

Data Representation

The CalcLib software uses generic data types to represent values manipulated in calculations so that the algorithm can be expressed without being specific to the representation of those values. Algorithms expressed on these generic types use interfaces based on the core Java interface within CalcLib:

```
package net.myorb.data.abstractions;

public interface SpaceDescription<T>
{
    /**
     * compute internal representation for scalar
     * @param x simple integer value to convert to
     * @return internal representation for value
     */
    T newScalar (int x);

    /**
     * determine value of opposite sign
     * @param x value to be negated
     * @return negated value
     */
    T negate (T x);

    /**
     * is the value zero
     * @param x value to be checked
     * @return TRUE for zero value
     */
    boolean isZero (T x);

    /**
     * is the value negative
     * @param x value to be checked
     * @return TRUE for negative value
     */
    boolean isNegative (T x);

    /**
     * compute sum of pair of terms
     * @param x the value of the left side of computation
     * @param y the value of the right side of computation
     * @return result of computation
     */
    T add (T x, T y);

    /**
     * compute product x*y
     * @param x the value of the left side of computation
     * @param y the value of the right side of computation
     * @return result of computation
     */
    T multiply (T x, T y);

    /**
     * compute division 1/x
     * @param x the value of the divisor of the computation
     * @return result of computation
     */
    T invert (T x);

    /**
     * get the conjugate of the specified value
     * @param x the value to be used to compute conjugate
     * @return the conjugate of the parameter
     */
    T conjugate (T x);

    /**
     * compute comparison x < y
     * @param x the value of the left side of computation
     * @param y the value of the right side of computation
     * @return result of computation
     */
    boolean lessThan (T x, T y);

    /**
     * get the value zero in this representation (additive identity)
     * @return the value zero
     */
    T getZero ();

    /**
     * get the value one in this representation (multiplicative identity)
     * @return the value zero
     */
    T getOne ();
}
```

This collection of methods defines a mathematical field operating on the data representation specified by the generic parameter <T>

Implemented Types

The following representations are implemented in the core of the CalcLib implementation:

Default

The most primitive form represents values as real numbers stored as Java Double objects supplying a default of 16 digit precision, common 64 bit floating point

Complex

Java objects that hold real and imaginary parts as separate Java Double values

Fraction

Java objects that hold numerator and denominator values as separate BigInteger values supplying arbitrary precision

Factorization

Java objects that hold prime numbers described as BigInteger each with an integer exponent which together represent an integer ratio with numerator and denominator held as prime factors

Generic Algorithms

Algorithms implemented using the generic `<T>` type and using the `SpaceDescription` interface have the advantage of fully working with any data type provided in the CalcLib build. The only requirement is that all arithmetic done in the implementation is provided by reference to the interface methods.

For Example:

```
/*
 * compute x^n for larger exponents
 * @param x base of the exponentiation
 * @param n exponent for the computation
 * @return computed result
 */
public T pow (T x, int n)
{
    if (n < 0) return inverted (pow (x, -n));
    else if (n == 0) return discrete (1);
    else if (n == 1) return x;

    Value<T> v = forValue (x);
    Value<T> square = v.squared ();
    Value<T> result = square;

    while ((n -= 2) >= 2)
    {
        result = result.times (square);
    }

    if (n == 1)
        return result.times (v).getUnderlying ();
    else return result.getUnderlying ();
}
public Value<T> pow (Value<T> x, int exponent)
{
    return forValue (pow (x.getUnderlying (), exponent));
}
```

Algorithm implementations can extend the class `net.myorb.math.Arithmetic` which constructs with an implementer of `SpaceDescription` providing simple and more readable coding of arithmetic operations in the implementation.

The obvious great advantage of this implementation style is that polynomials can be expressed with only the most simple arithmetic operations. With Taylor expansions most trig and power functions can be made available to any field described with a `SpaceDescription` implementation.

Consider Newton's root method, using fractions or factors one can get a SQRT approximation expressed as an integer ratio and can go to arbitrary precision.

FloatDemoAiry.txt

READ JavaMath.txt

INFINITY = 100

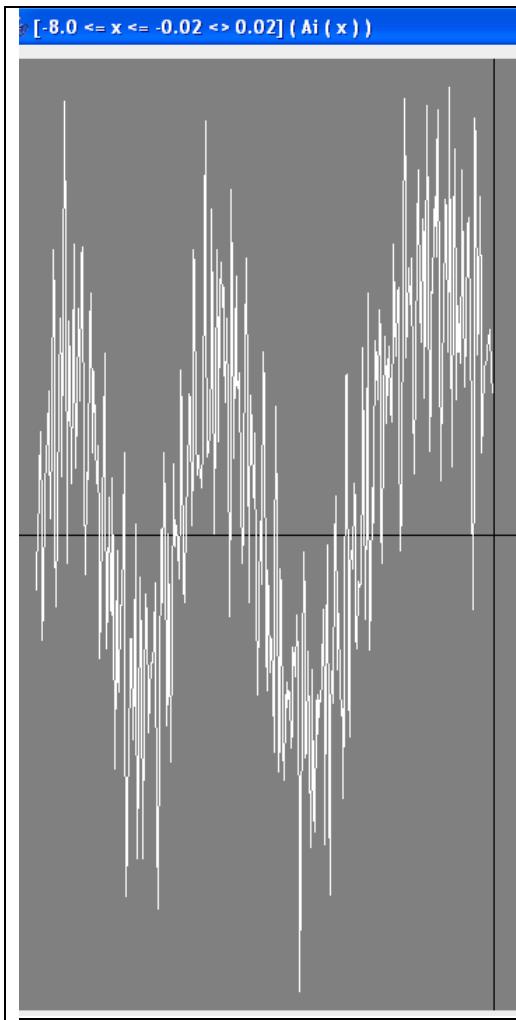
dt = 1 / INFINITY

// airy

!! $Ai(x) = \frac{1}{\pi} \int [0 \leq t \leq \infty] (\cos(\text{POW}(t, 3)/3 + x*t) * dt)$

!! $Bi(x) = \frac{1}{\pi} \int [0 \leq t \leq \infty] ((\exp(-\text{POW}(t, 3)/3 + x*t) + \sin(\text{POW}(t, 3)/3 + x*t)) * dt)$

graph [-8 <= x < 0 <> 0.02] (Ai(x))



This is a good demonstration of the difficulty of finding means of computing functions that have extremely long convergence requirements. The combination of the infinite integral and ‘dt’ having the implication of ‘as small as possible’ along with terms that have no diminishing factor all contrive to disallow any ‘brute force’ attempt to achieve a useful approximation of the value of the series.

A later test shows the attempt to evaluate Airy in the complex space using Bessel functions. While that result is more successful at showing a smooth curve, the further the evaluation is attempted from the 0 of the x-axis, the more error is seen to complicate the approximation

Functions

$$Ai(x) = \frac{1}{\pi} \int [0 \leq t \leq \infty] (\cos(\text{POW}(t, 3)/3 + x*t) * dt)$$

$$Bi(x) = \frac{1}{\pi} \int [0 \leq t \leq \infty] ((\exp(-\text{POW}(t, 3)/3 + x*t) + \sin(\text{POW}(t, 3)/3 + x*t)) * dt)$$

Airy.txt

```

LIBRARY AiFunc
net.myorb.math.specialfunctions.AiryFunctions

!+ AI_IMPORT(x) = AiFunc.Ai
!! Ai(x) = AI_IMPORT x

!+ AIJ_IMPORT(x) = AiFunc.Aij
!! AiJ(x) = AIJ_IMPORT x

!+ BI_IMPORT(x) = AiFunc.Bi
!! Bi(x) = BI_IMPORT x

!! Ai2(x) = Ai''(x <> 0.001)

!! err(x) = Ai2(x) - x*Ai(x)

```

A spline created from Stokes equation and the Bessel identities is shown here producing a smooth plot

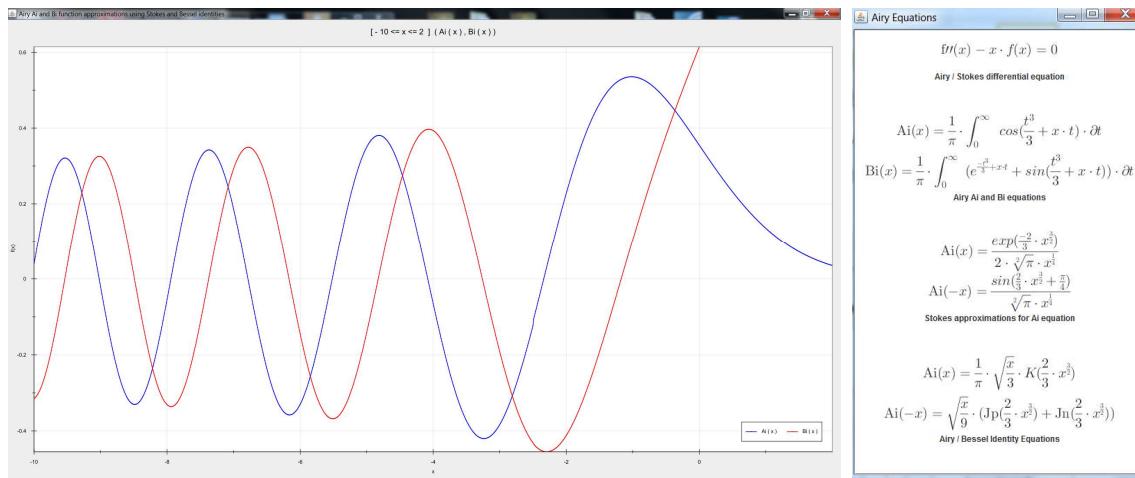
The knot is at $x = -2.5$ where a small jump in the plot can be seen

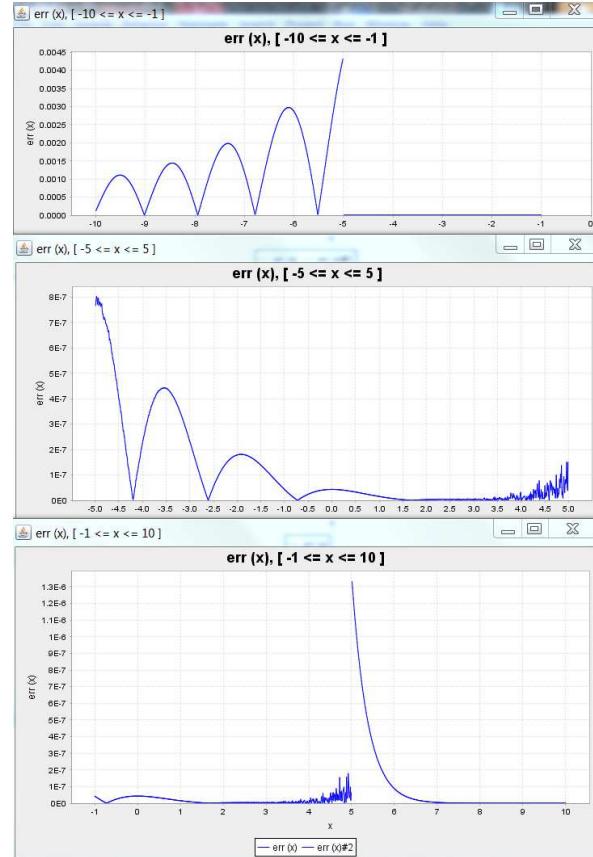
The error plots are done separately on either side of the knot since $x < -2.5$ is an approximation and for $x > -2.5$ identities with Bessel J and K functions are used

note that the error is so small for $x > -2.5$ that the graphing software could not even generate the error tick values

max_err = 1.8135048351908267E-7
min_err = 8.545179841501493E-11

Equations and Plots for $Ai(x)$ and $Bi(x)$





The domain below the low knot at -5 shows the expected errors coming from the Stokes approximation function

Between the low knot at -5 and the hi knot at +5 the error stays very small at $< 1\text{E-}6$ where the computation is done using the Bessel identities based on J_p and K_a . A knot exists at $x=0$ but shows no jump at that location in the plot

The hi knot at +5 separates the use of Bessel functions $[-5 .. +5]$ and the Stokes decay approximation used for $x > +5$. Being an exponential decay the jump at +5 is very small and the error drops very rapidly

AirySpline.txt

```

READ VC31Prep.txt

LIBRARY AiFunc net.myorb.math.specialfunctions.AiryFunctions

n = 4
wid = 1.5
nwid = n * wid
samplestep = n * 0.1
plotstep = n * 0.01

// Ai(x)
!+ AI_IMPORT(x) = AiFunc.Ai
!! Ai(x) = AI_IMPORT x

a = [-nwid <= x <= nwid+0.05 <> samplestep] ( Ai(x) )
calc LENGTH a ; graph a ; ac = VC31INTERP (a)

!! AiS(x) = ac @^n (x/n)
!! erra(x) = AiS''(x>0.001) - x * AiS(x)

PLOTF AiS [ -nwid <= x <= nwid <> plotstep ]
PLOTF erra [ -nwid <= x <= nwid <> plotstep ]

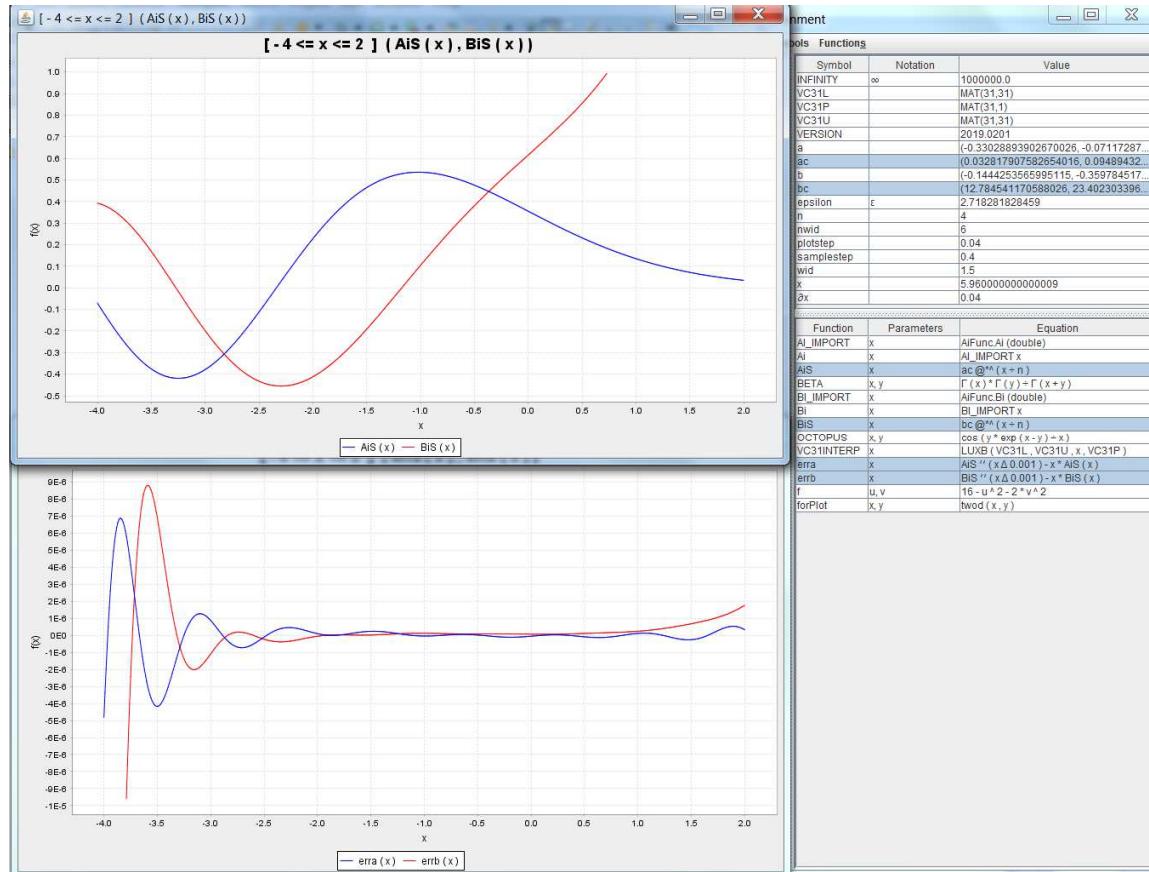
// Bi(x)
!+ BI_IMPORT(x) = AiFunc.Bi
!! Bi(x) = BI_IMPORT x

b = [-nwid <= x <= nwid+0.05 <> samplestep] ( Bi(x) )
calc LENGTH b ; graph b ; bc = VC31INTERP (b)

!! Bis(x) = bc @^n (x/n)
!! errb(x) = Bis''(x>0.001) - x * Bis(x)

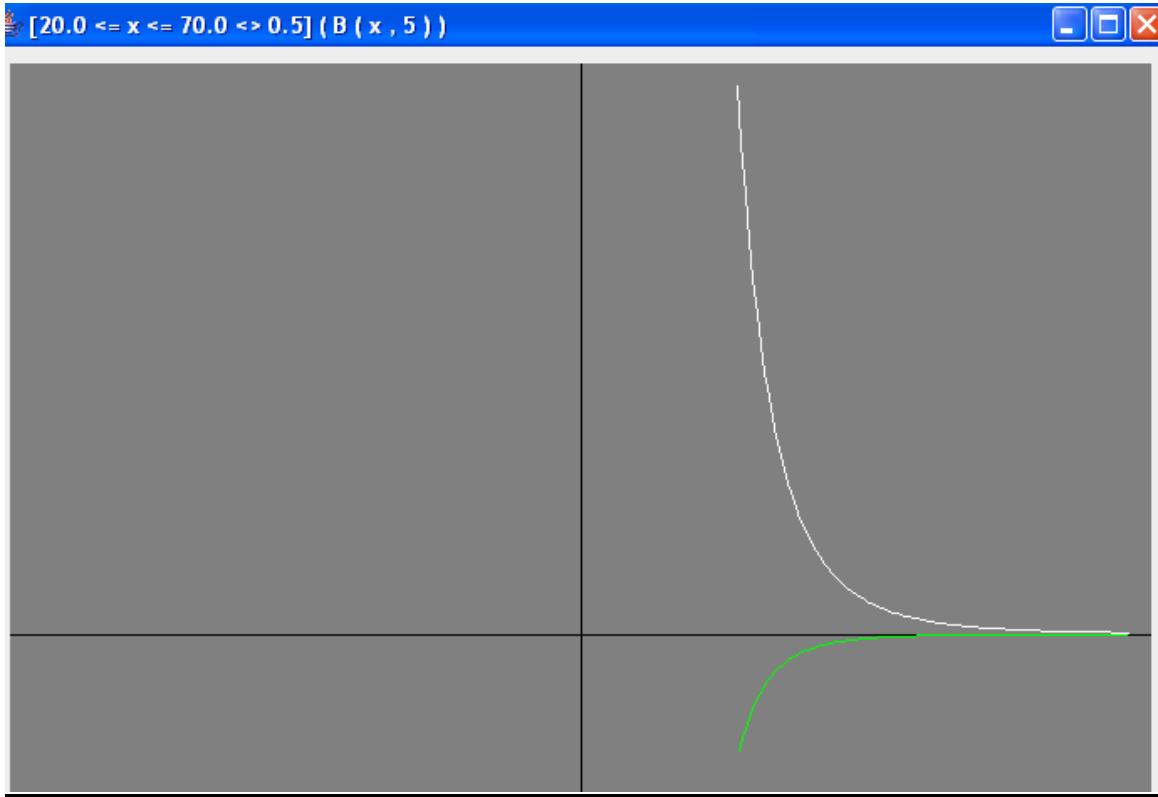
PLOTF Bis [ -nwid <= x <= nwid <> plotstep ]
PLOTF errb [ -nwid <= x <= nwid <> plotstep ]

```



FloatDemoBeta.txt

```
INFINITY = 100  
dt = 1 / INFINITY  
// beta / beta incomplete / inv / ln  
!! B(x,y) = INTEGRAL [0 < t < 1 >> dt] ( t^(x-1) * (1-t)^(y-1) * dt )  
graph [20 <= x <= 70 >> 0.5] (B(x,5))
```



The integration over the 0 to 1 range provides an easier approximation task. The computational value of 'dt' can easily be seen as a trade-off for the user with the understanding that the reduction of the value increases the precision of the result and requires extra time for the extra calculations.

Functions

$$B(x, y) = \int [0 < t < 1 \Delta dt] (t^{x-1} * (1-t)^{y-1}) dt$$

FloatDemoCalculus.txt

READ FloatDemoCalculus.txt

Reading... C:\workspace\MathFields\scripts\FloatDemoCalculus.txt

SCRIPTPRINT FloatDemoCalculus.txt

```
// computations of polynomial derivative (POLYDER) and integral (POLYINT) are not
// approximations but are actual term-by-term integrals and derivative applied to the
// coefficients of the polynomial
```

```
p = (2, -3, 4, -5, 6)
```

```
pder = POLYDER p
pint = POLYINT p
```

```
POLYPRINT p
POLYPRINT pder
POLYPRINT pint
```

```
// by convention the value of 'delta' is assigned as a constant.
// this is a literal selection of precision made by the user with an
// understanding that the smaller delta results in higher calculation time
dx = 0.005
```

```
// starting the integral at 0 reduces the result by the -INFINITY to 0 INTEGRAL
e_minus1_approximation = INTEGRAL [0 <= x <= 1 <> dx](exp(x) * dx)
PRETTYPRINT e_minus1_approximation
```

```
!! f(x) = 2\/(1 - x^2)
// this is the approximation of the area of the unit circle
pi_approximation = 4 * INTEGRAL [0 <= x <= 1 <> dx] ( f(x) * dx )
PRETTYPRINT pi_approximation
```

```

// this is an approximation of the
// Euler-Mascheroni gamma value using the INTEGRAL of the H(x) series

gamma_approximation = INTEGRAL [0 <= x <= 1 <> dx] (HARMONIC(x) * dx)
PRETTYPRINT gamma_approximation

// infinite series approximation.
// expanding the Taylor series for EXP(1) give a fast convergence to the value of e

e_approximation = SIGMA [0 <= i <= 10] (1/i!)
PRETTYPRINT e_approximation

// another infinite series approximation.
// slower convergence so number of terms significantly higher

!! zeta(x) = SIGMA [1 <= n <= 1000] (n^(-x))

Apery = zeta(3)
// the Apery constant is equivalent to zeta(3)
PRETTYPRINT Apery

// CALCLIB accepts the F'(x) notation for requesting derivative approximation.
// the addition of the specification of the delta value allows the user to choose precision.
// again the trade-off is increased precision for increased calculation time
// as smaller delta is chosen

fprime = f'( 0.5 <> dx )
PRETTYPRINT fprime

```

OUTPUT:

```
// the polynomial  
p =  
2 - 3 * x + 4 * x^2 - 5 * x^3 + 6 * x^4
```

```
// actual term-by-term derivative
```

```
pder =  
- 3 + 8 * x - 15 * x^2 + 24 * x^3
```

```
// and actual term-by-term integral
```

```
pint =  
2 * x - 1.5 * x^2 + 1.33333333333333 * x^3  
- 1.25 * x^4 + 1.2000000000000002 * x^5
```

```
// results of integral approximations by brute force area computation
```

```
e_minus1_approximation =
```

```
1.7139897036402167
```

```
pi_approximation =
```

```
3.1511769448395244
```

```
gamma_approximation =
```

```
0.5746638357147642
```

```
// approximations made from infinite series
```

```
e_approximation =
```

```
2.7182818011463845
```

```
Apery =
```

```
1.2020564036593433
```

```
// output of the approximated derivative requested as F'(x)
```

```
fprime =
```

```
-0.5773534767267385
```

FloatDemoCombo.TXT

READ FloatDemoCombo.txt

Reading... C:\workspace\MathFields\scripts\FloatDemoCombo.txt

Reading... C:\workspace\MathFields\scripts\FloatDemoCombo.TXT

// Bernoulli numbers are notoriously difficult to calculate.

// brute force calculation is beyond basic capabilities of most computers

SCRIPTPRINT FloatDemoCombo.TXT

n=1

SCRIPTPRINT ComputeNthBernoulli.txt

ITERATE 10 ComputeNthBernoulli.txt

// note the odd numbered values which should be 0 as evidence that even

// the low numbered results introduce error quickly, higher ones become impossible.

// in a later demo script the Bernoulli sequence is computed using prime fractions.

// the first 30 values are computed precisely as can be verified with online sites.

// the GAMMA function evaluated at positive integers GAMMA(n) = (n-1)!

x = 6

SCRIPTPRINT GammaCompute.txt

SCRIPTPRINT GammaIteration.txt

READ GammaCompute.txt

// the correct value of GAMMA(6) = 5! = 120, so this approximation is off by 2%

OUTPUT:

Reading... C:\workspace\MathFields\scripts\ComputeNthBernoulli.txt

bn = BERNOULLI(n)

PRETTYPRINT bn

n = n + 1

bn = 0.5

bn = 0.16666666666666663

bn = -2.220446049250313E-16

bn = -0.0333333333333399

bn = 1.7763568394002505E-15

bn = 0.02380952380956758

bn = 3.794742298168785E-13

bn = -0.03333333333154209

bn = -1.0449863196981823E-11

bn = 0.07575757526731397

Iteration 10 has completed

*** Maximum iteration count exceeded

Reading... C:\workspace\MathFields\scripts\GammaCompute.txt

!! GammaFactor (t,n) = n / (t+n) * (n / (n-1))^t

```
g = x^2 + x
g = 1 / g
n = 2
```

ITERATE 1000 GammaIteration.txt

PRETTYPRINT g

```
// the correct value of GAMMA(6) = 5! = 120, so this approximation is off by 2%
// and this is at 1000 iterations, at 5000 it is not much closer
// this is very slow convergence
```

ITERATE 4000 GammaIteration.txt

PRETTYPRINT g

ITERATE 5000 GammaIteration.txt

PRETTYPRINT g

ITERATE 3000 GammaIteration.txt

PRETTYPRINT g

// IMPORTANT NOTE: look at the formula for the Nth GAMMA factor.

// note that for integer t the computation can be done as integer exponentiation.

// if t is real, computation becomes exp(ln(n)*t), much more expensive

Reading... C:\workspace\MathFields\scripts\GammaIteration.txt

g = g * GammaFactor(x,n); n=n+1

Iteration 1000 has completed

g = 117.51405987584201

Iteration 5000 has completed

g = 119.49737453500546

Iteration 10000 has completed

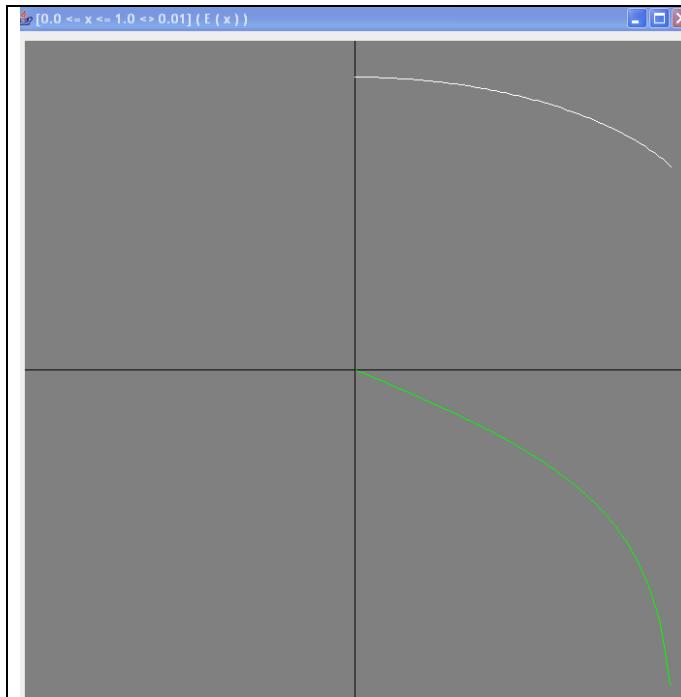
g = 119.74834401649464

Iteration 13000 has completed

g = 119.80635745853313

FloatDemoEllip.txt

Again the integration range being 0 – 1 allows a reasonable approximation using
INFINITY = 100 and dx = 0.01 resulting in a smooth curve in the plot, but how accurate?



