
   d=0.0
   for k=1:n
      d=d+(x[i,k]-x[j,k])*(x[i,k]-x[j,k])
   end # for k=1:n
   d=sqrt(d)
   return d
end # function

#    r1= bat_opt(func,m, xmin,xmax,gamma,alpha0,alphan,ngeneration)
function bat_opt(ff,mi, xxmin,xxmax,gammai,alpha0i,alphani,ngenerationi)
      # initial population set
      n=length(xxmin) # number of objective function
      m=mi
      println("m= $m n= $n")
      x=zeros(Float64,m+1,n)
      x1=zeros(Float64,m+1)
      I0=zeros(Float64,m+1)
      xmin=zeros(Float64,n)
      # x=new double[m+1,n];
      # I0=new double[m+1];
      # xmin=new double[n];
      gamma=gammai
      alpha0=alpha0i
      alphan=alphani
      alpha=alpha0i
      ngeneration=ngenerationi
      x2=0.0
      imin=0
      for j=1:n
          xmin[j]=xxmin[j]+rand()*(xxmax[j]-xxmin[j])
      end # for j=1:n
      ymin=ff(xmin)
      # end of init
      l=0;
      d=0.0;
      t=0.0;
      while l<ngeneration
```

```
        # find minimum of population
        for i=1:m
           for j=1:n
              x2=xxmin[j]+rand()*(xxmax[j]-xxmin[j])
              x1[j]=x2
              x[i,j]=x2
           end # for j=1:n
           I0[i]=ff(x1)
           if I0[i]<ymin
              ymin=I0[i]
              imin=i
              for k=1:n
                 xmin[k]=x[i,k]
              end # for k=1:n
           end # if I0[i]<ymin
        end # for i=1:m
        for j=1:n
           x[m+1,j]=xmin[j]
        end # for j=1:n
        I0[m]=ymin
        l=l+1
        fi=0.0
        fj=0.0
        for i=1:m+1
           for j=1:n
              if I0[i]<I0[j]
                 d=dist(x,i,j,n)
                 beta= I0[i]*exp(-gamma*d*d)
                 for k=1:n
                    #The best one will remain the rest will change
                    if i!=imin
                       x[i,k]=(1.0-beta)*x[i,k]+beta*x[j,k]+alpha*(rand()-0.5)
                    end # if i!=imin
                 end # for k=1:n
              else
                 for k=1:n
                    if i!=imin
                       x[i,k]=x[i,k]+alpha*(rand()-0.5)
                    end # i!=imin
                 end # for k=1:n
              end #  if I0[i]<I0[j]
           end # for j=for j=1:n
        end # for i=1:m+1
        #alpha will change by time
        #shrink to a smaller value
        alpha=alphan+(alpha0-alphan)*exp(-t)
        t=0.1*l
        end # while l<ngeneration
        return xmin
end # function

xmin=[0.0,0.0];
xmax=[2.0,2.0];
m=500
gamma=1.0
alpha0=0.5
alphan=0.01
ngeneration=500
r1= bat_opt(func,m, xmin,xmax,gamma,alpha0,alphan,ngeneration)
y=func(xmin)
print("xmin = $r1 fmin= $y")
```

```
---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" bat_opt.jl
m= 500 n= 2
xmin = [0.99584182480588, 0.9912614074873121] fmin= 1.0
> Terminated with exit code 0.
```

PROBLEMS

PROBLEM 1

In a distillation tank operates with saturated steam benzen and toluen is seperated. In order to obtain purest toluen in liquid phase (maximize the toluen in the mixture) what would be the ideal temperature?

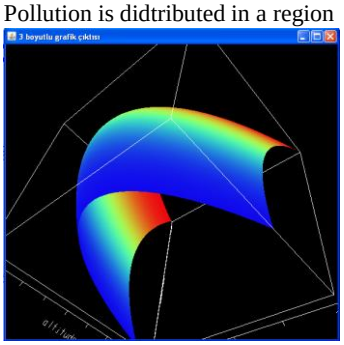
$x_{tolüen}P_{doyma\_tolüen} + x_{benzen}P_{doyma\_benzen} = P = 760$ mmHg

$\log_{10}(P_{doyma\_benzen}) = 6.905 - 1211/(T+221)$
$\log_{10}(P_{doyma\_tolüen}) = 6.953 - 1344/(T+219)$

temperature degree C, pressure mmHg

PROBLEM 2

Two dimensional distribution of pollution in a channel can be given by the equation
$C(x,y)=7.9+0.13x+0.21y-0.05x^2-0.016y^2-0.007xy$
Pollution is didtributed in a region $-10 <= x <= 10$ and $0 <= y <= 20$, find the place where the pollution is maximum



PROBLEM 3

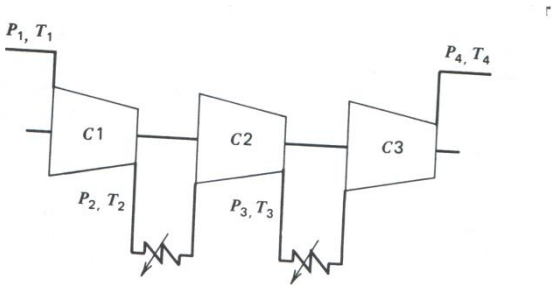
A petrochemical company developing a new fuel additive for commercial airplanes. Additive is consist of three components X,Y and Z. Fort he best performance total additive in the fuel should be at least 6 mL/L. Due o security reasons the total of most flammable X and Y components should be less than 3 mL/L. Componnt X should always be equal or more than the component Y, and componet Z should be more than half of the component Y. The cost of X,Y and Z components are 0.15,0.025 ve 0.05 TL respectively. Find the mixing component tha will give the minimum cost.

PROBLEM 4

A food company is trying to design a new container box for preserved food that will minimise the cost of material. Total 320 cm^3 volume is requested. Estetic concers limits diameter of the box in between 3 to 10 cm. and the height in between 2 to 10 cm. Box will be made of steel sheet of 0.4 mm thich with density of 7800 kg/m^3. The cost of box material is 37 TL. Assume that box including top and bottom covers are made of the same material, find the box size that will be given the minimum cost. And the cost per box.

PROBLEM 5

Air enters at 1 bar 300K to a 3 stage compressor system. After each stage air is cooled down again to 300 K before entering to the next stage. Calculate internal pressure for each stage to minimize work input to the compressors.
The work requirement for eah stage of the compressor:
$W=C_p(T_i - T_e)=C_pT_i(1-T_e/T_i)=kRT_i/(k-1)*[1-(P_e/P_i)^{(k-1)/k}]$
Total work fort he thre stages.
$W=kRT_i/(k-1)*[1-(P_2/P_1)^{(k-1)/k}]+ kRT_i/(k-1)*[1-(P_3/P_2)^{(k-1)/k}]+ kRT_i/(k-1)*[1-(P_4/P_3)^{(k-1)/k}]$
$W= kRT_i/(k-1)*[3-(P_2/P_1)^{(k-1)/k}-(P_3/P_2)^{(k-1)/k}-(P_4/P_3)^{(k-1)/k}]$
$(P_2/P_1) *(P_3/P_2)*(P_4/P_3)= (P_4/P_1)=27$
$k=1.4$ $C_p=1.005$ KJ/kg K



Work written as java function :

```
class f1 extends f_xj
```

```
{
public double func(double x[])
{
 double ff=1.4*8.3145*300/(1.4-1)*(1-x[0]*(1.4-1)/1.4)+ 1.4*8.3145*300/(1.4-1)*(1-(x[1]/x[0])*(1.4-1)/1.4)
 +1.4*8.3145*300/(1.4-1)*(1-(27/x[1])*(1.4-1)/1.4);
return ff; //minimum değil maxsimum istendiğinden - ile çarptik
}}
```

answer : 3 bar and 9 bar

PROBLEM 6

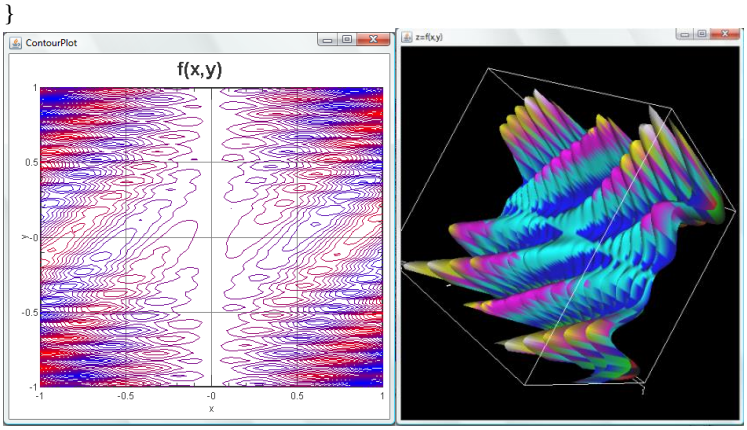
By using the data below find the aximum point. You can use an interpolation polynomial then mximise it.

X	Y
0	2
1	6
2	8
3	12
4	14
5	13
6	11
7	9
8	13
9	14.5
10	7

PROBLEM 7

Function below is given, find the maximum. -1<x [0]<1 -1<x [1]<1