INFINITY = 100

dt = 1 / INFINITY

// ellipj ellipke

!! K(k) = INTEGRAL [0<=t<=1 >> dt] (1 / sqrt ((1-t^2) * (1-k^2*t^2)) * dt)

!! E(k) = INTEGRAL [0 <= t <= 1 >> dt] (sqrt ((1-k^2*t^2) / (1-t^2)) * dt)

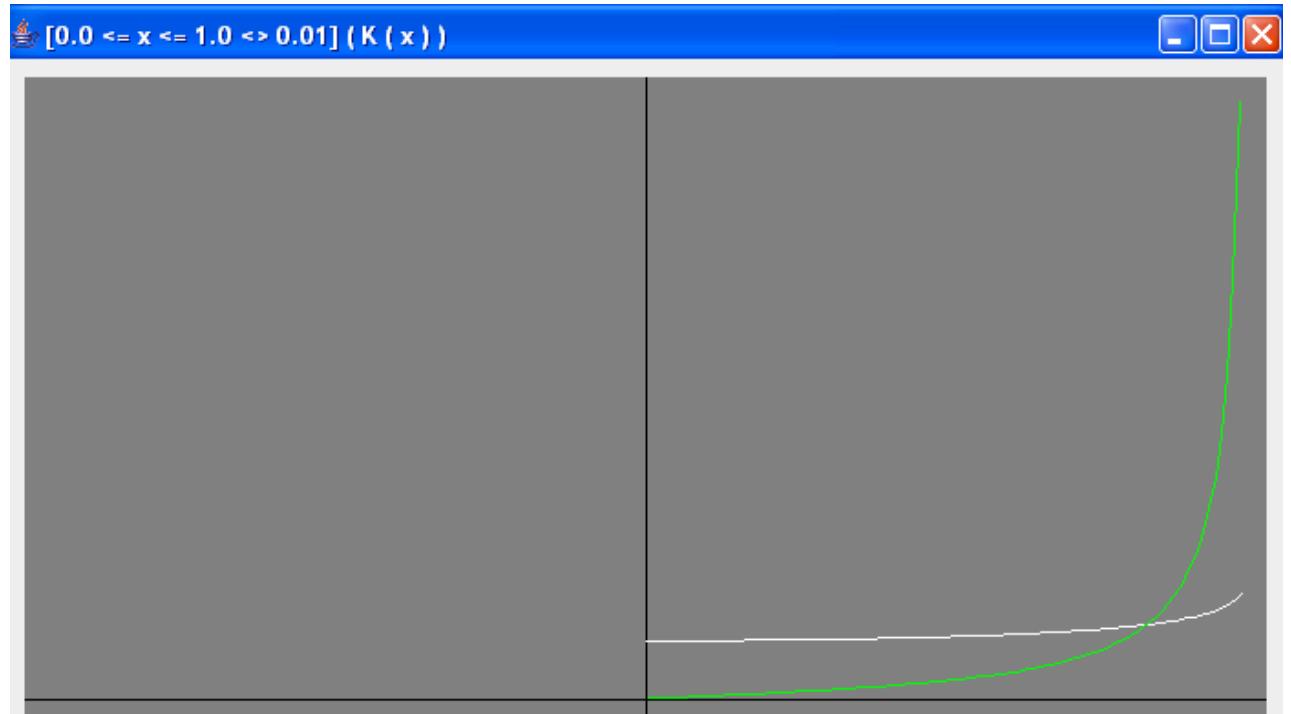
graph [0 <= x <= 1 >> dt] (E(x))

graph [0 <= x <= 1 >> dt] (K(x))

Functions

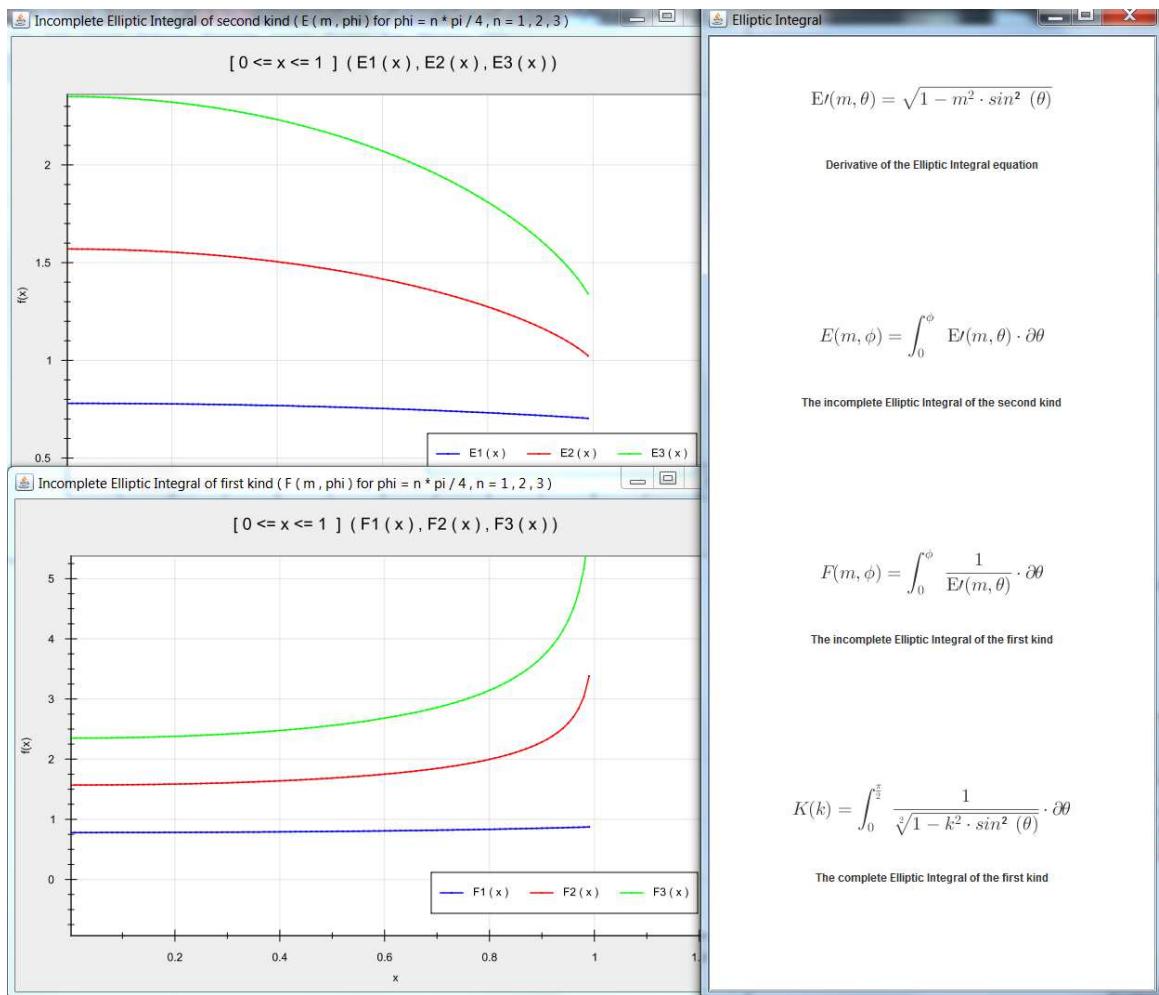
$$E(k) = \int [0 \leq t \leq 1] (\sqrt{(1 - k^2 t^2) / (1 - t^2)}) * dt$$

$$K(k) = \int [0 \leq t \leq 1] (\sqrt{((1 - t^2) * (1 - k^2 * t^2))} * dt)$$



EllipticInt.txt

<pre> INFINITY = 100 dt = 1 / INFINITY qPi = pi #/# 4 // complete first kind !! K(k) = INTEGRAL [0 < theta < pi/2 <> dt] (1 / 2\k^2 * sinsq theta) * <> theta) graph [0 <= x <= 1 <> dt] (K(x)) ENTITLED Complete Elliptic Integral of first kind (K(k)) // complete second kind !! E'(m, theta) = sqrt (1 - m^2 * sinsq theta) !! E(m, phi) = INTEGRAL [0 < theta <= phi <> dt] (E'(m,theta) * <> theta) !! F(m, phi) = INTEGRAL [0 < theta <= phi <> dt] (1 / E'(m,theta) * <> theta) SELECT "Elliptic Integral" RENDERF E' RENDER "Derivative of the Elliptic Integral equation " TOP RENDERF E RENDER "The incomplete Elliptic Integral of the second kind " TOP RENDERF F RENDER "The incomplete Elliptic Integral of the first kind " TOP RENDERF K RENDER "The complete Elliptic Integral of the first kind " TOP // degenerate case is simple circle // graph [0 < x < 1 <> dt] (E'(x, 2 * qPi)) !! E1(x) = E(x, qPi) !! E2(x) = E(x, 2 * qPi) !! E3(x) = E(x, 3 * qPi) !! F'(m, theta) = 1 / E'(m, theta) !! F(m, phi) = INTEGRAL [0 < theta <= phi <> dt] (F'(m, theta) * <> theta) !! F1(x) = F(x, qPi) !! F2(x) = F(x, 2 * qPi) !! F3(x) = F(x, 3 * qPi) graph [0 <= x <= 1 <> dt] (F1(x),F2(x), F3(x)) ENTITLED Incomplete Elliptic Integral of first kind (F(m,phi) for phi = n*pi/4, n=1,2,3) graph [0 <= x <= 1 <> dt] (E1(x), E2(x), E3(x)) ENTITLED Incomplete Elliptic Integral of second kind (E(m,phi) for phi = n*pi/4, n=1,2,3) // !! Er(k) = INTEGRAL [0 < t < 1 <> dt] (sqrt ((1-k^2*t^2) / (1-t^2)) * dt) // !! dif2(x) = E2(x) - Er(x) // !! dif3(x) = E3(x) - Er(x) // graph [0 < x < 1 <> dt] (dif2(x), dif3(x)) </pre>	<p>Description of integrals of first and second kind</p> <p>Equation renders</p> <p>Prepare plots for phi = n * pi/4</p> <p>Incomplete first kind</p> <p>N = 1 N = 2 N = 3</p> <p>Incomplete second kind</p>
--	--



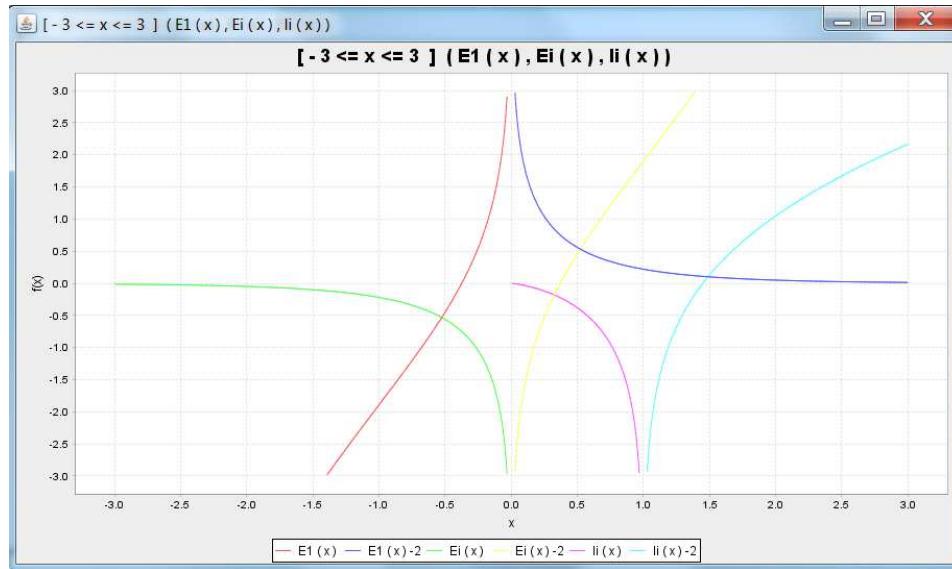
FloatDemoExpIntegral.txt

READ JavaMath.txt

```
INFINITY = 70  
dt = 1 / INFINITY
```

```
// exp integral  
!! Ei(x) = - INTEGRAL [-x <= t <= INFINITY] ( EXP(-t) / t * dt )  
!! E1(x) = INTEGRAL [x <= t <= INFINITY] ( EXP(-t) / t * dt )
```

```
graph [0.2 <= x <= 1.3 <> 0.01] (E1(x))
```



Exponential decay term-by-term, so we would expect fast convergence

Functions

$$E1(x) = \int [x \leq t \leq \infty] (\exp(-t) / t) dt$$

$$Ei(x) = - \int [-x \leq t \leq \infty] (\exp(-t) / t) dt$$

FloatDemoRegression.txt

```
READ FloatDemoRegression.txt
Reading... C:\workspace\MathFields\scripts\FLOATDemoRegression.txt
```

```
SCRIPTPRINT FloatDemoRegression.txt
READ regress_data.txt
```

```
PRETTYPRINT y_line
line_coefficients = FITLINE (x,y_line)
!! line(x) = line_coefficients +*^ x
PRETTYPRINT line_coefficients
```

```
PRETTYPRINT y_exp
exp_coefficients = FITEXP (x,y_exp)
!! aebx(x) = exp_coefficients *^# x
PRETTYPRINT exp_coefficients
```

```
PRETTYPRINT y_poly
poly_coefficients = FITPOLY (x,y_poly)
!! poly(x) = poly_coefficients +*^ x
PRETTYPRINT poly_coefficients
```

```
PRETTYPRINT y_har
har_coefficients = FITHARMONIC (x_har,y_har,0.1)
!! har(x) = har_coefficients +#* x
PRETTYPRINT har_coefficients
```

```
SHOW Functions
```

FloatDemoRegression.txt

```
x = (-3, -2, 1, 4, 5)
```

```
y_poly = (4, -7.1, 9.5, -13.5, 16.2)
```

```
y_line = (4, 7.1, 9.5, 13.5, 16.2)
```

```
y_exp = (0.1, 0.3, 6, 23.5, 66.2)
```

```
y_har = (4, -7.1, 9.5, -13.5, 16.2)
```

```
x_har = (1.1, 3.3, 3.9, 5.1, 6.3)
```

```
// linear regression using least squares methodology
```

```
Reading... C:\workspace\MathFields\scripts\regress_data.txt
```

```
// prepare a sample of data
```

```
y_line =
```

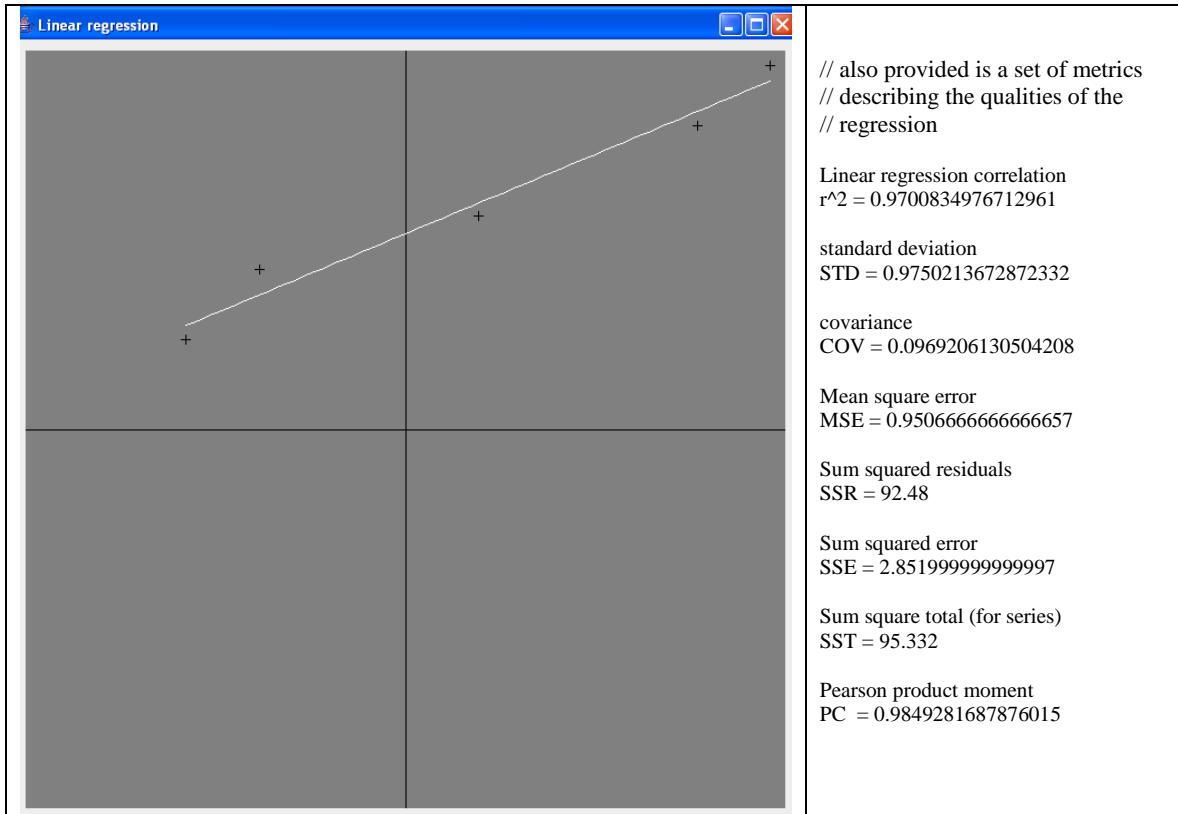
```
[  
 4  
 7.1  
 9.5  
 13.5  
 16.2
```

```
]
```

```
// the result is a set of coefficients describing a line, specifically a slope and a y-intercept
```

```
Linear regression
```

$$Y = 8.700000000000001 + 1.36*x$$



```
// Non-linear regression uses least squares between Ln Y as a function of X
```

```
y_exp =
```

```
[  
 0.1  
 0.3  
 6  
 23.5  
 66.2  
]
```

Non-Linear regression

$$\ln Y = 0.3456968531967063 + 0.7812796380035235 * X$$

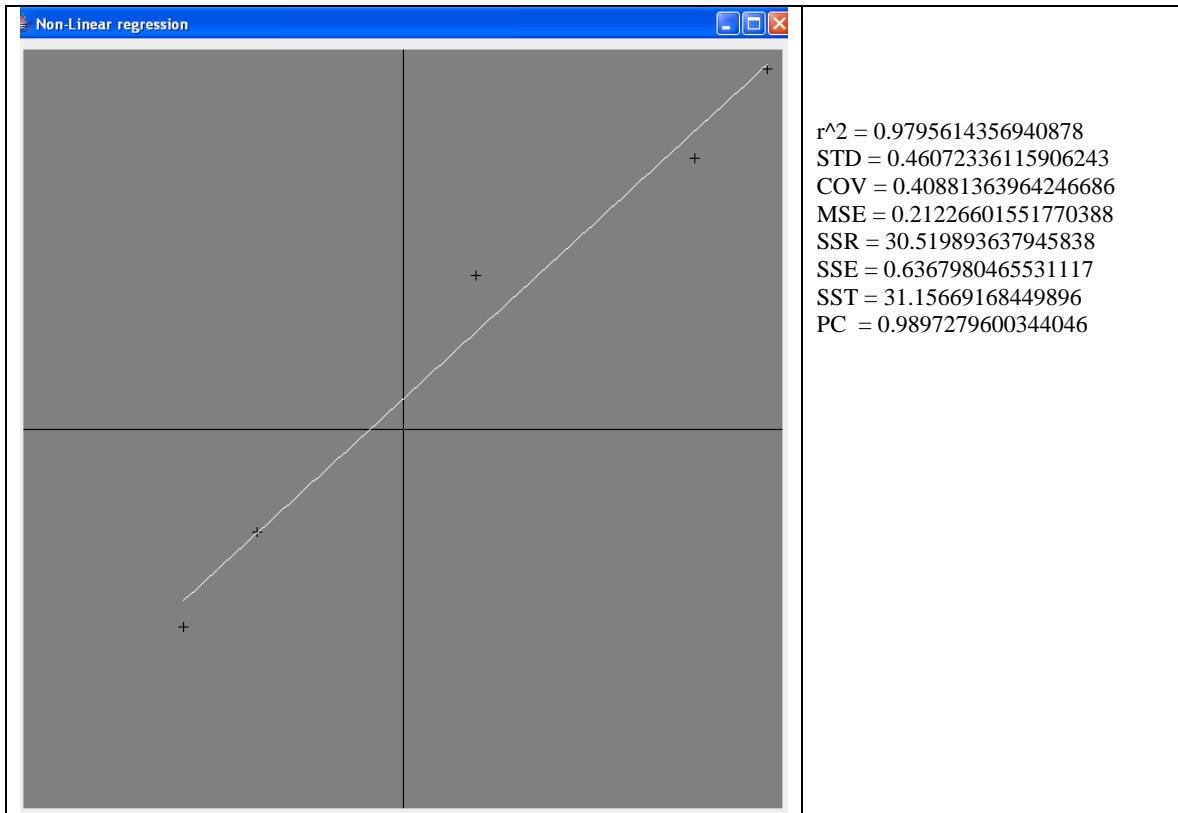
// The linear solution taking the form

$$// \quad \ln Y = a + b * X$$

// Solving for Y gives:

$$// \quad Y = \text{EXP} (a + b * X) = \text{EXP}(a) * \text{EXP} (b * X)$$

// A solution of the form $Y = C * \text{EXP} (b * X)$ { where $C = \text{EXP}(a)$ }



```
// Polynomial interpolation uses linear algebra to solve for polynomial coefficients.
// Vandermonde matrix is built from sample data with coefficients as the variables.
// solution for coefficients realized using matrix inversion for small sample sets,
// or by application of Gaussian elimination for larger matrices
```

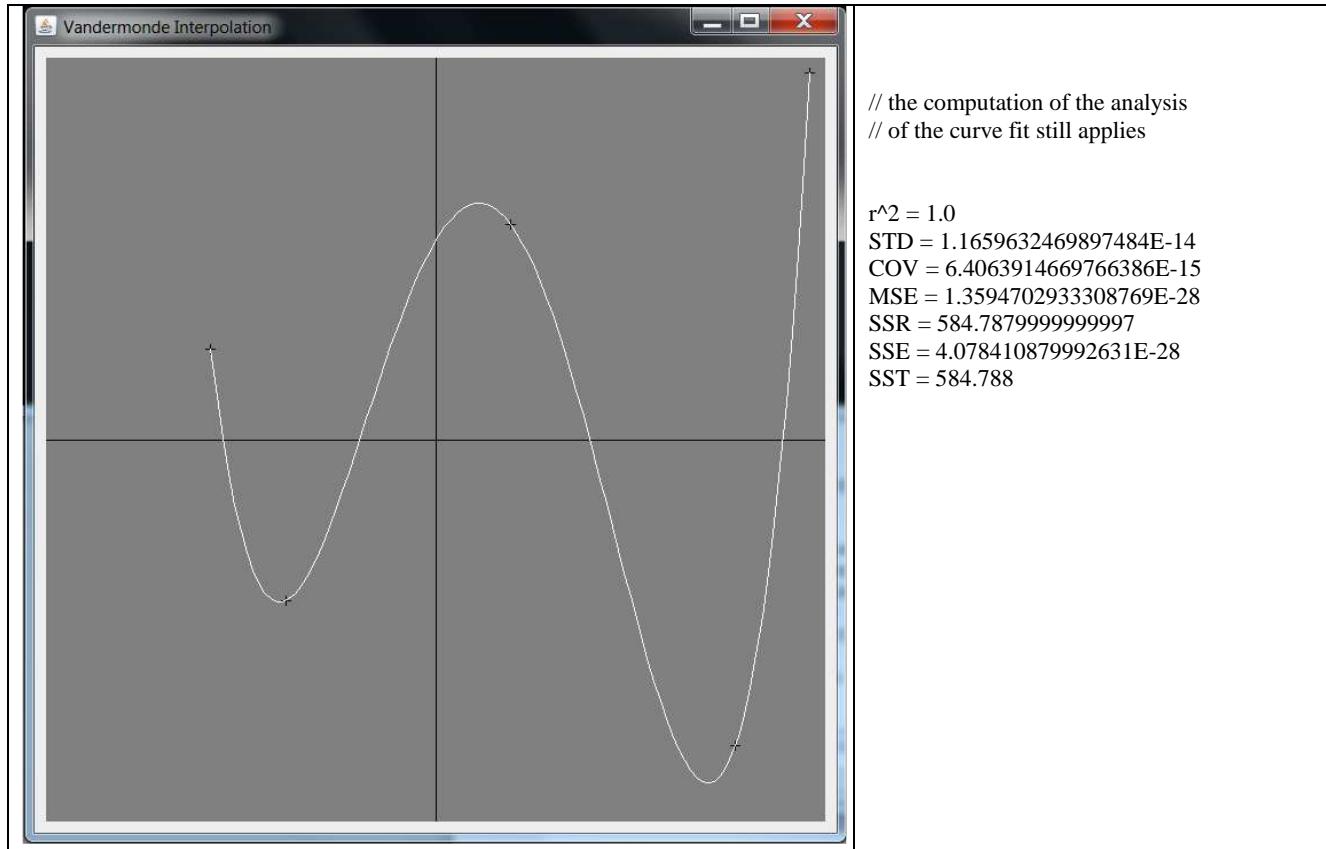
```
READ regress_data.txt
Reading... C:\workspace\MathFields\bin\scripts\regress_data.txt
```

```
y_poly =
[
    4
    -7.1
    9.5
    -13.5
    16.2
]
x_FITPOLY_y_poly = FITPOLY (x, y_poly)
```

// the solution is a set of coefficients making up a polynomial

Vandermonde Interpolation

$$Y = 8.771428571428572 + 5.586904761904762 * x - 4.269642857142857 * x^2 \\ - 0.908333333333333 * x^3 + 0.3196428571428571 * x^4$$



```

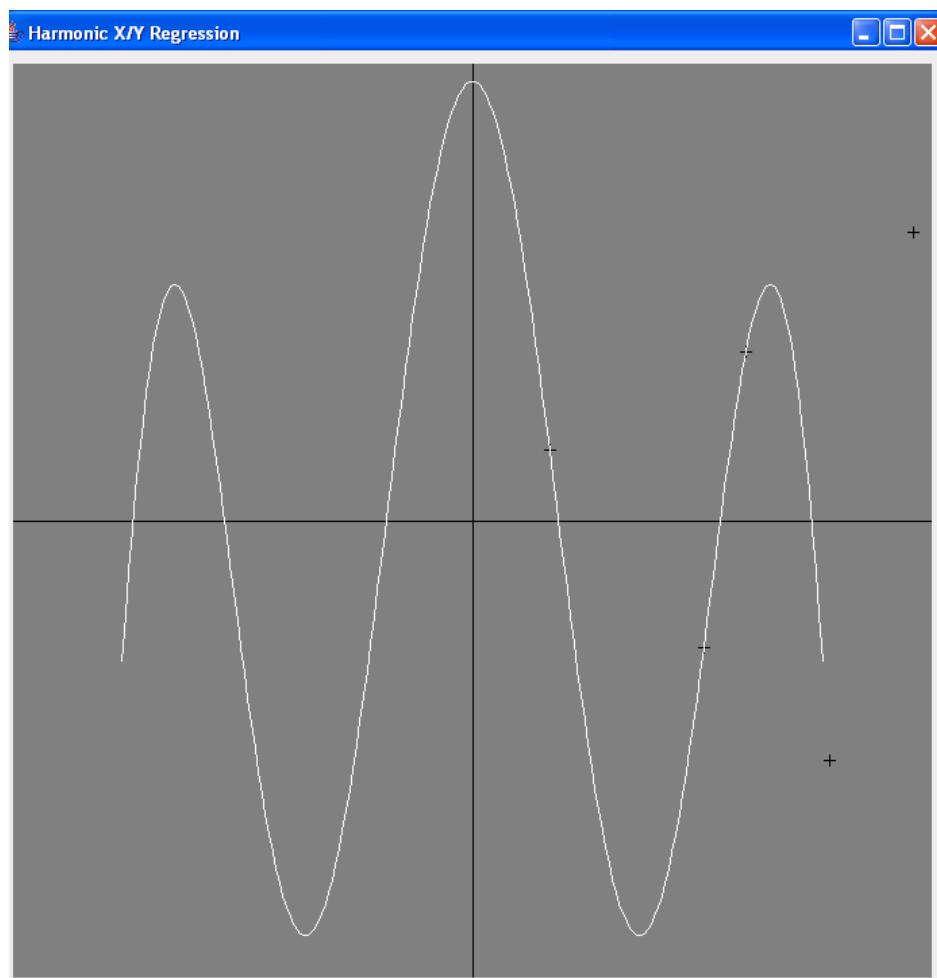
// harmonic regression using linear algebra to solve for Fourier series coefficients

y_har =
[
  4
  -7.1
  9.5
  -13.5
  16.2
]

har_coefficients =
[
  510461.2340641936
  -1215404.4917130603
  1098328.6886200758
  -478250.9996191938
  84890.2593673543
]

```

// fitting the series $Y = c_0 + c_1 \cos wX + c_2 \cos 2wX + c_3 \cos 3wx + \dots$
// where w (actually omega is common) is the base period, each a different solution

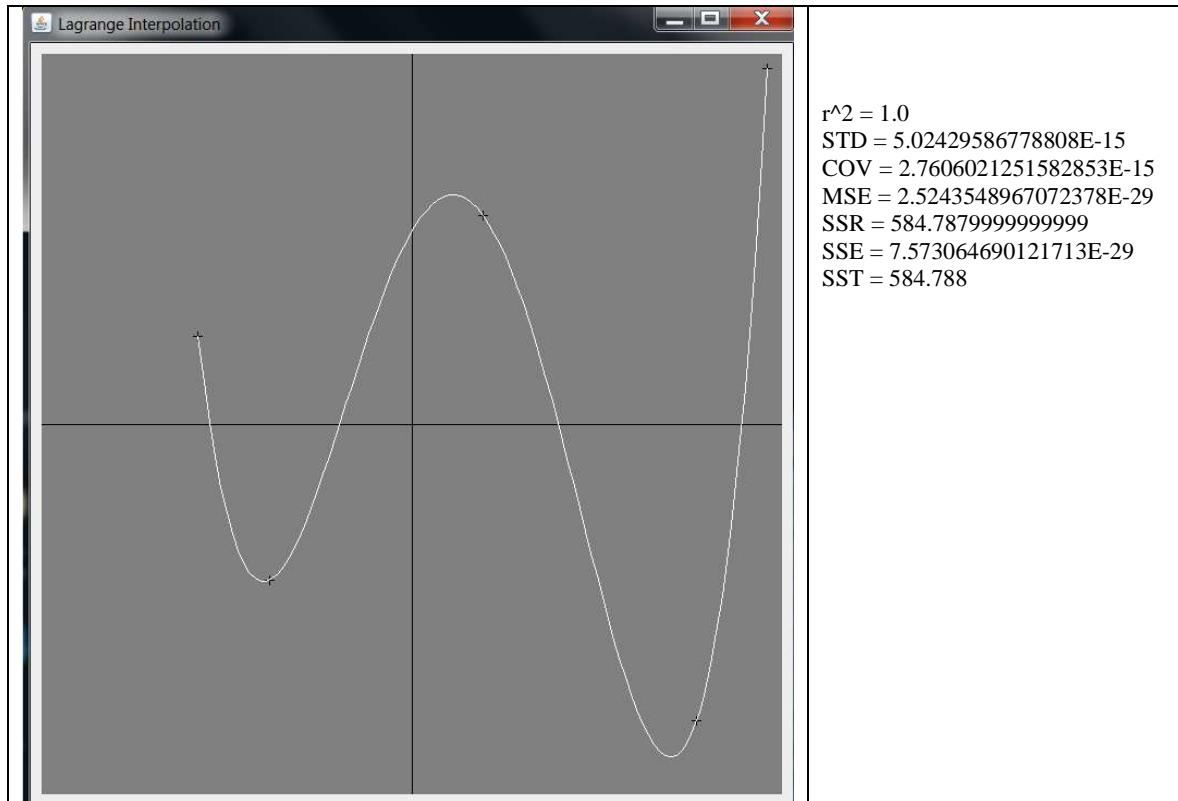


```
// Lagrange Interpolation derives the polynomial best fitting the data
// using computation made directly on the data sample with no matrix intermediates.
// this is an alternative to Vandermonde matrix construction
```

```
x_LAGRANGE_y_poly = LAGRANGE (x, y_poly)
```

Lagrange Interpolation

$$Y = 8.771428571428572 + 5.586904761904761 * x - 4.269642857142857 * x^2 \\ - 0.908333333333332 * x^3 + 0.3196428571428571 * x^4$$



Note that Vandermonde and Lagrange produced nearly identical polynomials
 (in this case it is seen, perhaps others will show differences?)
 Also, SSE comes up different, interesting

// The functions created in the regression script:

Functions

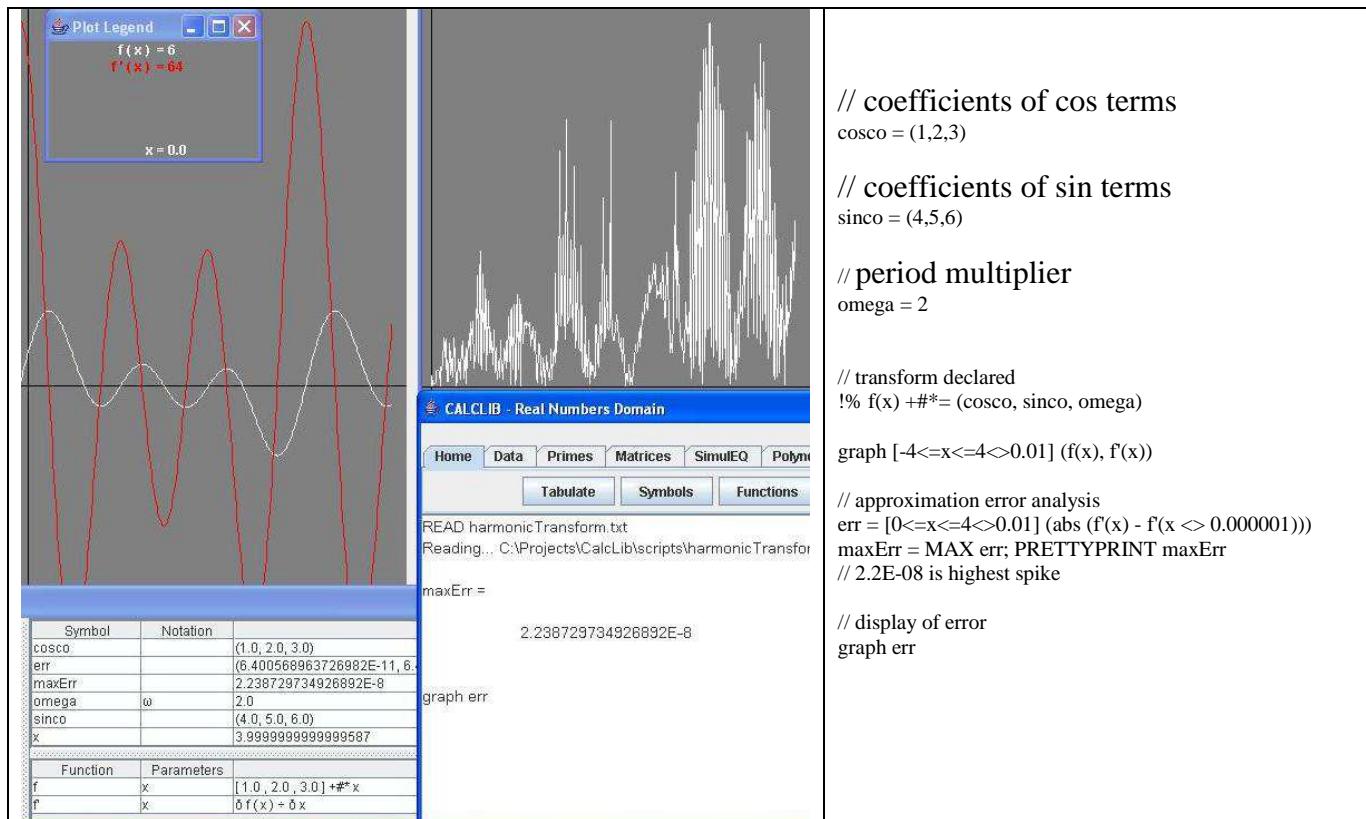
aebx (x)	exp_coefficients *^# x
har (x)	har_coefficients +#* x
line (x)	line_coefficients +*^ x
poly (x)	poly_coefficients +*^ x

4 symbols found and displayed

HarmonicTransform.txt

Harmonic Transform describes the Fourier series form

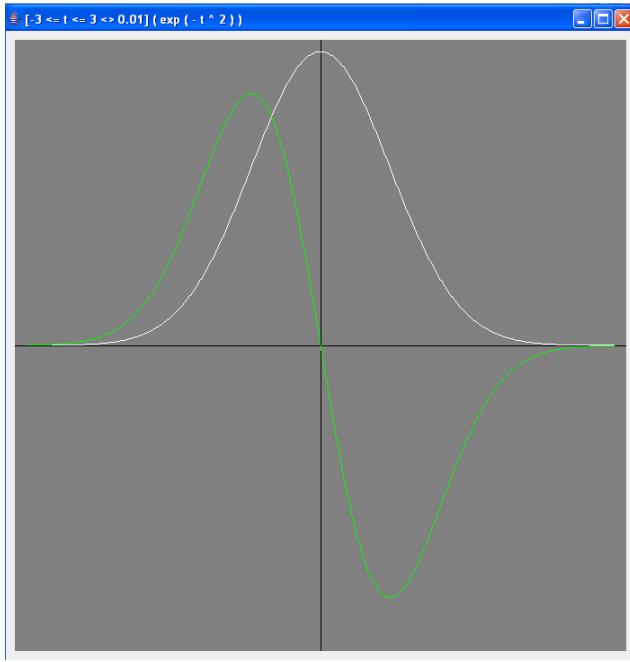
$$F(x) = c_1 \cos wx + c_2 \cos 2wx + c_3 \cos 3wx + \dots + s_1 \sin wx + s_2 \sin 2wx + \dots$$



the error analysis shows the difference between the computations of the function derivative from the function transform as opposed to the rise/run approximation calculation at each point. Using a delta of 1E-06 the maximum error peak found in the difference is 2.2E-08

FloatDemoErf.txt

Again, short range on the integral, but now in addition we have $\exp(-x)$ as a factor giving extremely fast convergence since term values now diminish exponentially



```
SCRIPTPRINT FloatDemoErf.txt

// constants for approximation purposes

INFINITY = 100
dt = 1 / INFINITY

// create an array function describing the curve
curve = [-3 <= t <= 3 >> dt] ( exp ( - t ^ 2 ) )

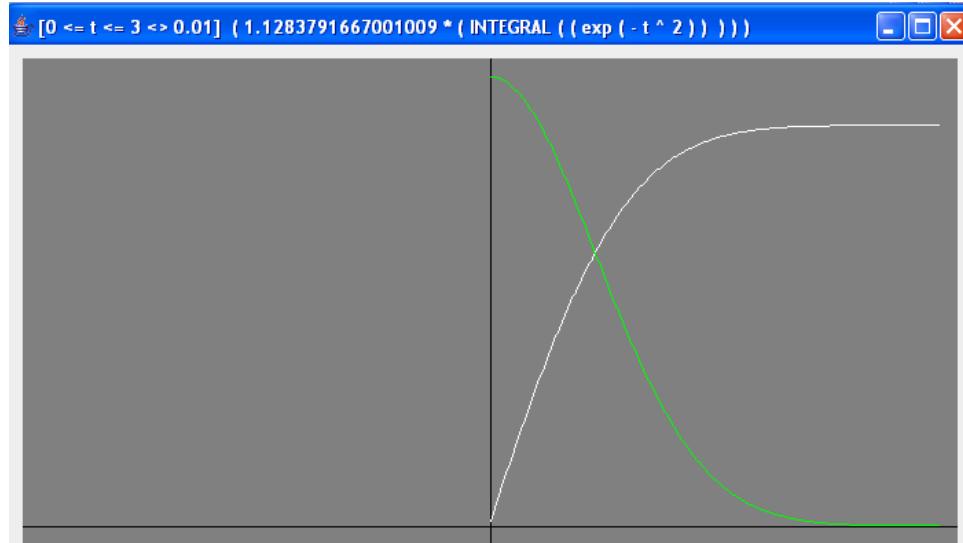
// show a plot of the curve
graph curve

// show a plot of the integral of the curve
graph 2 / sqrt pi * ARRAYINT (curve, 0, 3)

// capture integral as array function
erf = 2 / sqrt pi * ARRAYINT (INTERVAL (curve, 0, 3))

// define erfc function
!! erfc(x) = 1 - ( erf @# x )
graph [0 <= t <= 3 >> dt] ( erfc (t) )
```

(continued next page)



Functions

$$2 \div \sqrt(\pi) * \int [0 <= t <= 3 \Delta 0.01] (\exp (-t^2))$$

Polynomial Interpolation

```
// produce a polynomial interpolation of erfx (t)
// reduce the domain to provide proper size sample set
samples = [0 <= t <= 3 >> 0.3] ( erf @# t )

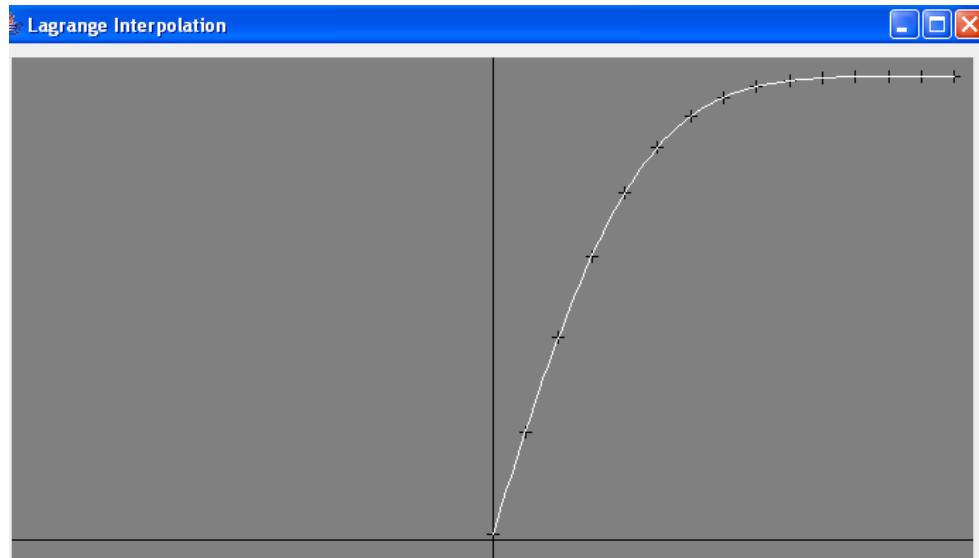
// perform polynomial interpolation
erfpoly = INTERPOLATE samples
!! erfx (x) = erfpoly +*^ x

GRAPH [0 <= t <= 3 >> dt] ( erfx (t) )
```

Lagrange Interpolation

$$Y = 0.001128 + 1.131666 * x - 0.031665 * x^2 - \\ 0.257873 * x^3 - 0.239321 * x^4 + 0.396119 * x^5 - \\ 0.195003 * x^6 + 0.039859 * x^7 - 5.018E-4 * x^8 - \\ 0.001058 * x^9 + 1.1998855568731703E-4 * x^{10}$$

$$r^2 = 0.9999999999999938 \\ STD = 2.61643420047861E-8 \\ COV = 3.3337018563070156E-8 \\ MSE = 6.845727925434144E-16 \\ SSR = 1.4281441558144792 \\ SSE = 8.899446303064386E-15 \\ SST = 1.4281441240103792$$



```
// the full set of generated polynomial coefficients
```

```
PRETTYPRINT p
```

```
p =
```

0	0.0011283791667001009	6	-0.19500366487283216
1	1.1316668575887787	7	0.03985950459153642
2	-0.03166565082125494	8	-5.018255826056972E-4
3	-0.2578731233342175	9	-0.0010581619105540785
4	-0.23932128353545323	10	1.1998855568731703E-4
5	0.3961193902536131		

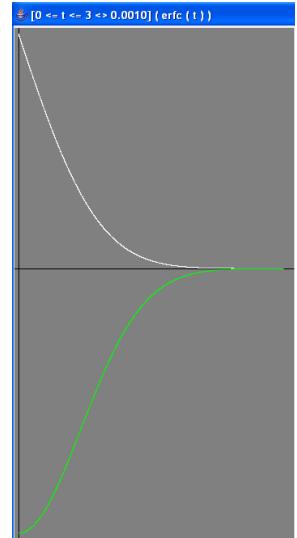
```
]
```

Applied Approximation Theory

Full expression of a function can be used to calculate points along a curve. With complicated expression using other complex function evaluations the computation time can become unacceptable where many evaluations are necessary. Polynomial interpolation used within approximation theory seeks the optimal polynomial (or polynomials) of Spline describing the function in segments) that will minimize the maximum value of the function

$$e = | P(x) - f(x) | \quad \text{where } P(x) \text{ is the approximation function and } f(x) \text{ is the correct value of the function}$$

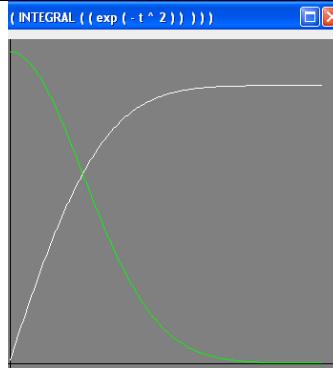
The definition of $\text{erfc}(t)$ was done in the script based on an array function. This would be a relatively fast function but extrapolation is used to compute values between elements and this may introduce unwanted errors



Interpolation could also be done on $\text{erfc}(x)$ and a polynomial approximation published for this function

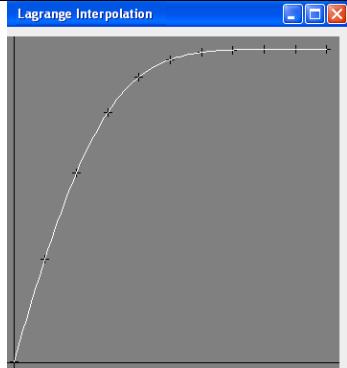
Function Analysis

The full expression of the function is used to create an array function. The long expression tends to be expensive to compute and the array function introduces errors where extrapolation is required



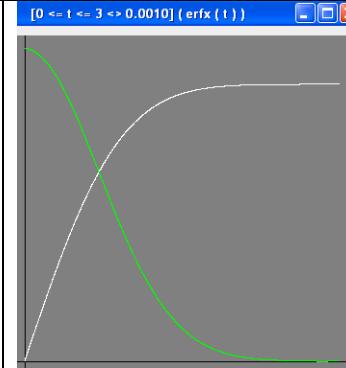
Interpolation

The interpolation step produces a set of polynomial coefficients that will approximate the function over the domain equivalent to the domain specified in the array function definition



Publishing

The interpolated polynomial coefficients can be used to define a function based on a polynomial. This function will be necessarily continuous and the accuracy needs to be evaluated to determine usefulness



Functions

$$\begin{aligned} \text{erfc}(x) &= 1 - (\text{erf} @\# x) && // \text{array function evaluation based on the 'erf' array} \\ \text{erfx}(x) &= \text{erfpoly} + *^{\wedge} x && // \text{polynomial based on 'erfpoly' coefficients} \end{aligned}$$

2 symbols found and displayed

FloatDemoSpline.txt

```
// Divide the domain of a function into segments.  
// Polynomial interpolation is done on each segment separately.  
// A "spline" is defined based on the set of polynomials. All function evaluation  
// is then done based on the segment referenced.
```

```
READ FloatDemoSpline.txt
```

```
Reading... C:\workspace\MathFields\scripts\FloatDemoSpline.txt
```

```
SCRIPTPRINT FloatDemoSpline.txt
```

```
// constants for approximation purposes  
INFINITY = 1000; dt = 1 / INFINITY
```

```
// create an array function describing the curve  
curve = [-3 <= t <= 3 >> dt] ( exp ( - t ^ 2 ) )
```

```
// select segment domains and collect samples
```

```
seg1 = [0 <= t <= 1 >> 0.1] ( curve @# t )  
seg2 = [1 <= t <= 2 >> 0.1] ( curve @# t )  
seg3 = [2 <= t <= 3 >> 0.1] ( curve @# t )
```

```
s1poly = INTERPOLATE seg1  
s2poly = INTERPOLATE seg2  
s3poly = INTERPOLATE seg3
```

```
// define a spline of 3 segments  
!$ bell(x) = (s1poly, s2poly, s3poly)
```

```
GRAPH [0 <= x <= 3 >> 0.001] (bell (x))
```

```
// evaluation of the error curve
```

```
!! P(x) = bell(x)  
!! f(x) = curve @# x
```

```
err = [0 <= x <= 3 >> 0.001] (abs (P(x) - f(x)))
```

```
maxError = MAX err  
PRETTYPRINT maxError  
GRAPH err
```

OUTPUT:

// interpolation of seg1

Lagrange Interpolation

$$Y = 0.999999999999999 - 8.463247910261984E-7 * x - \\ 0.9999750038276538 * x^2 - 3.0337950116177126E-4 * x^3 + \\ 0.5020299854145378 * x^4 - 0.008398561960802908 * x^5 - \\ 0.14401729277506092 * x^6 - 0.04054632573752315 * x^7 + \\ 0.08925391123739246 * x^8 - 0.03450164736568695 * x^9 + \\ 0.004338601056701918 * x^10$$

// (captured as s1poly)

r^2 = 1.0
STD = 3.459645151802116E-10
COV = 4.671725253261122E-10
MSE = 1.1969144576387888E-19
SSR = 0.5120456723098887
SSE = 1.0772230118749099E-18
SST = 0.5120456713082211
PC = -0.9858678255264112

// interpolation of seg2

Lagrange Interpolation

$$Y = 0.8286200879697234 + 0.9736284837995299 * x - \\ 3.200753017808287 * x^2 + 2.470769299660251 * x^3 - 0.8012268409784729 * \\ x^4 + 0.09684142852894684 * x^5$$

// (captured as s2poly)

r^2 = 1.0
STD = 2.2102786549547502E-13
COV = 1.5062381178003443E-12
MSE = 4.88533173254858E-26
SSR = 0.08995776735634785
SSE = 1.954132693019432E-25
SST = 0.08995776735615145
PC = -0.9582660424707687

```
// interpolation of seg3
```

```
Lagrange Interpolation
```

```
Y = - 2.389979813072614 + 13.52847449275123 * x -
24.492748482913157 * x^2 + 22.94192603632837 * x^3 - 13.011997020365925
* x^4 + 4.7541003534429365 * x^5 - 1.133624246020645 * x^6 +
0.17138242630778677 * x^7 - 0.014972771910288785 * x^8 +
5.7754667090959E-4 * x^9
```

```
// (captured as s3poly)
```

```
r^2 = 0.9999999996805409
STD = 1.1429547976886718E-7
COV = 2.242888597175639E-5
MSE = 1.3063456695595527E-14
SSR = 3.2714460471502753E-4
SSE = 1.0450765356476422E-13
SST = 3.2713940689165356E-4
PC = -0.8952813088488132
```

```
// the request to define a spline is accepted,
// verification on continuity is done generating appropriate errors;
// the segment table is then displayed in domain order
```

```
bell is found to be continuous on the interval 0.0 to 3.0
```

s1poly	0.0	1.0
s2poly	1.0	2.0
s3poly	2.0	3.0

```
maxError =
```

```
8.746921299832011E-7
```

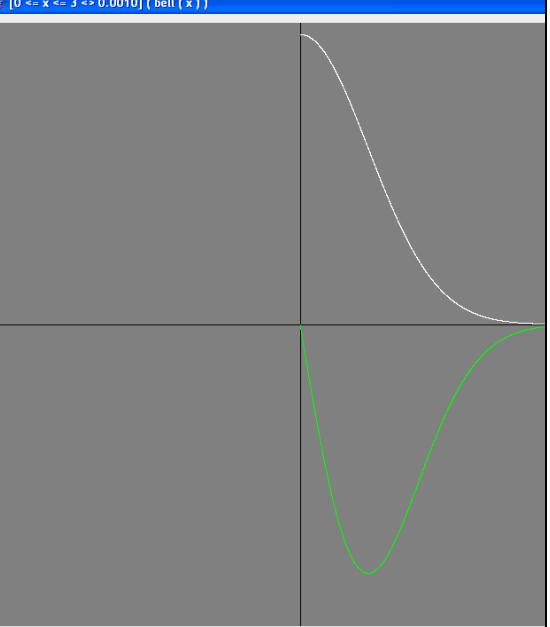
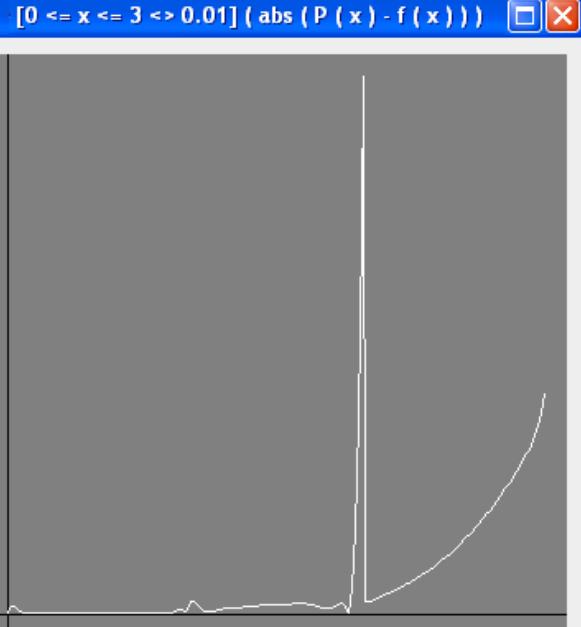
```
// see the chart, the largest error occurs at the "knot" point
// where segments change (x = 2) as might be expected
```

```
// the ENCODE command exports the function to a Java class
```

```
ENCODE bell
```

Spline Plot

The error plot does show that there are spikes at the knots (segment borders).
The peak at $x=2$ seems large until one notes that the MAX is 6.5E-07,
So the scale makes the image misleading

	
At a distance the curve looks exactly like the original, it seems impressive that there are no readily noticeable differences	The MAX error value at 6.5E-07 for a function that goes from 0 – 1 makes it reasonable that the error is not obvious

The increased error on the interval $2 < x < 3$ indicates some difficulty for the interpolation maintaining accuracy as the function value becomes very small relative to the full range of function values

Spline Export

Splines can be exported producing a Java class. This class can be compiled and added to the CLASSPATH allowing the function to be imported in the same way as external library functions. JAR files can be built making transport of function libraries as simple as any Java product. Used in conjunction with the CALCLIB JAR the function library JAR can make the functions available in any Java program.

```
public class Bell extends net.myorb.math.expressions.symbols.SplineInRealDomain
{
    public void initialize ()
    {
        setName ("Bell");
        setSegmentLoConstraints
        (
            0, 1, 2
        );
        setHiConstraint (3);
        addSegmentPolynomial
        (
            0.9999999999999999,
            -8.463247910261984E-7,
            -0.9999750038276538,
            -3.0337950116177126E-4,
            0.5020299854145378,
            -0.008398561960802908,
            -0.14401729277506092,
            -0.04054632573752315,
            0.08925391123739246,
            -0.03450164736568695,
            0.004338601056701918
        );
        addSegmentPolynomial
        (
            1.0891112255399094,
            -0.6151861959406233,
            0.8686902279332571,
            -3.2547740892405272,
            4.029346242787142,
            -2.3866178912721807,
            0.7430333815646009,
            -0.1088799341887352,
            0.002376469596924835,
            7.800020653121464E-4
        );
        addSegmentPolynomial
        (
            -2.389979813072614,
            13.52847449275123,
            -24.492748482913157,
            22.94192603632837,
            -13.011997020365925,
            4.7541003534429365,
            -1.133624246020645,
            0.17138242630778677,
            -0.014972771910288785,
            5.7754667090959E-4
        );
    }
}
```

Access To Exported Functions

// with the exported "Bell" class, an interface can be coded:

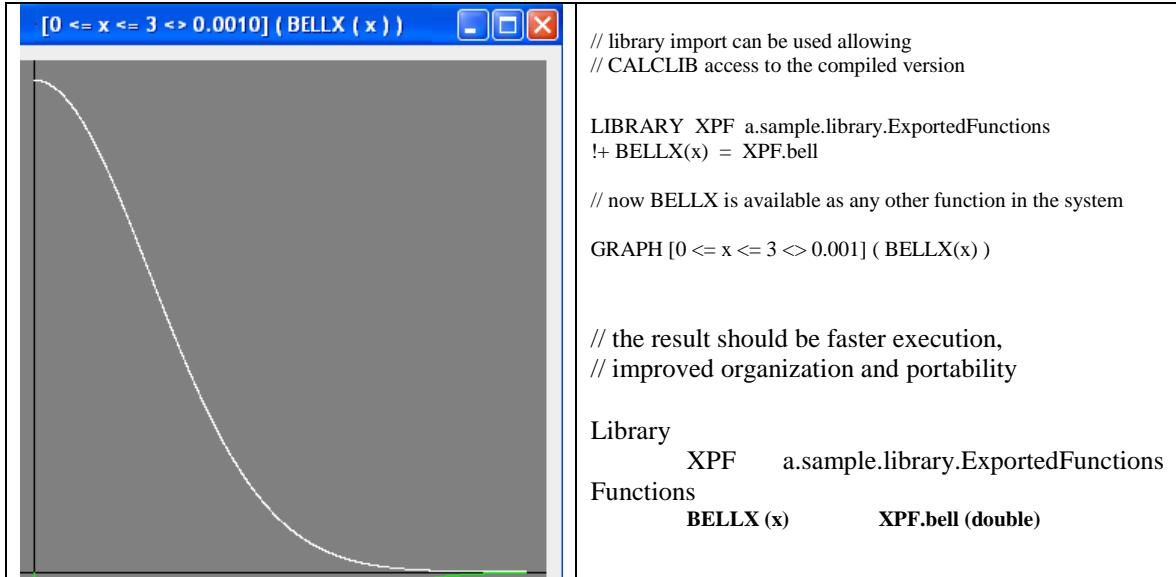
```
package a.sample.library;
import net.myorb.math.MultiDimensional;

/**
 * provide static interface to Bell function
 */
public class ExportedFunctions
{
    public static double bell (double x)
    {
        if (bell == null) bell = new Bell ().getFunction ();
        return bell.f (x);
    }
    static MultiDimensional.Function<Double> bell = null;
}
```

// this can be referenced in any Java program:

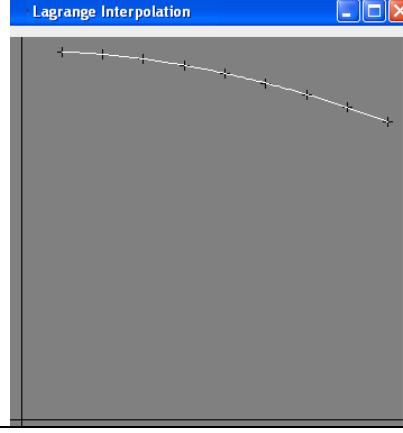
```
package a.sample.library;

/**
 * unit test for Bell class using static interface
 */
public class ExpFunTest
{
    public static void main(String[] args)
    {
        System.out.println (ExportedFunctions.bell(2));
    }
}
```

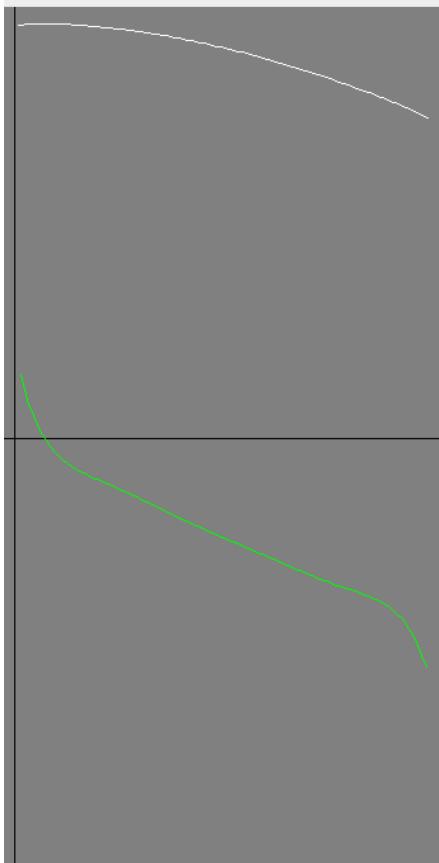


FloatDemoVanChe.txt

Comparison of LaGrange polynomial interpolation against Chebyshev T function interpolation

	<pre>// Bessel J0(x) over interval [0 < x < 1] x = (0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9) jx = (0.997501, 0.990024, 0.9776262, 0.960398, 0.9384, 0.912, 0.8812008, 0.84628, 0.8075237) // LaGrange interpolation produces x_LAGRANGE_jx = LAGRANGE(x,jx) !! If(x) = x_LAGRANGE_jx +*^ x</pre>
<pre>// Vandermonde matrix of Chebyshev polynomials vc = VANCHE x ; vcInv = INV vc; jm = MATRIX(jx,9,1) vcInv_MATMUL_jm = MATMUL(vcInv,jm) jxc = vcInv_MATMUL_jm#1 // producing the Chebyshev polynomial interpolation of jx // using the Clenshaw operator allows an // efficient function to be defined !! cf(x) = jxc @*^ x // look at the error function between LaGrange and Chebyshev !! P(x) = cf(x); !! f(x) = If(x) !! err(x) = abs (P(x) - f(x)) // evaluate the error function across the interval erf = [0 < x < 1 < 0.01] (err(x)) // query maximum value of error maxErf = MAX erf; PRETTYPRINT maxErf // show the plot GRAPH erf</pre>	<p>OUTPUT:</p> <p>Reading... C:\Projects\CalcLib\scripts\FloatDemoVanChe.txt</p> <p>Lagrange Interpolation</p> $Y = 0.9917303000000143 + 0.21729748928555637 * x - 2.4849073442426004 * x^2 + 11.988874097151438 * x^3 - 37.1869706594598 * x^4 + 69.12134722175688 * x^5 - 75.7669097214548 * x^6 + 45.13501984081631 * x^7 - 11.258184523720729 * x^8$ <p>$r^2 = 1.0$ STD = 3.4329155185658895E-11 COV = 3.717532401497196E-11 MSE = 1.178490895761051E-21 SSR = 0.03574957820309609 SSE = 8.249436270327357E-21 SST = 0.03574957822936 PC = -0.9780255354339753</p> <p>maxErf =</p> $1.9089778279246516E-6$
<pre>jxc = [-40.95140998426814 77.09301545187009 -60.27713885060075 39.40757141249924 -21.317397422270005 9.256728076843729 -3.071351473209347 0.7052343269186707 -0.08795456729584572]</pre>	<pre>x_LAGRANGE_jx = [0.9917303000000143 0.21729748928555637 -2.4849073442426004 11.988874097151438 -37.1869706594598 69.12134722175688 -75.7669097214548 45.13501984081631 -11.258184523720729]</pre>

[0.01 <= x <= 0.99 >> 0.01] (cf(x))



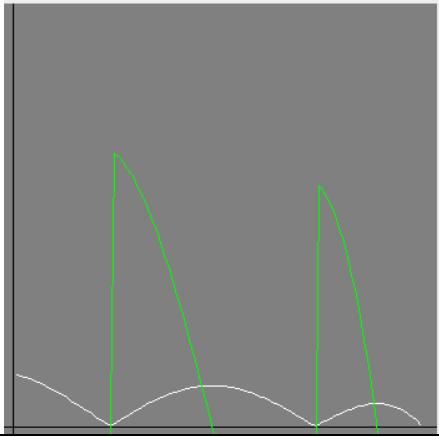
```
// Vandermonde matrix of Chebyshev polynomials  
vc = VANCHE x ; vcInv = INV vc; jm = MATRIX(jx,9,1)  
vcInv_MATMUL_jm = MATMUL(vcInv,jm)  
jxc = vcInv_MATMUL_jm|#1  
  
// producing the Chebyshev polynomial interpolation of jx  
// using the Clenshaw operator allows an efficient function to be defined  
!! cf(x) = jxc @*^ x
```