```
double func(double x[])
 { double ff= 34+x[0]*(Math.sin(3.0*Math.PI*(x[0]-x[1]))+x[1]*Math.sin(20.0*Math.PI*x[1]));
   return ff;
}
```

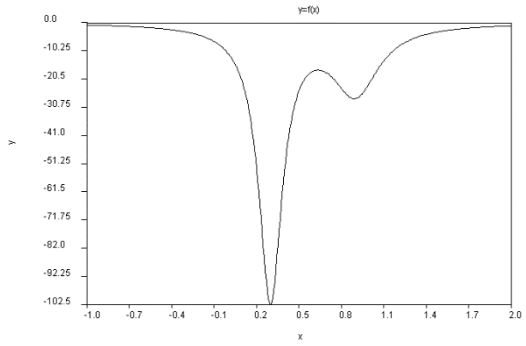


PROBLEM 8

An antibiotic producing microorganism has a specific growth rate(g) is function of food concentration (c)
If $g = 2*c/(4+0.8c+c^2+0.2c^3)$
Find the c that will give te maximum growth rate. In low food concentrations growth rate drops to zero, similarly in very high food concentrations growth rate will drop to zero due to food poisining effect. Because of this reason, values bigger than c = 10 mg/L should not be investigated.

PROBLEM 9

Find minimum of function
f(x)=-(1.0/((x-0.3)*(x-0.3)+0.01)+1.0/((x-0.9)*(x-0.9)+0.04))



PROBLEM 10

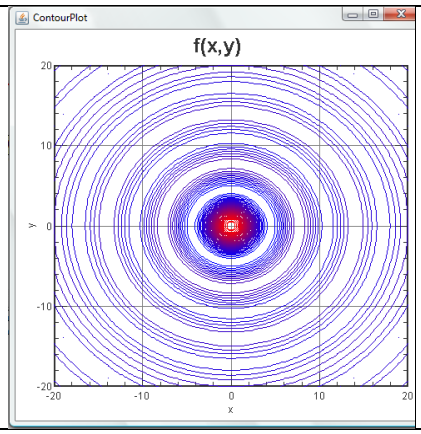
Function

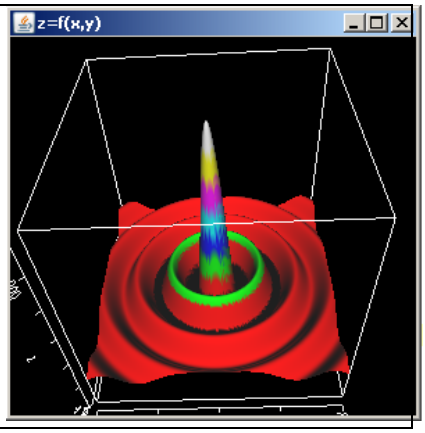
$$z(x,y)=\frac{\sin(\sqrt{x^2+y^2+\varepsilon})}{\sqrt{x^2+y^2+\varepsilon}}$$

$-20 \le x \le 20$ **ve**

$-20 \le y \le 20$

is given. Find the maximum of the function for ε=1e-2 starting from P(18,18)





PROBLEM 11

Function

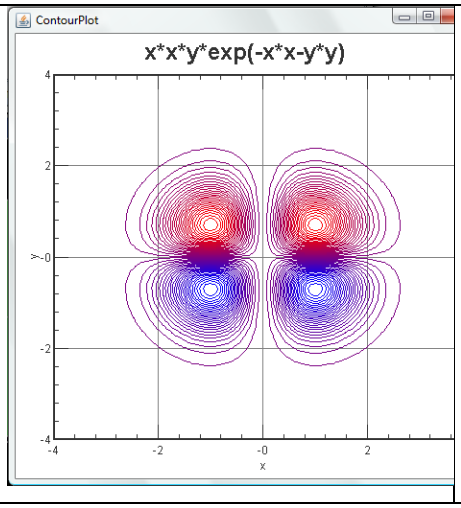
$$z(x,y)=x^2 y\, e^{(-x^2-y^2)}$$

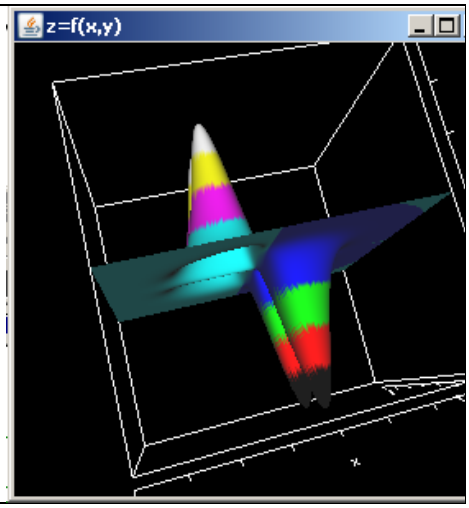
With boundaries

$-4 \le x \le 4$ **ve**

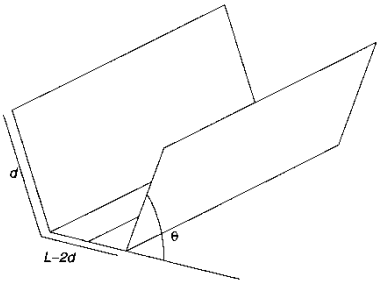
$-4 \le y \le 4$

is given. Find the minimum of the function starting from P(3,3)





PROBLEM 12



In order to construct water channel in a house , a flat sheet will be banded as shown in the figure. Water carrying capacity of the channel can be written as

$V(d,\theta) = Z(d^2 \sin\theta \cos\theta + (L-2d)d\sin\theta)$

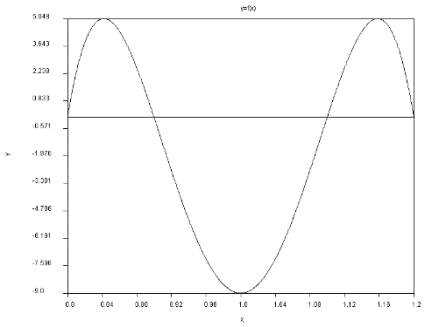
In this equation z is the length of the sheet (sheet area is zL). Find the d and θ to maximize water holding capacity of the channel.

PROBLEM 13

Function below is given

$f(x) = 2500x^6-15000x^5+14875x^4+40500x^3-97124x^2+73248x-19008$

find the minimum between x_0=0.8, x_1=1.2 values.



PROBLEM 14

Cities, A,B,C, and D located at corrdinates(0,0),(1,5),(5,2) and (7,3) respectively. The roads from all cities are merged at point P. Find point P that makes the total distance of the roads minimum.

PROBLEM 15

Function below is given

$f(x_0, x_1, x_2, x_3) = (x_0+10\ x_1)^2 +5(x_2-10\ x_3)^2+(x_1-2\ x_2)^4+10(x_1- x_4)^4$

find the minimum starting from P(1,-1,0,1)

PROBLEM 16

Function below is given

$f(x_0, x_1) = 3(x_0 - 1)^2 +2(x_1-1)\ (x_0-1)(x_1-2)+(x_1- 2)^4$

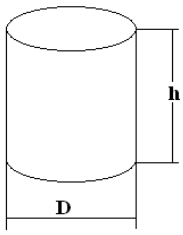
Minimum of the function is given at point (1,2). Located the minimum at 3D plot, then by using any 3D nonlinear optimization method find the minimum points starting at P (0,0)

PROBLEM 17

Find theminimum of function

$$f(x) = \frac{15x}{(4x^2 - 3x + 4)}$$ fin the range of 0 to10.

PROBLEM 18



One of the very basic optimization problem is the minimum cost of container problem. The cost of a box usually is a function of the surface area. Therefore we should minimize the area for a given volume For example if the volume of the container V=0.5 liter=$0.5x10^{-3}$ m^3 :

Volume $V = \frac{\pi D^2}{4} h$ or from this equation h, height is obtained as $h = \frac{4V}{\pi D^2}$.

Surface area of the cylinder : $A = 2\frac{\pi D^2}{4} + \pi D h = \frac{\pi D^2}{2} + \frac{4V}{D}$.

Analytical solution of the minimization problem

$$\frac{dA}{dD} = \pi D - \frac{4V}{D^2} = 0$$

$D = \sqrt[3]{\frac{4V}{\pi}}$. From here solution is $D = \sqrt[3]{\frac{4*0.5x10^{-3}}{\pi}}$ =0.086025401 m and .

$h = \frac{4*0.5x10^{-3}}{\pi D^2}$ = 0.086025401 m.

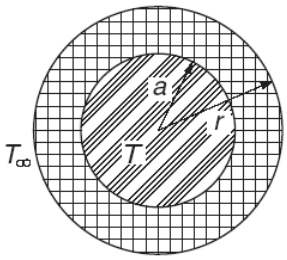
Now obtain this results by using nuerical optimization methods.

For the range of 0.01<=D<=0.2

a) Graphic method
b) Golden ratio (Fibonnachi)
c) Newton-Raphson root finding
d) Bisection method
e) Newton-Raphson method
f) Secant method
g) Brent method
h) Second degree polynomials
i) Third degree polynomials

Note: Some of these methods are defined only as root finding method, is not defined as optimization methods.

PROBLEM 19



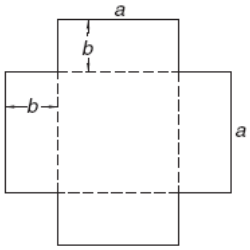
An isolation covered a wire carrying electrical current. The outlet diameter of electrical insulation is r. The electrical resistance on the wire caused heat. Thi heat crried through electrical insulation to the air around the wire with a temperature of T_∞. The temperature of the wire can be calculated as

$T = \frac{q}{2\pi}\left(\frac{\ln(r/a)}{k} + \frac{1}{hr}\right) + T_\infty$ In this equation

q = heat generation in the wire = 50 W/m
a = diameter of the wire = 5 mm = $5x10^{-3}$ m
k = coeeficient of thermal conductivity of insulation = 0.2 W/(mK)
h=thermal convetivity coefficient for air = 30 W/(m^2K)
T_∞ = environmental temperature = 300 K.

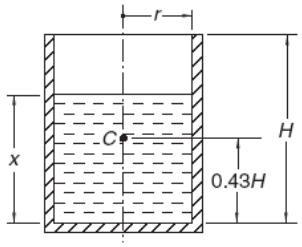
Find radius r that wil make temperature T minimum.

PROBLEM 20



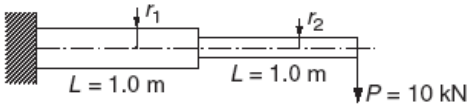
In order to make a cartoon box, the shape above should be used.After bending from the dotted linet he volume of the box will be 1 m^3. In order to spent the minimum amount of the cartoon, what a and b dimensions should be?

PROBLEM 21



A cylindrical container has a mass of M and has an empty center of gravity of C=0.43H. If the height of water in the cup is x, In order to have the common center of gravity of water and the cup as low as possible, what sholud be the value of x ?

PROBLEM 22



Two cylindrical beams with two different diameters is placed as in the figure. It is desired beam to have the minimum cross sectional area. Operational condions are:

$\sigma_1 \le 180$ MPa, $\sigma_2 \le 180$ MPa, and bending $\delta \le 25$ mm dir

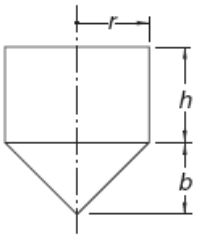
$\sigma_1 = \dfrac{8PL}{\pi\ r_1^3}$ =maximum stress in the left beam

$\sigma_2 = \dfrac{8PL}{\pi\ r_2^3}$ = maximum stress in the right beam

$\delta = \dfrac{4PL^3}{3\pi\ E}\left(\dfrac{7}{r_1^4} + \dfrac{1}{r_2^4}\right)$ =bending at the end of the beam

ve E =Young modüulus=200 GPa. Calculate r_1 and r_2.

PROBLEM 23



It is desired to have volume of the cone base shape shown in the figure as 1 m^3. Calculate r,h and b dimensions to minimise the surface area.

$V = \pi r^2\left(\dfrac{b}{3} + h\right)$

$S = \pi r\left(2h + \sqrt{b^2 + r^2}\right)$

PROBLEM 24

Lennard-Jones potential in between two molecules can be expressed as

$V = 4\varepsilon\left[\left(\dfrac{\sigma}{r}\right)^{12} - \left(\dfrac{\sigma}{r}\right)^{6}\right]$ In this equation ε and σ values are constants. Find $\left(\dfrac{\sigma}{r}\right)$ values that makes. V minimum

PROBLEM 25

Wave function for hydrogen atoms

$\psi = C(27 - 18\sigma + 2\sigma^2)e^{-\sigma/3}$ In this equation

$\sigma = zr / a_0$

$$C = \frac{1}{81\sqrt{3\pi}}\left(\frac{z}{a_0}\right)^{2/3}$$

z = nucleus electrical load
a_0 = Bohr diameter
r =radial distance.
Find σ value that make ψ wave function minimum.

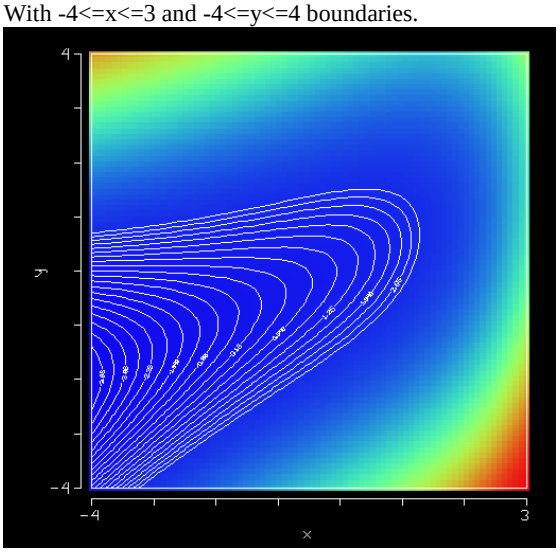
PROBLEM 26
Find optimum of function

$$f = e^{x_1} - x_1 x_2 + x_2^2$$

$$g = 2x_1 + x_2 - 2 \le 0$$

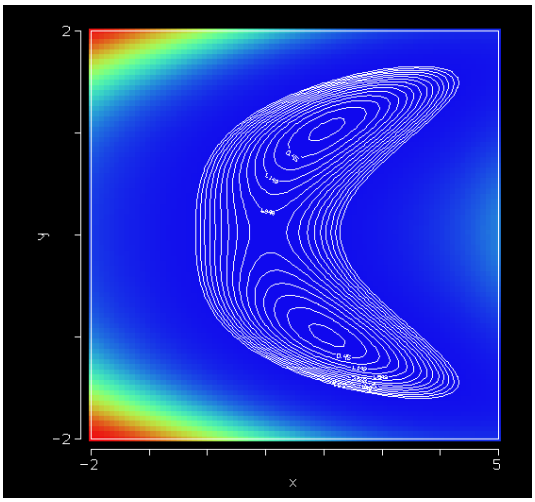
$$h = x_1^2 + x_2^2 - 4 = 0$$

With -4<=x<=3 and -4<=y<=4 boundaries.



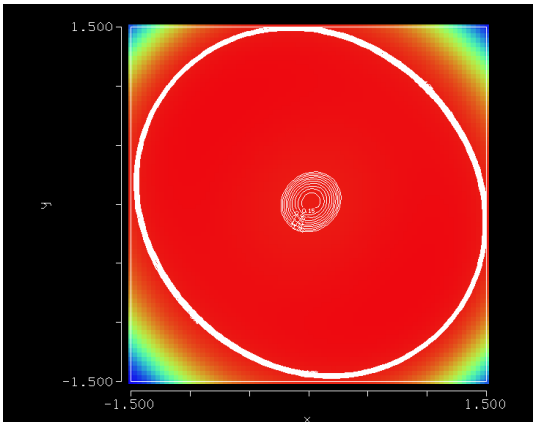
PROBLEM 27
Find the minimum of

$$f = (x-2)^2 + (x-2y^2)^2 \quad .$$



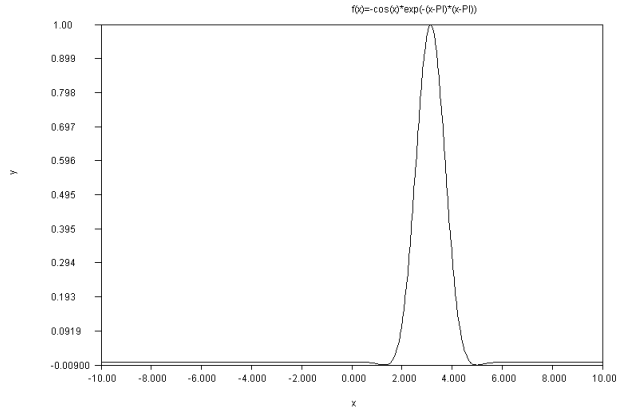
PROBLEM 28
Find the minimum of

$$f = 25x^2 - 12x^4 + 6xy + 25y^2 - 24x^2y^2 - 12y^4 \quad .$$



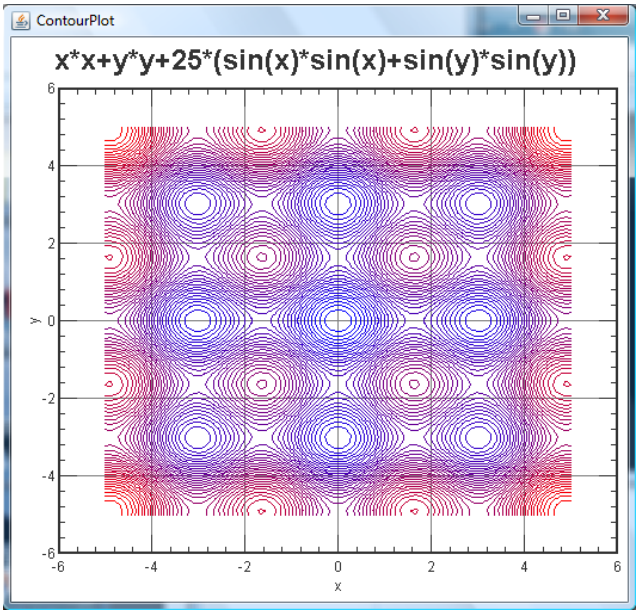
PROBLEM 29 Find maximum of Easoms's function

$f = -\cos(x)e^{-(x-\pi)^2}$ between limits [-10,10] by using Genetic algorithms



PROBLEM 30 Egg rate function

$f = x^2 + y^2 + 25[\sin^2(x) + \sin^2(y)]$ has the global minimum f=0 at (0,0) in the domain [-5,5]

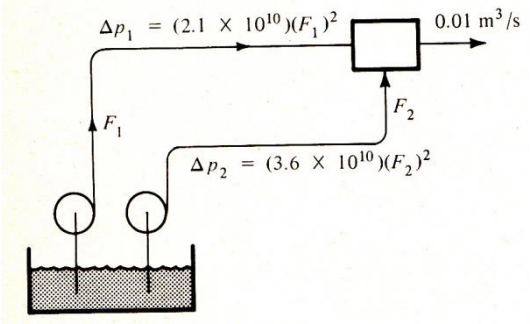


PROBLEM 31 : Two parallel pump-pipe assemblies, shown in the figure, deliver water from a common source to a common destination. The total flow rate required at the destination is $F=F_1+F_2=0.01$ m^3/s. The drops in pressure in two lines are functions of the square of flow rates.

$\Delta p_1 = 2.1x10^{10} F_1^2$ Pa and $\Delta p_2 = 3.6x10^{10} F_2^2$ Pa where F_1 and F_2 are respective flow rates in Cubic meters per second. pumping power for a pump is given by the equation

$Power = \frac{F\Delta p}{\rho\eta}$ where F is flow rate(m^3/s), Δp is pressure drop (Pa), ρ=1000 kg/m^3 is density and is pump efficiency.

η_1=0.81 and η_2=0.86

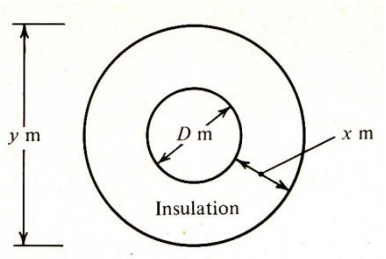


So pumping power is equal to: $Power = \frac{F_1\Delta p_1}{\rho\eta_1} + \frac{F_2\Delta p_2}{\rho\eta_2}$

$$Power = \frac{F_1\Delta p_1}{\rho\eta_1} + \frac{F_2\Delta p_2}{\rho\eta_2} = \frac{C_1F_1^3}{\rho\eta_1} + \frac{C_2F_2^3}{\rho\eta_2} = \frac{C_1F_1^3}{\rho\eta_1} + \frac{C_2}{\rho\eta_2}(F - F_1)^3$$

Solve the values of F_1 and F_2 for minimum power requirements.

PROBLEM 32: A pipe carrying high temperature water is to be insulated and then mounted in a restricted space, as shown in the figure. The choice of the pipe diameter D m and the insulation thickness x m are to be such that the OD of the insulation



Y is a minimum, but the total annual operation cost of the insulation is limited to 40000 TL. This annual operation cost has two components, the water pumping cost and the cost of the heat loss:

Pumping cost = $\frac{8}{D^5}$ and Heat cost= $\frac{1500}{x}$

If an objective function is created for this problem:

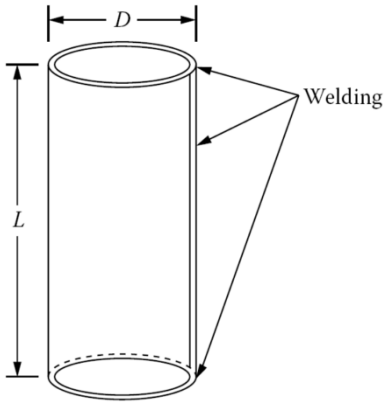
$y = D + 2x$

Subject to:

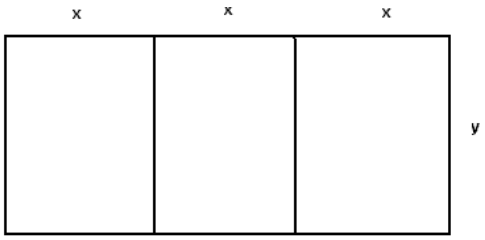
$\frac{8}{D^5} + \frac{1500}{x} = 40000$ Equation can be arranged as a single equation as:

$$y = D + 3000\left[\frac{1}{40000 - \frac{8}{D^5}}\right]$$ Find the minimum y and corresponding x and D values.

PROBLEM 33: A manufacturer of steel cans wants to minimize costs. As a first approximation, the cost of making a can consists of the cost of the metalplus the cost of welding the longitudinal seam and the top and bottom. The can may have any diameter *D* and length *L*, for a given volume *V*. The wall thickness *d* is 1 mm. The cost of the material is \$0.50/kg and the cost of welding is \$0.1/m of the weld. The density of the material is 10^4 kg/m^3 Find the dimensions of the can that will minimize cost.



PROBLEM 33: A farmer needs 150 meters of fencing to fence three adjacent gardens. What would be maximum area of each garden, and what would be values of x and y



$4y + 6x = 150$

$y = 37.5 - 1.5x$

$A(x) = 3(37.5 - 1.5x)x = 112.5x - 4.5x^2$

$\frac{dA(x)}{dx} = 112.5 - 9x = 0$

$x = 12.5$ y=18.75

$A = 703.125$ total of 3 garden

Calculate by using a numerical method

CURVE FITTING

Data is obtained by measurements in the scientific world. Scientits are generalized the measurement results by using various formulas and theories. Creating functions by using experimental numerical data is called curve fitting. Formulas are generally formed from the theories of physical laws or freely selected by the researchers. In the fitted equation usually a good fit to the experimental data is a primary requirement. Fitting means assigning values to unknown coefficients in the equations. The coefficients can be linearly or non-linearly dependent the the variables of the function(s). Interpolation is also similar process, but in the interpolation a function is not always required. Interpolation could be just a repeatetive calculation process by following a certain rule. Different curve fitting methods will be investigated in this chapter. At the end of the chapter Approximation of the functions will also be covered.

LINEAR CURVE FITTING : LEAST SQUARE METHODS

Least square methods are one of the most used curve fitting methods. Polinomial equation version is widely used. Assuming having a linear function $f(a_j,x)$ where a_j are m linear coefficient and x are independent variable and $\phi_j(x)$ are m sub-funtions linearly multiplied with the coefficients

$f(a_i, x) = \sum_{j=0}^{m} a_j^{(m)} \phi_j(x)$ (9.1-1)

It should be noted again that linearity is only for the coefficients, $\phi_j(x)$ functions does not have to be linear.

Sample functions: $f(a_i, x) = a_0 \sin(x) + a_1 \exp(x) + a_2 \ln(x) + a_3 x^2$

It is desired to fit data x_i, f_i, i=0..n into the equation so that the difference between data and fitted function dependent values for all points will be minimum. In order to establish this, the following function H will be minimized with respect to a_j

$$H(a_0^{(m)},..,a_0^{(m)}) = \sum_{i=1}^{n} w(x_i)\left[f(x_i) - \sum_{j=0}^{m} a_j^{(m)}\phi_j(x)\right]^2 \qquad (10.1-1)$$

Where $w(x_i)$ values in the equation are called weight function. In order to minimiz the function root of the derivative of the function can be calculated.

$$\frac{\partial H(a_0^{(m)},..,a_0^{(m)})}{\partial a_k^{(m)}} = 2\sum_{i=1}^{n} w(x_i)\left[f(x_i) - \sum_{j=0}^{m} a_j^{(m)}\phi_j(x)\right]\phi_k(x) = 0 \quad k = 0,\dots,m \qquad (10.1-2)$$

For weight function to be taken equal to unity , $w(x_i)$ =1, equation in the open form can be written in the following form.

$$\begin{bmatrix} \sum_{i=1}^n w(x_i)\phi_0(x_i)\phi_0(x_i) & \sum_{i=1}^n w(x_i)\phi_0(x_i)\phi_1(x_i) & \sum_{i=1}^n w(x_i)\phi_0(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^n w(x_i)\phi_0(x_i)\phi_n(x_i) \\ \sum_{i=1}^n w(x_i)\phi_1(x_i)\phi_0(x_i) & \sum_{i=1}^n w(x_i)\phi_1(x_i)\phi_1(x_i) & \sum_{i=1}^n w(x_i)\phi_1(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^n w(x_i)\phi_2(x_i)\phi_n(x_i) \\ \sum_{i=1}^n w(x_i)\phi_2(x_i)\phi_0(x_i) & \sum_{i=1}^n w(x_i)\phi_2(x_i)\phi_1(x_i) & \sum_{i=1}^n w(x_i)\phi_2(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^n w(x_i)\phi_2(x_i)\phi_n(x_i) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \sum_{i=1}^n w(x_i)\phi_n(x_i)\phi_0(x_i) & \sum_{i=1}^n w(x_i)\phi_n(x_i)\phi_1(x_i) & \sum_{i=1}^n w(x_i)\phi_n(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^n w(x_i)\phi_n(x_i)\phi_n(x_i) \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdots \\ a_n \end{Bmatrix} =$$

$$\begin{Bmatrix} \sum_{i=1}^n w(x_i)\phi_0(x_i)f(x_i) \\ \sum_{i=1}^n w(x_i)\phi_1(x_i)f(x_i) \\ \sum_{i=1}^n w(x_i)\phi_2(x_i)f(x_i) \\ \cdots \\ \sum_{i=1}^n w(x_i)\phi_n(x_i)f(x_i) \end{Bmatrix} \quad (10.1\text{-}3)$$

If $w(x_i)$=1 matrix will be

$$\begin{bmatrix} \sum_{i=1}^n \phi_0(x_i)\phi_0(x_i) & \sum_{i=1}^n \phi_0(x_i)\phi_1(x_i) & \sum_{i=1}^n \phi_0(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^n \phi_0(x_i)\phi_n(x_i) \\ \sum_{i=1}^n \phi_1(x_i)\phi_0(x_i) & \sum_{i=1}^n \phi_1(x_i)\phi_1(x_i) & \sum_{i=1}^n \phi_1(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^n \phi_2(x_i)\phi_n(x_i) \\ \sum_{i=1}^n \phi_2(x_i)\phi_0(x_i) & \sum_{i=1}^n \phi_2(x_i)\phi_1(x_i) & \sum_{i=1}^n \phi_2(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^n \phi_2(x_i)\phi_n(x_i) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \sum_{i=1}^n \phi_n(x_i)\phi_0(x_i) & \sum_{i=1}^n \phi_n(x_i)\phi_1(x_i) & \sum_{i=1}^n \phi_n(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^n \phi_n(x_i)\phi_n(x_i) \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdots \\ a_n \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^n \phi_0(x_i)f(x_i) \\ \sum_{i=1}^n \phi_1(x_i)f(x_i) \\ \sum_{i=1}^n \phi_2(x_i)f(x_i) \\ \cdots \\ \sum_{i=1}^n \phi_n(x_i)f(x_i) \end{Bmatrix} \quad (10.1\text{-}4)$$

This equation is an m+1 linear system of equation. It can easily be solved by using a system of equation solving method. A special form of the above equation a form of a polinomial can be assumed. If $\phi_j(x) = x^j$, equation becomes

$$f(a_i, x) = \sum_{j=0}^{m} a_j^{(m)} x^j \qquad (10.1-5)$$

In this case the minimisation equation becomes

$$\frac{\partial H(a_0^{(m)},..,a_0^{(m)})}{\partial a_k^{(m)}} = 2\sum_{i=1}^{n} w(x_i)\left[f(x_i) - \sum_{j=0}^{m} a_j^{(m)} x^j\right]x^k = 0 \quad k = 0,\dots,m \qquad (10.1-6)$$

$$\begin{bmatrix} \sum_{i=1}^n w(x_i) & \sum_{i=1}^n w(x_i)x_i & \sum_{i=1}^n w(x_i)x_i^2 & \cdots & \sum_{i=1}^n w(x_i)x_i^n \\ \sum_{i=1}^n w(x_i)x_i & \sum_{i=1}^n w(x_i)x_i^2 & \sum_{i=1}^n w(x_i)x_i^3 & \cdots & \sum_{i=1}^n w(x_i)x_i^{n+1} \\ \sum_{i=1}^n w(x_i)x_i^2 & \sum_{i=1}^n w(x_i)x_i^3 & \sum_{i=1}^n w(x_i)x_i^4 & \cdots & \sum_{i=1}^n w(x_i)x_i^{n+2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \sum_{i=1}^n w(x_i)x_i^n & \sum_{i=1}^n w(x_i)x_i^{n+1} & \sum_{i=1}^n w(x_i)x_i^{n+2} & \cdots & \sum_{i=1}^n w(x_i)x_i^{2n} \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdots \\ a_n \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^n w(x_i)f(x_i) \\ \sum_{i=1}^n w(x_i)x_i f(x_i) \\ \sum_{i=1}^n w(x_i)x_i^2 f(x_i) \\ \cdots \\ \sum_{i=1}^n w(x_i)x_i^n f(x_i) \end{Bmatrix} \quad (10.1-7)$$

Or if $w(x_i) = 1$

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \cdots & \sum_{i=1}^{n} x_i^n \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 & \cdots & \sum_{i=1}^{n} x_i^{n+1} \\ \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^4 & \cdots & \sum_{i=1}^{n} x_i^{n+2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \sum_{i=1}^{n} x_i^n & \sum_{i=1}^{n} x_i^{n+1} & \sum_{i=1}^{n} x_i^{n+2} & \cdots & \sum_{i=1}^{n} x_i^{2n} \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdots \\ a_n \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^{n} f(x_i) \\ \sum_{i=1}^{n} x_i f(x_i) \\ \sum_{i=1}^{n} x_i^2 f(x_i) \\ \cdots \\ \sum_{i=1}^{n} x_i^n f(x_i) \end{Bmatrix} \quad (10.1-8)$$

Linear polynomial regression

Consider the general polynomial curve fitting equation and take only first two column and row, equation will be for the linear polynomial $f(x)=a_0+a_1x$

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^{n} f(x_i) \\ \sum_{i=1}^{n} x_i f(x_i) \end{Bmatrix} \quad (10.1-9)$$

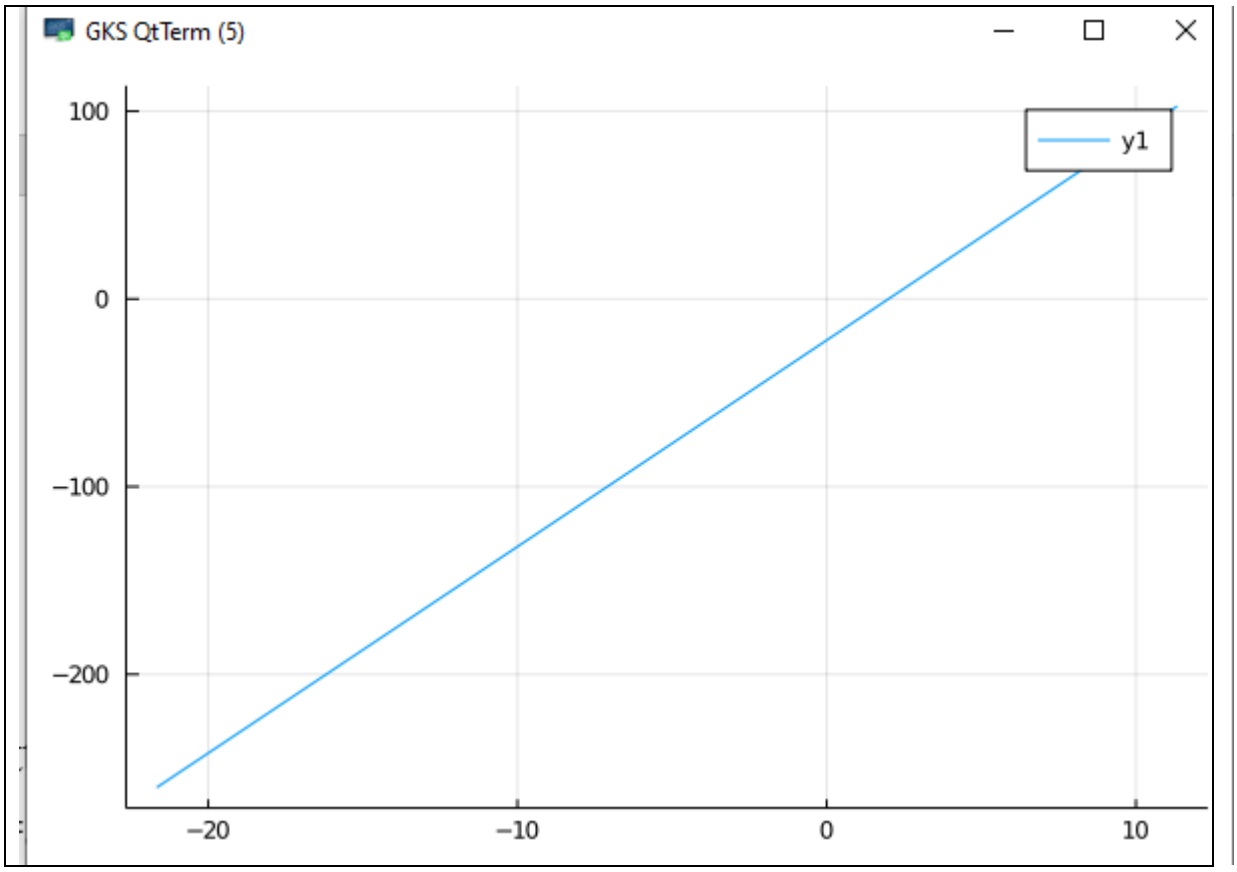
Sample data:
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64