Functions

P(x)	cf(x)
cf(x)	jxc @*^ x
err(x)	abs(P(x) - f(x))
f(x)	lf(x)
lf(x)	x_LAGRANGE_jx +*^ x

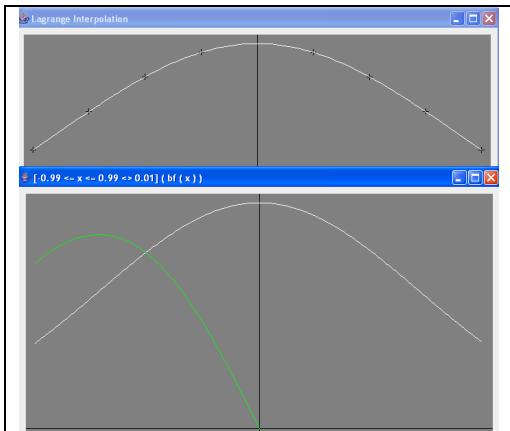
5 symbols found and displayed

[0.01 <= x <= 0.99 >> 0.01] (err(x))



```
// look at the error function between LaGrange and Chebyshev  
  
!! P(x) = cf(x); !! f(x) = lf(x)  
!! err(x) = abs( P(x) - f(x) )  
  
// evaluate the error function across the interval  
erf = [0 < x < 1 >> 0.01] (err(x))  
  
// query maximum value of error  
maxErf = MAX erf; PRETTYPRINT maxErf  
  
// show the plot  
GRAPH erf  
-----  
maxErf =  
1.9089778279246516E-6
```

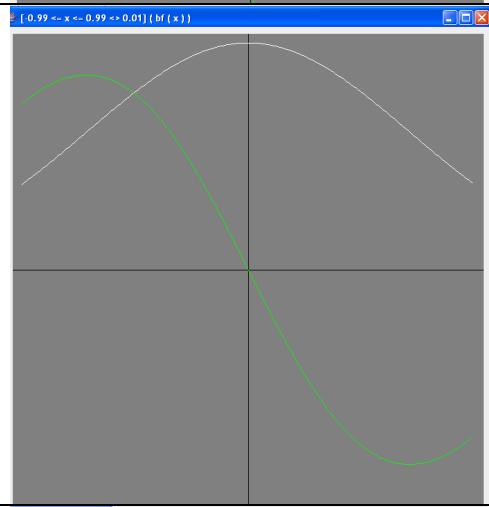
Chebyshev interpolation applied to Bell curve



```
xaxis = [-1 < x < 1 <> 0.2] (x)
bell = [-1 < x < 1 <> 0.2] ( exp ( - x ^ 2 ) )
graph bell
```

```
xaxis_LAGRANGE_bell = LAGRANGE (xaxis, bell)
```

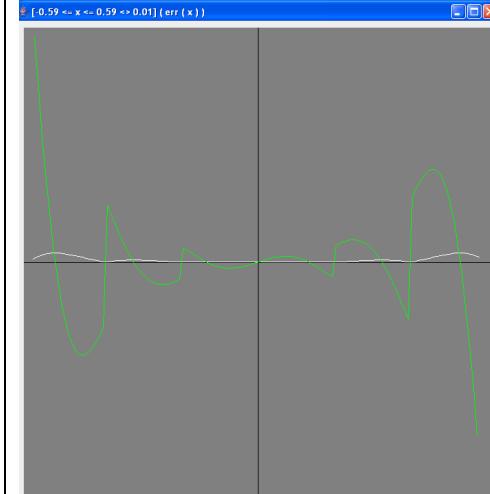
```
vc = VANCHE (xaxis)
vcInv = INV vc
```



```
bm = MATRIX (bell, 9, 1)
mm = MATMUL (vcInv, bm)
bcpc = mm#1
```

```
!! bf(x) = bcpc @*^ x
```

```
graph [-1 < x < 1 <> 0.01] ( bf (x) )
```



```
!! P(x) = bcpc @*^ x ; !! f(x) = exp (-x^2)
!! err(x) = abs ( P(x) - f(x) )
```

```
erf = [-1 < x < 1 <> 0.01] ( err (x) )
```

```
maxErf = MAX erf
```

```
PRETTYPRINT maxErf
```

```
maxErf =
```

```
1.5179754979666171E-6
```

```
GRAPH erf
```

The VANCHE Concept Extended

The Chebyshev polynomial are constrained to the interval [-1, 1] in both domain and range. Interpolations for functions that are difficult (expensive) to compute are done making function evaluation and calculus on these functions easier and less (time) expensive. For these purposes a quick tool for the interpolation process is useful. Given the Chebyshev polynomial constraints and the need to calculate points to build the interpolation, the process can be made a simple set of constants. Taking the constraint of $-1 \leq x \leq 1$ the function can be evaluated at equidistant points of 0.1 making a constant Vandermonde matrix of Chebyshev T[n] polynomials calculated at values of $[-1 \leq x \leq 1 \Delta 0.1]$:

$$X = [-1 \leq xi \leq 1 \Delta 0.1] (xi)$$

$$VC21 = VANCHE X$$

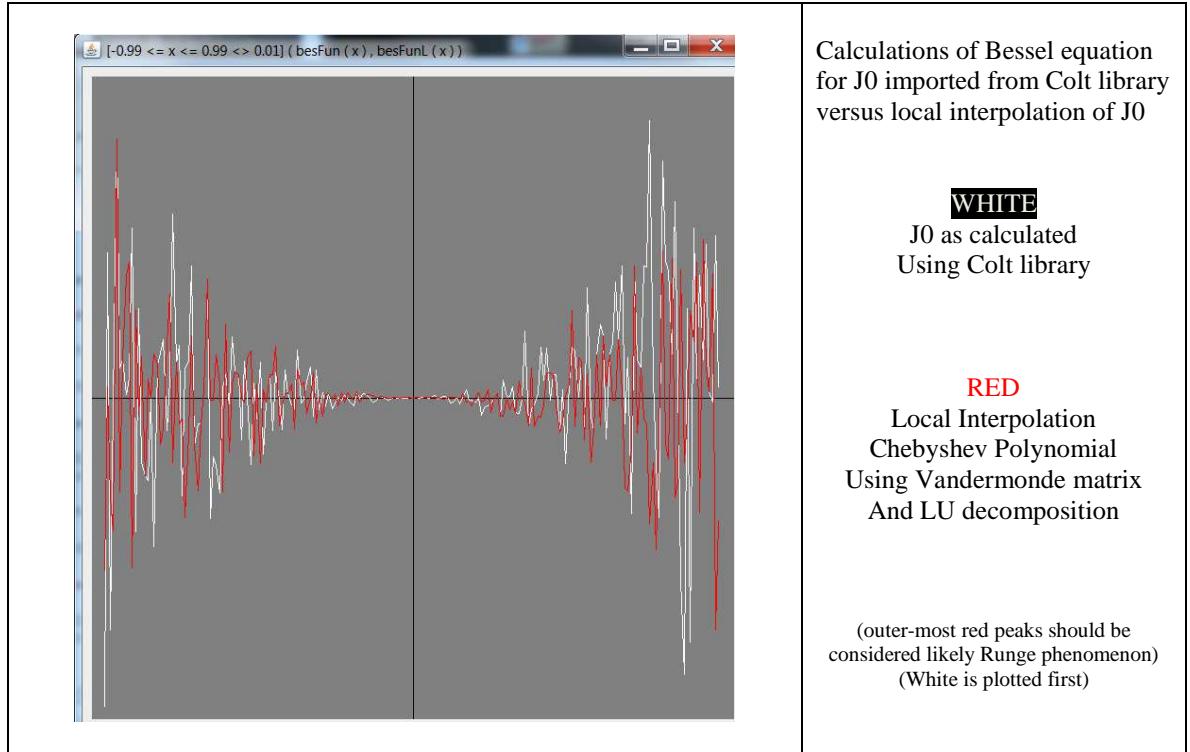
This gives the constant Vandermonde Chebyshev matrix for T[n] over $-1 \leq x \leq 1$ with delta 0.1
Using the Colt library to perform decomposition takes it to the next step:

```
READ  ColtLib.txt
CALC  MatRpt  VC21
VC21L = GetLudL()
VC21U = GetLudU()
```

VC21L and VC21U are constants that can be used with any sequence of 21 equally spaced function evaluation points to produce a Chebyshev polynomial interpolation; Coordinate translation for the [-1,1] interval must be done and the coefficient solution becomes a simple application of $LUx = b$ but the heavy lifting is embedded in the constants.

As a simple test of this concept, the Colt library J0 function was used to capture data on the interval [-1, 1] with 0.1 as increments. The VC21 constant matrix was used to interpolate the function as a Chebyshev polynomial. This function was introduced to the system as "j0lcl". The Bessel function was used to evaluate J0 and j0lcl since a correct J0 function would be constant 0 return from the Bessel function if J0 provides proper function values. The Bessel function was plotted over the [-1, 1] interval and the error values were collected. The MAX value of the collected errors was calculated. The script and the plotted and printed results are on the following page.

Extended Concept Test



```

READ BesselFunctions.txt
jVals = [-1<=x<=1<>0.1](J0(x))
mj = MATRIX(jVals,21,1)

j0coef = SolveWithLUD(VC21,mj)
j0c = j0coef#1

!! j0lcl(x) = j0c @*^ x

// // //

alpha=0
dx = 0.00001

!! besFun (x) = x^2 * J0"(x <> dx) + x * J0'(x <> dx) + (x^2 - alpha^2)* J0(x)
GRAPH [-1<x<1<>0.01](besFun(x)) ; err = [-1 < x < 1 <> 0.01] (abs(besFun(x)))
CALC MAX err ; // 2.455309003113193E-6

!! besFunL (x) = x^2 * j0lcl"(x <> dx) + x * j0lcl'(x <> dx) + (x^2 - alpha^2)* j0lcl(x)
GRAPH [-1<x<1<>0.01](besFunL(x)) ; errL = [-1 < x < 1 <> 0.01] (abs(besFunL(x)))
CALC MAX errL ; // 2.0533536184697E-6

GRAPH [-1<x<1<>0.01](besFun(x), besFunL(x)); // ( actual graph displayed above )

```

Max error of the interpolation is smaller than that of the original function, interesting little mystery

VC21 decomposed as L and U matrices

Upper Triangular Matrix

PRETTYPRINT VC21L 5

VC21L=

```

1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
1   0.5  1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
1   0.8   0.64  1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
1   0.15  0.51  -0.92969  1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
1   0.3   0.84  -0.875  0.72398  1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
1   0.95  0.19  0.44531  0.11054  0.81429  1   0   0   0   0   0   0   0   0   0   0   0   0   0
1   0.05  0.19  -0.44531  0.55268  -0.50893  0.24038  1   0   0   0   0   0   0   0   0   0   0   0
1   0.65  0.91  0.71094  -0.17647  -0.8125  -0.53728  -0.74503  1   0   0   0   0   0   0   0
1   0.9   0.36  0.75  0.12411  0.85714  0.97166  0.22456  -0.427  1   0   0   0   0   0   0
1   0.4   0.96  -0.5  0.33096  0.7619  0.14395  0.36595  -0.28653  -0.67102  1   0   0   0
1   0.1   0.36  -0.75  0.86878  -0.4  0.15115  0.59384  -0.06642  -0.34222  0.816  1   0   0
1   0.75  0.75  0.97656  -0.0808  -0.44643  -0.37955  -0.35086  0.54945  -0.51471  0.23011
1   0.2   0.64  -1   0.99289  0.45714  -0.08637  -0.29942  0.10047  0.42353  -0.88364
1   0.85  0.51  0.92969  0.07692  0.49583  0.51524  0.23815  -0.42621  0.79852  -0.119
1   0.55  0.99  0.25781  -0.10666  -0.39286  -0.18556  -0.34308  0.38374  0.35948
1   0.25  0.75  -0.97656  0.88882  0.81845  -0.07732  -0.25016  0.11194  0.41939
1   0.7   0.84  0.875  -0.1448  -0.73333  -0.55421  -0.64042  0.93122  -0.43617
1   0.35  0.91  -0.71094  0.52941  0.975  0.09211  0.25544  -0.17143  -0.48176
1   0.6   0.96  0.5  -0.16548  -0.68571  -0.38866  -0.62877  0.77363  0.36235
1   0.45  0.99  -0.25781  0.14932  0.4125  0.1169  0.27018  -0.24176  -0.45294

```

Pivot Table

VC31P=

```

[
  0, 30, 15, 24, 5, 10, 28, 2, 20, 29, 1, 12, 26, 7, 18, 3, 22,
  27, 4, 14, 25, 8, 17, 6, 23, 11, 21, 9, 19, 13, 16
]

```

Lower Triangular Matrix

PRETTYPRINT VC21U 5

VC2IU =

1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1
0	2	-0	2	-0	2	-0	2	-0	2	0	2	0	2	-0	2	-0	2	0	2	-0
0	0	-2	0	0	0	-2	0	0	-2	-0	-0	0	-2	0	0	-0	-2	0	-0	0
0	0	0	-1.536	-1.8432	-0.67584	1.03219	0.37847	-0.57803	-1.0721	-0.7085	-1.31409	-0.86841	0.272	1.19481	-0.37422	-1.64388	-1.59843	-0.27242	-0.26665	-0.04575
0	0	0	0	-3.7128	0.74256	1.03958	0.29702	-0.54058	-0.93741	-0.73852	0.25877	-2.80016	1.56234	1.27035	-0.45275	-1.54107	-1.36331	-0.414	1.28003	-0.2226
0	0	0	0	0	-1.6128	1.6128	0.77414	-2.10309	0.67351	1.15709	-1.91998	0.4913	0.31058	-0.0611	0.29013	-0.36767	-0.66541	0.3079	0.23934	0.05676
0	0	0	0	0	0	-3.41453	-2.73162	1.36358	-1.47508	-1.36581	1.46633	-0.05964	-0.53356	-0.24231	-0.03359	0.79701	0.68978	-0.96418	-1.77365	-1.72168
0	0	0	0	0	0	0	2.95488	-2.95488	2.24571	0.18911	-0.40416	1.06943	-0.46919	0.23684	0.25334	-0.66379	0.25177	-0.38632	1.27799	-0.37917
0	0	0	0	0	0	0	0	-5.28353	2.11341	2.5361	-0.9299	-0.5833	-1.3323	0.62287	1.28252	-0.30635	-0.27942	-0.31149	0.78329	-1.50367
0	0	0	0	0	0	0	0	0	2.25604	2.70729	0.99267	-0.72194	-1.89871	-1.74711	-1.16262	-0.66072	0.23627	1.15807	1.74965	1.3069
0	0	0	0	0	0	0	0	0	0	3.02776	2.42221	-2.66443	-2.32532	1.04639	-0.43794	-2.42151	0.52828	2.53907	0.01474	-0.19686
0	0	0	0	0	0	0	0	0	0	0	-2.96479	2.71383	1.77887	-1.94107	1.90221	0.32636	-0.04345	-0.76976	-1.21119	0.69323
0	0	0	0	0	0	0	0	0	0	0	0	0.25361	0.50722	-3.55053	-1.54195	0.38143	1.6442	1.29994	0.3142	0.05032
0	0	0	0	0	0	0	0	0	0	0	0	-1.88351	1.88351	2.26021	-3.45059	1.59722	0.31281	-0.93804	2.97948	
0	0	0	0	0	0	0	0	0	0	0	0	0	-0.20247	-0.80939	3.31848	1.74827	0.845	-0.22922	-0.32225	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1.64087	-0.98884	4.15314	3.24341	-2.81263	-3.99334	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1.02024	0.40809	3.18313	-1.5834	-3.34311	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.58766	0.23506	-2.23309	-1.10949	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-0.41136	0.08227	1.92516		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.16876	0.03375		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-0.07594		

VC31 Spline Organization

```

          start                                low                               knot                                high
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30
.   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
-1.5 .4 .3 .2 .1 -1 .9 .8 .7 .6 .5 .4 .3 .2 .1 0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1 .1 .2 .3 .4 .5
[ low end knot ] [ standard Chebyshev constraints specify interval of [-1, 1] for best behavior ]
[ not used ]
[Runge phenomenon]

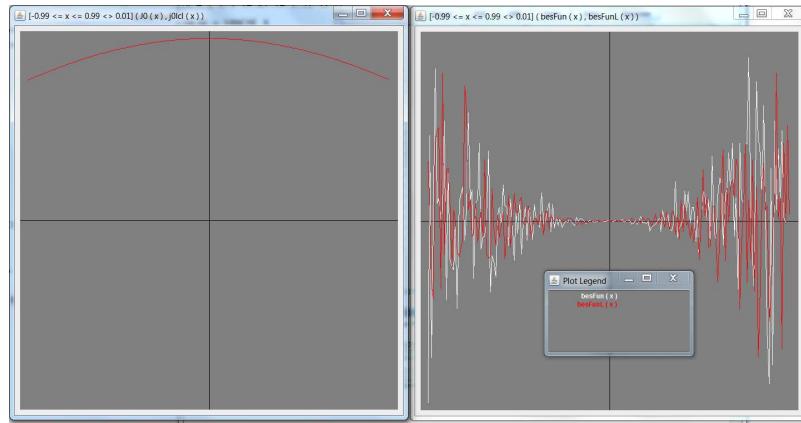
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
.   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
1.5 .6 .7 .8 .9 2 .1 .2 .3 .4 .5 .6 .7 .8 .9 3 .1 .2 .3 .4 .5
          low                               knot                                high

```

Note positions of knots and overlap points of segments around knots

In Search Of: The Perfect Spline

Restricted to the $[-1, 1]$ interval the knots of a spline made of these polynomials would force the useful portion of the interval to perhaps $\frac{1}{2}$ leaving the other $\frac{1}{2}$ for overlay in the knots. This opens the question of “what happens if we drift a bit past the $[-1, 1]$ interval to perhaps $[-1.5, 1.5]$?



These are plots of the same J0 experiment as before on the interval $[-1.5, 1.5]$ using 31 points of 0.1 increments building VC31, VC31U, and VC31L as was done with VC21. The left side plot is J0 in white plotted first and then j0lcl in red plotted after; note on left that red completely over writes the white plot indicating that the difference is smaller than the plot precision.

The error analysis documents:

```
errL = [-1.2 < x < 1.2 >> 0.01] (abs(besFunL(x)));  CALC MAX errL  
3.3008504037868036E-6  
  
err = [-1.2 < x < 1.2 >> 0.01] (abs(besFun(x)));  CALC MAX err  
4.9065843416018495E-6
```

Showing that the error evaluation as we step beyond $[-1, 1]$, while larger, is less than the increase in the original function. On this point, interesting to note:

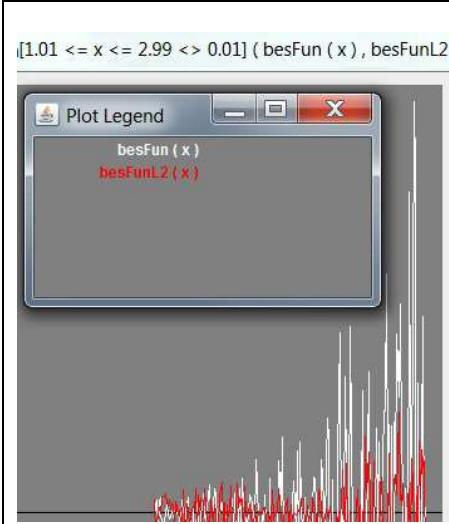
```
errL = [-1 < x < 1 >> 0.01] (abs(besFunL(x)));  CALC MAX errL  
1.994155862661273E-6  
  
err = [-1 < x < 1 >> 0.01] (abs(besFun(x)));  CALC MAX err  
2.455309003113193E-6
```

The gap between errors on interval $[-1, 1]$ widened using VC31, while the original J0 plot retains a fairly constant error, the interpolation error drops...

Conclusions:

This method may produce useful spline segments in 2 unit increments. Three units of equally spaced points should be sampled into 31 points. The VC31 triangular matrices can be used to solve the $LUX = b$ equation for the X coefficients of the Chebyshev polynomial. This polynomial should perform very well on the $[-1, 1]$ interval with the real X coordinates of the segment translated on to that interval. The lower 5 and upper 5 points of the interpolation are being discarded as being intended overlays of the knot between the segments

Building A Spline



The script for the second segment:

```
j0xx = [0.5<=xx<=3.6>>0.1](J0(xx))
mjxx = MATRIX(j0xx,31,1)

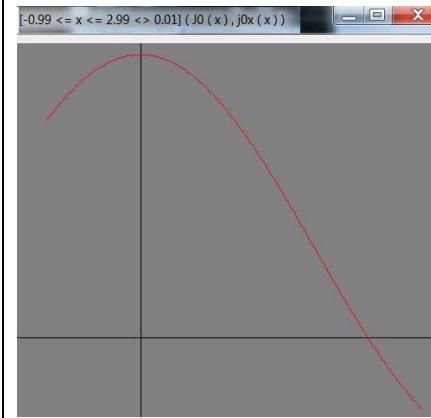
j0xxcoef = SolveWithLUD(VC31,mjxx)
j0xxc = j0xxcoef#1

!! j0xxlcl(x) = j0xxc @*^ x
!! j0lcl2(x) = j0xxlcl(x-2)

calc j0xxlcl(0); // 0.22389078190857192
calc j0lcl2(2); // 0.22389078190857192
calc J0(2); // 0.2238907819085722

!! besFunL2 ( x ) = x^2 * j0lcl2"(x >> dx) + x * j0lcl2'(x >> dx) + (x^2 - alpha^2)* j0lcl2(x)
graph [1<x<3>>0.01] (besFun ( x ), besFunL2 ( x ))
```

Looks like RED wins again

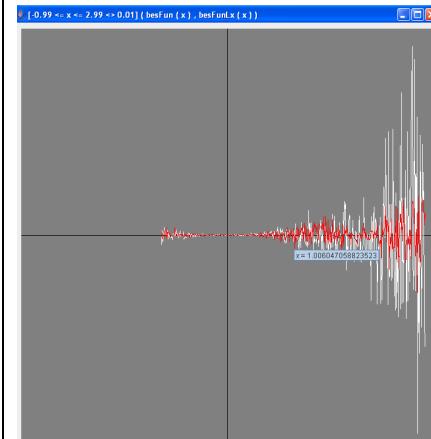


```
err = [1<x<3>>0.01](besFun(x))
errL = [1<x<3>>0.01](besFunL2(x))

calc MAX err; // 6.013422098871679E-5
calc MAX errL; // 1.464845459953068E-5 // so far, so good

// the function implementing the knot
!! j0x(x) = (x>1)? j0lcl2(x) : j0lcl(x)

// a graph of the overlay
graph [-1 < x < 3 >> 0.01] ( J0(x), j0x(x) )
```



```
// the Bessel equation for testing the spline
!! besFunLx ( x ) = x^2 * j0x"(x >> dx) + x * j0x'(x >> dx) + (x^2 - alpha^2)* j0x(x)

// a graph to compare the error as before
graph [-1 < x < 3 >> 0.01] ( besFun ( x ), besFunLx ( x ) )

// RED Chebyshev polynomial interpolation RULES !!!
// note the lack of error spike at the knot (x = 1)

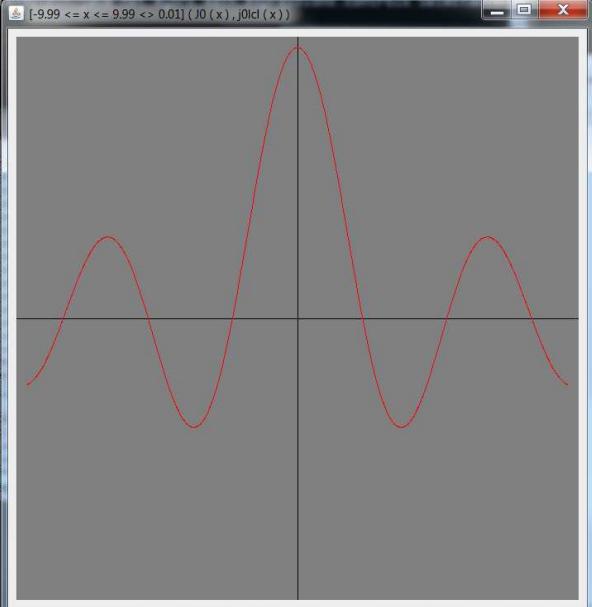
err = [-1 < x < 3 >> 0.01](abs (besFun ( x )))
errL = [-1 < x < 3 >> 0.01](abs (besFunLx ( x )))

calc MAX err; // 5.182550320093071E-5
calc MAX errL; // 1.4990911813006491E-5
```

One Additional Feature (an obvious notion)

The CalcLib tools offer function definition commands. The ones that export VC31 spline functions publish function definitions from matrix objects requesting a name for the function and a low knot value.

Optionally a type of function can be selected (Odd, Even) which forces the assumption that the low knot value is 0 and the input parameter is absolute function (abs(x))

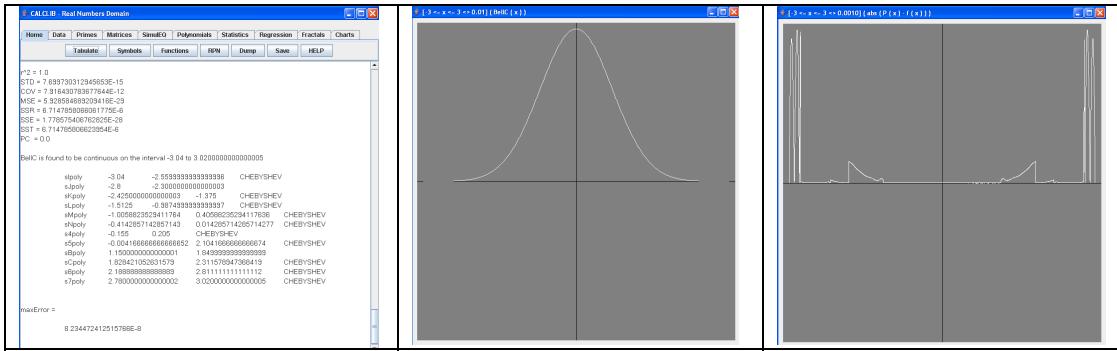
	<p>READ VC31Prep.txt</p> <p>READ BesselFunctions.txt</p> <pre>// the knot is assumed to be 0 // the -0.5 knot overlap must be present jVals = [-0.5<=x<=15<>0.1](J0(x)) // the interpolation remains the same j0co = VC31 jVals // function exported as EVEN type // the definition shows the abs(x) & 0 knot !! j0lcl(x) = EVALSPLINE (j0co, 0, abs(x)) // the graph shows the y-axis mirror graph [-10 < x < 10 <> 0.01] (J0(x), j0lcl(x))</pre>
--	---

Alternatively:

Odd and Even functions can use the GENKNOT function to produce the knot value to be pre-pended to a data sequence. The VC31 interpolation is based on data sequences having a 0.1 increment between x-axis points, but scaling can be used.

<pre>// function evaluation that starts at 0 // the GENKNOT function reflects low 5 values jVals = [0.0 <= x <= 12 <> 0.1] (J0(x)) knot = GENKNOT jVals // append the knot values to the front of the sequence // the interpolation now works just as it did in the previous example jv = APPEND (knot, jVals); j0co = VC31 jv !! j0lcl (x) = EVALSPLINE (j0co, 0, abs(x)) graph [-10 < x < 10 <> 0.01] (J0(x), j0lcl(x))</pre>	<pre>// x-axis scaling allows more of the curve per segment // increment of 0.2 requires ½ as many samples jVals2 = [0.0 <= x <= 12 <> 0.2] (J0(x)) // combined sub-expressions reduce the spline generation to: j0co2 = VC31 (APPEND (GENKNOT jVals2, jVals2)) // evaluation done on x/2 scales the function back !! j0x(x) = EVALSPLINE (j0co2, 0, abs(x/2)) graph [-10 < x < 10 <> 0.1] (J0(x), j0x(x))</pre>
---	--

Mix and Match using High-Order Polynomials



Interpolation

Segments are chosen and the interpolation produces a polynomial for the segment. Refinements are made reducing the maximum error evaluation. The final max-error on this attempt was 8.2E-08 with the error spikes remaining at the extreme ends of the x-axis. The middle of the range show errors in the 1E-11 up to 1E-09

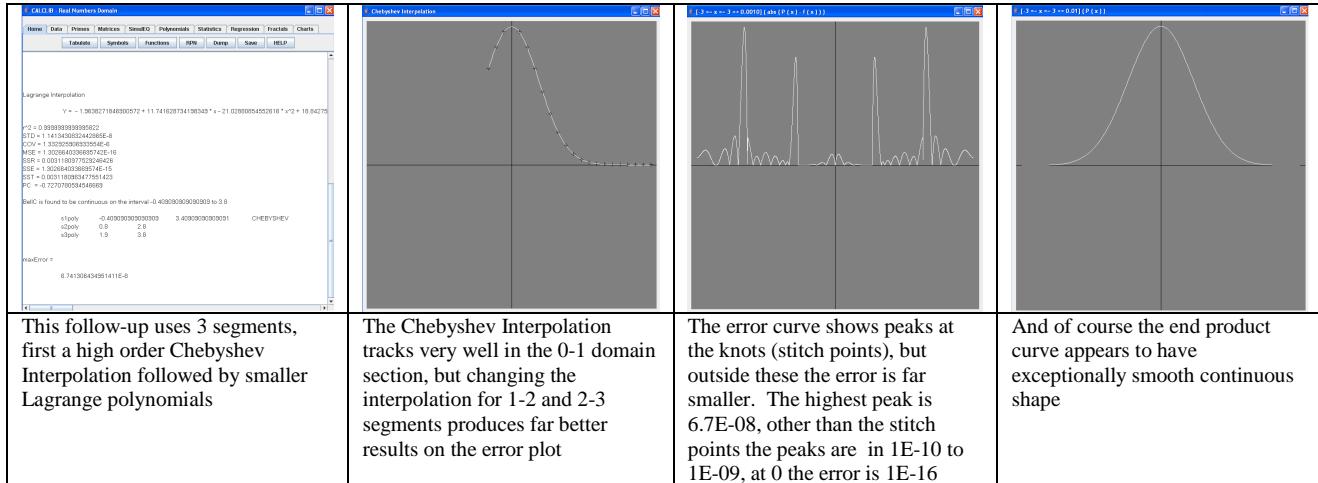
The Bell Curve

The plot of the produced function is very smooth and appears the well known shape of the Bell curve. No obvious errors are seen and even the ends appear to approach zero but remain just far enough above that the x-axis can still be seen under the plot of the curve

The Error Plot

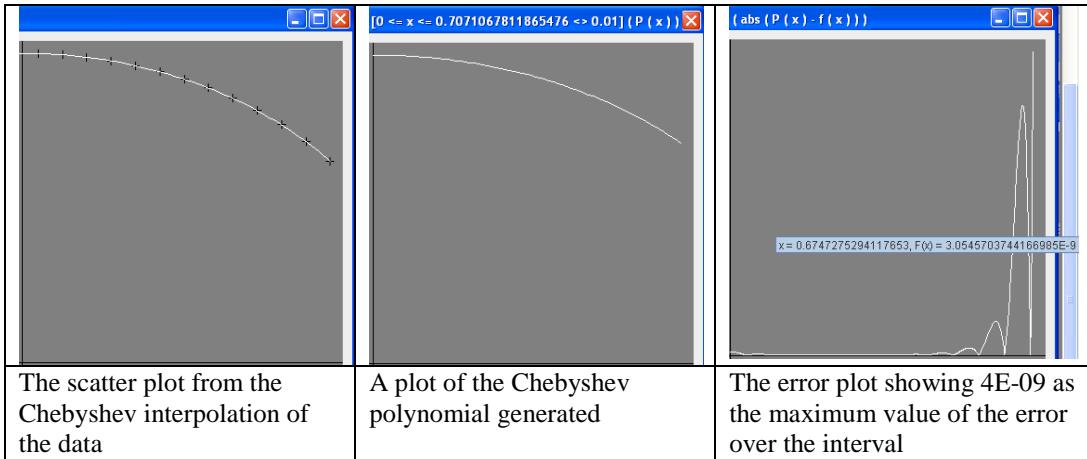
In the range -1.5 to 1.5 on the x-axis no error registers at all. The peak at 1.81 show an error of 1E-08 which is 1/8 the size of the highest peak. The end of curve error peaks start at 2.7 and reach values of 8.2E-08. The value of the function is 6.8E-04, so even at worst the error is 1 part in 8000 while at best we get 1 in 100,000,000

The building blocks for this test can be seen in scripts/FloatDemoRungeSegmented.txt where a test is built using Chebyshev polynomial interpolation. This produced results with errors in the 1E-05 range. The errors anticipated by Carl Runge were being seen as well as problems for $|x| > 1$. An additional series of tests was built found in scripts/FloatDemoGRungeSegmented.txt where other types of interpolation where used to generate functions with smaller errors in the areas found to be problematic. This result documented here shows errors in the 1E-11 to 8E-08 range. Another set of tests is attempting to get the error down to the 1E-15 optimal range.



ChebCircle8th.txt

Clebsch Quadrature is based on Chebyshev Polynomial Interpolation, so with a successful interpolation in place a numerical integration can easily be done using the generated polynomial. The function used here is $\sqrt{1-x^2}$ plotted from 0 to $\sqrt{2}/2$ which is 1/8 of the circle. To make it a slice there is a shared piece of the area that calculates as 0.25, this is called excess in the variables in the script



```

// equation for a unit circle
!! f(x) = 2\ (1 - x^2)

// 45DEG is 1/8 of circle, sin=cos=sqrt(2)/2
r2o2 = 2\ / 2

// area of the shared portion of the slice 0 - pi/4
excessArea = r2o2 ^ 2 / 2

// map out the slice to be interpolated
c00 = [-0.20 <= x <= 0.8 >> 0.06] ( f (x) )
c00poly = CHEBINTERP c00

// the interpolated Chebyshev polynomial
!! P(x) = c00poly @*^ x

// the error calculated at 1000 points
err = [0 <= x <= r2o2 >> 0.001] (abs (P(x) - f(x)))
maxError = MAX err

// maximum error found
PRETTYPRINT maxError

```

```

// use Clebsch quadrature to formulate
// the anti-derivative and calculate the area

area = CLENQUAD (c00poly, r2o2) ;
area0 = CLENQUAD (c00poly, 0)

calculatedArea = area - (area0 + excessArea)

PRETTYPRINT calculatedArea

// this should be pi / 8 since area of full unit circle = pi r^2 = pi
approximationOfPi = 8 * calculatedArea
PRETTYPRINT approximationOfPi

// plots of interpolation and error curve
GRAPH [0 <= x <= r2o2 >> 0.01] (P(x))
GRAPH err

```

Symbol	Notation
approximationOfPi	3.1415926542866544
area	4.453178685699542
area0	3.81047960391371
c00	(0.9797958971132712, 0.990)
c00poly	(-8.512669289364831, 17.69)
calculatedArea	0.3926990817858318
err	(4.5718095975644246E-11, 4)
excessArea	0.25000000000000000000000000000006
maxError	4.073859471986907E-9
r2o2	0.7071067811865476

Note the approximation of PI is correct to 9 decimal places with maxError printing at 4E-09

The integral was actually computed using the anti-derivative of the Chebyshev polynomial, but of course the polynomial was an interpolation of data (with a nice small max error)

Visualization of the realized solution

<p>Unit circle \Rightarrow radius = 1</p>	
<p>The integral $0 - \cos(\pi/4)$ of the full circle equation gives the area of the slice but includes the full area of the square. Subtracting half the square area leaves just the slice</p>	<p>An alternative is to break the area down and compute areas of sections using simple geometry and this leaves a smaller integral</p> $\text{CrustLineSlope} = -(1 - 2\sqrt{2}/2) / (2\sqrt{2}/2) = 1 - \sqrt{2}$
	<pre>// equation for a unit circle !! f(x) = sqrt(1 - x^2) // 45DEG is 1/8 of circle, sin=cos=sqrt(2)/2 r2 = sqrt(2); r2o2 = r2 / 2 crustLineSlope = 1 - r2 !! crustLine(x) = 1 + crustLineSlope * x !! crust(x) = f(x) - crustLine(x) c00 = [-0.20 <= x <= 0.8 <> 0.06] (crust(x)) c00poly = CHEBINTERP c00 !! P(x) = c00poly @*^ x areaCrust = CLENQUAD (c00poly, 0, r2o2)</pre>
<pre>PRETTYPRINT areaCrust areaCrust = 0.03914569119254674 CombinedAreaOfTriangles = r2 / 4 sliceArea = areaCrust + CombinedAreaOfTriangles</pre>	<pre>// LowTriArea = (r2 / 2) ^ 2 / 2 // HiTriArea = [(1 - r2) * r2 / 2] / 2 // CombinedAreaOfTriangles = (r2 - 1) / 4 + 1 / 4 = r2 / 4</pre>
<pre>PRETTYPRINT sliceArea sliceArea = 0.3926990817858205</pre>	<pre>piApprox = sliceArea*8 piApprox = 3.14159265 // So the approximations are nearly identical</pre>
<p>Compare the interpolated area approximation</p>	<p>To use of Tanh-Sinh quadrature built-in as \$ function suffix</p>
<pre>// try built-in TanhSinh quadrature tsq = f \$ (0, r2o2, 1E-8) // lo, hi, dx // this produces 5 additional digits of precision calc 8 * (tsq - excessArea) 3.1415926535897</pre>	<pre>// using TanhSinh on the crust equation crustTsq = crust \$ (0, r2o2, 1E-8) // lo, hi, dx // this produces 2 more digits than on the left calc 8 * (crustTsq + 2\sqrt{2}/4) 3.141592653589793</pre>

FloatDemoChebyshevSegmented.txt

This script steps through the use of the tool to generate a spline

```
SCRIPTPRINT FloatDemoChebyshevSegmented.txt
```

```
// constants for approximation purposes  
INFINITY = 1000; dt = 1 / INFINITY
```

This is this capture of points of the target function over the domain of interest

```
// create an array function describing the curve  
curve = [-3 <= t <= 3.5 >> dt] ( exp ( -t^2 ) )
```

The captured points are broken into segments which will be separately interpolated and the resulting equations will be joined with spline knots

```
// select segment domains and collect samples
```

```
seg1 = [-0.2 <= t <= 1.2 >> 0.2] ( curve @# t )  
seg2 = [0.8 <= t <= 2.4 >> 0.2] ( curve @# t )  
seg3 = [1.8 <= t <= 3.4 >> 0.2] ( curve @# t )
```

```
s1poly = CHEBINTERP seg1  
s2poly = CHEBINTERP seg2  
s3poly = CHEBINTERP seg3
```

The segments are joined with knots using the !\$ prefix

```
!$ BellC(x) = (s1poly, s2poly, s3poly)
```

```
GRAPH [0 <= x <= 3 >> 0.001] (BellC (x))
```

Compare the spline calculated points with the original values

```
// evaluation of the error curve
```

```
!! P(x) = BellC(x)  
!! f(x) = curve @# x
```

```
err = [0 <= x <= 3 >> 0.001] (abs (P(x) - f(x)))
```

Evaluate the resulting errors

```
maxError = MAX err  
PRETTYPRINT maxError  
GRAPH err
```

Generate code for the spline (written to the 'exports' sub-directory)

```
encode BellC
```

The binary distribution directory structure contains 'a sample' package with compiled sources

```
LIBRARY XPF a.sample.library.ExportedFunctions
```

```
!+ BELLCX(x) = XPF.bellc
```

```
GRAPH [0 <= x <= 3 >> 0.001] ( BELLCX(x) )
```

This is the output of the script run

The interpolation of each segment produces the collection of statistics for the quality

Chebyshev Interpolation

```
Y = 0.63113 * T[0](x) + 0.02631 * T[1](x) - 0.33504 * T[2](x) + 0.0166 * T[3](x) +
0.02787 * T[4](x) + 0.00603 * T[5](x) - 0.00596 * T[6](x) + 0.00095 * T[7](x)
```

```
r^2 = 1.0
STD = 1.1830084923285172E-16
COV = 1.6889569508434738E-16
MSE = 1.3995090929213912E-32
SSE = 8.397054557528348E-32
SSR = 0.6037417381563968
SST = 0.6037417381563965
```

Chebyshev Interpolation

```
Y = 3.04342 * T[0](x) - 4.23353 * T[1](x) + 2.58546 * T[2](x) - 1.51837 * T[3](x) +
0.63124 * T[4](x) - 0.16363 * T[5](x) + 0.02537 * T[6](x) - 0.00217 * T[7](x) + 0.00008 * T[8](x)
```

```
r^2 = 1.0
STD = 2.5228156550928536E-12
COV = 1.600319504926352E-11
MSE = 6.364598829581584E-24
SSE = 4.4552191807071987E-23
SSR = 0.2736000978145655
SST = 0.2736000978178316
```

Chebyshev Interpolation

```
Y = 1.70603 * T[0](x) - 1.06794 * T[1](x) - 0.57618 * T[2](x) + 0.44567 * T[3](x) -
0.12161 * T[4](x) + 0.01706 * T[5](x) - 0.00124 * T[6](x) + 0.00004 * T[7](x)
```

```
r^2 = 1.0
STD = 1.289824592900256E-14
COV = 1.4688452379395254E-12
MSE = 1.663647480450311E-28
SSE = 9.981884882701866E-28
SSR = 0.0013263617043913348
SST = 0.0013263617043919478
```

A table of the segment descriptions is formatted to output

BellC is found to be continuous on the interval -2.7755575615628914E-17 to 3.0000000000000001

s1poly	-2.7755575615628914E-17	1.0	CHEBYSHEV
s2poly	1.0	2.1999999999999997	CHEBYSHEV
s3poly	2.0	3.000000000000001	CHEBYSHEV

The error output is also displayed

```
maxError =
5.985661350960125E-6
```

A sample program that is able to access the generated function

```
package net.myorb.testing.splines;

import net.myorb.math.MultiDimensional;
import net.myorb.math.expressions.symbols.SplineInRealDomain;
import java.util.ArrayList;

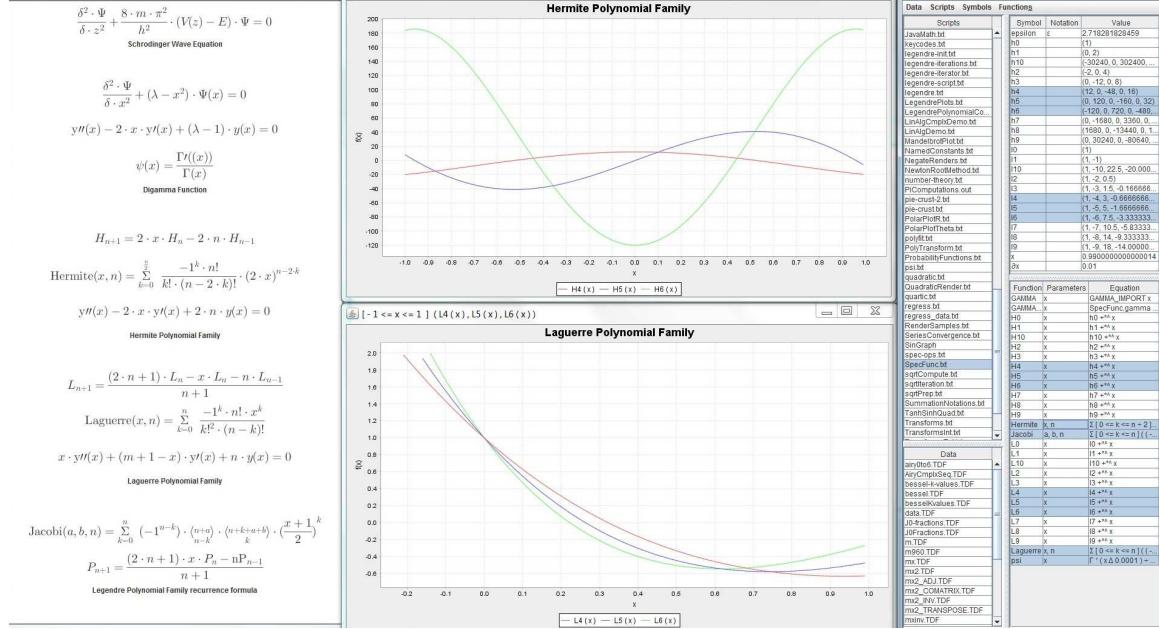
public class BellC extends SplineInRealDomain
{
    public void initialize ()
    {
        setName ("BellC");
        setHiConstraint (3.00000000000001);
        setSegmentLoConstraints (-2.7755575615628914E-17, 1.0, 2.0);
        addSegmentChebyshevPolynomial (...);
        addSegmentChebyshevPolynomial (...); // values removed from display
        addSegmentChebyshevPolynomial (...);
    }

    public static void main (String[] args)
    {
        ArrayList<Double> a = new ArrayList<> ();
        MultiDimensional.Function<Double> bell = (new BellC ()).getFunction ();
        for (double x = 0; x <= 2.5; x += 0.01) a.add (bell.f(x));
        System.out.println (a);
    }
}

// to run the test program we can capture the output and plot the points
a = (1.0, 0.9999005979257615, 0.9996011955333208, 0.9991019681153207, 0.9984032126069633, 0.9975053471396506, ...
graph [0 < i < 200] (a#i)
```

Special Functions for Engineering Applications

Polynomial Families of functions can be introduced in configuration. The FAMILY command can then be used to post the polynomial coefficient arrays and the functions into the symbol table



The SpecFunc script demonstrates this functionality

```
// Schrodinger
render delta^2 * PSI / (delta*x^2) + 8*m*pi^2/h^2 * (V(z) - E) * PSI = 0
RENDER "Schrodinger" Wave Equation" TOP

render delta^2 * PSI / (delta*x^2) + (lambda - x^2) * PSI(x) = 0
render y''(x) - 2*x*y'(x) + (lambda-1)*y(x) = 0

LIBRARY SpecFunc net.myorb.math.specialfunctions.Library

// GAMMA spline
!+ GAMMA_IMPORT(x) = SpecFunc.gamma
!! GAMMA(x) = GAMMA_IMPORT x
!! psi(x) = GAMMA'(x < 0.0001)/GAMMA(x)
RENDERF psi
RENDER "Digamma Function" TOP

// Hermite Polynomials
!! Hermite(x, n) = SIGMA [ 0 <= k <= n ] ( (-1)^k * n!
( k! * (n - 2k)! ) * (2*x)^(n - 2k) )
RENDEF#(n+1) = 2*x*H#n - 2*n*H#(n-1)
RENDERF Hermite
render y''(x) - 2*x*y'(x) + 2*n*y(x) = 0
FAMILY Hermite_10
GRAPH [ -1 <= x <= 1 < 0.01 ] ( H4(x), H5(x), H6(x) )
RENDER "Hermite Polynomial Family" TOP

// Laguerre Polynomials
!! Laguerre(x,n) = SIGMA [ 0 <= k <= n ] ( (-1)^k * n!
( (n+a)##(n-k)) * ((n+k+a+b)##k) * ((x+1)/2)^k )
RENDEF Laguerre
RENDER Laguerre_10

// Jacobi Polynomials
!! Jacobi(a,b,n) = SIGMA [ 0 <= k <= n ] ( (-1)^k * ((n+a)##(n-k)) * ((n+k+a+b)##k) * ((x+1)/2)^k )
RENDEF Jacobi
RENDER Jacobi_10

// Legendre Polynomials
render P#(n+1) = ( (2*n+1)*x*P#n - n*P#(n-1) ) / (n+1)
RENDER "Legendre Polynomial Family recurrence formula" TOP
```

The interface for a class that introduces a polynomial family is net.myorb.math.PolynomialFamily

```
1 package net.myorb.math;
2
3
4 import java.util.List;
5
6 /**
7  * properties of a family of polynomials
8  * @param <T> the data type to operate on
9  * @author Michael Druckman
10 */
11 public interface PolynomialFamily<T>
12 {
13
14     /**
15      * @return the name of the family
16     */
17     String getName ();
18
19     /**
20      * @param identifier the identifier for the kind of functions
21      * @param upTo the highest order requested
22      * @return the list of power functions
23      */
24     List<Polynomial.PowerFunction<T>> getFunctions (String identifier, int upTo);
25
26     /**
27      * @param kind a name for the kind (typically first & second, null if only one)
28      * @return the letter which typically identifies the functions
29      */
30     String getIdentifier (String kind);
31
32     /**
33      * @param manager data type manager
34      */
35     void init (SpaceManager<T> manager);
36
37 }
```

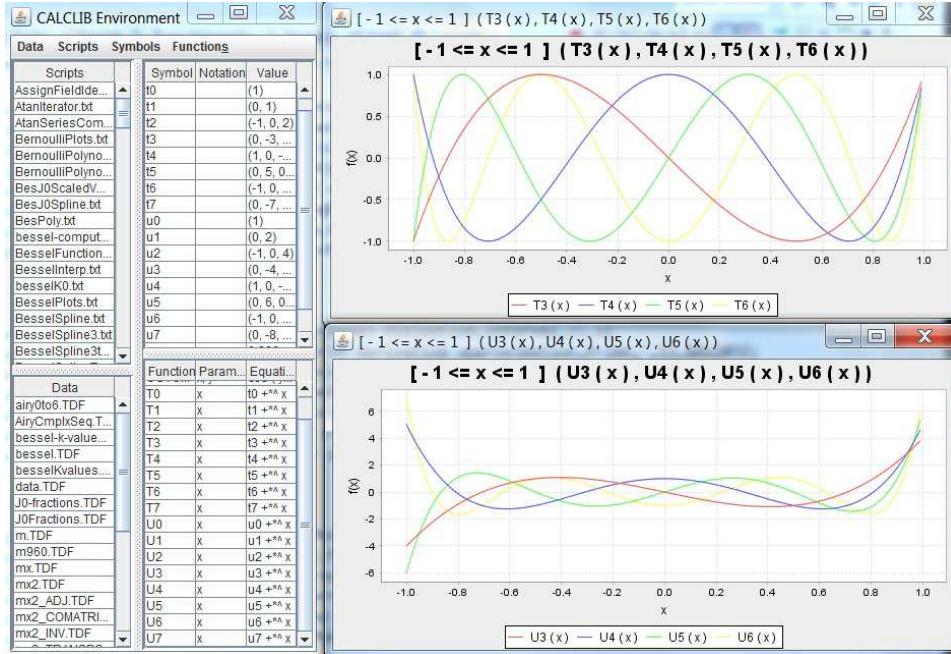
Typically such a class would contain a recurrence formula coded to produce the function of order 0..n which would be called by the getFunctions method passing parameter upTo as the value for n.

The Chebyshev Polynomial class implements the `PolynomialFamily` interface. This allows the recurrence formula to be invoked from the `FAMILY` command and a number of order of the polynomials can be imported into the execution environment. Note that First kind ("T") and Second kind ("U") polynomials can be individually or both imported by use of the `KIND` parameter on the `FAMILY` command. Note also that the polynomial coefficient arrays (`t0..t7`, `u0..u7`) and the polynomial functions (`T0..T7`, `U0..U7`) appear in the Environment table

```
// ChebyshevFamily.txt

FAMILY Chebyshev 7 First
GRAPH [-1 <= x <= 1 <> 0.01] ( T3(x), T4(x), T5(x), T6(x) )

FAMILY Chebyshev 7 Second
GRAPH [-1 <= x <= 1 <> 0.01] ( U3(x), U4(x), U5(x), U6(x) )
```



Spline for Gamma Function

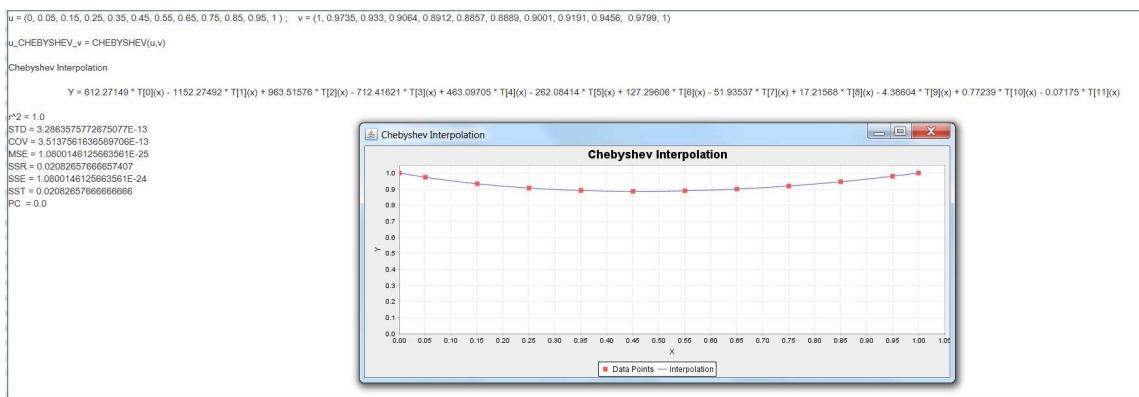
CalcLib was used to build a spline for the Gamma Function. The core interval is the domain 1 to 2. Special treatment is given to $x=1$, $x=1.5$, and $x=2$. Other parts of the domain are calculated by the generating functions

$$\text{GAMMA}(x+1) = x * \text{GAMMA}(x)$$

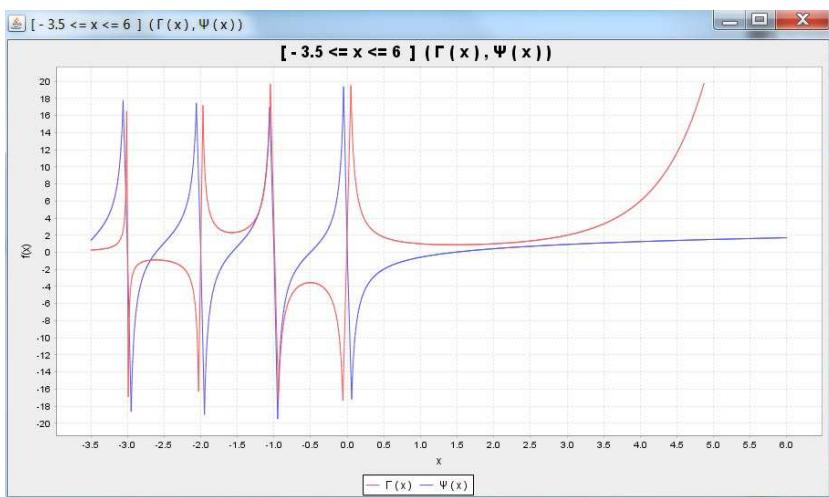
Or

$$\text{GAMMA}(x) = \text{GAMMA}(x+1) / x$$

Chebyshev interpolation provided the coefficients for the polynomial



```
LIBRARY SpecFunc net.myorb.math.specialfunctions.Library
!+ GAMMA(x) = SpecFunc.gamma
!! PSI(x) = GAMMA'(x <> 0.0001)/GAMMA(x)
GRAPH [-3.5 <= x <= 6 <> 0.01] (GAMMA(x), PSI(x))
```

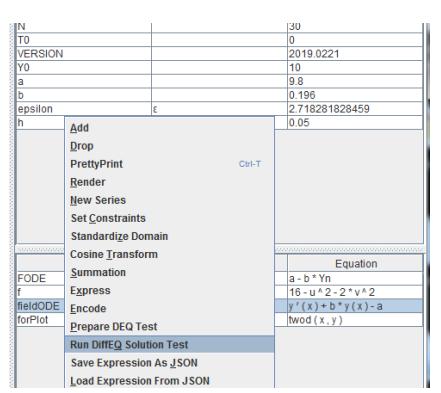
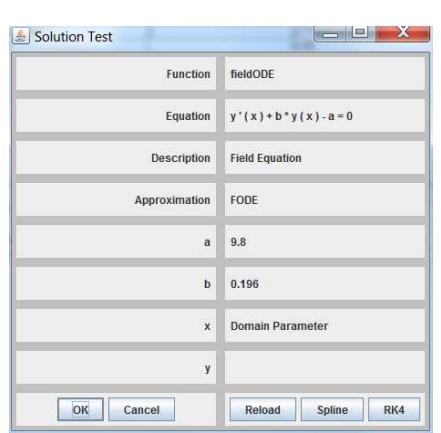
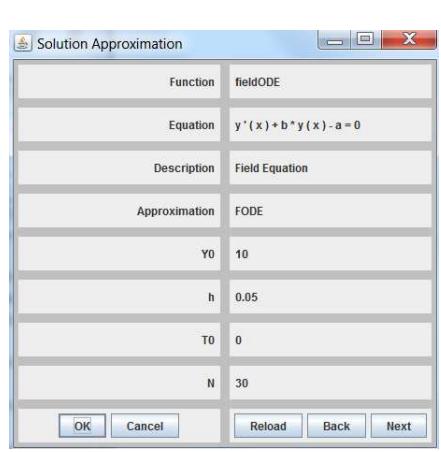


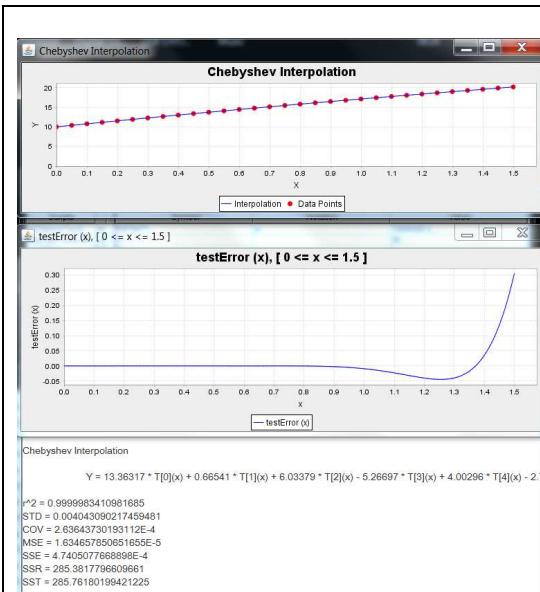
```

44✉ /**
45  * establish Chebyshev polynomial coefficients
46  * collected by regression over domain 1:2
47  * spline will use Clenshaw to compute results
48  * requested in this domain
49  */
50✉ public Gamma ()
51 {
52     super (new DoubleFloatingFieldManager ());
53     coefficients = (new GeneratingFunctions<Double>(manager)).newCoefficients
54     (
55         612.2714891668269,
56         -1152.274915868479,
57         963.5157625310783,
58         -712.4162057423769,
59         463.09704516096866,
60         -262.0841412593909,
61         127.2960555329889,
62         -51.93536578366297,
63         17.215678004111524,
64         -4.386044338750759,
65         0.7723942678398508,
66         -0.07175167115412054
67     );
68 }
69 GeneratingFunctions.Coefficients<Double> coefficients;
70
71 /**
72  * use Clenshaw generating function
73  * to evaluate Chebyshev spline for domain [1-2]
74  * @param x value of parameter within domain of [1-2]
75  * @return GAMMA(x) as calculated by spline
76  */
77 public Double eval12 (Double x)
78 {
79     if (x == 1.5)
80         return GAMMA_HALF / 2.0;
81     Value<Double> v = forValue (x - 1); // spline has shifted x-axis
82     Value<Double> result = evaluatePolynomialV (coefficients, v); // Chebyshev requires [0-1] domain
83     return result.getUnderlying ();
84 }
85
86
87 /**
88  * recursive evaluation for x > 1.5
89  * @param x value of function parameter
90  * @return value of function at parameter
91  */
92 public Double gammaGt (Double x)
93 {
94     if (x == 2.0) return 1.0;
95     else if (x > 2.0) return (x - 1) * gammaGt (x - 1);
96     else return eval12 (x);
97 }
98
99
100 /**
101  * recursive evaluation for x < 1.5
102  * @param x value of function parameter
103  * @return value of function at parameter
104  */
105 public Double gammalT (Double x)
106 {
107     if (x == 0) asymptotic ();
108     else if (x == 0.5) return GAMMA_HALF;
109     else if (x < 1) return gammalT (x + 1) / x;
110     else if (x == 1) return 1.0;
111     return eval12 (x);
112 }
113 void asymptotic () { throw new RuntimeException ("Gamma asymptotic for Integer <= 0"); }
114
115
116 /**
117  * (non-Javadoc)
118  * @see net.myorb.math.Function#eval(java.lang.Object)
119  */
120 public Double eval (Double x)
121 {
122     if (x == 1.5) return GAMMA_HALF / 2.0;
123     else if (x > 1.5) return gammaGt (x);
124     else return gammalT (x);
125 }
126

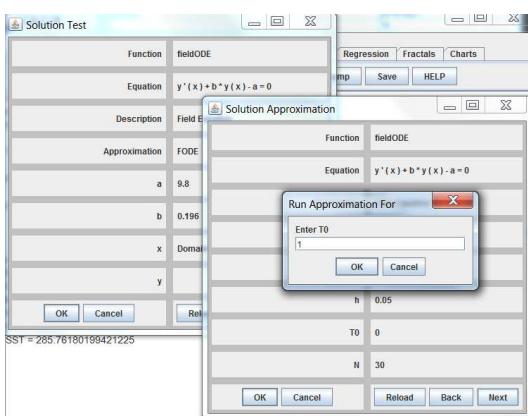
```