```
 using SpecialFunctions
using Plots
 # Linear least curve fitting (regression)  y=a0+a1x
function gauss_pivot(A1,b)
            n=size(A1)[1]
            carpan=0.0
            toplam=0.0
            global x=Array{Float64}(undef,n)
            global a=Array{Float64}(undef,n,n)
            for i=1:n
              for j=1:n
                a[i,j]=A1[i,j]
              end
            end
            for k=1:n
            #pivoting
               p=k
               buyuk=abs(a[k,k])
               for ii=k+1:n
                 dummy=abs(a[ii,k])
                 if dummy > buyuk
                    buyuk=dummyp=ii
                 end #if dummy
               end #for ii
               if p!=k
                 for jj=k:n
                   dummy=a[p,jj]
                   a[p,jj]=a[k,jj]
                   a[k,jj]=dummy
                   end #for jj

                 dummy=b[p]
                 b[p]=b[k]
                 b[k]=dummy
               end #if p!
            # gauss elimination
   for i=k+1:n
               carpan=a[i,k]/a[k,k]
               a[i,k]=0.0
               for j=k+1:n
                 a[i,j]=a[i,j]-carpan*a[k,j]
               end #for j
               b[i]=b[i]-carpan*b[k]
```

```
            end #for i
            end #k
                        # back substituting
              x[n]=b[n]/a[n,n]
              n1=n-1
              for i=n1:-1:1
                toplam=0.0
                for j=i+1:n
                  toplam=toplam+a[i,j]*x[j]
                end #for j
                x[i]=(b[i]-toplam)/a[i,i]
              end #for i
            return x
end #function

function poly_1(x,y)
   n=length(x)
   x1=0.0
   x2=0.0
   y1=0.0
   y2=0.0
   for i=1:n
      x1+=x[i]
      x2+=x[i]*x[i]
      y1+=y[i]
      y2+=x[i]*y[i]
   end # for i=1:n

   a=zeros(Float64,2,2)
   a[1,1]=n
   a[1,2]=x1
   a[2,1]=x1
   a[2,2]=x2;
   b=zeros(Float64,2,2)
   b[1]=y1
   b[2]=y2
   x=gauss_pivot(a,b)
   return x
end

function func(e,x)
   # this function calculates the value of
   # least square curve fitting function
   ff=e[1]+e[2]*x
   return ff
end

function convert_to_float(column)
   new_column = map(x -> (
         x = ismissing(x) ? "" : x;  # no method matching tryparse(::Type{Float64}, ::Missing)
         x = tryparse(Float64, x);   # returns: Float64 or nothing
         isnothing(x) ? missing : x; # options: missing, or "", or 0.0, or nothing
         ), column)  # input
   # returns Array{Float64,1} OR Array{Union{Missing, Float64},1}
   return new_column
end

function readDouble(name)
  open(name) do f
  # read till end of file
  m=0
  while ! eof(f)
    m+=1
    line = readline(f)
    s=split(line," ")
    n=length(s)
    global m1=m
    global n1=n
  end  #while
  close(f)
  end  #open
  global x=Matrix{Float64}(undef,n1,m1)
  open(name) do f
  # read till end of file
  m=0
  global z=Array{Float64}(undef,n1,0)
```

```
  while ! eof(f)
    m+=1
    line = readline(f)
    s=split(line," ")
    n=length(s)
    x1=convert_to_float(s)
    z=hcat(z,x1)
    global x=z
  end  #while
  close(f)
  end  #open
  #transpose matrix
  global y=x'
  return y
  end