Differential Equations

 <p>N T0 VERSION Y0 a b epsilon h</p> <p>Add Drop PrettyPrint Render New Series Set Constraints Standardize Domain Cosine Transform Summation Express Encode Prepare DEQ Test Run DiffEQ Solution Test Save Expression As JSON Load Expression From JSON</p>	<p>Select an equation</p> <p>“Run DiffEQ Solution Test” From the Function drop-down menu</p>																						
 <table border="1"><tr><td>Function</td><td>fieldODE</td></tr><tr><td>Equation</td><td>$y'(x) + b * y(x) - a = 0$</td></tr><tr><td>Description</td><td>Field Equation</td></tr><tr><td>Approximation</td><td>FODE</td></tr><tr><td>a</td><td>9.8</td></tr><tr><td>b</td><td>0.196</td></tr><tr><td>x</td><td>Domain Parameter</td></tr><tr><td>y</td><td></td></tr><tr><td>OK</td><td>Cancel</td></tr><tr><td>Reload</td><td>Spline</td></tr><tr><td>RK4</td><td></td></tr></table>	Function	fieldODE	Equation	$y'(x) + b * y(x) - a = 0$	Description	Field Equation	Approximation	FODE	a	9.8	b	0.196	x	Domain Parameter	y		OK	Cancel	Reload	Spline	RK4		<p>The “Solution Test” tool is shown</p> <p>The information about the equation is shown</p> <p>The required parameters are shown with their values</p> <p>The RK4 button starts the Runge-Kutta tool</p>
Function	fieldODE																						
Equation	$y'(x) + b * y(x) - a = 0$																						
Description	Field Equation																						
Approximation	FODE																						
a	9.8																						
b	0.196																						
x	Domain Parameter																						
y																							
OK	Cancel																						
Reload	Spline																						
RK4																							
 <table border="1"><tr><td>Function</td><td>fieldODE</td></tr><tr><td>Equation</td><td>$y'(x) + b * y(x) - a = 0$</td></tr><tr><td>Description</td><td>Field Equation</td></tr><tr><td>Approximation</td><td>FODE</td></tr><tr><td>Y0</td><td>10</td></tr><tr><td>h</td><td>0.05</td></tr><tr><td>T0</td><td>0</td></tr><tr><td>N</td><td>30</td></tr><tr><td>OK</td><td>Cancel</td></tr><tr><td>Reload</td><td>Back</td></tr><tr><td>Next</td><td></td></tr></table>	Function	fieldODE	Equation	$y'(x) + b * y(x) - a = 0$	Description	Field Equation	Approximation	FODE	Y0	10	h	0.05	T0	0	N	30	OK	Cancel	Reload	Back	Next		<p>The “Solution Approximation” screen shows the set values of the RK4 formula parameters</p> <p>Press the OK button to run the T0 approximation</p>
Function	fieldODE																						
Equation	$y'(x) + b * y(x) - a = 0$																						
Description	Field Equation																						
Approximation	FODE																						
Y0	10																						
h	0.05																						
T0	0																						
N	30																						
OK	Cancel																						
Reload	Back																						
Next																							



The Interpolation displays show the quality of the best fit attempt made for the approximation



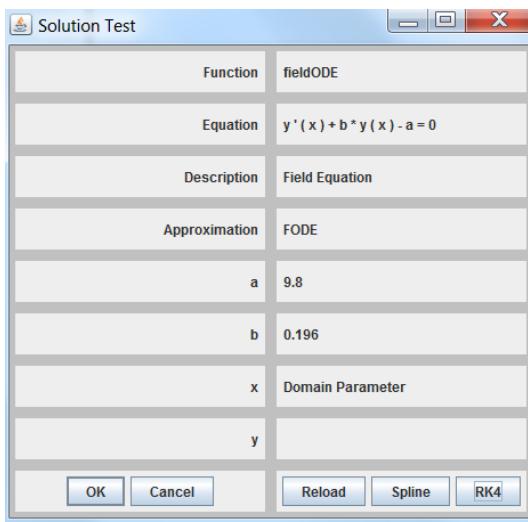
In addition to the interpolation displays a “testError” plot is shown

From the error plot the knot point should be chosen

Use the Next button on the RK4 Solution Approximation screen which will present a dialog requesting the T0 for the knot

This will present the next Interpolation

This repeats for as many knots as are required



Now the Solution Test “Spline” button will generate the spline for the solution approximation

CALCLIB Environment

Symbol	Notation	Value
INFINITY	∞	1000000.0
N	30	
SOLUTION_INTERPOLATION_C...		(9.45487513303952, 8.1667006...
T		(1, 1.05, 1.1, 1.15, 1.2, 1.25, 1.3, 1)
T0		1
Tn		2.499999999999998
VERSION		2013.02.22
f		(17.11872471442053, 17.43953...
Y0		17.11872471442053
Yn		25.49296094518933
a		9.8
b		0.196
epsilon	ϵ	2.719281828459
t		0.08
t0		30.0
x		2.499999999999998
dt		1.0
dx		0.0500000000000001

Function	Parameters	Equation
FODE	Tn, Yn	$a - b * Yn$
FODE	Tn, Yn	$FODE(Tn, Yn)$
fieldODE	x	$y'(x) + b^2 y(x) - a$
forPlot	x, y	$twoD(x, y)$
testError	x	$ f(x) - y(x) $
testError	x	Function Spline
expr.json		
J0-fractions TDF	y	Function Derivative Spline
J0Fractions.TDF	y'	Second Derivative Spline

Now the solution function "y" can be found in the functions list along with the derivatives y' and y''

CALCLIB Environment

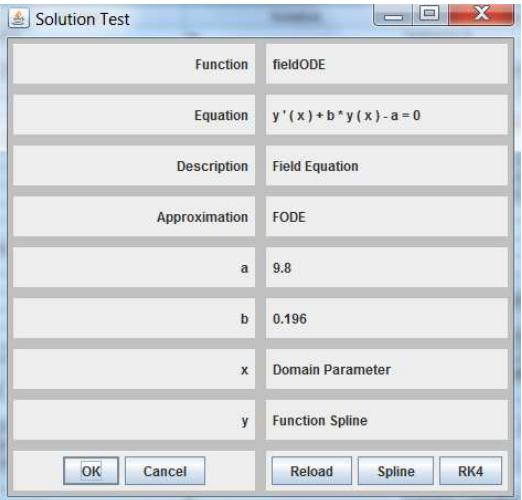
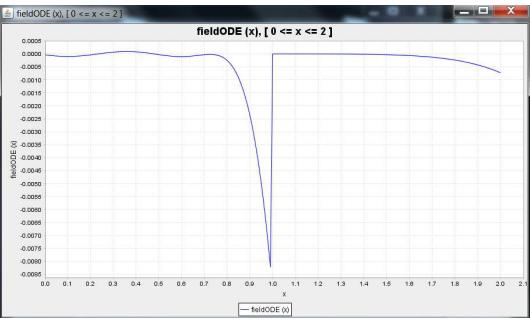
Symbol	Notation	Value
INFINITY	∞	1000000.0
N	30	
SOLUTION_INTERPOLATION_C...		(9.45487513303952, 8.1667006...
T		(1, 1.05, 1.1, 1.15, 1.2, 1.25, 1.3, 1)
T0		1
Tn		2.499999999999998
VERSION		3.1743953
f		
Y0		
Yn		
a		
b		
epsilon	ϵ	
t		
t0		
x		
dt		
dx		

Function	P	Options
FODE	Tn, Yn	Express
fieldODE	x	Encode
forPlot	x, y	Prepare DEQ Test
testError	x	Run DIFEQ Solution Test
testError	x	Save Expression As JSON
expr.json		Load Expression From JSON
J0-fractions.TDF	y	Apply Rule (Differentiation)
J0Fractions.TDF	y'	Apply Rule (Integration)

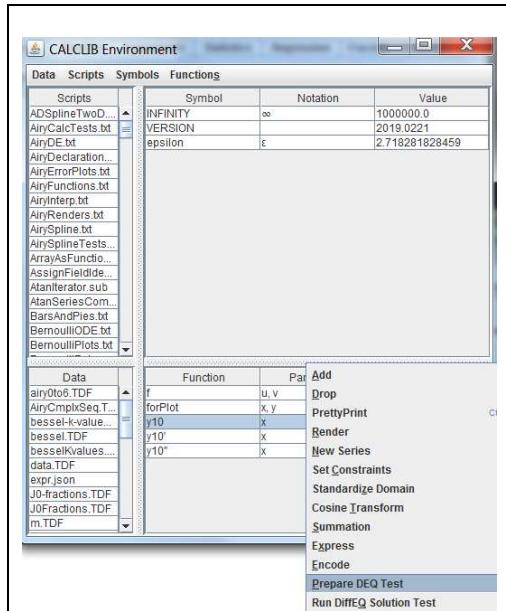
The function can be saved as a JSON expression for later import as a common singleton function

Hence the solution is available as a function and the accuracy across the domain is known from the error plot

Test the Solution Quality

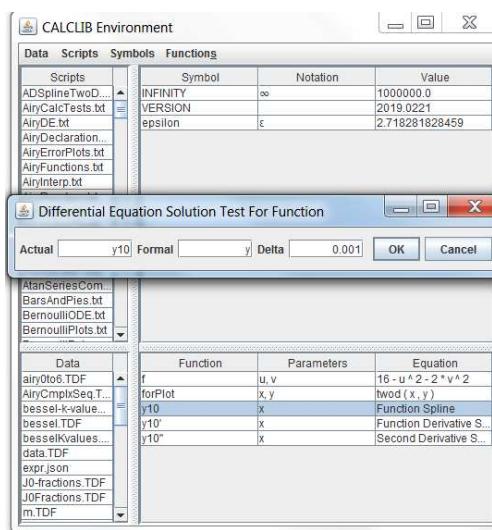
	The Solution Test screen now shows the solution "y" has a "Function Spline" available for test
	The OK button show a form for entering domain parameters
	A plot of the solution error for the domain is shown An error spike is typically seen at each knot

Test function as Solution

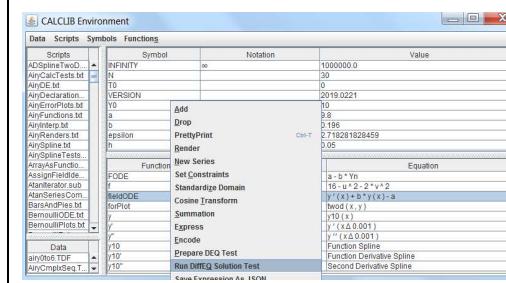


Select function

“Prepare DEQ Test” from function menu

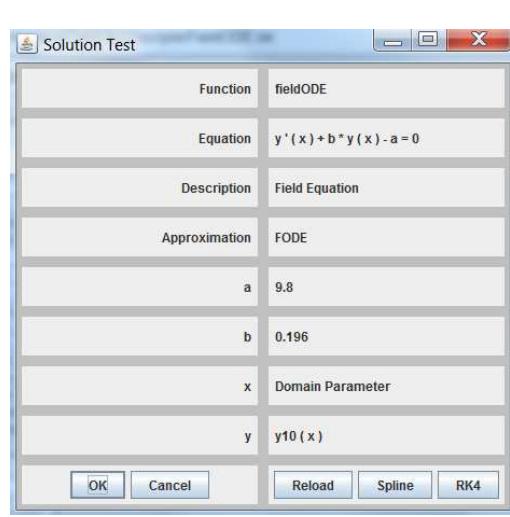
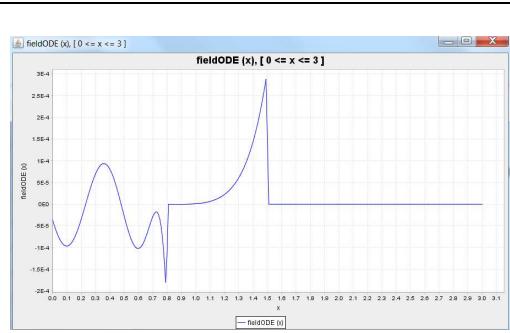


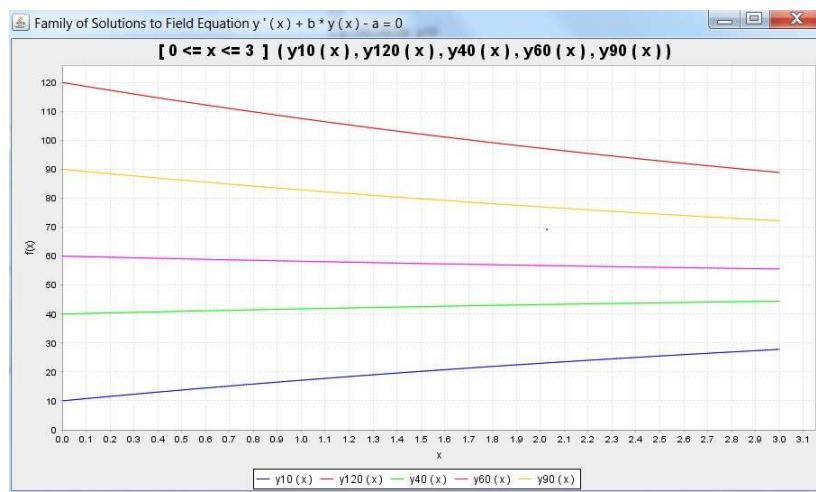
A dialog will request the formal parameter name for the function within the differential equation



Select the differential equation

“Run DiffEQ Solution Test” from the Function drop-down menu

	<p>The solution test form comes up Now the formal parameter is shown to reference the selected actual parameter Function $y = y10(x)$ Press OK button</p>
	<p>Enter domain description</p>
	<p>The error plot is displayed</p>



RungeKutta.txt

Use Runge-Kutta approximation and check the error computed against solution vector

```
// Runge-Kutta error test

// generate interpolation
T = [ 0 <= t <= N ] ( T0 + t * h )
solution = CHEBYSHEV (T, Y)

// generate alias for solution and for derivative
!! y (x) = solution @*^ x
!! y' (x) = solution @*^' x

// improve polynomial use efficiency
OPTIMIZE y ; OPTIMIZE y'

// post error test
!! testError (x) = y' (x) - f ( x, y(x) )

// plot error against domain described by test parameters
PLOTF testError [ T0 <= x <= T0 + h * N <> h/10 ]
```

SIDEBYSIDE "Regression Versus Error Plot"

RungeKuttaRiccati.txt

A simple example of use of RungeKutta.txt

// Riccati

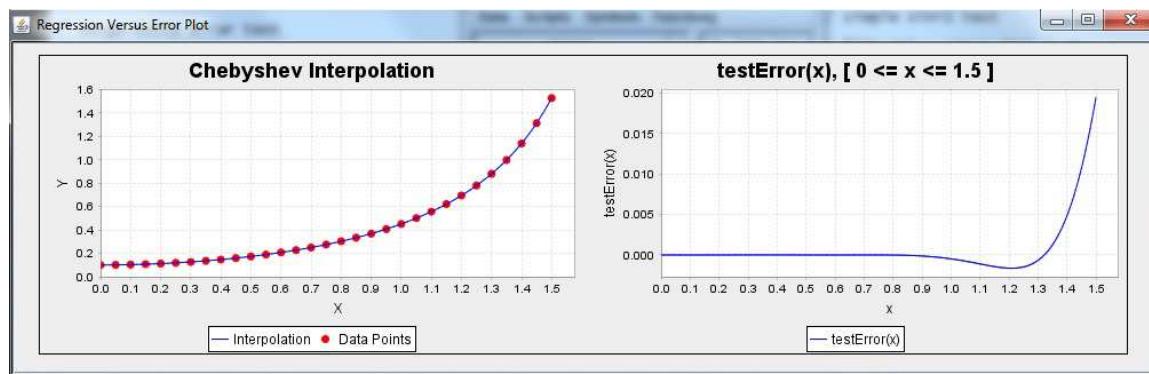
!! $u(x) = x / 2$

$Y_0 = 0.1 ; T_0 = 0 ; h = 0.05 ; N = 30$

!! $f(T_n, Y_n) = Y_n^2 + T_n * u(T_n) * Y_n + u(T_n)$

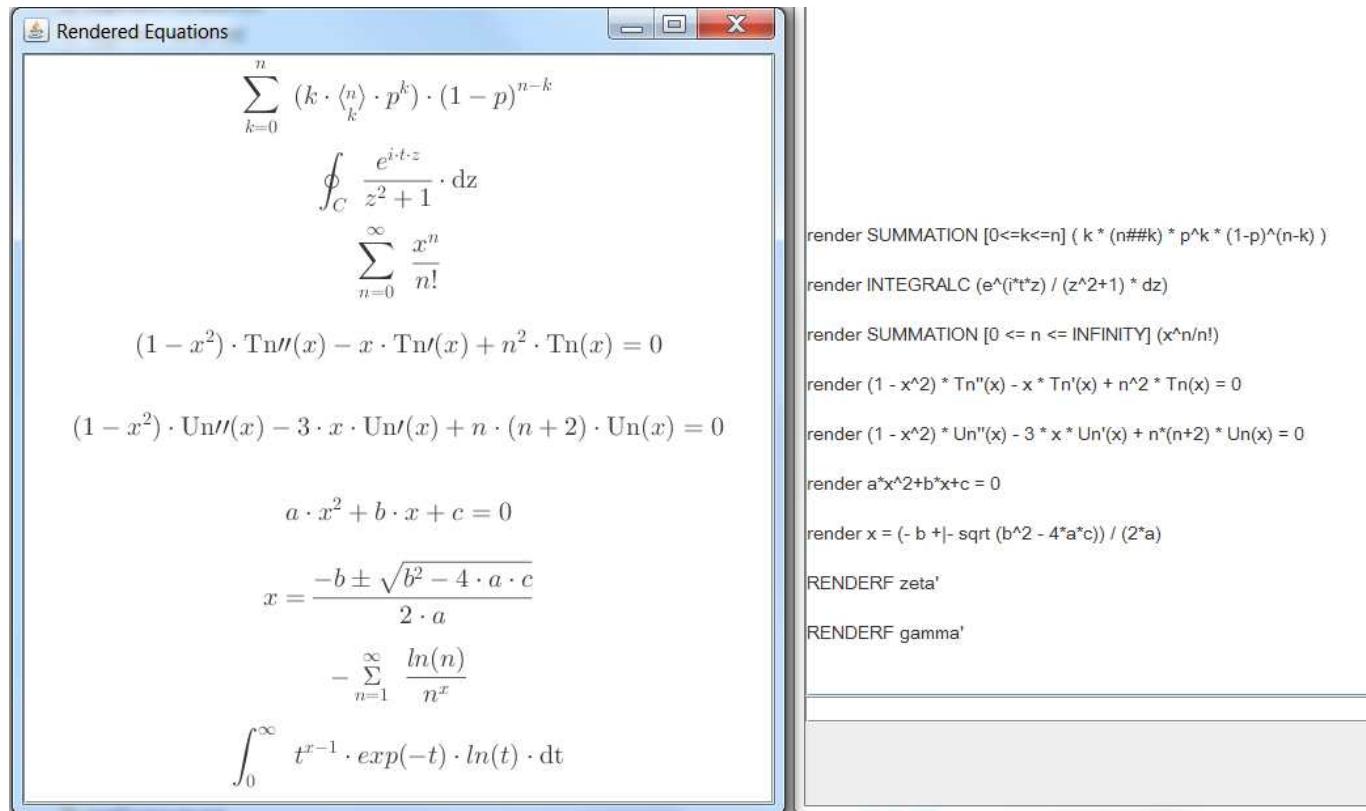
$Y = (0.1, 0.10113171857626335, 0.10354424073431215, 0.10726542307837783, 0.11232678123700574, 0.11876514779425473, 0.12662446126207821, 0.1359577239358084, 0.14682917975976403, 0.15931677951018194, 0.1735150208399999, 0.18953827665792933, 0.20752475923532693, 0.22764131262151488, 0.250089287107206, 0.2751118334633071, 0.303003071635968, 0.3341197537209428, 0.36889627761184174, 0.40786425183380254, 0.45167832094096894, 0.5011507269701952, 0.5572982587771222, 0.6214070869571133, 0.6951239492603989, 0.7805870513163335, 0.8806183876003942, 0.999013878270224, 1.1409946276569765, 1.3139341820639356, 1.5285808552014561)$

READ RungeKutta.txt



Equation Renders

RENDER commands are available to supply MathML / LaTeX type renders of sophisticated equations.



Functions found in the Functions list can be rendered using:

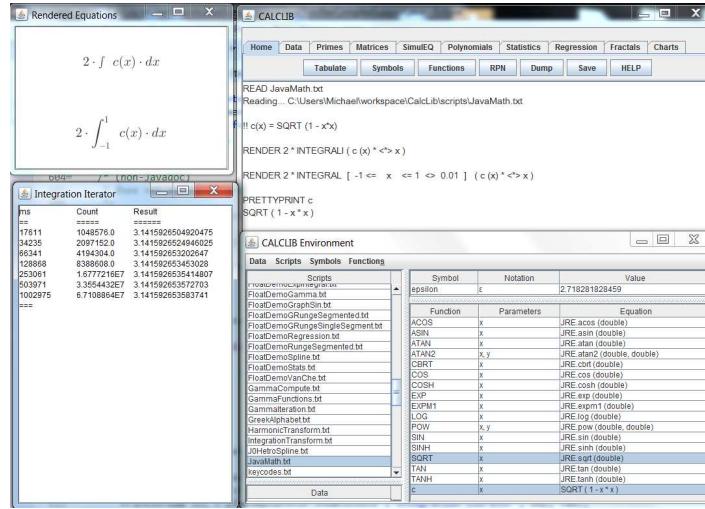
RENDERF function-name

As seen above equations can be rendered directly from the command line e.g.

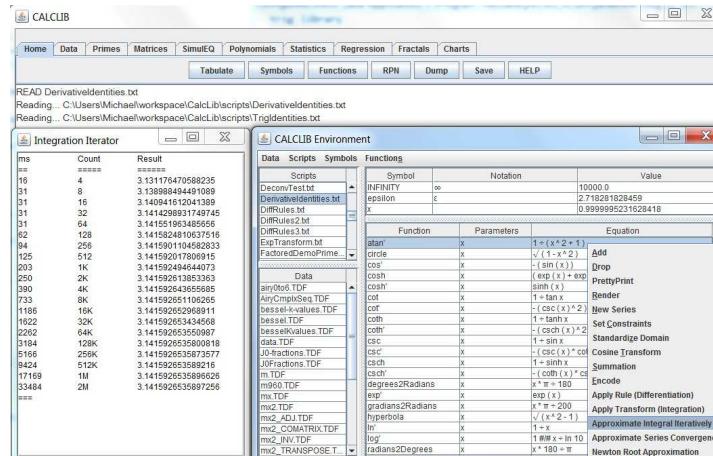
```
RENDER (1 - x^2) * Tn''(x) - x * Tn'(x) + n^2 * Tn(x) = 0
```

Numeric Integration

Brute force calculation of the sum of function values over a specified range can result in an approximate value for a definite integral. Doing the process iteratively allows the pattern of convergence to be identified and the efficiency of the method to be evaluated in a time VS accuracy assessment.

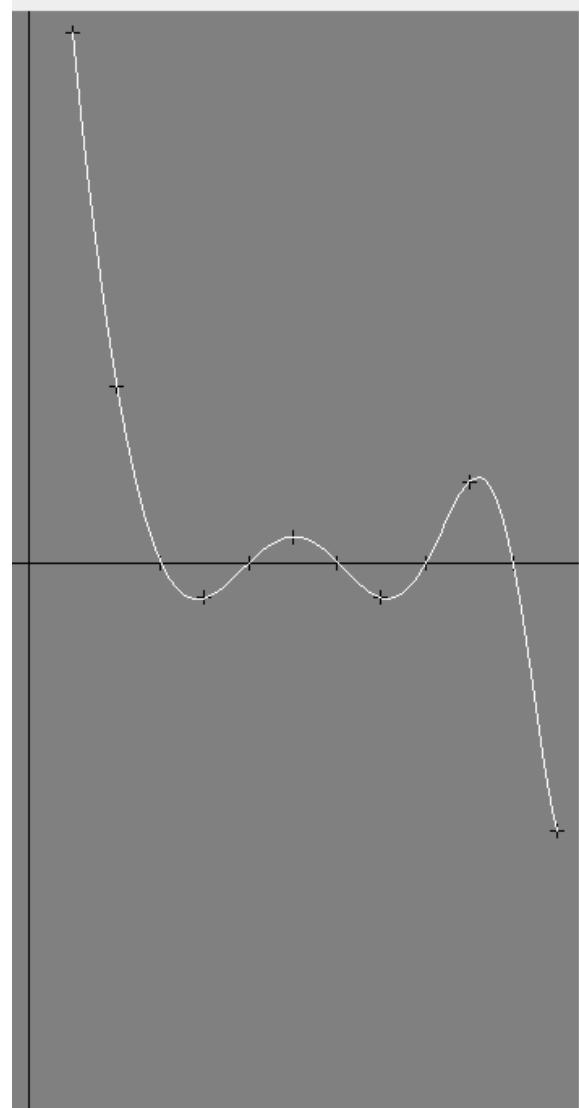


The Integration Approximation of the function selection menu produces the above output. A multiplier can be selected to allow the output to be more intelligible (e.g. pi VS pi/2 in the illustration) and the low to hi bounds of the integration interval are entered into the GUI. The indefinite and then the definite form of the integral are rendered. The iteration count is requested at which point the request can be canceled. The iteration count is relative to the log of the count of calculations. 2^iterations evaluations of the function will be computed. The display runs in the background and will show time, function evaluation count, and the approximation at that point of the integral.



BernoulliPolynomialRegression.txt

// this is just a fun exercise to see how well regression can fit the Bernoulli sequence



```
READ BernoulliPolynomialRegression.txt  
Reading... BernoulliPolynomialRegression.txt
```

```
a_FITPOLY_b = FITPOLY(a,b)
```

```
Polynomial regression
```

```
Y = 1.2531135531134323 - 1.3467492296060017*x +  
1.0447195450047913*x^2 - 0.6801772589547119*x^3 +  
0.30259420381926283*x^4 - 0.0890846611133917*x^5 +  
0.01781262969283056*x^6 - 0.00244056651571436*x^7 +  
2.2449744919974645E-4*x^8 - 1.3145933284817513E-5*x^9 +  
4.393846159717669E-7*x^10 - 6.341028166422841E-9*x^11
```

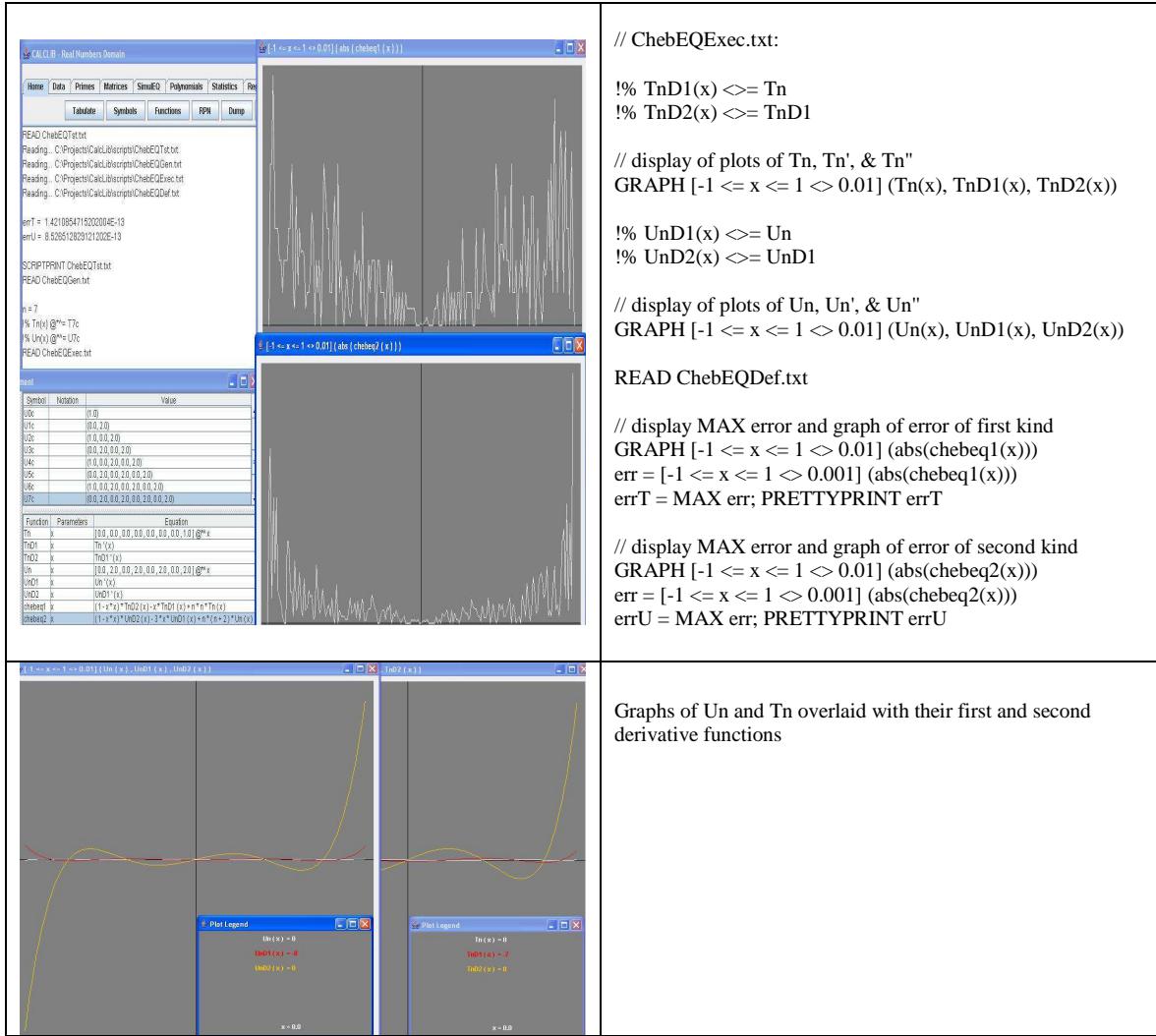
```
// note the coefficients decay fairly rapidly
```

```
r^2 = 1.0 // looks pretty good  
STD = 1.840308006797505E-12  
COV = 4.946471196610345E-11  
MSE = 3.386733559883005E-24  
SSR = 0.3337625103863371  
SSE = 3.386733559883005E-23  
SST = 0.33376251038338967
```

```
// solve for the roots of this polynomial, it shows:  
// (3, 5, 7, 9, 11) which is pretty impressive
```

ChebEQTst.txt

Chebyshev differential equations being used to demonstrate the implemented concept of transform functions. One of the transform types is ‘Calculus’ providing for integration (anti-derivative) and derivation (derivative function determination).



CmplxDemoBesselAiry.txt

```
INFINITY = 10

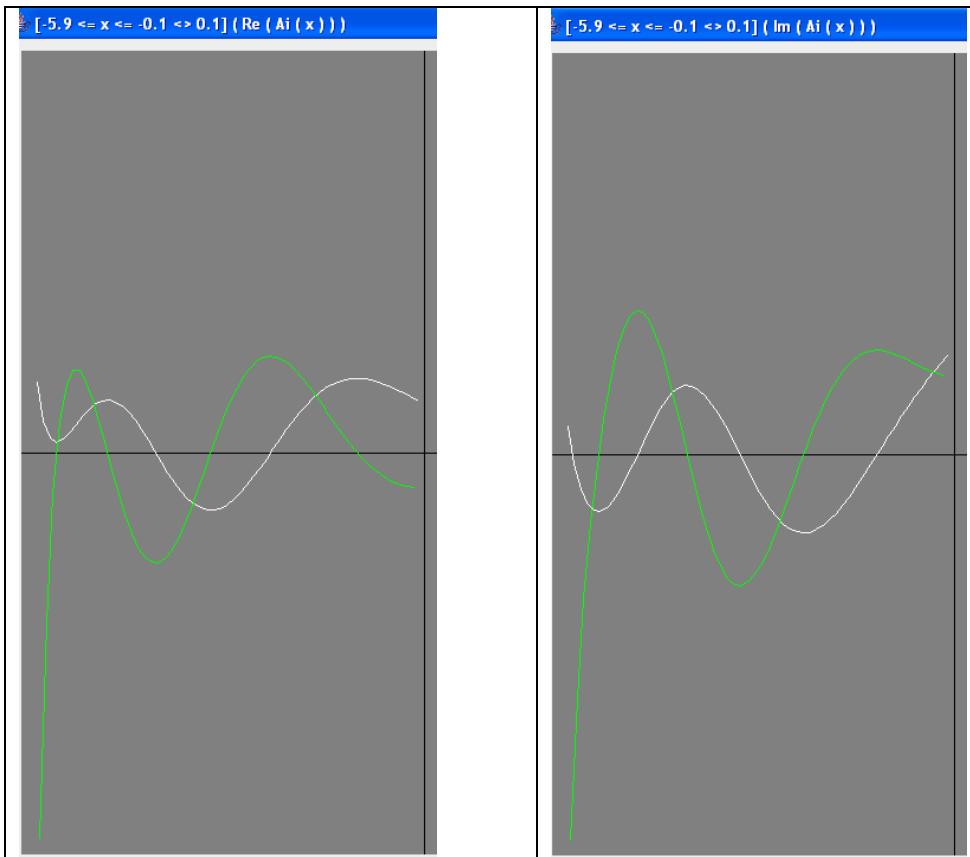
LIBRARY JRE java.lang.Math; !+ POW(x,y) = JRE.pow; !+ SIN(x) = JRE.sin
LIBRARY Gamma cern.jet.stat.Gamma; !+ gamma(x) = Gamma.gamma

// modified Bessel Ialpha and Kalpha
!! Ialpha(alpha,x) = SIGMA [0<=m<=INFINITY] ( (x/2)^(2*m+alpha) / m! / gamma (m+alpha+1) )
!! Kalpha(alpha,x) = pi/2 * ( Ialpha(-alpha,x) - Ialpha(alpha,x) ) / SIN (pi*alpha)

// Airy
!! Ai(x) = 1/pi * sqrt(x/3) * Kalpha( 1/3, 2/3 * sqrt(POW(x,3)) )
!! Bi(x) = sqrt(x/3) * ( Ialpha( 1/3, 2/3 * sqrt(POW(x,3)) ) + Ialpha( -1/3, 2/3 * sqrt(POW(x,3)) ) )

aivals = [-10<x<0<>0.1](Ai(x))
graph [-6<x<0<>0.1](Re (Ai(x)))
graph [-6<x<0<>0.1](Im (Ai(x)))

// as mentioned before this is the complex version using Bessel functions
```



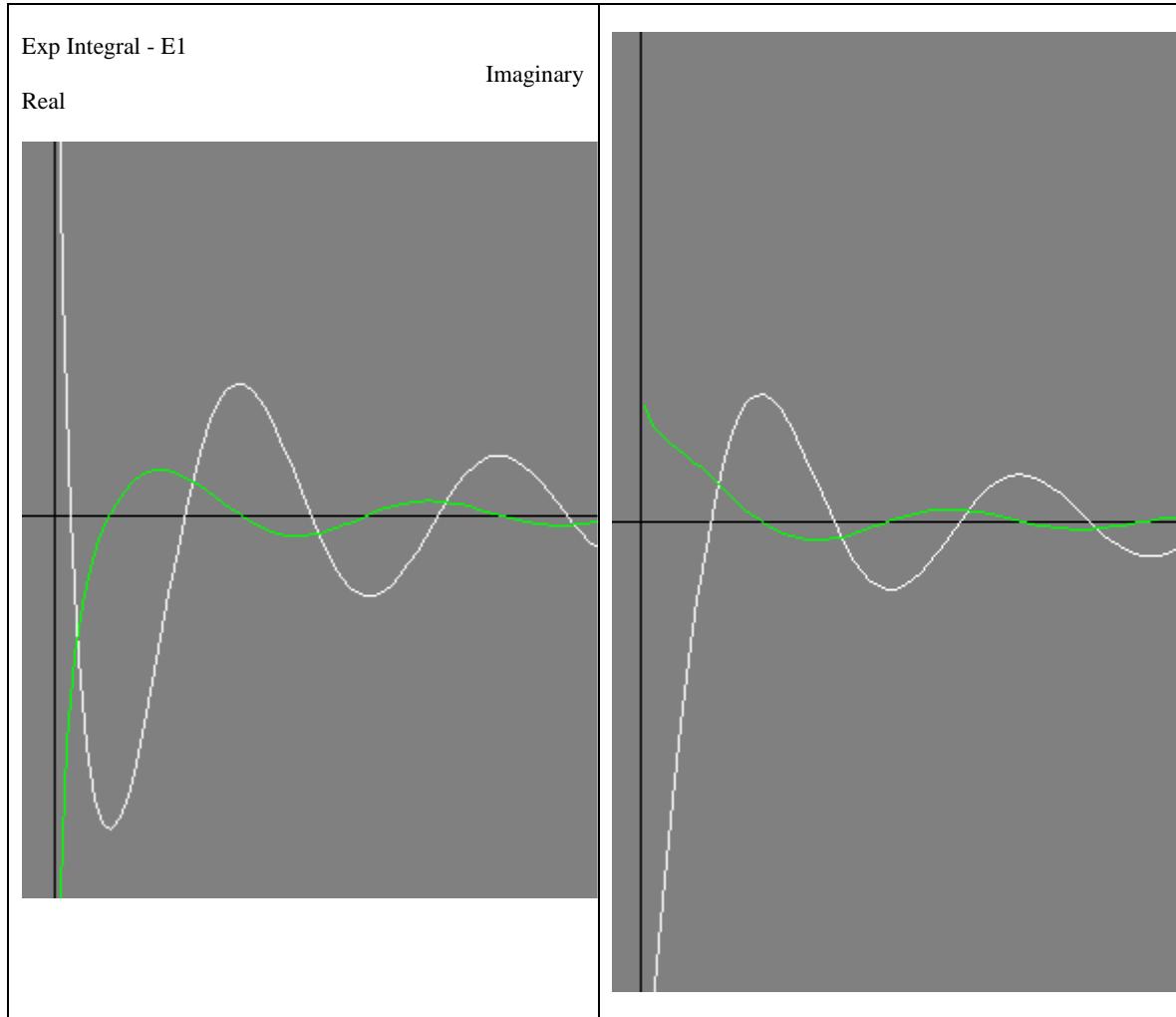
Functions

```
Ai (x) = 1 ÷ π * √(x ÷ 3) * Kalpha (1 ÷ 3, 2 ÷ 3 * √(POW (x, 3)))
Bi (x) = √(x ÷ 3) * ( Ialpha (1 ÷ 3, 2 ÷ 3 * √(POW (x, 3))) + Ialpha (-1 ÷ 3, 2 ÷ 3 * √(POW (x, 3))))
Ialpha (α, x) = Σ [0 <= m <= ∞] ((x ÷ 2)^(2 * m + α) ÷ m! ÷ γ (m + α + 1))
Kalpha (α, x) = π ÷ 2 * ( Ialpha (-α, x) - Ialpha (α, x)) ÷ SIN (π * α)
```

CmplxDemoExpInt.txt

```
INFINITY = 20  
dt = 1 / INFINITY  
  
// exp integral  
  
!! Ei(x) = - INTEGRAL [-x <= t <= INFINITY >> dt] ( exp(-t) / t * dt )  
!! E1(x) = INTEGRAL [x <= t <= INFINITY >> dt] ( exp(-t) / t * dt )  
  
E1vals = [0.2 <= x <= 16 >> 0.2] (E1(i*x))  
  
graph [0<=ii<LENGTH(E1vals)]( Re(E1vals#ii) )  
graph [0<=ii<LENGTH(E1vals)]( Im(E1vals#ii) )
```

again with the infinite integrals, but being exponential and complex they become diminishing cyclic functions



Functions

$$\begin{aligned} E1(x) &= \int [x \leq t \leq \infty] (\exp(-t) / t^*) dt \\ Ei(x) &= - \int [-x \leq t \leq \infty] (\exp(-t) / t^*) dt \end{aligned}$$

CmplxDemoMatrix.txt

```
// simple examples of the matrix primitives
```

```
READ CmplxDemoMatrix.txt
Reading... C:\workspace\MathFields\scripts\CmplxDemoMatrix.txt
```

```
SCRIPTPRINT CmplxDemoMatrix.txt
```

```
IMPORT mx2 MX2.TDF
PRETTYPRINT mx2
```

```
mx2_DET = DET(mx2)
PRETTYPRINT mx2_DET
```

```
mx2_TRANSPOSE = TRANSPOSE(mx2)
PRETTYPRINT mx2_TRANSPOSE
```

```
mx2_COMATRIX = COMATRIX(mx2)
PRETTYPRINT mx2_COMATRIX
```

```
mx2_ADJ = ADJ(mx2)
PRETTYPRINT mx2_ADJ
```

```
mx2_INV = INV(mx2)
PRETTYPRINT mx2_INV
```

```
mx2_TR = TR(mx2)
PRETTYPRINT mx2_TR
```

```
mx2_CHARACTERISTIC = CHARACTERISTIC(mx2)
```

```
mx2_EIGENVALUE = EIG(mx2,mx2_EIGENVECTOR)
```

OUTPUT:

Import from file: C:\workspace\MathFields\data\MX2.TDF
Import to matrix: mx2
Columns per row: 3
Rows read: 3

data read:

$$\begin{pmatrix} (-7 - 11i) & (-11 + 10i) & (14 - 5i) \\ (-11 - 16i) & (-13 + 11i) & (19 - 4i) \\ (18 + 12i) & (12 - 18i) & (-15 + 12i) \end{pmatrix}$$

=EOD=

mx2 =

$$\begin{pmatrix} (-7 - 11i) & (-11 + 10i) & (14 - 5i) \\ (-11 - 16i) & (-13 + 11i) & (19 - 4i) \\ (18 + 12i) & (12 - 18i) & (-15 + 12i) \end{pmatrix}$$

mx2_DET =

$$(-369 + 108i)$$

mx2_TRANSPOSE =

$$\begin{pmatrix} (-7 - 11i) & (-11 - 16i) & (18 + 12i) \\ (-11 + 10i) & (-13 + 11i) & (12 - 18i) \\ (14 - 5i) & (19 - 4i) & (-15 + 12i) \end{pmatrix}$$

mx2_COMATRIX =

$$\begin{pmatrix} (-93 + 69i) & (33 + 48i) & (-54 - 36i) \\ (33 - 30i) & (-75 + 3i) & (-36 + 54i) \\ (-42 + 15i) & (-57 + 12i) & -69 \end{pmatrix}$$

mx2_ADJ =

$$\begin{pmatrix} (-93 + 69i) & (33 - 30i) & (-42 + 15i) \\ (33 + 48i) & (-75 + 3i) & (-57 + 12i) \\ (-54 - 36i) & (-36 + 54i) & -69 \end{pmatrix}$$

mx2_INV =

$$\begin{pmatrix} (0.28256 - 0.10429i) & (-0.10429 + 0.05078i) & (0.1158 - 0.00676i) \\ (-0.04731 - 0.14393i) & (0.18941 + 0.04731i) & (0.15105 + 0.01169i) \\ (0.10849 + 0.12932i) & (0.12932 - 0.10849i) & (0.17224 + 0.05041i) \end{pmatrix}$$

mx2_TR =

$$(-35 + 12i)$$

Characteristic polynomial : $- (369 - 108i) + (237 - 72i) * x - (35 - 12i) * x^2 - x^3$

Von Mises dominant Eigen-Pair

Dominant Eigenvalue is: 42.720018726587654

Dominant Eigenvector is:

$$\begin{bmatrix} [0.45054741724475555 + 0.06575787393774088i], \\ (0.5946656941881084 + 0.005590057346825491i), \\ (-0.6128582806028051 - 0.25185138105731264i) \end{bmatrix}$$

CmplxDemoRoots.txt

```
READ CmplxDemoRoots.txt
Reading... C:\workspace\MathFields\scripts\CmplxDemoRoots.txt
Reading... C:\workspace\MathFields\scripts\CmplxDemoRoots.TXT
SCRIPTPRINT CmplxDemoRoots.TXT
```

```
x = 5
SCRIPTPRINT sqrtCompute.txt
SCRIPTPRINT sqrtIteration.txt
READ sqrtCompute.txt
```

```
x=(1, 1, -1)
SCRIPTPRINT QUADRATIC.txt
READ QUADRATIC.txt
```

```
calc ROOTS x
```

```
x=(2,-3,4,-5)
SCRIPTPRINT CUBIC.txt
READ CUBIC.txt
```

```
calc ROOTS x
```

```
x=(2,-3,4,-5,1)
SCRIPTPRINT QUARTIC.txt
READ QUARTIC.txt
```

```
calc ROOTS x
```

```
Reading... C:\workspace\MathFields\scripts\sqrtCompute.txt
```

```
// looking for 10 decimal places of precision
```

```
TOLERANCE = 10^(-10)
```

```
// polynomial solutions "c - x^2 = 0" gives sqrt(c) as a solution
```

```
sqrt_polynomial = (x,0,-1)
```

```
polyprint sqrt_polynomial
```

```
// Newton's methods uses the first derivative for the solution
```

```
sqrt_poly_derivative = POLYDER sqrt_polynomial
```

```
polyprint sqrt_poly_derivative
```

```
// initial approximation
```

```
sqrtx = 1
```

```
ITERATE 10 SqrtIteration.txt
```

OUTPUT:

```
Reading... C:\workspace\MathFields\scripts\sqrtIteration.txt

// compute polynomial value
f = sqrt_polynomial +*^ sqrtx

// compute derivative value
fprime = sqrt_poly_derivative +*^ sqrtx

// iteration contribution computed
sqrt_iteration = f/fprime

// new value for this iteration
sqrtx = sqrtx - sqrt_iteration

// check status of result
sqrtx_squared = sqrtx ^ 2

// display intermediate results
PRETTYPRINT sqrt_iteration
PRETTYPRINT sqrtx_squared
PRETTYPRINT sqrtx

// look for convergence
ASSERT "Convergence Complete" TOLERANCE >|| sqrt_iteration

sqrt_polynomial =
      5 - x^2

sqrt_poly_derivative =
      - 2*x
```

Reading... C:\workspace\MathFields\scripts\SqrtIteration.txt

sqrt_iteration =

-2

sqrtx_squared =

9

sqrtx =

3

sqrt_iteration =

0.6666666666666666

sqrtx_squared =

5.444444444444455

sqrtx =

2.3333333333333335

sqrt_iteration =

0.09523809523809545

sqrtx_squared =

5.009070294784581

sqrtx =

2.238095238095238

sqrt_iteration =

0.002026342451874501

sqrtx_squared =

5.000004106063731

sqrtx =

2.2360688956433634

```
sqrt_iteration =
9.181433852065489E-7

sqrtx_squared =
5.0000000000000843

sqrtx =
2.236067977499978

sqrt_iteration =
1.8847399291451707E-13

sqrtx_squared =
5.000000000000001

sqrtx =
2.23606797749979

Script interrupted in iteration 6
Assertion "Convergence Complete" has been validated, TOLERANCE >|| sqrt_iteration
*** Script has terminated
```

Reading... C:\workspace\MathFields\scripts\QUADRATIC.txt

```
a = x#2; b = x#1; c = x#0
quad_common = -b / (2 * a)
quad_d = b^2 - 4*a*c; quad_sqrt = 2\quad_d
quad_roots = quad_common +|- quad_sqrt / (2 * a)
PRETTYPRINT quad_roots

quad_roots =
[
    -0.6180339887498949
    1.618033988749895
]

// computed real roots
(-0.6180339887498949, 1.618033988749895)
```

Reading... C:\workspace\MathFields\scripts\CUBIC.txt

```
delta#3 = 0
a = x#3; b = x#2; c = x#1; d = x#0
delta#0 = b^2 - 3*a*c; delta#1 = 2*b^3 - 9*a*b*c + 27*a^2*d
delta#3 = delta#1 ^ 2 - 4 * delta#0 ^ 3
C = 3\(( ( delta#1 + 2\delta#3 ) / 2 )
u = APPEND (-1/2 +|- i*2\3 / 2, 1)
cubic_roots = [0 <= k <= 2] (-(b + u#k * C + delta#0 / (u#k * C)) / (3*a))
PRETTYPRINT cubic_roots

cubic_roots =
[
    (0.03533842849427097 + 0.7397330112915744*i)
    (0.03533842849427097 - 0.7397330112915744*i)
    0.7293231430114581
]

// computed real roots
(0.7293231430114581)
```

Reading... C:\workspace\MathFields\scripts\QUARTIC.txt

```
delta#3 = 0
a = x#4; b = x#3; c = x#2; d = x#1; e = x#0
delta#0 = c^2 - 3*b*d + 12*a*e
delta#1 = 2*c^3 - 9*b*c*d + 27*b^2*e + 27*a*d^2 - 72*a*c*e
delta#3 = delta#1 ^ 2 - 4 * delta#0 ^ 3
p = (8*a*c - 3*b^2) / (8*a^2)
q = (b^3 - 4*a*b*c + 8*a^2*d) / (8*a^3)
Q = 3\((delta#1 + 2\((delta#3)) / 2)
S = 2\((Q + delta#0 / Q) / (3*a) - 2*p/3) / 2
common = -b / (4*a)
// quartic_roots_12 = common - S +|- 2\((q/S - 4*S^2 - 2*p) /2
// quartic_roots_34 = common + S +|- 2\((-q/S - 4*S^2 - 2*p) /2
// quartic_roots = APPEND (quartic_roots_12, quartic_roots_34)
CMPS = common -|+ S
quartic_roots = APPEND ( CMPS#0 +|- 2\((q/S - 4*S^2 - 2*p) /2 , CMPS#1 +|- 2\((-q/S - 4*S^2 - 2*p) /2 )
PRETTYPRINT quartic_roots
quartic_roots =
[
    (0.004423084318876391 + 0.7714190717315003*i)
    (0.004423084318876391 - 0.7714190717315003*i)
    4.188847029536467
    0.8023068018257802
]
// computed real roots
(0.8023068018257804, 4.188847029536467)
```

Iterations Improving Accuracy

Using FACTORIZATION mode, values are maintained as ratios of prime factors. By selecting algorithms that are based on series of integer ratios it is possible to compute irrational values out to arbitrary numbers of significant digits providing that the numerator and denominator are allowed to grow to arbitrary numbers of digits. Having implemented this methodology the remaining issue is how fast the series converges since this impacts how many terms are required to give an accurate number of how many digits. The sample below uses the Ramanujan series to calculate 1/pi. This series converges very rapidly. Computation of 10 terms of the series gives a calculation of 85 digits of pi. The representation of the ratio has numerator and denominator values that are significantly longer than 85 digits

Script Source

```
SCRIPTPRINT ComputePiRamanujan.txt
Reading... C:\Users\Michael\workspace\CalcLib\scripts\ComputePiRamanujan.txt

x = 2
READ sqrtCompute.txt
// requires 7 iterations of Newton's method
// sqrt(2) = 1.4142_13562_37309_50488_01688_72420_96980_78569_67187_53769_48073_17667_97379_90732_47846_21070_38850_38753_43276 // 95 digits
// = 1.4142_13562_37309_50488_01688_72420_96980_78569_67187_53769_48073_17667_97379_90732_47846_21070_38850_38753_43276 // per OEIS A002193
// = 494604117625520187877508648757351061418968498177 / 349737925575794117202851852070562919437964212608
// = (7681 * 1492993 * 431302713980890947612633357964569696769) / (2^7 * 3 * 17 * 257 * 577 * 1409 * 11777 * 665857 * 2393857 * 2448769 * 55780318173953)

radical2 = sqrt(x

// 1 / pi = ( 2 * sqrt(2) / 9801 ) * SIGMA [0 <= k <= INFINITY] ( (4*k)! * (1103 + 26390*k) / ((k!)^4 * 396^(4*k)) )

!! series (n) = SIGMA [0 <= k <= n] ( (4*k)! * (1103 + 26390*k) / ((k!)^4 * 396^(4*k)) )

s = series(10)
// s = 1103.00002683197457346381340888213305633693640546388967
// = 3872430076450268064456794609054361181459412148133735536353909359385271078094930941516221071647971468276373108204166977
// 206077712680271927759102018748518486070845634112299488263564859436406918411999470275
// / 35108159403881632092975628161374616891448276678347860097892396840833649930820528350216578632712876164962142367521057398
// 56126714936857216329479003182021378136916689419412298749856707187360789430272
// = (5^2 * 7^3 * 13 * 23 * 29 * 31 * 37 * 41 * 43 * 47 * 53 * 59 * 61 * 67 * 71 * 73 * 79 * 83 * 191 * 1451 *
// 4552468928477187574050463918838755530501788164295356574114208913461935568005590671309567043
// 440364087806034470850425552964221299682289083320181488913830968277491841459)
// / ( 2^154 * 3^153 * 11^77 )
// PRETTYPRINT s 50

c = 2 * radical2 / 9801
// c = 0.0002885856522254770917287801738796002011420709630
// = 494604117625520187877508648757351061418968498177 / 17138907042841790713488184501071793586705743623885504
// = (7681 * 1492993 * 431302713980890947612633357964569696769) / ( 2^6 * 3^5 * 11^2 * 17 * 257 * 577 * 1409 * 11777 * 665857 * 2393857 * 2448769 *
// 55780318173953 )
// PRETTYPRINT c 50

piApproximation = 1 / (c*s)
PRETTYPRINT piApproximation 86
// piApproximation = 3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510_58209_74944_59230_78164_06286_20899_86280_3
// = 1063088298856439570523868829133082411252165861973797244208822758794531096590230104814300659761592544166124366527517669023328669747469931773952
// / 338525374596355722607287291604043659184224783071313742055861437609455035446281079201455765711868066989195644732872439287244601612731402376166467685
// = ( 2^80 * 3^83 * 11^39 * 257 * 577 * 1409 * 11777 * 665857 * 2393857 * 2448769 * 55780318173953
// / ( 5 * 7^2 * 13 * 19 * 23 * 29 * 31 * 37 * 41 * 43 * 7681 * 1492993 * 431302713980890947612633357964569696769 *
// 8385498739767342831426911893732450575073715160975667674966403041328086345417661675621 )
// pi =
// 3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510_58209_74944_59230_78164_06286_20899_86280_34825_
// 34211_70679_82148_08651_32823_06647_09384_46095_50582_23172_53594_08128
// 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110 115 120 125 130 135 140 145 150
```

Script Execution

```
READ ComputePiRamanujan.txt
Reading... C:\Users\Michael\workspace\CalcLib\scripts\sqrtCompute.txt

sqrt_polynomial =
    2 - x^2

sqrt_poly_derivative =
    - 2 * x

Reading... C:\Users\Michael\workspace\CalcLib\scripts\SqrtIteration.txt
sqrt_iteration =
    -0.500000000000000 = -1 / 2 = ( -1 ) / ( 2 )

sqrtx_squared =
    2.250000000000000 = 9 / 4 = ( 3^2 ) / ( 2^2 )

sqrtx =
    1.500000000000000 = 3 / 2 = ( 3 ) / ( 2 )

sqrt_iteration =
    0.08333333333334 = 1 / 12 = ( 1 ) / ( 2^2 * 3 )

sqrtx_squared =
    2.006944444444445 = 289 / 144 = ( 17^2 ) / ( 2^4 * 3^2 )

sqrtx =
    1.4166666666666667 = 17 / 12 = ( 17 ) / ( 2^2 * 3 )

sqrt_iteration =
    0.0024509803921569 = 1 / 408 = ( 1 ) / ( 2^3 * 3 * 17 )

sqrtx_squared =
    2.0000060073048828 = 332929 / 166464 = ( 577^2 ) / ( 2^6 * 3^2 * 17^2 )

sqrtx =
    1.4142156862745099 = 577 / 408 = ( 577 ) / ( 2^3 * 3 * 17 )

sqrt_iteration =
    0.0000021238998199 = 1 / 470832 = ( 1 ) / ( 2^4 * 3 * 17 * 577 )

sqrtx_squared =
    2.000000000045110 = 443365544449 / 221682772224 = ( 665857^2 ) / ( 2^8 * 3^2 * 17^2 * 577^2 )

sqrtx =
    1.4142135623746900 = 665857 / 470832 = ( 665857 ) / ( 2^4 * 3 * 17 * 577 )

sqrt_iteration =
    1.5948618246059560E-12 = 1 / 627013566048 = ( 1 ) / ( 2^5 * 3 * 17 * 577 * 665857 )

sqrtx_squared =
    2.000000000000001 = 786292024016459316676609 / 393146012008229658338304 = ( 257^2 * 1409^2 * 2448769^2 ) / ( 2^10 * 3^2 * 17^2 * 577^2 * 665857^2 )

sqrtx =
    1.4142135623730951 = 886731088897 / 627013566048 = ( 257 * 1409 * 2448769 ) / ( 2^5 * 3 * 17 * 577 * 665857 )

sqrt_iteration =
    8.992983216504540E-25 = 1 / 1111984844349868137938112 = ( 1 ) / ( 2^6 * 3 * 17 * 257 * 577 * 1409 * 665857 * 2448769 )

sqrtx_squared =
    2.000000000000001 = 2473020588127600939387543243786675530709484249089 / 1236510294063800469693771621893337765354742124544 =
        ( 11777^2 * 2393857^2 * 55780318173953^2 ) / ( 2^12 * 3^2 * 17^2 * 257^2 * 577^2 * 1409^2 * 665857^2 * 2448769^2 )

sqrtx =
    1.4142135623730951 = 1572584048032918633353217 / 1111984844349868137938112 =
        ( 11777 * 2393857 * 55780318173953 ) / ( 2^6 * 3 * 17 * 257 * 577 * 1409 * 665857 * 2448769 )
```

```

sqrt_iteration =
2.859283843339520E-49 = 1 / 3497379255757941172020851852070562919437964212608 =
( 1 ) / ( 2^7 * 3 * 17 * 257 * 577 * 1409 * 11777 * 665857 * 2393857 * 2448769 * 55780318173953 )

sqrtx_squared =
2.0000000000000001 =
2446323317211940977293404419928347279246091129334268572773850449273611455484253634058690852323329 /
12231661658605970488646702209964173639623045564667134286386925224636805727742126817029345426161664 =
( 7681^2 * 1492993^2 * 431302713980890947612633357964569696769^2 ) /
( 2^14 * 3^2 * 17^2 * 257^2 * 577^2 * 1409^2 * 11777^2 * 665857^2 * 2393857^2 * 2448769^2 * 55780318173953^2 )

sqrtx =
1.4142135623730951 = 4946041176255201878775086487573351061418968498177 / 3497379255757941172020851852070562919437964212608 =
( 7681 * 1492993 * 431302713980890947612633357964569696769 ) / ( 2^7 * 3 * 17 * 257 * 577 * 1409 * 11777 * 665857 * 2393857 * 2448769 * 55780318173953 )

Script interrupted in iteration 7
Assertion "Convergence Complete" has been validated, TOLERANCE >|| sqrt_iteration
*** Script has terminated

s =
1103.00002683197457346381340888213305633693640546388967 =
116354295547844200479625409627050350445031010498388307062857519290687784871920308555177681218916232885 /
105488932654005296215338934589125951145247950655030308540653845613856673595635654924408113432887296 =
( 5 * 7^2 * 13 * 17 * 19 * 23 * 29 * 31 * 37 * 41 * 43 * 838549873976734283142691189373245057037151609756676749664030413280866345417661675621 ) / ( 2^74 * 3^78 * 11^37 )

c =
0.0002885855652225470917287801738796002011420709630 = 4946041176255201878775086487573351061418968498177 /
17138907042841790713488184501071793586705743623885504 =
( 7681 * 1492993 * 431302713980890947612633357964569696769 ) / ( 2^6 * 3^5 * 11^2 * 17 * 257 * 577 * 1409 * 11777 * 665857 * 2393857 * 2448769 * 55780318173953 )

piApproximation =
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862804 =
10635088298856439557025238688291130082411252165861976379724420882275879453109659023010481430065976159254416616124366527517669023328669747469931773952 /
33852537459635572260728729160404365911842247830713137420558614376094550354462810792014557657118680669891956744732872439287244601612731402376166467685 =
( 2^80 * 3^83 * 11^39 * 257 * 577 * 1409 * 11777 * 665857 * 2393857 * 2448769 * 55780318173953 ) /
( 5 * 72^2 * 13 * 19 * 23 * 29 * 31 * 37 * 41 * 43 * 7681 * 1492993 * 431302713980890947612633357964569696769 *
838549873976734283142691189373245057037151609756676749664030413280866345417661675621 )

```

Represent Irrational Numbers with Primes

Compute irrational values out to arbitrary numbers of significant digits. This can be seen as an alternate way to describe an irrational number that carries additional attributes such as the specification of the equation and the number of terms that have been used to compute it. The sample below uses the Ramanujan series to calculate 1/pi. This series converges very rapidly. Using just 2 iterations of the summation results in an accurate double float equivalent calculation.