print("file name : ")
name=readline()
a=readDouble(name)
c=a'
n=size(a)[1]
m=size(a)[2]
global x=c[1,:]
global y=c[2,:]
println("x = $x \ny =  $y")
e=poly_1(x,y)
n1=100
x1=zeros(Float64,n1)
y1=zeros(Float64,n1)
xmax=maximum(x)
xmin=minimum(x)
dx=(xmax-xmin)/(n1-1)
for i=1:n1
    x1[i]=xmin+i*dx
    xx=x1[i]
    y1[i]=func(e,xx)
end # for i
println("x1 0 $x1 \ny1 =  $y1")
p=plot(x1,y1)
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" poly_least_linear.jl
file name : a.txt
x = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
y =  [1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
x1 0 [-21.666666666666668, -21.333333333333332, -21.0, -20.666666666666668, -20.333333333333332, -20.0, -19.666666666666668,
-19.333333333333332, -19.0, -18.666666666666668, -18.333333333333332, -18.0, -17.666666666666668, -17.333333333333336, -
17.0, -16.666666666666668, -16.333333333333336, -16.0, -15.666666666666668, -15.333333333333334, -15.0, -14.666666666666668,
-14.333333333333334, -14.0, -13.666666666666668, -13.333333333333334, -13.0, -12.666666666666668, -12.333333333333334, -
12.0, -11.666666666666668, -11.333333333333334, -11.0, -10.666666666666668, -10.333333333333334, -10.0, -9.666666666666668, -
9.333333333333334, -9.0, -8.666666666666668, -8.333333333333334, -8.0, -7.666666666666668, -7.333333333333334, -7.0, -
6.666666666666668, -6.333333333333334, -6.0, -5.666666666666668, -5.333333333333336, -5.0, -4.666666666666668, -
4.333333333333336, -4.0, -3.666666666666668, -3.3333333333333357, -3.0, -2.666666666666668, -2.3333333333333357, -2.0, -
1.6666666666666679, -1.3333333333333357, -1.0, -0.6666666666666679, -0.3333333333333357, 0.0, 0.33333333333333215,
0.6666666666666643, 1.0, 1.3333333333333321, 1.6666666666666643, 2.0, 2.333333333333332, 2.6666666666666643, 3.0,
3.333333333333332, 3.6666666666666643, 4.0, 4.333333333333332, 4.666666666666664, 5.0, 5.333333333333332,
5.666666666666664, 6.0, 6.333333333333332, 6.666666666666664, 7.0, 7.333333333333332, 7.666666666666664, 8.0,
8.333333333333332, 8.666666666666664, 9.0, 9.333333333333332, 9.666666666666664, 10.0, 10.333333333333329,
10.666666666666664, 11.0, 11.333333333333329]
y1 =  [-260.33333333333337, -256.66666666666663, -253.0, -249.33333333333334, -245.66666666666666, -242.0, -
238.33333333333334, -234.66666666666666, -231.0, -227.33333333333334, -223.66666666666666, -220.0, -216.33333333333334, -
212.66666666666669, -209.0, -205.33333333333334, -201.66666666666669, -198.0, -194.33333333333334, -190.66666666666669, -
187.0, -183.33333333333334, -179.66666666666669, -176.0, -172.33333333333334, -168.66666666666669, -165.0, -
161.33333333333334, -157.66666666666669, -154.0, -150.33333333333334, -146.66666666666669, -143.0, -139.33333333333334, -
135.66666666666669, -132.0, -128.33333333333334, -124.66666666666667, -121.0, -117.33333333333334, -113.66666666666667, -
110.0, -106.33333333333334, -102.66666666666667, -99.0, -95.33333333333334, -91.66666666666667, -88.0, -84.33333333333334, -
80.66666666666669, -77.0, -73.33333333333334, -69.66666666666669, -66.0, -62.33333333333334, -58.66666666666669, -55.0, -
51.33333333333334, -47.66666666666669, -44.0, -40.33333333333334, -36.66666666666669, -33.0, -29.333333333333346, -
25.666666666666693, -22.0, -18.333333333333346, -14.666666666666693, -11.0, -7.333333333333346, -3.6666666666666927, 0.0,
3.6666666666666536, 7.333333333333307, 11.0, 14.666666666666657, 18.333333333333307, 22.0, 25.666666666666657,
29.333333333333307, 33.0, 36.66666666666666, 40.33333333333331, 44.0, 47.66666666666666, 51.333333333333314, 55.0,
58.66666666666666, 62.333333333333314, 66.0, 69.66666666666666, 73.33333333333331, 77.0, 80.66666666666666,
84.33333333333331, 88.0, 91.66666666666661, 95.33333333333331, 99.0, 102.66666666666661]
```



Regression with quadratic polynomial

Consider the general polynomial curve fitting equation and take only first 3 column and row, equation will be for the quadratic polynomial $f(x)=a_0+a_1x+ a_2x^2$

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 \\ \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^4 \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^{n} f(x_i) \\ \sum_{i=1}^{n} x_i f(x_i) \\ \sum_{i=1}^{n} x_i^2 f(x_i) \end{Bmatrix} \quad (10.1-10)$$

Sample data:
1.0 1.0
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100

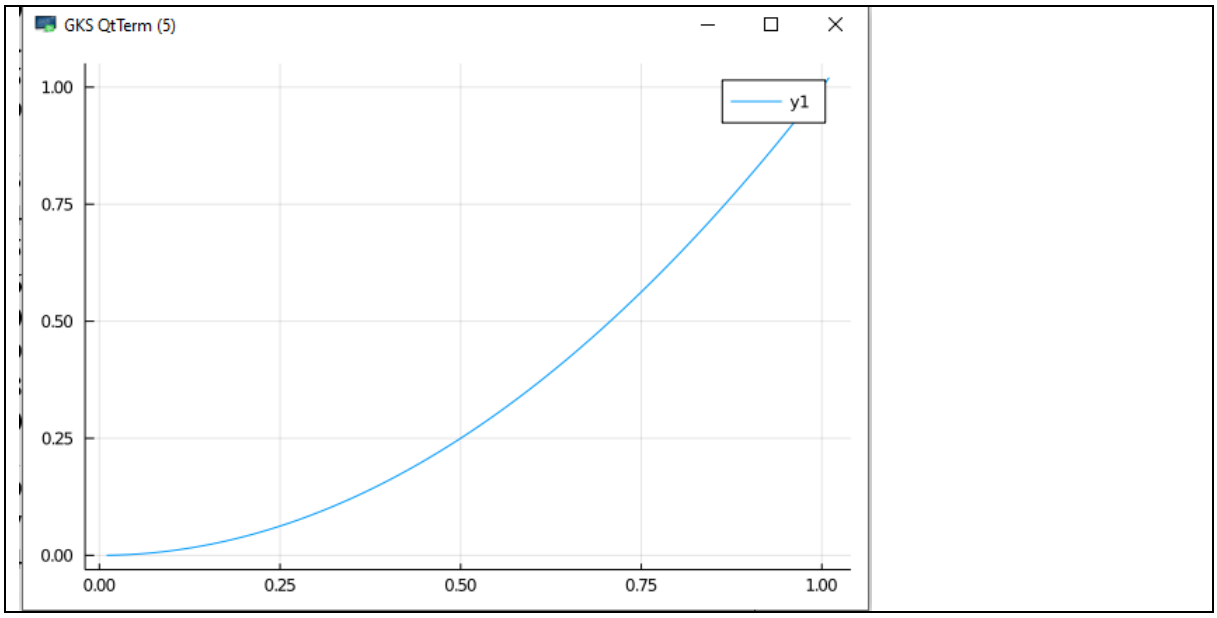
```
 using SpecialFunctions
using Plots
 # Linear least curve fitting (regression)  y=a0+a1x
function gauss_pivot(A1,b)
            n=size(A1)[1]
            carpan=0.0
            toplam=0.0
            global x=Array{Float64}(undef,n)
            global a=Array{Float64}(undef,n,n)
```

```
            for i=1:n
              for j=1:n
                a[i,j]=A1[i,j]
              end
            end
            for k=1:n
            #pivoting
               p=k
               buyuk=abs(a[k,k])
               for ii=k+1:n
                 dummy=abs(a[ii,k])
                 if dummy > buyuk
                    buyuk=dummyp=ii
                 end #if dummy
               end #for ii
               if p!=k
                 for jj=k:n
                   dummy=a[p,jj]
                   a[p,jj]=a[k,jj]
                   a[k,jj]=dummy
                   end #for jj

                 dummy=b[p]
                 b[p]=b[k]
                 b[k]=dummy
               end #if p!
            # gauss elimination
   for i=k+1:n
                 carpan=a[i,k]/a[k,k]
                 a[i,k]=0.0
                 for j=k+1:n
                   a[i,j]=a[i,j]-carpan*a[k,j]
                 end #for j
                 b[i]=b[i]-carpan*b[k]
            end #for i
            end #k
                          # back substituting
              x[n]=b[n]/a[n,n]
              n1=n-1
              for i=n1:-1:1
                toplam=0.0
                for j=i+1:n
                  toplam=toplam+a[i,j]*x[j]
                end #for j
                x[i]=(b[i]-toplam)/a[i,i]
              end #for i
            return x
end #function

function poly_1(x,y)
   n=length(x)
   x1=0.0
   x2=0.0
   x3=0.0
   x4=0.0
   y3=0.0
   y1=0.0
   y2=0.0
   for i=1:n
      x1+=x[i]
      x2+=x[i]*x[i]
      x3+=x[i]*x[i]*x[i]
      x4+=x[i]*x[i]*x[i]*x[i]
      y1+=y[i]
      y2+=x[i]*y[i]
      y3+=x[i]*x[i]*y[i]
   end # for i=1:n
   a=zeros(Float64,3,3)
   a[1,1]=n
   a[1,2]=x1
   a[1,3]=x2
   a[2,1]=x1
   a[2,2]=x2
   a[2,3]=x3
   a[3,1]=x2
   a[3,2]=x3
```

```
    a[3,3]=x4
    b=zeros(Float64,3)
    b[1]=y1
    b[2]=y2
    b[3]=y3
    x=gauss_pivot(a,b)
    return x
end

function func(e,x)
    # this function calculates the value of
    # least square curve fitting function
    ff=e[1]+e[2]*x+e[3]*x*x
    return ff
end

function convert_to_float(column)
    new_column = map(x -> (
            x = ismissing(x) ? "" : x;  # no method matching tryparse(::Type{Float64}, ::Missing)
            x = tryparse(Float64, x);   # returns: Float64 or nothing
            isnothing(x) ? missing : x; # options: missing, or "", or 0.0, or nothing
            ), column)  # input
    # returns Array{Float64,1} OR Array{Union{Missing, Float64},1}
    return new_column
end

function readDouble(name)
  open(name) do f
  # read till end of file
  m=0
  while ! eof(f)
    m+=1
    line = readline(f)
    s=split(line," ")
    n=length(s)
    global m1=m
    global n1=n
  end  #while
  close(f)
  end  #open
  global x=Matrix{Float64}(undef,n1,m1)
  open(name) do f
  # read till end of file
  m=0
  global z=Array{Float64}(undef,n1,0)
  while ! eof(f)
    m+=1
    line = readline(f)
    s=split(line," ")
    n=length(s)
    x1=convert_to_float(s)
    z=hcat(z,x1)
    global x=z
  end  #while
  close(f)
  end  #open
  #transpose matrix
  global y=x'
  return y
  end


print("file name : ")
name=readline()
a=readDouble(name)
c=a'
n=size(a)[1]
m=size(a)[2]
global x=c[1,:]
global y=c[2,:]
println("x = $x \ny =  $y")
e=poly_1(x,y)
n1=100
x1=zeros(Float64,n1)
y1=zeros(Float64,n1)
xmax=maximum(x)
```

```
xmin=minimum(x)
dx=(xmax-xmin)/(n1-1)
for i=1:n1
   x1[i]=xmin+i*dx
   xx=x1[i]
   y1[i]=func(e,xx)
end # for i
println("x1 0 $x1 \ny1 =  $y1")
p=plot(x1,y1)
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" poly_least_quadratic.jl
file name : a.txt
x = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
y =  [1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
x1 0 [0.010101010101010102, 0.020202020202020204, 0.030303030303030304, 0.04040404040404041, 0.05050505050505051,
0.06060606060606061, 0.07070707070707072, 0.08080808080808081, 0.09090909090909091, 0.10101010101010102,
0.11111111111111112, 0.12121212121212122, 0.13131313131313133, 0.14141414141414144, 0.15151515151515152,
0.16161616161616163, 0.17171717171717174, 0.18181818181818182, 0.19191919191919193, 0.20202020202020204,
0.21212121212121213, 0.22222222222222224, 0.23232323232323235, 0.24242424242424243, 0.25252525252525254,
0.26262626262626265, 0.27272727272727276, 0.2828282828282829, 0.29292929292929293, 0.30303030303030304,
0.31313131313131315, 0.32323232323232326, 0.33333333333333337, 0.3434343434343435, 0.3535353535353536,
0.36363636363636365, 0.37373737373737376, 0.38383838383838387, 0.393939393939394, 0.4040404040404041,
0.4141414141414142, 0.42424242424242425, 0.43434343434343436, 0.4444444444444445, 0.4545454545454546,
0.4646464646464647, 0.4747474747474748, 0.48484848484848486, 0.494949494949495, 0.5050505050505051, 0.5151515151515152,
0.5252525252525253, 0.5353535353535354, 0.5454545454545455, 0.5555555555555556, 0.5656565656565657, 0.5757575757575758,
0.5858585858585859, 0.595959595959596, 0.6060606060606061, 0.6161616161616162, 0.6262626262626263, 0.6363636363636365,
0.6464646464646465, 0.6565656565656566, 0.6666666666666667, 0.6767676767676768, 0.686868686868687, 0.696969696969697,
0.7070707070707072, 0.7171717171717172, 0.7272727272727273, 0.7373737373737375, 0.7474747474747475, 0.7575757575757577,
0.7676767676767677, 0.7777777777777778, 0.787878787878788, 0.797979797979798, 0.8080808080808082, 0.8181818181818182,
0.8282828282828284, 0.8383838383838385, 0.8484848484848485, 0.8585858585858587, 0.8686868686868687, 0.8787878787878789,
0.888888888888889, 0.8989898989898991, 0.9090909090909092, 0.9191919191919192, 0.9292929292929294, 0.9393939393939394,
0.9494949494949496, 0.9595959595959597, 0.9696969696969697, 0.9797979797979799, 0.98989898989899, 1.0,
1.0101010101010102]
y1 =  [0.0001020304050607081, 0.0004081216202428324, 0.0009182736455463729, 0.0016324864809713297,
0.002550760126517703, 0.0036730945821854917, 0.0049994898479746985, 0.006529945923885319, 0.008264462809917356,
0.010203040506070812, 0.01234567901234568, 0.014692378328741967, 0.017243138455259672, 0.019997959391898794,
0.022956841138659322, 0.026119783695541274, 0.029486787062544647, 0.03305785123966942, 0.03683297622691563,
0.04081216202428325, 0.044995408631772274, 0.04938271604938272, 0.053974084277114594, 0.05876951331496787,
0.06376900316294257, 0.06897255382103869, 0.07438016528925621, 0.07999183756759518, 0.0858075706560555,
0.09182736455463729, 0.09805121926334048, 0.1044791347821651, 0.11111111111111113, 0.11794714825017859,
0.12498724619936745, 0.1322314049586777, 0.1396796245281094, 0.14733190490766251, 0.15518824609733703,
0.163248648097133, 0.17151311090705035, 0.1799816345270891, 0.1886542189572493, 0.19753086419753088,
0.20661157024793392, 0.21589633710845837, 0.22538516477910422, 0.23507805325987147, 0.24497500255076016,
0.25507601265177027, 0.26538108356290185, 0.27589021528415475, 0.286603407815529, 0.29752066115702486,
0.308641975308642, 0.3199673502703807, 0.33149678604224064, 0.343230282624222, 0.35516784001632495,
0.36730945821854916, 0.3796551372308949, 0.39220487705336193, 0.40495867768595056, 0.4179165391286604,
0.4310784613814917, 0.44444444444444453, 0.4580144883175187, 0.47178859300071435, 0.4857667584940313,
0.4999489847974698, 0.5143352719110296, 0.5289256198347108, 0.5437200285685135, 0.5587184981124376, 0.5739210284664832,
0.5893276196306501, 0.6049382716049383, 0.6207529843893481, 0.6367717579838792, 0.652994592388532, 0.6694214876033059,
0.6860524436282014, 0.7028874604632181, 0.7199265381083564, 0.7371696765636161, 0.7546168758289972, 0.7722681359044997,
0.7901234567901235, 0.808182838485869, 0.8264462809917357, 0.8449137843077238, 0.8635853484338335, 0.8824609733700643,
0.9015406591164169, 0.9208244056728907, 0.9403122130394859, 0.9600040812162026, 0.9799000102030406, 1.0,
1.020304050607081]
> Terminated with exit code 0.
```



Degree n polynomial

Polynomial curve fitting equation can be solved for any degree of polynomials. Restriction of the degree of polynomial is not needed. In practical applications degree of polynomial rarly taken above degree 10

Data (a.txt)

```
1.0 1.0
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

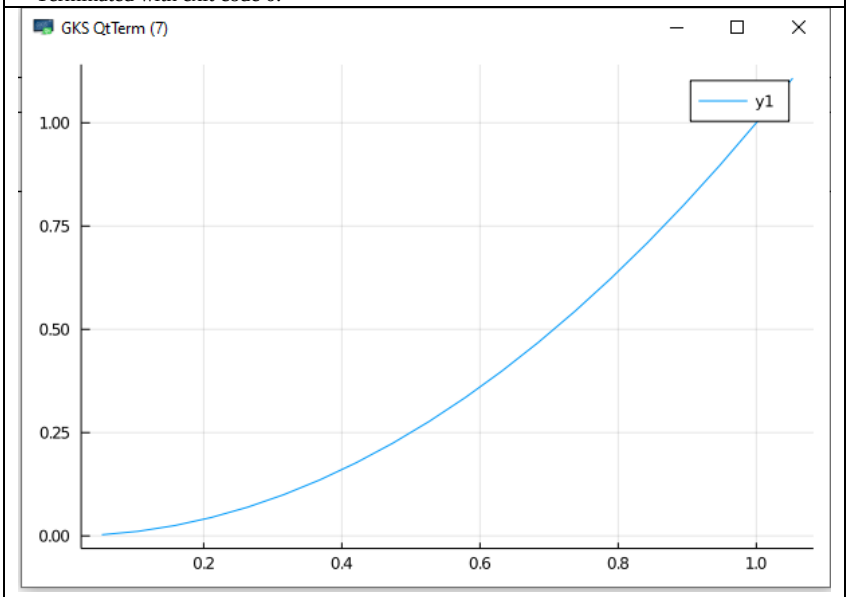
```
 using SpecialFunctions
using Plots
 # Linear least curve fitting (regression)  y=a0+a1x
function gauss_pivot(A1,b)
                n=size(A1)[1]
                carpan=0.0
                toplam=0.0
                global x=Array{Float64}(undef,n)
                global a=Array{Float64}(undef,n,n)
                for i=1:n
                  for j=1:n
                    a[i,j]=A1[i,j]
                  end
                end
                for k=1:n
                #pivoting
                   p=k
                   buyuk=abs(a[k,k])
                   for ii=k+1:n
                     dummy=abs(a[ii,k])
                     if dummy > buyuk
                        buyuk=dummyp=ii
                     end #if dummy
                   end #for ii
                   if p!=k
```

```
                    for jj=k:n
                      dummy=a[p,jj]
                      a[p,jj]=a[k,jj]
                      a[k,jj]=dummy
                      end #for jj

                    dummy=b[p]
                    b[p]=b[k]
                    b[k]=dummy
                  end #if p!
              # gauss elimination
    for i=k+1:n
                  carpan=a[i,k]/a[k,k]
                  a[i,k]=0.0
                  for j=k+1:n
                    a[i,j]=a[i,j]-carpan*a[k,j]
                  end #for j
                  b[i]=b[i]-carpan*b[k]
              end #for i
              end #k
                          # back substituting
                x[n]=b[n]/a[n,n]
                n1=n-1
                for i=n1:-1:1
                  toplam=0.0
                  for j=i+1:n
                    toplam=toplam+a[i,j]*x[j]
                  end #for j
                  x[i]=(b[i]-toplam)/a[i,i]
                end #for i
              return x
end #function

function poly_least_n(x,y,n)
    #Polynomial least square
    m=length(x)
    np1=n+1
    A =zeros(Float64,np1,np1)
    B =zeros(Float64,np1)
    X =zeros(Float64,np1)
    x1=0.0
    for i=1:np1
        for j=1:np1
            if i==1 && j==1
                A[i,j]=m;
            else
                for k=1:m
                    x1=x[k]
                    pp=x1^(i+j-2)
                    A[i,j]= A[i,j]+pp
                end # for k=1:m
            end # for if
         end # j=1:
        for k=1:m
            if i==1
                B[i]=  B[i]+y[k];
            else
                x1=x[k];
                B[i]= B[i]+x1^(i-1)*y[k]
            end # if i==0
        end # for k=1:m
    end # for i=1:np1
    show(IOContext(stdout, :limit=>false), MIME"text/plain"(), A)
    show(IOContext(stdout, :limit=>false), MIME"text/plain"(), B)
    X=gauss_pivot(A,B)
    return X
end

function func(e,x)
    # this function calculates the value of
    # least square curve fitting function
    n=length(e);
    if n!=0.0
        ff=e[n]
        for i=n-1:-1:1
            ff=ff*x+e[i]
```

```
        end # for i=n-1:-1:1
    else
        ff=0.0
    end # n!=0.0
    return ff
end

function convert_to_float(column)
    new_column = map(x -> (
            x = ismissing(x) ? "" : x;  # no method matching tryparse(::Type{Float64}, ::Missing)
            x = tryparse(Float64, x);   # returns: Float64 or nothing
            isnothing(x) ? missing : x; # options: missing, or "", or 0.0, or nothing
            ), column)  # input
    # returns Array{Float64,1} OR Array{Union{Missing, Float64},1}
    return new_column
end

function readDouble(name)
  open(name) do f
  # read till end of file
  m=0
  while ! eof(f)
    m+=1
    line = readline(f)
    s=split(line," ")
    n=length(s)
    global m1=m
    global n1=n
  end  #while
  close(f)
  end  #open
  global x=Matrix{Float64}(undef,n1,m1)
  open(name) do f
  # read till end of file
  m=0
  global z=Array{Float64}(undef,n1,0)
  while ! eof(f)
    m+=1
    line = readline(f)
    s=split(line," ")
    n=length(s)
    x1=convert_to_float(s)
    z=hcat(z,x1)
    global x=z
  end  #while
  close(f)
  end  #open
  #transpose matrix
  global y=x'
  return y
  end


print("file name : ")
name=readline()
a=readDouble(name)
c=a'
n=size(a)[1]
m=size(a)[2]
global x=c[1,:]
global y=c[2,:]
println("x = $x \ny =  $y")
e=poly_least_n(x,y,2)
print(" e = $e")
n1=20
x1=zeros(Float64,n1)
y1=zeros(Float64,n1)
xmax=maximum(x)
xmin=minimum(x)
dx=(xmax-xmin)/(n1-1)
for i=1:n1
    x1[i]=xmin+i*dx
    xx=x1[i]
    y1[i]=func(e,xx)
end # for i
# println("x1 0 $x1 \ny1 =  $y1")
```

```
p=plot(x1,y1)
```

```
 ---------- Capture Output ----------
> "C:\co\Julia\bin\julia.exe" poly_least_n.jl
file name : a.txt
x = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
y =  [1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
3Ã—3 Matrix{Float64}:
  10.0    55.0    385.0
  55.0   385.0   3025.0
 385.0  3025.0  25333.03-element Vector{Float64}:
   385.0
  3025.0
 25333.0 e = [0.0, 0.0, 1.0]
> Terminated with exit code 0.
```



A general function version of the least square program will look like :

$$\begin{bmatrix} \sum_{i=1}^{n}\phi_0(x_i)\phi_0(x_i) & \sum_{i=1}^{n}\phi_0(x_i)\phi_1(x_i) & \sum_{i=1}^{n}\phi_0(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^{n}\phi_0(x_i)\phi_n(x_i) \\ \sum_{i=1}^{n}\phi_1(x_i)\phi_0(x_i) & \sum_{i=1}^{n}\phi_1(x_i)\phi_1(x_i) & \sum_{i=1}^{n}\phi_1(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^{n}\phi_2(x_i)\phi_n(x_i) \\ \sum_{i=1}^{n}\phi_2(x_i)\phi_0(x_i) & \sum_{i=1}^{n}\phi_2(x_i)\phi_1(x_i) & \sum_{i=1}^{n}\phi_2(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^{n}\phi_2(x_i)\phi_n(x_i) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \sum_{i=1}^{n}\phi_n(x_i)\phi_0(x_i) & \sum_{i=1}^{n}\phi_n(x_i)\phi_1(x_i) & \sum_{i=1}^{n}\phi_n(x_i)\phi_2(x_i) & \cdots & \sum_{i=1}^{n}\phi_n(x_i)\phi_n(x_i) \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdots \\ a_n \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^{n}\phi_0(x_i)f(x_i) \\ \sum_{i=1}^{n}\phi_1(x_i)f(x_i) \\ \sum_{i=1}^{n}\phi_2(x_i)f(x_i) \\ \cdots \\ \sum_{i=1}^{n}\phi_n(x_i)f(x_i) \end{Bmatrix} \quad (10.1-11)$$

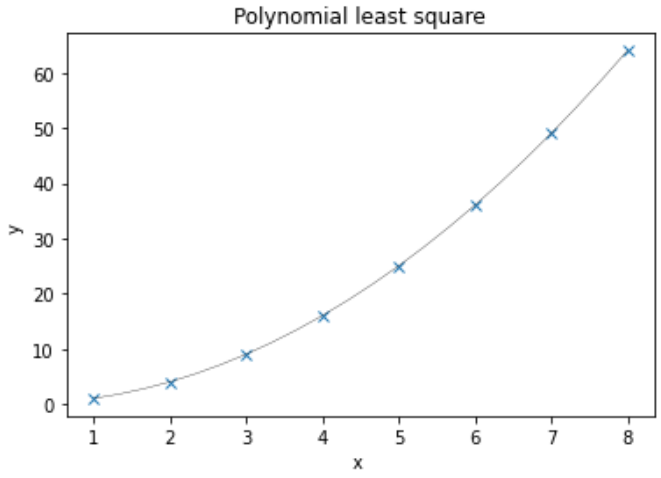
Program 6.1-4 General least square curve fitting (Python version)

```
 # -*- coding: utf-8 -*-
"""
Created on Sat Aug 25 09:31:06 2018

@author: Mustafa Turhan Çoban
"""
from math import *
from gauss import *;
import numpy as np;
import matplotlib.pyplot as plt;
from abc import ABC, abstractmethod


class f_xr(ABC):

    @abstractmethod
```

```
    def func(self,x,equation_ref):
        pass;

class f1(f_xr):
  def func(this,x,i):
      xx=0;
      if i==0: xx=1.0;
      elif i==1:xx=x;
      elif i==2:xx=x*x;
      elif i==3:xx=x*x*x;
      elif i==4:xx=x*x*x*x;
      elif i==5:xx=x*x*x*x*x;
      elif i==6:xx=x*x*x*x*x*x;
      return xx;

def PolynomialLSQ(f,xi,yi,n):
    #Polynomial least square
    m=len(xi);
    print("l=",m);
    np1=n+1;
    print("np1=",np1);
    A = [[0 for x in range(np1)] for y in range(np1)];
    B =[0 for x in range(np1)];
    X =[0 for x in range(np1)];
    for i in range(np1):
        for j in range(np1):
            for k in range(m):
                pp=pow(xi[k],(i+j));
                A[i][j]= A[i][j]+f.func(xi[k],i)*f.func(xi[k],j);
        for k in range(m):
            B[i]= B[i]+f.func(xi[k],i)*yi[k];
    X=gauss(A,B);
    return X;

def funcPolynomialLSQ(f,e,x):
    # this function calculates the value of
    # least square curve fitting function
    n=len(e);
    ff=0;
    if n!=0.0:
        for i in range(n-1,-1,-1):
            ff=ff+e[i]*f.func(x,i);
    return ff;

def listPolynomialLSQ(f,E,xi,aradegersayisi):
     #aradegersayisi:  x--o--o--x--o--o--x zincirinde x deneysel noktalar ise
                 # ara değer sayisi 2 dir
     n=len(xi);
     nn=(n-1)*(aradegersayisi+1)+1;
     z= [[0 for x in range(nn)] for y in range(2)];
     dx=0;
     k=0;
     for i in range(n-1):
        z[0][k]=xi[i];
        z[1][k]=funcPolynomialLSQ(f,E,z[0][k]);
        k=k+1;
        for j in range(aradegersayisi):
            dx=(xi[i+1]-xi[i])/(aradegersayisi+1.0);
            z[0][k]=z[0][k-1]+dx;
            z[1][k]=funcPolynomialLSQ(f,E,z[0][k]);
            k=k+1;
        z[0][k]=xi[i+1];z[1][k]=funcPolynomialLSQ(f,E,z[0][k]);
     return z;

def read_array2D(d):
    path = d
    file1 = open(path,'r')
    file1.seek(0,2)  #jumps to the end
    endLocation=file1.tell()
    file1.seek(0) #jumps to the start
    nn=0
    m=0
    while True:
        line=file1.readline()
        ee=line.split()
        nn+=1
```

```
        if  file1.tell()==endLocation:
            break
    m=len(ee)
    print("m=",m)
    a=[[0.0 for i in range(nn)] for j in range(m)]
    file1.seek(0) #jumps to the start
    for i in range(nn):
        line=file1.readline()
        ee=line.split()
        for j in range(m):
            a[j][i]=float(ee[j])
    file1.close()
    return a

a=read_array2D('a.txt')
print("a",a)
x=a[0]
y=a[1]
print("x",x)
print("y",y)
f=f1()
E=PolynomialLSQ(f,x,y,2);
print("E=",E)
Z=listPolynomialLSQ(f,E,x,2);
print("Z=",Z[0])
plt.title('Polynomial least square')
plt.xlabel('x ')
plt.ylabel('y ')
plt.plot(x,y,'x',Z[0],Z[1],'k',linewidth=0.3)
```

```
runfile('E:/okul/SCO1/gen_poly_least_sqr.py', wdir='E:/okul/SCO1')
Reloaded modules: gauss
m= 2
a [[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0], [1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0]]
x [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
y [1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0]
l= 8
np1= 3
E= [0.0, -0.0, 1.0]
Z= [1.0, 1.3333333333333333, 1.6666666666666665, 2.0, 2.3333333333333335, 2.666666666666667, 3.0, 3.3333333333333335,
3.666666666666667, 4.0, 4.333333333333333, 4.666666666666666, 5.0, 5.333333333333333, 5.666666666666666, 6.0,
6.333333333333333, 6.666666666666666, 7.0, 7.333333333333333, 7.666666666666666, 8.0]
```



An excel or Octave version of polynomial least square curve fitting is given below. In order to carry out curve fitting, data (x,y) and x^2,x^3,x^4,.. should be defined and LINEST function (DOT in turkish version) should be used with ctrl-shift-enter

=DOT(G2:G9;B2:F9;DOĞRU;DOĞRU)

=LINEST(G2:G9;B2:F9;TRUE;TRUE)

x	x^2	x^3	x^4	x^5	y	f(x)=a0+a1*x+..
1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1

2.0000	4.0000	4.0000	16.0000	32.0000	4.0000	4
3.0000	9.0000	9.0000	81.0000	243.0000	9.0000	9
4.0000	16.0000	16.0000	256.0000	1024.0000	16.0000	16
5.0000	25.0000	25.0000	625.0000	3125.0000	25.0000	25
6.0000	36.0000	36.0000	1296.0000	7776.0000	36.0000	36
7.0000	49.0000	49.0000	2401.0000	16807.0000	49.0000	49
8.0000	64.0000	64.0000	4096.0000	32768.0000	64.0000	64

a0	a1	a2	a3	a4	a5	turkish	english	
6.91E-19	-9.81031E-18	1	0	0	-	DOT	LINEST	ctrl-shift-enter

An excel or Octave version of polynomial least square curve fitting with our own gauss elimination linear system of equation solution is given below:

n	x	y	x^2	x^3	x^4	x*y	x^2*y
1	1	1.2452	1	1	1	1.2452	1.2452
1	2	4.34534	4	8	16	8.69068	17.38136
1	3	8.9746	9	27	81	26.9238	80.7714
1	4	15.2	16	64	256	60.8	243.2
1	5	22	25	125	625	110	550
1	6	34.8975	36	216	1296	209.385	1256.31
1	7	50.12378	49	343	2401	350.86646	2456.0652
1	8	63.61761	64	512	4096	508.94088	4071.527
1	9	81.74675	81	729	6561	735.72075	6621.4868
1	10	97	100	1000	10000	970	9700
10	55	379.15078	385	3025	25333	2982.5728	24997.987

GAUSS ELIMINATION 3 UNKNOWN

A B

	10	55	385	379.15078
	55	385	3025	2982.5728
	385	3025	25333	24997.987

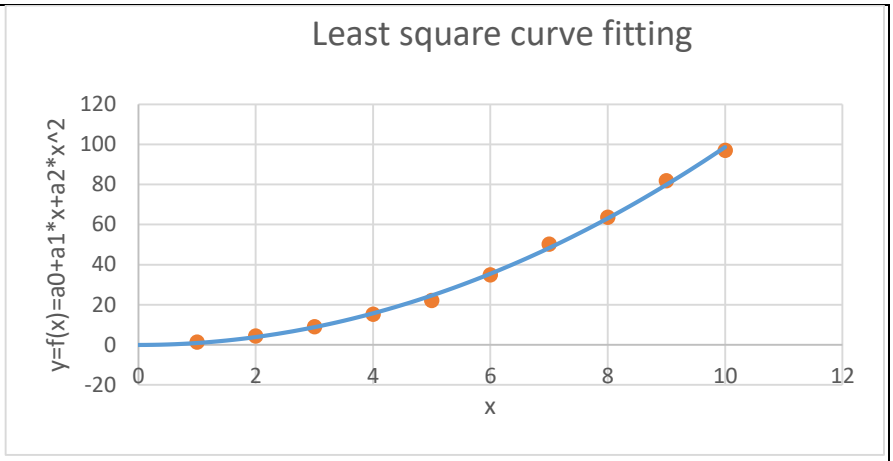
	10	55	385	379.15078
5.5	55	385	3025	2982.5728
38.5	385	3025	25333	24997.987

	10	55	385	379.15078
7.8571429	55	385	3025	2982.5728
	0	3025	25333	24997.987

x0	-0.0411622
x1	-0.0062948
x2	0.9867756

created data

x	y
0	-0.0411622
0.25	0.0189376
0.5	0.2023843
0.75	0.509178
1	0.9393186
1.25	1.4928062
1.5	2.1696408
1.75	2.9698223
2	3.8933507
2.25	4.9402261
2.5	6.1104485
2.75	7.4040178
3	8.820934
3.25	10.361197
3.5	12.024807
3.75	13.811765
4	15.722069
4.25	17.75572



10.2 LINEAR CURVE FITTING : ORTOGONAL POLYNOMIAL LEAST SQUARE

In the previous subsection , polynomial least square method is investigated. In order to find coefficient of polynomial least square a matrix solution process should be carried out. Furthermore, the equation tends to create ill matrices, so that chance of having an error in the coefficient while solving the matrix is high. Orthogonal polynomials has the following usefull property

$$\begin{Bmatrix} p_i(x)p_j(x) = 0 & if\ \ i \neq j \\ p_i(x)p_j(x) = A & if\ \ i = j \end{Bmatrix} \quad (10.2-1)$$

Now Assume to have a jth order orthogonal polynomial$p_j(x)$. And assume that least square polynomial equation is

$$y_m(x) = \sum_{j=0}^{m} b_j^{(m)} p_j(x) \quad (10.2-2)$$

If x_i, f_i i=1,…,n is given as a data to be curve fit, The minimization function can be defined as

$$H\left(b_0^{(m)},..,b_0^{(m)}\right) = \sum_{i=1}^{n} w(x_i)[f(x_i) - y_m(x)]^2 = \sum_{i=1}^{n} w(x_i)\left[f(x_i) - \sum_{j=0}^{m} b_j^{(m)} p_j(x)\right]^2 \quad (10.2-3)$$

In order to minimize the function, derivative can be set as equal to zero

$$\frac{\partial H\left(b_0^{(m)},..,b_0^{(m)}\right)}{\partial b_k^{(m)}} = \sum_{i=1}^{n} w(x_i)[f(x_i) - y_m(x)]^2$$

$$= \sum_{i=1}^{n} w(x_i)\left[f(x_i) - \sum_{j=0}^{m} b_j^{(m)} p_j(x)\right] p_k(x) = 0 \ \ k = 0,..,m \quad (9.2-4)$$

for a shorter notation new variables can be defined as:

$$d_{jk} \sum_{i=1}^{n} w(x_i)p_j(x_i)\,p_k(x_i) \quad (10.2-5)$$

$$\omega_{jk} = \sum_{i=1}^{n} w(x_i)p_n(x_i)f(x_i) \quad (10.2-8)$$

$$\sum_{j=0}^{m} d_{jk}\left[b_j^{(m)}\right] = \omega_{jk} \quad k = 0,..,m \quad (10.2-7)$$

This equation came to the following matrix form

$$\begin{bmatrix} \sum_{i=1}^{n} w(x_i)p_0(x_i)\,p_0(x_i) & \sum_{i=1}^{n} w(x_i)p_0(x_i)\,p_1(x_i) & \sum_{i=1}^{n} w(x_i)p_0(x_i)\,p_2(x_i) & \cdots & \sum_{i=1}^{n} w(x_i)p_0(x_i)\,p_n(x_i) \\ \sum_{i=1}^{n} w(x_i)p_1(x_i)\,p_0(x_i) & \sum_{i=1}^{n} w(x_i)p_1(x_i)\,p_1(x_i) & \sum_{i=1}^{n} w(x_i)p_1(x_i)\,p_2(x_i) & \cdots & \sum_{i=1}^{n} w(x_i)p_1(x_i)\,p_n(x_i) \\ \sum_{i=1}^{n} w(x_i)p_2(x_i)\,p_0(x_i) & \sum_{i=1}^{n} w(x_i)p_2(x_i)\,p_1(x_i) & \sum_{i=1}^{n} w(x_i)p_2(x_i)\,p_2(x_i) & \cdots & \sum_{i=1}^{n} w(x_i)p_2(x_i)\,p_n(x_i) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \sum_{i=1}^{n} w(x_i)p_n(x_i)\,p_0(x_i) & \sum_{i=1}^{n} w(x_i)p_n(x_i)\,p_1(x_i) & \sum_{i=1}^{n} w(x_i)p_n(x_i)\,p_2(x_i) & \cdots & \sum_{i=1}^{n} w(x_i)p_n(x_i)\,p_n(x_i) \end{bmatrix} \begin{Bmatrix} b_0 \\ b_1 \\ b_2 \\ \cdots \\ b_n \end{Bmatrix} =$$

$$\begin{Bmatrix} \sum_{i=1}^{n} w(x_i)p_0(x_i)f(x_i) \\ \sum_{i=1}^{n} w(x_i)p_1(x_i)f(x_i) \\ \sum_{i=1}^{n} w(x_i)p_2(x_i)f(x_i) \\ \cdots \\ \sum_{i=1}^{n} w(x_i)p_n(x_i)f(x_i) \end{Bmatrix} \quad (10.2\text{-}8a)$$

Writing the equation this way does not give any advantage, But considering that $p_k(x_i)$ is an orthogonal function, due to the propertiy of the orthogonal functions. In this case the equation that formed a matrix converted to a simple multiplication equation becomes

$$\begin{bmatrix} \sum_{i=1}^{n} w(x_i)p_0(x_i)\,p_0(x_i) & & & & \\ & \sum_{i=1}^{n} w(x_i)p_1(x_i)\,p_1(x_i) & & & \\ & & \sum_{i=1}^{n} w(x_i)p_2(x_i)\,p_2(x_i) & \cdots & \\ & & & & \sum_{i=1}^{n} w(x_i)p_n(x_i)\,p_n(x_i) \end{bmatrix} \begin{Bmatrix} b_0 \\ b_1 \\ b_2 \\ \cdots \\ b_n \end{Bmatrix} =$$

$$\begin{Bmatrix} \sum_{i=1}^{n} w(x_i)p_0(x_i)f(x_i) \\ \sum_{i=1}^{n} w(x_i)p_1(x_i)f(x_i) \\ \sum_{i=1}^{n} w(x_i)p_2(x_i)f(x_i) \\ \cdots \\ \sum_{i=1}^{n} w(x_i)p_n(x_i)f(x_i) \end{Bmatrix} \quad (10.2\text{-}8b)$$

$$d_{kk}\left[b_j^{(m)}\right] = \omega_{kk} \quad k = 0, \ldots, m \quad (10.2-8c)$$

To solve $b_j^{(m)}$

$$\left[b_j^{(m)}\right] = \frac{\omega_{kk}}{d_{kk}} = \frac{\sum_{i=1}^{n} w(x_i)p_k(x_i)f(x_i)}{\sum_{i=1}^{n} w(x_i)p_k(x_i)\,p_k(x_i)} \quad k = 0, .., m \quad (10.2-8d)$$

Anthony Ralston,Philip Rabinowitz orthogonal polynomial least square:

Several different orthogonal polynomials are available. The orthogonal polynomial that we will utilise in our first program has the following form. This equation is taken from "**A First Course in Numerical Analysis, Anthony Ralston,Philip Rabinowitz, Mc Graw Hill ISBN 0-07-051158-6" [4]**

$$p_{j+1}(x) = (x - \alpha_{j+1})p_j(x) - \beta_j p_{j-1}(x) \quad j = 0,1, \ldots \quad (10.2-9)$$

$p_0(x) = 1 \quad p_{-1}(x) = 0$

α_{j+1} and β_j are polynomial constants to be calculated.

$$\beta_k = \frac{\sum_{i=1}^{n} w(x_i)x_i p_{k-1}(x_i)\,p_k(x_i)}{\sum_{i=1}^{n} w(x_i)x_i p_{k-1}(x_i)\,p_{k-1}(x_i)} = \frac{\sum_{i=1}^{n} w(x_i)p_k(x_i)\,p_k(x_i)}{\sum_{i=1}^{n} w(x_i)p_{k-1}(x_i)\,p_{k-1}(x_i)} \quad 10.2-10)$$

$$\alpha_k = \frac{\sum_{i=1}^{n} w(x_i)x_i p_k(x_i)\,p_k(x_i)}{\sum_{i=1}^{n} w(x_i)p_{k-1}(x_i)\,p_{k-1}(x_i)} \quad (10.2-11)$$

Least square algorithm can be given as follows:

$$y_m(x) = \sum_{j=0}^{m} b_j^{(m)} p_j(x) \qquad (10.2.12)$$

$$b_j = \frac{\omega_j}{\gamma_j}$$

$$\omega_j = \sum_{i=1}^{n} w(x_i)p_j(x_i)f(x_i)$$

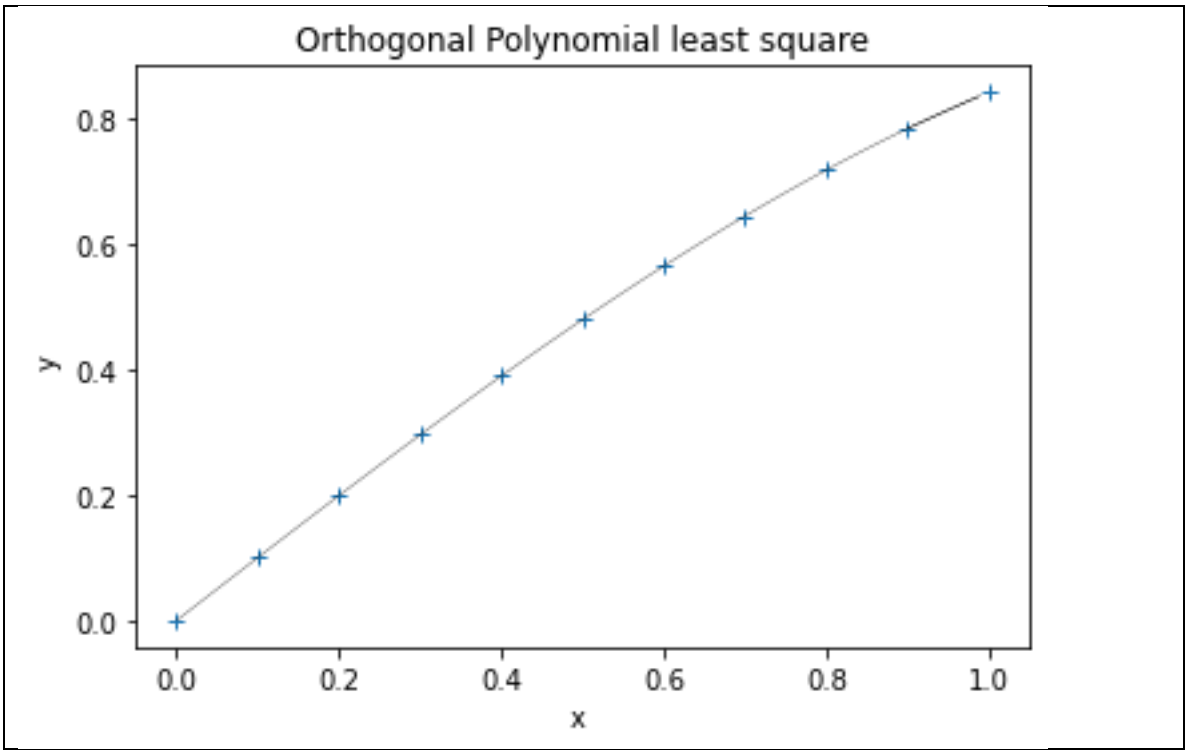
$$\gamma_j = \sum_{i=1}^{n} w(x_i)p_j(x_i)p_j(x_i)$$

Because Orthogonal polynomial equations are not solving any system of equations, roundoff errors due to matrix solutions will not be existed. But the polynomial equation used has a more complex form compare to polynomial form.

```
from math import *
import numpy as np;
import matplotlib.pyplot as plt

def  OPEKK(xi,fi,m):
    #ortogonal polinomial least square method
    #Referance : A First Course in Numerical Analysis
    #Anthony Ralston,Philip Rabinowitz, Mc Graw Hill ISBN 0-07-051158-6
    #m degree of polynomial xi fi input data
    # int i,j,k;
    n=len(xi)
    mp2=n+2
    mp1=n+1
    p=[[0.0 for j in range(n)] for i in range(mp2)]
    gamma=[0.0 for j in range(mp1)]
    beta=[0.0 for j in range(mp1)]
    omega=[0.0 for j in range(mp1)]
    alpha=[0.0 for j in range(mp1)]
    b=[0.0 for j in range(mp1)]
    wi=[0.0 for j in range(n)]
    a=[[0.0 for j in range(mp1)] for i in range(3)]
    for i in range(n):
        p[1][i]=1.0
        p[0][i]=0.0
        wi[i]=1.0
    gamma[0]=0
    for i in range(n):
        gamma[0]=gamma[0]+wi[i]
    beta[0]=0.0
    for j in range(m+1):
        omega[j]=0;
        for i in range(n):
            omega[j]=omega[j]+wi[i]*fi[i]*p[j+1][i]
        b[j]=omega[j]/gamma[j]
        if j != m:
            alpha[j+1]=0
            for i in range(n):
                alpha[j+1]+=wi[i]*xi[i]*p[j+1][i]*p[j+1][i]/gamma[j]
            for i in range(n):
                p[j+2][i]=(xi[i]-alpha[j+1])*p[j+1][i]-beta[j]*p[j][i]
            gamma[j+1]=0
            for i in range(n):
                gamma[j+1]+=wi[i]*p[j+2][i]*p[j+2][i]
            beta[j+1]=gamma[j+1]/gamma[j]
    for j in range(m+1):
       a[0][j]=b[j]
       a[1][j]=alpha[j]
       a[2][j]=beta[j]
    return a


def func(a,x):
    #polinom değerleri hesaplama fonksiyonu
    yy=0
    k=0
    m=len(a[0])-1
    mp2=m+2;
    q=[0.0 for j in range(mp2)]
    for k in range(m-1,-1,-1):
        q[k]=a[0][k]+(x-a[1][k+1])*q[k+1]-a[2][k+1]*q[k+2]
        yy=q[k]
    return yy;

def funcOPEKK(xi,yi,polynomialcoefficient,numberofmidpoints):
    #number of midpoints:  x--o--o--x--o--o--x  x data points midpoints are 2
    n=len(xi)
    nn=(n-1)*(numberofmidpoints+1)+1
    z=[[0.0 for j in range(nn)] for i in range(2)]
    E=OPEKK(xi,yi,polynomialcoefficient);
    dx=0.0
    k=0
    for i in range(n-1):
        z[0][k]=xi[i]
        z[1][k]=func(E,z[0][k])
```

```
        k=k+1
        for j in range(numberofmidpoints):
            dx=(xi[i+1]-xi[i])/(float(numberofmidpoints)+1.0)
            z[0][k]=z[0][k-1]+dx
            z[1][k]=func(E,z[0][k])
            k=k+1
        z[0][k]=xi[i]
        z[1][k]=func(E,z[0][k])
    return z

x=[0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
y=[0.0,0.099833,0.1986693,0.2955202,0.38941834,0.479425,0.5646424,0.6442176,0.717356,0.783326,0.84147]
z3=funcOPEKK(x,y,3,8);
print("z3",z3)
plt.title('Orthogonal Polynomial least square')
plt.xlabel('x ')
plt.ylabel('y ');
plt.plot(x,y,'+',z3[0],z3[1],'k',linewidth=0.3,)
```

```
runfile('E:/okul/SCO1/OPEKK.py', wdir='E:/okul/SCO1')
z3 [[0.0, 0.011111111111111112, 0.022222222222222223, 0.03333333333333333, 0.044444444444444446, 0.05555555555555556,
0.06666666666666667, 0.07777777777777778, 0.08888888888888889, 0.1, 0.11111111111111112, 0.12222222222222223,
0.13333333333333333, 0.14444444444444443, 0.15555555555555553, 0.16666666666666663, 0.17777777777777773,
0.18888888888888883, 0.2, 0.2111111111111111, 0.2222222222222222, 0.2333333333333333, 0.2444444444444444,
0.25555555555555554, 0.26666666666666666, 0.2777777777777778, 0.2888888888888889, 0.3, 0.3111111111111111,
0.32222222222222224, 0.33333333333333337, 0.3444444444444445, 0.3555555555555556, 0.36666666666666675,
0.3777777777777779, 0.3888888888888889, 0.4, 0.41111111111111115, 0.4222222222222223, 0.4333333333333334,
0.44444444444444453, 0.45555555555555566, 0.4666666666666668, 0.4777777777777779, 0.48888888888888904, 0.5,
0.5111111111111111, 0.5222222222222221, 0.5333333333333332, 0.5444444444444443, 0.5555555555555554,
0.5666666666666664, 0.5777777777777775, 0.5888888888888886, 0.6, 0.611111111111111, 0.6222222222222221,
0.6333333333333332, 0.6444444444444443, 0.6555555555555553, 0.6666666666666664, 0.6777777777777775,
0.6888888888888886, 0.7, 0.711111111111111, 0.7222222222222221, 0.7333333333333332, 0.7444444444444442,
0.7555555555555553, 0.7666666666666664, 0.7777777777777775, 0.7888888888888885, 0.8, 0.8111111111111111,
0.8222222222222222, 0.8333333333333333, 0.8444444444444443, 0.8555555555555554, 0.8666666666666665,
0.8777777777777775, 0.8888888888888886, 0.9, 0.9111111111111111, 0.9222222222222222, 0.9333333333333332,
0.9444444444444443, 0.9555555555555554, 0.9666666666666665, 0.9777777777777775, 0.9888888888888886, 0.9], [-
0.0001333131468531090 2, 0.011022273501971321, 0.02217203082505974, 0.03331477390831393, 0.04444931783763565,
0.05557447769892668, 0.06668906857808862, 0.0777919055610234, 0.0888818037336327, 0.09995757818181823,
0.1110180439914818, 0.1220620162485252, 0.1330883100388 5008, 0.14409574044835827, 0.15508312256295145,
0.16604927146853146, 0.17699300225099998, 0.1879131299962589, 0.19880846979020986, 0.20967783671875456,
0.22052004586779472, 0.23133391232323228, 0.24211825117096894, 0.25287187749690637, 0.2635936063869464,
0.2742822529269907, 0.2849366322029412, 0.2955555593006993, 0.3061378493061 6705, 0.31668231730524615,
0.32718777838383845, 0.3376530476278455, 0.3480769401231691, 0.35845827095571103, 0.3687958552113731,
0.379088507976057, 0.38933504433566435, 0.3995342793760972, 0.40968502818325714, 0.4197861058430459,
0.42983632744136524, 0.439834508064117, 0.4497794627972029, 0.4596700067265246, 0.46950495493798394,
0.4792831225174825, 0.4890033245509 2227, 0.49866437612420494, 0.5082650923232322, 0.5178042882339059,
0.5272807789421277, 0.5366933795337994, 0.5460409050948227, 0.5553221707110993, 0.5645359914685315,
0.5736811824530201, 0.5827565587504675, 0.5917609354467753, 0.6006931276278452, 0.6095519503795791,
0.6183362187878786, 0.6270447479386455, 0.6356763529177815, 0.6442298488111887, 0.6527040507047684,
0.6610977736844225, 0.6694098328360527, 0.6776390432455609, 0.6857842199988488, 0.693844178181818,
0.7018177328803704, 0.7097036991804079, 0.7175008921678322, 0.7252081269285447, 0.7328242185484475,
0.740347982113442, 0.7477782327094304, 0.7551137854223142, 0.7623534553379951, 0.7694960575423752,
0.7765404071213559, 0.7834853191608392, 0.7903296087467265, 0.7970720909649198, 0.8037115809013209,
0.8102468936418314, 0.8166768442723531, 0.8230002478787879, 0.8292159195470372, 0.8353226743630031,
0.7834853191608392]]
```



Chebyshev polynomial orthogonal polynomial least square curve fitting:
A second example to orthogonal polynomial curve fitting will be **Chebyshev polynomial** curve fitting. Chebyshev polynomials defined at (-1, 1) region.General definition of Chebyshev polynomials

$T_n(x) = \cos(n.cos^{-1}(x)$ (10.2-13)

If this equation is written in open form

$T_0(x)=1$

$T_1(x)=x$

$T_2(x)=2x^2-x$ (10.2-13a)

…..

$T_{n+1}(x)=2x\ T_n(x)- T_{n-1}(x)$

It is clear that it is a serial equation. The weight function for the Chebchev polynomials

$w(x) = \frac{1}{\sqrt{1-x^2}}$ (6.2-13)

The root of Chebchev polynomials

$t_k = \cos\left(\pi \frac{2k+1}{2n+2}\right)$ k=0,…,n (10.2-14)

n in this equation indicates the degree of polynomial. In order to curve fit data y=f(x) using this set as an orthogonal polynomial data should be given at the root points of chebchev polynomials. Furthermore due to regional limit of the data, to apply this equation to real data a conversion process is required. If the real data is in region (a,b) The convertion equations will be

$x_k = \left(\frac{b-a}{2}\right) t_k + \left(\frac{a+b}{2}\right)$ (10.2-15)

$t_k = \left(\frac{x_k-a}{b-1}\right) - 1$ (10.2-156

If x_k and corresponding y_k ,is given

Coefficient of polynomial $P_n(x) = \sum_{k=0}^{n} C_k T_k(x)$ can be calculated by using least sqaure method. Due to orthogonality of the polynomials, matrix solution is not required, coefficients is found directly.

$C_0 = \frac{1}{n+1} \sum_{k=0}^{n} y_k T_0\,(t_k)$

$C_j = \frac{2}{n+1} \sum_{k=0}^{n} y_k T_j\,(t_k)\ \ j = 1,2,\dots,n$ (10.2-17)

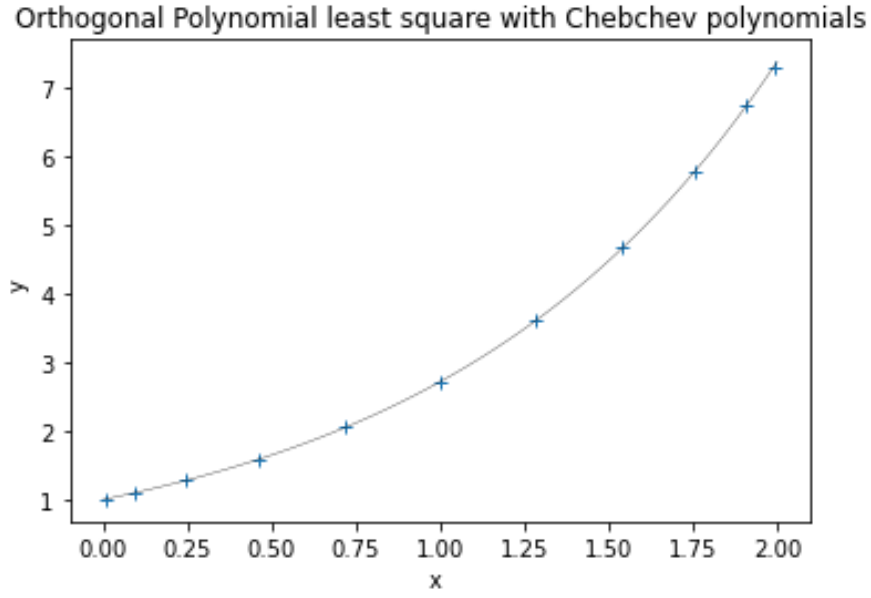
As an example problem $f(x)=e^x$ function in the limit 0 to 2 will be investigated. If Chebchev polynomials are to be used in curve fitting, experiments should be design so that the data should be taken exactly in the roots of the Chebchev polynomials.

x_k	y_k
0.010178558	1.010231
0.090368005	1.094577
0.244250426	1.276664
0.459359183	1.583059
0.718267443	2.050877
1	2.718282
1.281732557	3.602877
1.540640817	4.66758
1.755749574	5.787784
1.909631995	6.750604
1.989821442	7.314228

```
#Checychev Orthogonal Polynomial Curve Fitting  chebchev_OPEKK.py
from math import *
import numpy as np;
import matplotlib.pyplot as plt

def Ti(x,equation_ref):
    #Chebysev function
    T=[0.0 for j in range(equation_ref+1)]
    T[0]=1.0
    if equation_ref>=1:
        T[1]=x
    for i in range(2,equation_ref+1):
        T[i]=2.0*x*T[i-1]-T[i-2]
    return T[equation_ref]

def xi(a,b,N):
    nn=N+1;
    x=[0.0 for j in range(nn)]
    t=[0.0 for j in range(nn)]
    for k in range(nn):
        t[k]=cos((2.0*N+1-2.0*k)*pi/(2.0*N+2.0))
        x[k]=(b-a)/2.0*t[k]+(a+b)/2.0
    return x

def Chebyshev(a,b,y,N):
    #region a b
    nn=N+1;
    #int k,j;
    d=pi/(2.0*N+2)
    x=[0.0 for j in range(nn)]
    t=[0.0 for j in range(nn)]
    # -1<=x<=1 region convertion
    for k in range(nn):
        t[k]=cos((2.0*N+1-2.0*k)*pi/(2.0*N+2.0))
        x[k]=(b-a)/2.0*t[k]+(a+b)/2.0
    C=[0.0 for j in range(nn)]
    z=0.0
    for k in range(nn):
        z+=y[k]
    C[0]=z/(N+1)
    for j in range(1,N+1):
        z=0
        for k in range(N+1):
            z=z+y[k]*Ti(t[k],j)
        C[j]=2.0*z/(N+1)
    return C

def func(C,a,b,x):
    t=2.0*(x-a)/(b-a)-1;
    n=len(C)
    ff=0
    if n!=0.0:
        for i in range(n):
            ff=ff+C[i]*Ti(t,i)
    return ff
```

```
def funcChebyshev(a,b,y,polinomkatsayisi,verisayisi):
    n=verisayisi+1
    z=[[0.0 for j in range(n)] for i in range(2)]
    C=Chebyshev(a,b,y,polinomkatsayisi)
    dx=(b-a)/float(verisayisi)
    z[0][0]=a;
    for i in range(verisayisi+1):
        if i==0:
            z[0][i]=a
        else:
            z[0][i]=z[0][i-1]+dx
        z[1][i]=func(C,a,b,z[0][i])
    return z

a=0
b=2.0
y=[1.010230536,1.094577019,1.276664,1.583059208,2.05087687,2.718281828,
        3.60287651,4.66758038,5.787784491,6.750604088,7.314227631]
#x values are roots of chebchev polynomials
x= xi(a,b,10)
z=funcChebyshev(a,b,y,10,50)
print("Chebyschev orthogonal least square curve fitting\n",z)
plt.title('Orthogonal Polynomial least square with Chebchev polynomials')
plt.xlabel('x ')
plt.ylabel('y ');
plt.plot(x,y,'+',z[0],z[1],'k',linewidth=0.3,)
```

```
runfile('E:/okul/SCO1/chebychev_OPEKK.py', wdir='E:/okul/SCO1')
Chebyschev orthogonal least square curve fitting
 [[0, 0.04, 0.08, 0.12, 0.16, 0.2, 0.24000000000000002, 0.28, 0.32, 0.36, 0.39999999999999997, 0.43999999999999995,
0.4799999999999999, 0.5199999999999999, 0.5599999999999999, 0.6, 0.64, 0.68, 0.7200000000000001, 0.7600000000000001,
0.8000000000000002, 0.8400000000000002, 0.8800000000000002, 0.9200000000000003, 0.9600000000000003,
1.0000000000000002, 1.0400000000000003, 1.0800000000000003, 1.1200000000000003, 1.1600000000000004,
1.2000000000000004, 1.2400000000000004, 1.2800000000000005, 1.3200000000000005, 1.3600000000000005,
1.4000000000000006, 1.4400000000000006, 1.4800000000000006, 1.5200000000000007, 1.5600000000000007,
1.6000000000000008, 1.6400000000000008, 1.6800000000000008, 1.7200000000000009, 1.760000000000001,
1.800000000000001, 1.840000000000001, 1.880000000000001, 1.920000000000001, 1.960000000000001, 2.000000000000001],
[1.0000000003363947, 1.0408107741052222, 1.0832870676174815, 1.1274968516551822, 1.1735108711582891,
1.2214027583411369, 1.2712491504601175, 1.323129812413029, 1.3771277643597053, 1.4333294145629911,
1.4918246976585994, 1.5527072185721034, 1.6160744023111264, 1.68202764987093, 1.7506725005018824, 1.822118800597927,
1.8964808794760928, 1.973877732328336, 2.054433210638661, 2.138276220370504, 2.2255409282418266, 2.316366976418341,
2.4108997059686983, 2.509290389439452, 2.6116964729221515, 2.718281828000004, 2.8292170139773067,
2.9446795508112373, 3.064854203182635, 3.1899332761602364, 3.3201169229313003, 3.455613465090903, 3.596639726002278,
3.7434213777615013, 3.8961933023216746, 4.055199967354418, 4.2206958174501725, 4.392945681283381,
4.5722251953942035, 4.758821245265077, 4.9530324243980175, 5.155169512127402, 5.365555970932761, 5.584528464047143,
5.812437394188804, 6.049647464277367, 6.296538261030195, 6.553504862370688, 6.820958469617304, 7.099327065460759,
7.389056098776642]]
```



10.3 LINEAR CURVE FITTING : LEAST SQUARE METHOD WITH MULTI VARIABLES

Curve fitting of a multivariable function basically the same of general curve fitting of the single variable function. The only basic difference is definition of the function to be used. Assume that

$$f(x_0, x_{1,}.., x_n) = \sum_{j=0}^{m} a_j^{(m)} \phi_j((x_0, x_{1,}.., x_n) \quad (10.3-1)$$

j th degree function is given. It is desired to fit x_0, $x_{1,}$ x_2,…, x_n ,$f_{i,}$ i=0...n data set into this function. The best fitted $a_j^{(m)}$ values to be found. For this purpose Minimum value of

$$H(x_0, x_{1,}.., x_n) = \sum_{i=1}^{n} w(x_0, x_{1,}.., x_n)\left[f_i - \sum_{j=0}^{m} a_j^{(m)} \phi_j(x_0, x_{1,}.., x_n)\right]^2 \quad (10.3-2)$$

function should be found. $w(x_0, x_{1,}.., x_n)$ is called weight function and it should satisfy the condition $w(x_0, x_{1,}.., x_n) \geq 0 \quad i = 1, \dots, n.$ The minimum of the function is the root of the derivative of the function.

$$\frac{\partial H(x_0, x_{1,}.., x_n)}{\partial a_k^{(m)}} = 2\sum_{i=1}^{n} w(x_0, x_{1,}.., x_n)\left[f_i - \sum_{j=0}^{m} a_j^{(m)} \phi_j(x_0, x_{1,}.., x_n)\right] \phi_k(x_0, x_{1,}.., x_n) = 0 \quad k = 0,..,m \quad (10.3-3)$$

If unit value is taken as weight function, equation becomes:

$$2\sum_{i=1}^{n}\left[f_i - \sum_{j=0}^{m} a_j^{(m)} \phi_j(x_0, x_{1,}.., x_n)\right] \phi_k(x_0, x_{1,}.., x_n) = 0 \qquad k = 0,..,m \quad (10.3-3a)$$

As an example a surface polinomial can be defined as
$f(x,y) = a_0 + a_1x + a_2y + a_3x^2 + a_4y^2 + a_5xy + a_6x^3 + a_7y^3 + a_8x^2y + a_9xy^2 + a_{10}x^4 + a_{11}y^4 + a_{12}x^2y^2 + a_{13}x^3y + a_{14}xy^2$
This polynomial is defined in the example problem

```
from abc import ABC, abstractmethod
from math import *

class f_xir(ABC):

   @abstractmethod
   def func(self,x,equation_ref):
      pass;
```

Data file a.txt

```
0    0    0
0    1    1
0    2    4
0    3    9
0    4    16
1    0    1
1    1    2
1    2    5
1    3    10
1    4    17
2    0    4
2    1    5
2    2    8
2    3    13
2    4    20
3    0    9
3    1    10
3    2    13
3    3    18
3    4    25
4    0    16
4    1    17
4    2    20
4    3    25
4    4    32
```

```
from f_xir import *
from gauss import *
class f1(f_xir):
#f(x,y)=a0+a1*x+a2*y+a3*x^2+a4*y^2+a5*xy+a6*x^3+a7*y^3+a8*x^2y
#      +a9*xy^2+a10*x^4+a11*y^4+a12*x^2y^2+a13x^3y+a14*xy^3
   def func(self,x,i):
      xx=0.0
      if i==0: xx=1.0
      elif i==1:   xx=x[0]
      elif i==2:   xx=x[1]
      elif i==3:   xx=x[0]*x[0]
      elif i==4:   xx=x[1]*x[1]
      elif i==5:   xx=x[0]*x[1]
      elif i==6:   xx=x[0]*x[0]*x[0]
      elif i==6:   xx=x[0]*x[0]*x[0]
      elif i==7:   xx=x[1]*x[1]*x[1]
      elif i==8:   xx=x[0]*x[0]*x[1]
      elif i==9:   xx=x[0]*x[1]*x[1]
      elif i==10:   xx=x[0]*x[0]*x[0]*x[0]
      elif i==11:   xx=x[1]*x[1]*x[1]*x[1]
      elif i==12:   xx=x[0]*x[0]*x[1]*x[1]
      elif i==13:   xx=x[0]*x[0]*x[0]*x[1]
      elif i==14:   xx=x[0]*x[1]*x[1]*x[1]
      return xx

# General least square curve fitting with multivariable
def Transpose(left):
   # transpose matrix (if A=a(i,j) Transpose(A)=a(j,i)
   # int i,j;
   n=len(left)
   m=len(left[0])
   b=[[0 for j in range(n)] for i in range(m)]
   for i in range(n):
      for j in range(m):
         b[j][i]=left[i][j]
   return b

def GeneralLeastSquare(c,n):
   n1=len(c)
   n2=len(c[0])-1
   xi=[[0 for j in range(n2)] for i in range(n1)]
   yi=[0 for j in range(n1)]
   for i in range(n1):
      for j in range(n2):
         xi[i][j]=c[i][j]
      yi[i]=c[i][n2]
   return GLS(xi,yi,n)

def GLS(xi,yi,n):
# n dimensional surface general least square curve fitting
   f=f1()
   l=len(xi)
   np1=n+1
   A=[[0 for j in range(np1)] for i in range(np1)]
   B=[0 for j in range(np1)]
   X=[0 for j in range(np1)]
   for i in range(n+1):
      B[i]=0
      for j in range(np1):
         A[i][j]=0.0
         for k in range(l):
            A[i][j]=A[i][j]+f.func(xi[k],i)*f.func(xi[k],j)
      for k in range(l):
            B[i]=B[i]+f.func(xi[k],i)*yi[k]
   X=gauss(A,B)
   max=0
   for i in range(n+1):
      if abs(X[i]) > max: max=abs(X[i])
   for i in range(n+1):
      if abs(X[i])/max > 0 and abs(X[i]/max) < 1.0e-100: X[i]=0.0
   #print(X)
   return X
```

```
def func(e,x):
    #multidimensional function calculation
    f=f1()
    n=len(e)
    ff=0.0
    if n!=0:
        for i in range(n-1,-1,-1):
            ff=ff+e[i]*f.func(x,i)
    return ff

def funcGeneralLeastSquare(e,xi):
    # multidimensional function calcu0000000000 nnnmlation
    f=f1()
    n=len(e)
    m=len(xi)
    ff=[0 for j in range(m)]
    for k in range(m):
      if n!=0:
          for i in range(n-1,-1,-1):
           ff[k]=ff[k]+e[i]*f.func(xi[k],i)
    return ff

def read(name):
    path = name
    file1 = open(path,'r')
    file1.seek(0,2)  #jumps to the end
    endLocation=file1.tell()
    file1.seek(0) #jumps to the start
    nn=0
    while True:
        line=file1.readline()
        nn+=1
        if  file1.tell()==endLocation:
            break
    x=[0.0 for i in range(nn)]
    y=[0.0 for i in range(nn)]
    z=[0.0 for i in range(nn)]
    file1.seek(0) #jumps to the start
    for i in range(nn):
        line=file1.readline()
        ee=line.split()
        x[i]=float(ee[0])
        y[i]=float(ee[1])
        z[i]=float(ee[2])
        #print(x[i]," ",y[i]," ",z[i])
    file1.close()
    a=[x,y,z]
    return Transpose(a)
def output1(c,polinomkatsayisi,aradegersayisi):
    n1=len(c)
    n2=len(c[0])-1
    xi=[[0 for j in range(n2)] for i in range(n1)]
    yi=[0 for j in range(n1)]
    for i in range(n1):
        for j in range(n2):
            xi[i][j]=c[i][j]
        yi[i]=c[i][n2]
    return output2(xi,yi,polinomkatsayisi,aradegersayisi);

def output2(xi,yi,polinomkatsayisi,aradegersayisi):
    n=len(xi)
    nk=len(xi[0])
    nn=(n-1)*(aradegersayisi+1)+1
    E=GLS(xi,yi,polinomkatsayisi)
    x=[[0 for j in range(nk)] for i in range(nn)]
    c=[[0 for j in range(nk+1)] for i in range(nn)]
    yy=[0.0 for i in range(nn)]
    dx=[0.0 for i in range(nk)]
    k=0
    # int i,j,w;
    for i in range(n-1):
        for w in range(nk):
            x[k][w]=xi[i][w]
            dx[w]=(xi[i+1][w]-xi[i][w])/(float(aradegersayisi)+1.0)
        k=k+1
        for j in range(aradegersayisi):
```

```
            for w in range(nk):
                x[k][w]=x[k-1][w]+dx[w]
            k=k+1
    for w in range(nk):
        x[k][w]=xi[i][w]
    yy=funcGeneralLeastSquare(E,x)
    for i in range(len(x)):
        for w in range(nk):
            c[i][w]=x[i][w]
        c[i][nk]=yy[i]
    return c

a=read('a.txt')
e=GeneralLeastSquare(a,9)
#print(e)
y=output1(a,9,1)
print(y)
```

```
runfile('E:/okul/SCO1/multidimensional_LSCF.py', wdir='E:/okul/SCO1')
Reloaded modules: f_xir, gauss
[[0.0, 0.0, -2.1827872842550278e-14], [0.0, 0.5, 0.24999999999998757], [0.0, 1.0, 0.9999999999999939], [0.0, 1.5,
2.249999999999998], [0.0, 2.0, 4.0], [0.0, 2.5, 6.25], [0.0, 3.0, 9.0], [0.0, 3.5, 12.249999999999998], [0.0, 4.0, 15.999999999999996],
[0.5, 2.0, 4.249999999999998], [1.0, 0.0, 0.9999999999999859], [1.0, 0.5, 1.249999999999992], [1.0, 1.0, 1.999999999999996],
[1.0, 1.5, 3.249999999999998], [1.0, 2.0, 4.999999999999998], [1.0, 2.5, 7.249999999999997], [1.0, 3.0, 9.999999999999998], [1.0,
3.5, 13.249999999999996], [1.0, 4.0, 16.999999999999996], [1.5, 2.0, 6.249999999999999], [2.0, 0.0, 3.999999999999995], [2.0,
0.5, 4.249999999999998], [2.0, 1.0, 5.0], [2.0, 1.5, 6.25], [2.0, 2.0, 8.000000000000002], [2.0, 2.5, 10.25], [2.0, 3.0,
12.999999999999998], [2.0, 3.5, 16.25], [2.0, 4.0, 20.0], [2.5, 2.0, 10.250000000000002], [3.0, 0.0, 9.000000000000002], [3.0, 0.5,
9.250000000000004], [3.0, 1.0, 10.000000000000004], [3.0, 1.5, 11.250000000000004], [3.0, 2.0, 13.000000000000002], [3.0, 2.5,
15.25], [3.0, 3.0, 18.0], [3.0, 3.5, 21.25], [3.0, 4.0, 25.000000000000004], [3.5, 2.0, 16.249999999999996], [4.0, 0.0, 16.0], [4.0, 0.5,
16.25], [4.0, 1.0, 16.999999999999996], [4.0, 1.5, 18.249999999999996], [4.0, 2.0, 19.999999999999993], [4.0, 2.5,
22.249999999999993], [4.0, 3.0, 24.999999999999996], [4.0, 3.5, 28.25], [4.0, 3.0, 24.999999999999996]]
```

10.4 LEAST SQUARE CURVE FITTING OF NONLINEAR EQUATIONS

If curve fitting to a function $y = f(a; x)$ with non-linear coefficients is required. Process will be a little bit different compare to equations with linear coefficients.

$y = f(a; x)$ is given and data x_i,f_i, i=0...n will be fit into the equation. In this equation coefficients are placed in a non-linear form. For example : $y = a_0(1 - e^{-a_1 x})$

Least square error formula was defined as

$$H\left(a_j^{(m)}; x_0, x_{1,}.., x_n\right) = \sum_{i=1}^{n} w\left(x_0, x_{1,}.., x_n\right)\left[y_i - f\left(a_j^{(m)}; x_0, x_{1,}.., x_n\right)\right]^2 \quad (10.4-1)$$

This equation should be solved by using one of the multidimensional optimization methods such as Nelder-Mead, Newton-Raphson, Steepest descent, Generic algorithms

sample : (input3.txt)

```
5.0 16.0
10.0 25.0
15.0 32.0
20.0 33.0
25.0 38.0
30.0 36.0
```

A nonlinear function to fit the data:

pi=ai[0]*(1.0-exp(-ai[1]*x));

In this function ai are coefficients of the equations and x is independent variable. In order to solve non-linear system of equations Nelder-Mead optimisation method is used.

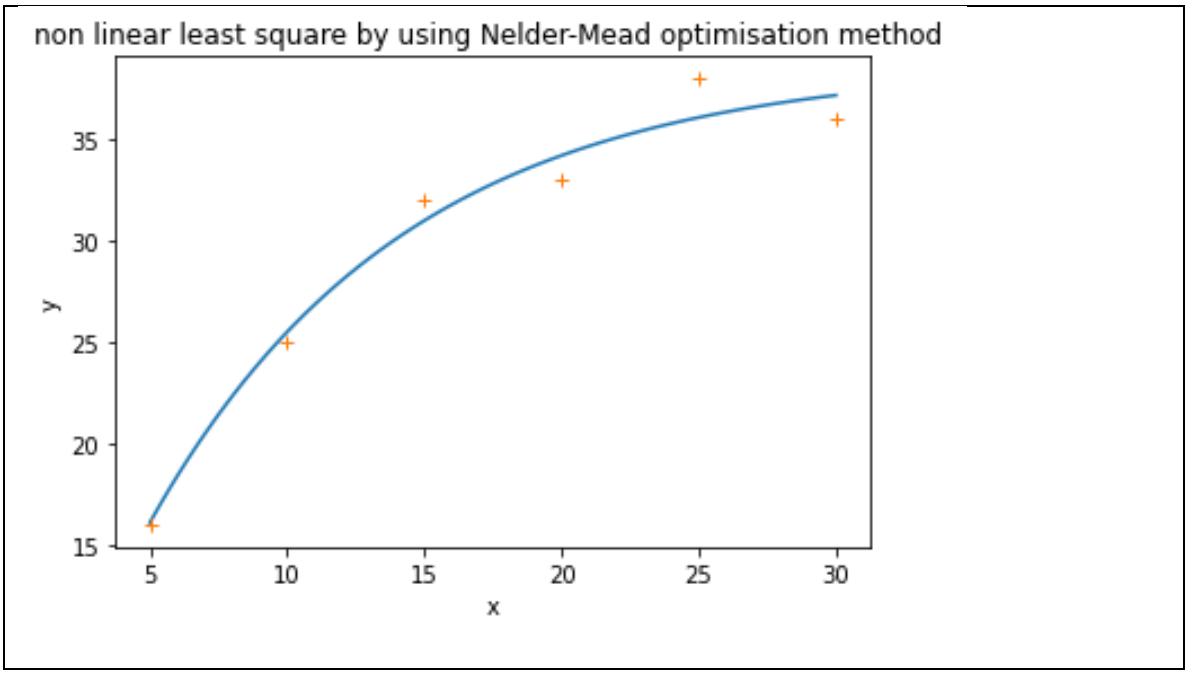
```
from math import *
from f_xj import *
from matplotlib import pyplot as plt
import numpy as np


class y(f_xj):
    def __init__(self,filename,ia):
        self.name = 'yy',
        self.filename=filename
        path = filename
        fin=open(path,'r')
        n=len(ia)
        self.a=ia
```

```
        path = self.filename
        file1 = open(path,'r')
        file1.seek(0,2)  #jumps to the end
        endLocation=file1.tell()
        file1.seek(0) #jumps to the start
        nn=0
        while True:
           line=file1.readline()
           nn+=1
           if  file1.tell()==endLocation:
              break
        xi=[0.0 for i in range(nn)]
        yi=[0.0 for i in range(nn)]
        file1.seek(0) #jumps to the start
        for i in range(nn):
           line=file1.readline()
           ee=line.split()
           xi[i]=float(ee[0])
           yi[i]=float(ee[1])
           #print(xi[i]," ",yi[i])
        file1.close()
        self.xi=xi
        self.yi=yi
        self.nn=nn

     def Ps(self,x,ai):
        pi=ai[0]*(1-exp(-ai[1]*x))
        return pi
     def func(self,ai):
        ff=0
        for i in range(self.nn):
           w=self.Ps(self.xi[i],ai)
           yy=self.yi[i]
           w=w-yy
           ff=ff+w*w
        return ff

def  nelder(fnelder,a,da,maxiteration,tolerance,printlist):
    #print("a",a)
    #print("da",da)
    n=len(a)
    m=n+1
    x=[[0.0 for j in range(n)] for i in range(n+1)]
    p=[[0.0 for j in range(m)] for i in range(m)]
    s=""
    NDIMS = len(x)-1
    NPTS = len(x)
    FUNC = NDIMS
    ncalls = 0;
    for i in range(m):
       for j in range(n):
          if i==j:
             x[i][j]=a[j]+da[j]
             p[i][j]=x[i][j]
          else:
             x[i][j]=a[j]
             p[i][j]=x[i][j]
       p[i][FUNC] = fnelder.func(p[i])
       #print("x[",i,"]=",x[i])
       #print("p[",i,"]=",p[i])
    #print("p",p)
    #Inlet variable definitions
    #fnelder : abstract multivariable function f(x)
    #x : independent variable set of n+1 simplex elements
    #maxiteration : maximum iteration number
    #tolerance :

    ##### construct the starting simplex ///////////////////
    z=[0.0 for i in range(NDIMS)]
    best = 1.0E99;
    ################ calculate the first function values for the simplex ////////////////
    iter=0;
    for iter in range(1,maxiteration):
       ########  define lo, nhi, hi (low high next_to_high ///////////////
       ilo=0
       ihi=0
```

```
    inhi = -1
    # -1 means missing
    flo = p[0][FUNC];
    fhi = flo
    #print("flo=",flo,"fhi=",fhi)
    #double pavg,sterr;
    for i in range(1,NPTS):
       if p[i][FUNC] < flo:
          flo=p[i][FUNC]
          ilo=i
       if  p[i][FUNC] > fhi:
          fhi=p[i][FUNC]
          ihi=i
       #print("i=",i,"flo=",flo,"fhi=",fhi,"ilo=",ilo,"ihi=",ihi,"pifunc=",p[i][FUNC])
    fnhi = flo
    inhi = ilo
    #print("fnhi=",fnhi,"inhi=",inhi)
    for i in range(NPTS):
       if (i != ihi) and (p[i][FUNC] > fnhi):
          fnhi=p[i][FUNC]
          inhi=i
    #////////// exit criteria ////////////////
    if (iter % 4*NDIMS) == 0:
       #calculate the avarage (including maximum value)
       pavg=0
       for i in range(NPTS):
           pavg=pavg+p[i][FUNC]
       pavg=pavg/NPTS
       tot=0.0
       if printlist!=0:
          print(iter)
          for j in range(NDIMS):
             print(p[ilo][j]," ")
       for i in range(NPTS):
          tot=(p[i][FUNC]-pavg)*(p[i][FUNC]-pavg)
       sterr=sqrt(tot/NPTS)
       for j in range(NDIMS):
          z[j]=p[ilo][j]
       best = p[ilo][FUNC];
    # end of if iter%4*NDMS)==0
    #//// calculate avarage without maximum value //////
    ave=[0.0 for i in range(NDIMS)]
    for j in range(NDIMS):
       ave[j] = 0
    for i in range(NPTS):
       if i != ihi:
          for j in range(NDIMS):
             ave[j] = ave[j]+p[i][j];
    for j in range(NDIMS):
       ave[j]=ave[j]/(NPTS-1)
    #print("ave[",j,"]=",ave[j])
    #//////// reflect ////////////////
    #print("reflect")
    r=[0.0 for i in range(NDIMS)]
    for j in range(NDIMS):
       r[j] = 2.0*ave[j] - p[ihi][j]
    fr = fnelder.func(r)
    if (flo <= fr) and (fr < fnhi): #in zone: accept
       for j in range(NDIMS):
          p[ihi][j] = r[j]
       p[ihi][FUNC] = fr
       continue
    #///(////////////// expand
    if fr < flo:
       #print("expand")
       e=[0.0 for i in range(NDIMS)]
       for j in range(NDIMS):
          e[j] = 3.0*ave[j] - 2.0*p[ihi][j];
       fe = fnelder.func(e);
       if fe < fr:
          for j in range(NDIMS):
             p[ihi][j] = e[j]
          p[ihi][FUNC] = fe
          continue
       else:
          for j in range(NDIMS):
```

```
                p[ihi][j] = r[j]
            p[ihi][FUNC] = fr
            continue
        #////////// shrink:
        if fr < fhi:
            #print("srink")
            c=[0.0 for i in range(NDIMS)]
            for j in range(NDIMS):
                c[j] = 1.5*ave[j] - 0.5*p[ihi][j]
            fc = fnelder.func(c);
            if fc <= fr:
                for j in range(NDIMS):
                    p[ihi][j] = c[j];
                p[ihi][FUNC] = fc
                continue
            else:   #////// shrink
                for i in range(NPTS):
                    if i != ilo:
                        for j in range(NDIMS):
                            p[i][j] = 0.5*p[ilo][j] + 0.5*p[i][j]
                        p[i][FUNC] = fnelder.func(p[i])
                continue
        #/////////////// contract
        if fr >= fhi:
            #print("contract")
            cc=[0.0 for i in range(NDIMS)]
            for j in range(NDIMS):
                cc[j] = 0.5*ave[j] + 0.5*p[ihi][j];
            fcc = fnelder.func(cc)
            if fcc < fhi:
                for j in range(NDIMS):
                    p[ihi][j] = cc[j];
                p[ihi][FUNC] = fcc
                continue
            else:
                for i in range(NPTS):
                    if i != ilo:
                        for j in range(NDIMS):
                            p[i][j] = 0.5*p[ilo][j] + 0.5*p[i][j]
                        p[i][FUNC] = fnelder.func(p[i]);
    return z

a=[30.0,1.0]
da=[1.0,0.2]
name="input3.txt"
f=y(name,a)
ai=nelder(f,a,da,100,1.0e-15,0)
print("ai = ",ai)
n1=100
xx1 =[0 for w in range(n1)]
yy1 =[0 for w in range(n1)]
dx=(30.0-5.0)/(n1-1)
for i in range(n1):
    xx1[i]=5.0+dx*i
    yy1[i]=f.Ps(xx1[i],ai)
plt.plot(xx1,yy1,f.xi,f.yi,"+")
plt.xlabel("x")
plt.ylabel("y")
plt.title("non linear least square by using Nelder-Mead optimisation method ")
plt.show()
```

```
runfile('E:/okul/SCO1/nelder_opt.py', wdir='E:/okul/SCO1')
Reloaded modules: f_xj
ai =  [38.727045551319954, 0.1074564960913112]
```


non linear least square by using Nelder-Mead optimisation method
35
30
25
20
15
y
5
10
15
20
25
30
x