Script Source

```
1/pi = 2^(1/2) / 9801 * ( SUMMATION [0 <= k <= INFINITY] ( (4*k)! * (1103 + 26390*k) / ((k!)^4 * 396^(4*k)) ) )

!! series (n) = SUMMATION [0 <= k <= n] ( (4*k)! * (1103 + 26390*k) / ((k!)^4 * 396^(4*k)) )

calc series 1
// 1103.0000268319743490 = 1130173253125 / 1024635744 = ( 5^5 * 7 * 4423 * 11681 ) / ( 2^5 * 3^7 * 11^4 )

calc sqrt 2
// 1.4142135623730951 = 886731088897 / 627013566048 = ( 257 * 1409 * 2448769 ) / ( 2^5 * 3 * 17 * 577 * 665857 )

c = 2 * sqrt 2 / 9801

calc 1 / (c * series 1)
```

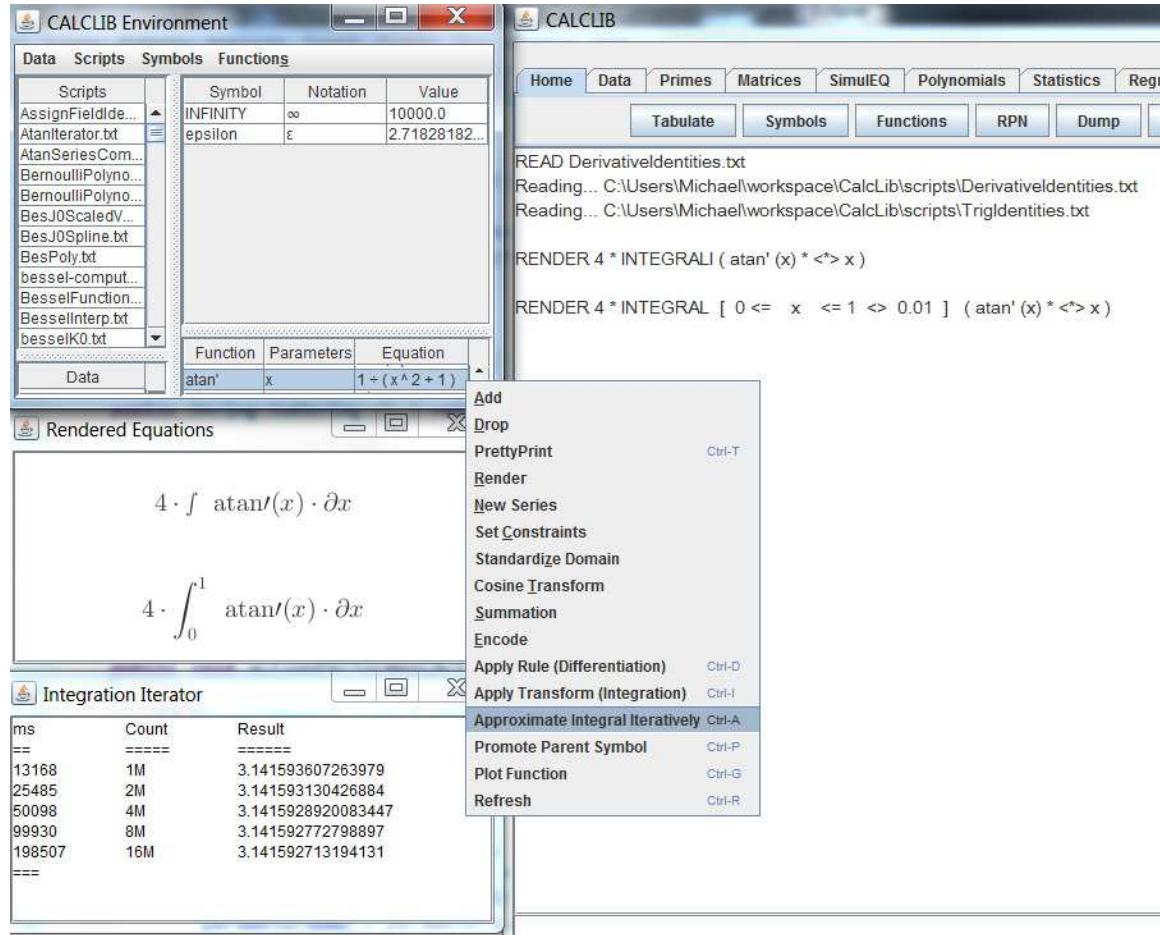
Script Execution (ComputePrimePI.txt)

```
3.141592653589793 =
3148377737809732379398656 / 1002159759385796058053125 =
(
    2^9 * 3^12 * 11^6 * 17 * 577 * 665857
)
/
(
    5^5 * 7 * 257 * 1409 * 4423 * 11681 * 2448769
)
```

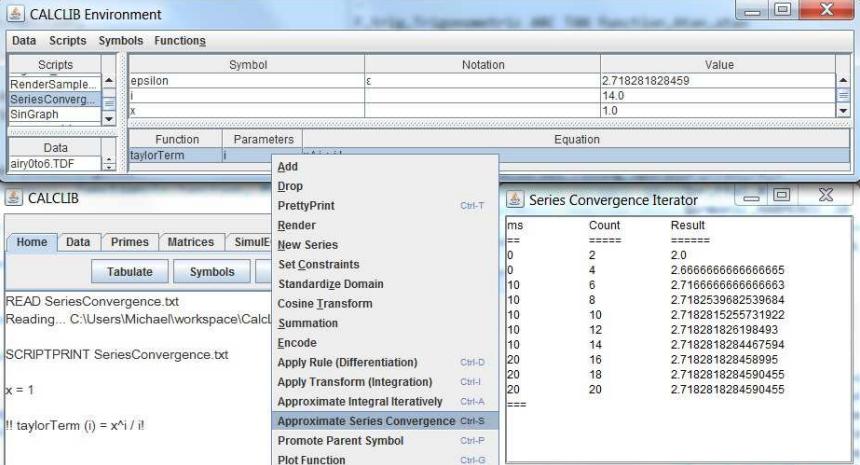
As mentioned above we can attribute this result as a 2 term approximation of the calculation using the Ramanujan series. This can be taken to imply that additional precision can be accomplished by using this as an interim result to which additional terms can be added

Approximate Integral Iteratively

Integral approximation using the trapezoidal rule is the implemented mechanism of the “Approximate Integral” functionality found in the Function List pop-up menu. Each iteration of the approximation doubles the count of function evaluations being used to compute the integral.

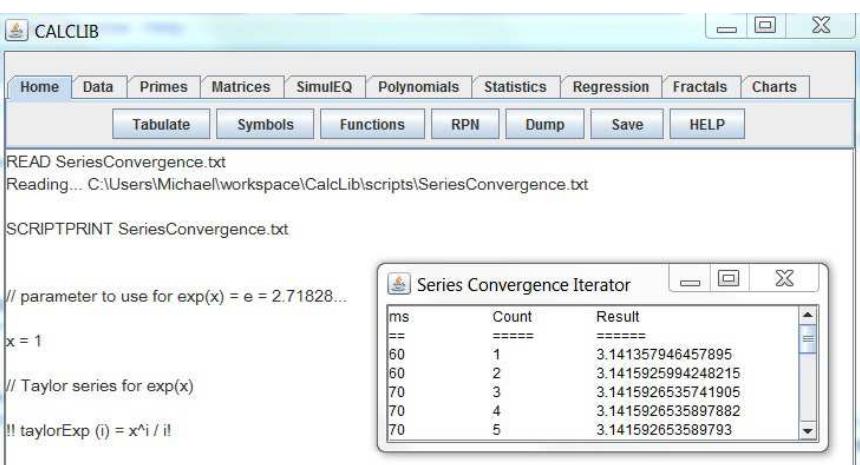


Approximate Series Convergence



The screenshot shows the CALCLIB Environment interface. In the top left, there's a table for 'Scripts' and 'Functions'. Below it is a 'Data' table with entries for 'epsilon' and 'x'. A 'Function' table contains 'taylorTerm'. A context menu for 'taylorTerm' is open, showing options like 'Add', 'Drop', 'PrettyPrint', 'Render', etc., with 'Approximate Series Convergence' highlighted. To the right is a 'Series Convergence Iterator' dialog box displaying a table with columns 'ms', 'Count', and 'Result'.

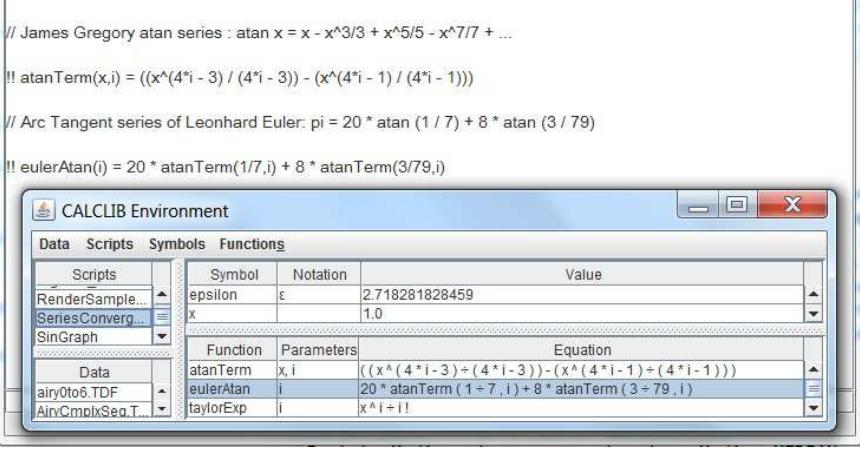
A tool for monitoring series convergence is also found in the function menu. A function that computes a series term based on the iteration number must be defined in the Function table. The iterator will accept parameters for the range and increment of the iteration number. The table built will then show the iteration count and the resulting sum generated from that count of iterations.



The screenshot shows the CALCLIB interface with a 'Series Convergence Iterator' dialog box. The dialog displays a table with columns 'ms', 'Count', and 'Result' for 5 terms of the series.

Another Example

This one shows the James Gregory ATAN series being used with the Leonhard Euler PI computation from integer ratios

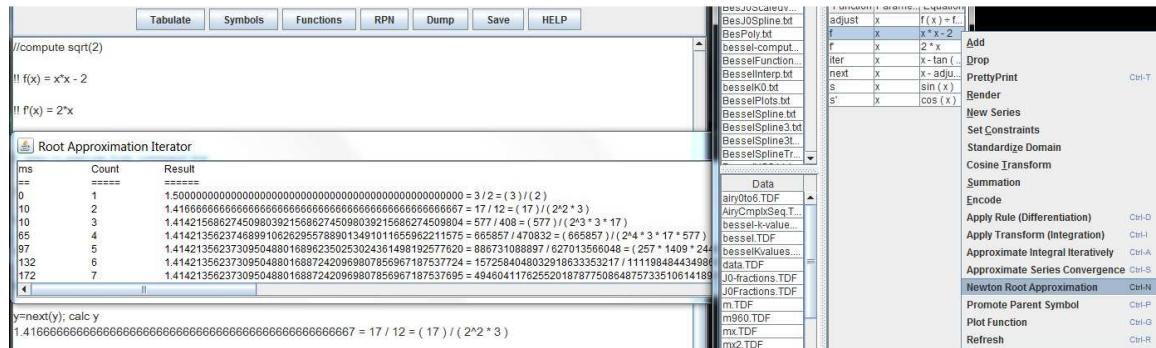


The screenshot shows the CALCLIB Environment with the 'Functions' table open. It contains definitions for 'atanTerm', 'eulerAtan', and 'taylorExp'.

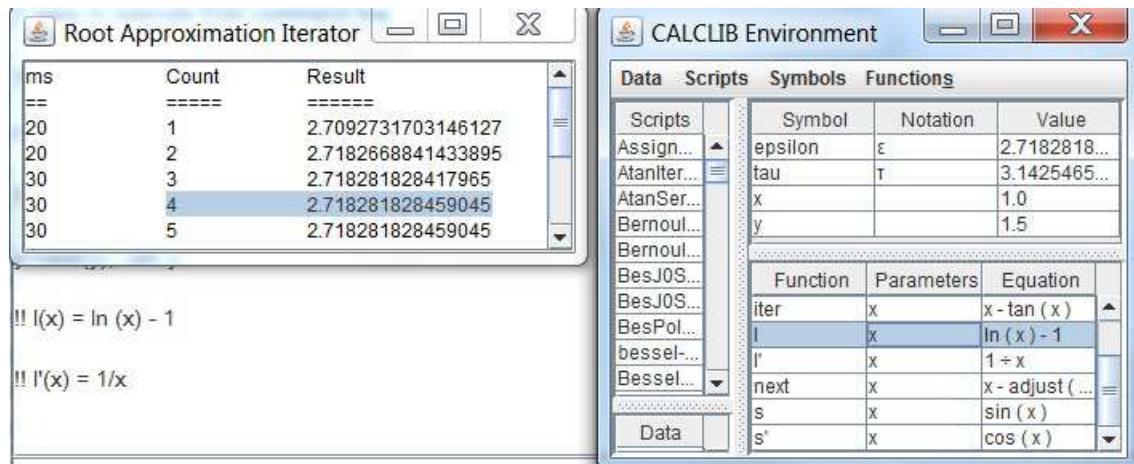
This shows that the convergence of 5 terms of the series (10 alternating sign terms) is adequate to generate 16 digits of PI correctly

Root Approximation

Root approximation using the Newton's rule is the implemented mechanism of the "Newton Root Approximation" functionality found in the Function List pop-up menu. The selected function must have a corresponding derivative function declaration ($F'(x)$) for a selected function declared as $f(x)$), the iterative adjustments are then $f(x)/f'(x)$. The user is prompted for the initial value of the approximation. This selection will drive the root that is found when multiple roots exist for the equation.



The example below shows the computation of “e” from solution of the root of the equation “ $\ln(x) - 1 = 0$ ”. Note that the solution is found in only four terms providing 16 correct digits of the value of the constant.



LinAlgDemo.txt

```
// this is an example of functional import from an external library.  
// Colt is a library of advanced mathematical functionality published as open source by CERN.  
// one simple class is added to each package to make desired functions available for import.  
// the LIBRARY command is used to recognize the items as available for import.  
// the functions can then be used exactly as if they were internally realized.  
  
IMPORT simple simple.TDF  
  
// this script imports the Colt library functions  
READ ColtLib.txt  
  
// MatRpt is the interface entry to the Colt Algebra class  
calc MatRpt simple  
  
// having used the Colt library to execute SV decomposition,  
// the by-products of the decomposition can be used as internal objects  
// Verification of SVD matrices generated by ColtLib  
  
S = GetSvdS()  
V = GetSvdV()  
U = GetSvdU()  
  
PRETTYPRINT S 5  
PRETTYPRINT V 5  
PRETTYPRINT U 5  
  
// to reconstitute the matrix the transpose of V is needed  
Vt= TRANSPOSE(V)  
prd = U * S * Vt  
  
PRETTYPRINT prd  
  
// the product should match the original  
PRETTYPRINT simple
```

OUTPUT:

```
READ ColtLib.txt  
Reading... C:\workspace\MathFields\scripts\ColtLib.txt  
  
READ LinAlgDemo.txt  
Reading... C:\workspace\MathFields\scripts\LinAlgDemo.txt  
Import from file: C:\workspace\MathFields\data\simple.TDF  
Import to matrix: simple  
Columns per row: 3  
Rows read: 3  
  
data read:  
 1   -3    2  
 8    6   -3  
 -2    1    7  
  
=EOD=
```

OUTPUT FROM COLT LIBRARY:

```
A = 3 x 3 matrix
1 -3  2
8  6 -3
-2 1  7

cond      : 3.395431804039953
det       : 235.0
norm1     : 12.0
norm2     : 11.221729908134112
normF     : 13.30413469565007
normInfinity : 17.0
rank      : 3
trace     : 14.0

density      : 1.0
isDiagonal   : false
isDiagonallyDominantByColumn : false
isDiagonallyDominantByRow   : false
isIdentity    : false
isLowerBidiagonal : false
isLowerTriangular  : false
isNonNegative   : false
isOrthogonal    : false
isPositive     : false
isSingular     : false
isSkewSymmetric : false
isSquare       : true
isStrictlyLowerTriangular : false
isStrictlyTriangular  : false
isStrictlyUpperTriangular : false
isSymmetric    : false
isTriangular   : false
isTridiagonal  : false
isUnitTriangular : false
isUpperBidiagonal : false
isUpperTriangular : false
isZero        : false
lowerBandwidth : 2
semiBandwidth : 3
upperBandwidth : 2
```

```
-----  
LUDecompositionQuick(A) --> isNonSingular(A), det(A), pivot, L, U, inverse(A)  
-----
```

```
isNonSingular = true  
det = 235.0  
pivot = [1, 0, 2]
```

```
L = 3 x 3 matrix  
1 0 0  
0.125 1 0  
-0.25 -0.666667 1
```

```
U = 3 x 3 matrix  
8 6 -3  
0 -3.75 2.375  
0 0 7.833333
```

```
inverse(A) = 3 x 3 matrix  
0.191489 0.097872 -0.012766  
-0.212766 0.046809 0.080851  
0.085106 0.021277 0.12766
```

```
-----  
QRDecomposition(A) --> hasFullRank(A), H, Q, R, pseudo inverse(A)  
-----
```

```
hasFullRank = true
```

```
H = 3 x 3 matrix  
1.120386 0 0  
0.963087 1.942239 0  
-0.240772 0.334942 2
```

```
Q = 3 x 3 matrix  
-0.120386 0.826813 -0.549442  
-0.963087 -0.231508 -0.137361  
0.240772 -0.512624 -0.824163
```

```
R = 3 x 3 matrix  
-8.306624 -5.176592 4.333891  
0 -4.382111 -1.24022  
0 0 -6.455947
```

```
pseudo inverse(A) = 3 x 3 matrix  
0.191489 0.097872 -0.012766  
-0.212766 0.046809 0.080851  
0.085106 0.021277 0.12766
```

```
-----  
CholeskyDecomposition(A) --> isSymmetricPositiveDefinite(A), L, inverse(A)  
-----
```

```
isSymmetricPositiveDefinite = false
```

```
L = 3 x 3 matrix  
1 0 0  
8 0 0  
-2 Infinity 0
```

```
inverse(A) = 3 x 3 matrix  
Infinity -Infinity NaN  
-Infinity Infinity NaN  
Infinity -Infinity NaN
```

EigenvalueDecomposition(A) --> D, V, realEigenvalues, imagEigenvalues

realEigenvalues = 1 x 3 matrix

3.870848 3.870848 6.258303

imagEigenvalues = 1 x 3 matrix

4.750436 -4.750436 0

D = 3 x 3 matrix

3.870848 4.750436 0

-4.750436 3.870848 0

0 0 6.258303

V = 3 x 3 matrix

0.106036 -1.021459 0.381526

-1.342451 0.779749 0.009024

0.564717 -0.044746 1.016627

SingularValueDecomposition(A) --> cond(A), rank(A), norm2(A), U, S, V

cond = 3.395431804039953

rank = 3

norm2 = 11.221729908134112

U = 3 x 3 matrix

-0.161259 -0.054457 0.985409

0.898826 -0.420443 0.123855

-0.407564 -0.905683 -0.116748

S = 3 x 3 matrix

11.22173 0 0

0 6.33641 0

0 0 3.304949

V = 3 x 3 matrix

0.699043 -0.253556 0.668616

0.487373 -0.515272 -0.704956

-0.523265 -0.81866 0.236621

CALCLIB OUTPUT:

// Verification of SVD matrices generated by ColtLib

S =

```
11.22173      0      0
  0   6.33641      0
  0      0   3.30495
```

V =

```
0.69904 -0.25356  0.66862
0.48737 -0.51527 -0.70496
-0.52327 -0.81866  0.23662
```

U =

```
-0.16126 -0.05446  0.98541
0.89883 -0.42044  0.12385
-0.40756 -0.90568 -0.11675
```

prd =

```
0.999999999999998 -3.000000000000001      2
7.999999999999964  5.99999999999999 -2.999999999999973
-1.99999999999998  1.000000000000042  7.000000000000036
```

simple =

```
1   -3    2
8     6   -3
-2    1    7
```

LinAlgCmplxDemo.txt

```
// example of exchange of data with external libraries understanding representation within each

IMPORT simple simple.TDF

READ ColtLib.txt

calc MatRpt simple

D = GetEvdD()
V = GetEvdV()
Dr = GetEvdDreal()
Di = GetEvdDimag()

PRETTYPRINT D 5
PRETTYPRINT V 5

// real and imaginary parts are separated in the Colt library
// this simple expression allows further expression to use complex values
eigenvalues = Dr + i * Di

PRETTYPRINT eigenvalues 5
```

OUTPUT:

D =

```
 3.87085  4.75044      0
 -4.75044  3.87085      0
      0      0    6.2583
```

V =

```
 0.10604  -1.02146   0.38153
 -1.34245   0.77975   0.00902
  0.56472  -0.04475   1.01663
```

eigenvalues =

```
[
  (3.87085 + 4.75044*i)
  (3.87085 - 4.75044*i)
  6.2583
]
```

legendre-iterator.txt

(run in domain of real numbers)

// using real numbers to build equations has the disadvantage of lost decimal places

READ legendre-iterator.txt

Reading... C:\workspace\MathFields\scripts\legendre-iterator.txt

Reading... C:\workspace\MathFields\scripts\legendre-script.txt

pn =

$$- 0.5 + 1.5 * x^2$$

pn =

$$- 1.5 * x + 2.5 * x^3$$

pn =

$$0.375 - 3.75 * x^2 + 4.375 * x^4$$

pn =

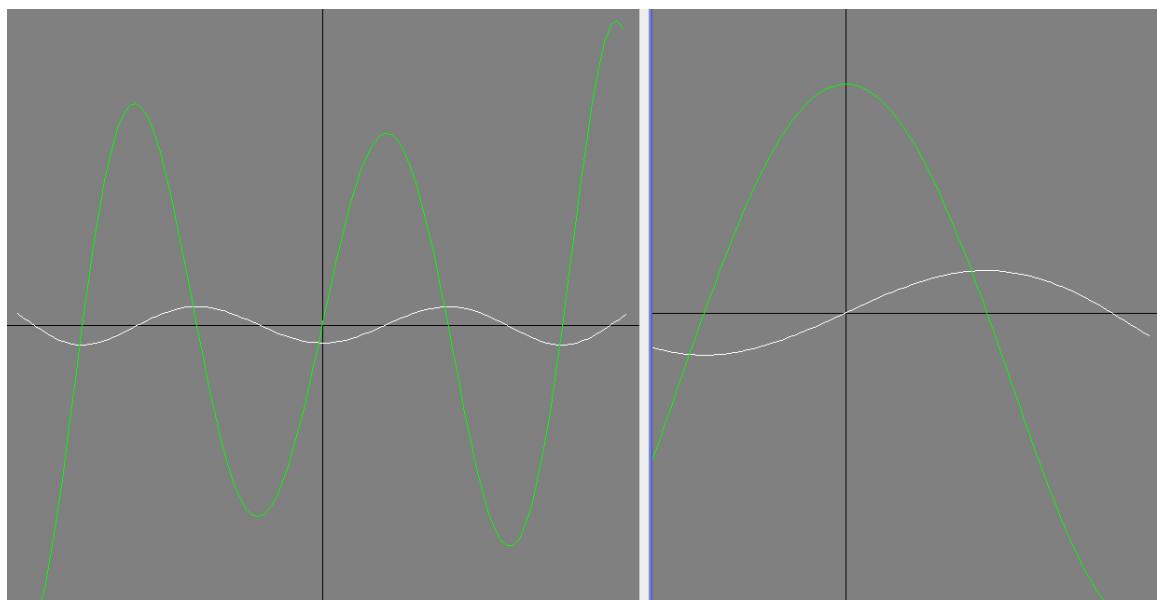
$$1.875 * x - 8.75 * x^3 + 7.875 * x^5$$

pn =

$$- 0.3125 + 6.5625 * x^2 - 19.6875 * x^4 + 14.4375 * x^6$$

Iteration 5 has completed

*** Maximum iteration count exceeded



legendre-iterator.txt

(run in domain of prime factored number fractions)

```
// this example shows the advantage of lossless computation using prime factored integer ratios
```

```
READ legendre-iterator.txt
Reading... C:\workspace\MathFields\scripts\legendre-iterator.txt
Reading... C:\workspace\MathFields\scripts\legendre-script.txt
```

```
pn =
```

$$- 2^{-1} + 2^{-1} * 3 * x^2$$

```
pn =
```

$$- 2^{-1} * 3 * x + 2^{-1} * 5 * x^3$$

```
pn =
```

$$2^{-3} * 3 - 2^{-2} * 3 * 5 * x^2 + 2^{-3} * 5 * 7 * x^4$$

```
pn =
```

$$2^{-3} * 3 * 5 * x - 2^{-2} * 5 * 7 * x^3 + 2^{-3} * 3^2 * 7 * x^5$$

```
pn =
```

$$- 2^{-4} * 5 + 2^{-4} * 3 * 5 * 7 * x^2 - 2^{-4} * 3^2 * 5 * 7 * x^4 + 2^{-4} * 3 * 7 * 11 * x^6$$

```
Iteration 5 has completed
*** Maximum iteration count exceeded
```

```
// the loss of decimal places in this small example may seem unimportant,
// but the point is to consider extrapolation to larger systems of equations
```

FactoredDemoPrimes.txt

```
// this demonstration shows the value of evaluating complicated expressions using ratios of integers

READ FactoredDemoPrimes.txt
Reading... C:\workspace\MathFields\scripts\FactoredDemoPrimes.txt
Reading... C:\workspace\MathFields\scripts\FactoredDemoPrimes.TXT

SCRIPTPRINT FactoredDemoPrimes.TXT

// the first example will compute SQRT(7) as a ratio of integer prime factors

x = 7
SCRIPTPRINT sqrtCompute.txt
SCRIPTPRINT sqrtIteration.txt
READ sqrtCompute.txt

n=1
SCRIPTPRINT ComputeNthBernoulli.txt
ITERATE 30 ComputeNthBernoulli.txt

Reading... C:\workspace\MathFields\scripts\sqrtCompute.txt

// looking for 10 decimal places of precision

TOLERANCE = 10^(-10)

// polynomial solutions "c - x^2 = 0" gives sqrt(c) as a solution

sqrt_polynomial = (x,0,-1)

polyprint sqrt_polynomial

// Newton's methods uses the first derivative for the solution

sqrt_poly_derivative = POLYDER sqrt_polynomial

polyprint sqrt_poly_derivative

// initial approximation

sqrtx = 1

ITERATE 10 SqrtIteration.txt

Reading... C:\workspace\MathFields\scripts\sqrtIteration.txt

// compute polynomial value

f = sqrt_polynomial +*^ sqrtx

// compute derivative value

fprime = sqrt_poly_derivative +*^ sqrtx

// iteration contribution computed

sqrt_iteration = f/fprime
```

```
// new value for this iteration
sqrtx = sqrtx - sqrt_iteration

// check status of result
sqrtx_squared = sqrtx ^ 2

// display intermediate results
PRETTYPRINT sqrt_iteration
PRETTYPRINT sqrtx_squared
PRETTYPRINT sqrtx

// look for convergence
ASSERT "Convergence Complete" TOLERANCE >|| sqrt_iteration

sqrt_polynomial =
    7 - x^2

sqrt_poly_derivative =
    - 2*x
```

Reading... C:\workspace\MathFields\scripts\SqrtIteration.txt

sqrt_iteration =

$$-3 = -3 / 1 = (-3) / (1)$$

sqrtx_squared =

$$16 = 16 / 1 = (2^4) / (1)$$

sqrtx =

$$4 = 4 / 1 = (2^2) / (1)$$

sqrt_iteration =

$$1.125 = 9 / 8 = (3^2) / (2^3)$$

sqrtx_squared =

$$8.265625 = 529 / 64 = (23^2) / (2^6)$$

sqrtx =

$$2.875 = 23 / 8 = (23) / (2^3)$$

sqrt_iteration =

$$0.220108695652174 = 81 / 368 = (3^4) / (2^4 * 23)$$

sqrtx_squared =

$$7.048447837901701 = 954529 / 135424 = (977^2) / (2^8 * 23^2)$$

sqrtx =

$$2.654891304347826 = 977 / 368 = (977) / (2^4 * 23)$$

sqrt_iteration =

$$0.0091242601575364 = 6561 / 719072 = (3^8) / (2^5 * 23 * 977)$$

sqrtx_squared =

$$7.000083252123423 = 3619494835009 / 517064541184 = (1902497^2) / (2^{10} * 23^2 * 977^2)$$

sqrtx =

$$2.64576704419029 = 1902497 / 719072 = (1902497) / (2^5 * 23 * 977)$$

```

sqrt_iteration =
1.57330789204E-5 = 43046721 / 2736064645568 = ( 3^16 ) / ( 2^6 * 23 * 977 * 1902497 )

sqrtx_squared =
7.00000000024753 = 52402348214943038423150209 / 7486049744727145462042624
= ( 127^2 * 449^2 * 126947839^2 ) / ( 2^12 * 23^2 * 977^2 * 1902497^2 )

sqrtx =
2.6457513111113693 = 7238946623297 / 2736064645568
= ( 127 * 449 * 126947839 ) / ( 2^6 * 23 * 977 * 1902497 )

sqrt_iteration =
4.67788E-11 = 1853020188851841 / 39612451854313553433195392
= ( 3^32 ) / ( 2^7 * 23 * 127 * 449 * 977 * 1902497 * 126947839 )

sqrtx_squared =
7 = 10984024393372164945081646087753767584561255399324929 /
1569146341910309277868316057704782153153799650033664
= ( 31231^2 * 313727^2 * 10696531080798721^2 ) /
( 2^14 * 23^2 * 127^2 * 449^2 * 977^2 * 1902497^2 * 126947839^2 )

sqrtx =
2.6457513110645907
= 104804696428033056657448577 /
39612451854313553433195392
= ( 31231 * 313727 * 10696531080798721 ) /
( 2^7 * 23 * 127 * 449 * 977 * 1902497 * 126947839 )

```

Script interrupted in iteration 6
Assertion "Convergence Complete" has been validated, TOLERANCE >|| sqrt_iteration
*** Script has terminated

```
// this example will compute the first 30 Bernoulli sequence numbers.  
// using prime factored fractions complicated values can be computed without loss.
```

```
Reading... C:\workspace\MathFields\scripts\ComputeNthBernoulli.txt  
bn = BERNOULLI(n)  
PRETTYPRINT bn  
n = n + 1
```

```
bn =  
0.5 = 1 / 2 = ( 1 ) / ( 2 )
```

```
bn =  
0.1666666666666667 = 1 / 6 = ( 1 ) / ( 2 * 3 )
```

```
bn =  
0  
  
bn =  
-0.033333333333334 = -1 / 30 = ( -1 ) / ( 2 * 3 * 5 )
```

```
bn =  
0
```

```
bn =  
0.0238095238095239 = 1 / 42 = ( 1 ) / ( 2 * 3 * 7 )
```

```
bn =  
0  
  
bn =  
-0.033333333333334 = -1 / 30 = ( -1 ) / ( 2 * 3 * 5 )
```

```
bn =  
0  
  
bn =  
0.0757575757575758 = 5 / 66 = ( 5 ) / ( 2 * 3 * 11 )
```

bn =

0

bn =

$$-0.2531135531135532 = -691 / 2730 = (-691) / (2 * 3 * 5 * 7 * 13)$$

bn =

0

bn =

$$1.1666666666666667 = 7 / 6 = (7) / (2 * 3)$$

bn =

0

bn =

$$-7.092156862745098 = -3617 / 510 = (-3617) / (2 * 3 * 5 * 17)$$

bn =

0

bn =

$$54.971177944862156 = 43867 / 798 = (43867) / (2 * 3 * 7 * 19)$$

bn =

0

bn =

$$-529.1242424242424 = -174611 / 330 = (-283 * 617) / (2 * 3 * 5 * 11)$$

bn =

0

bn =

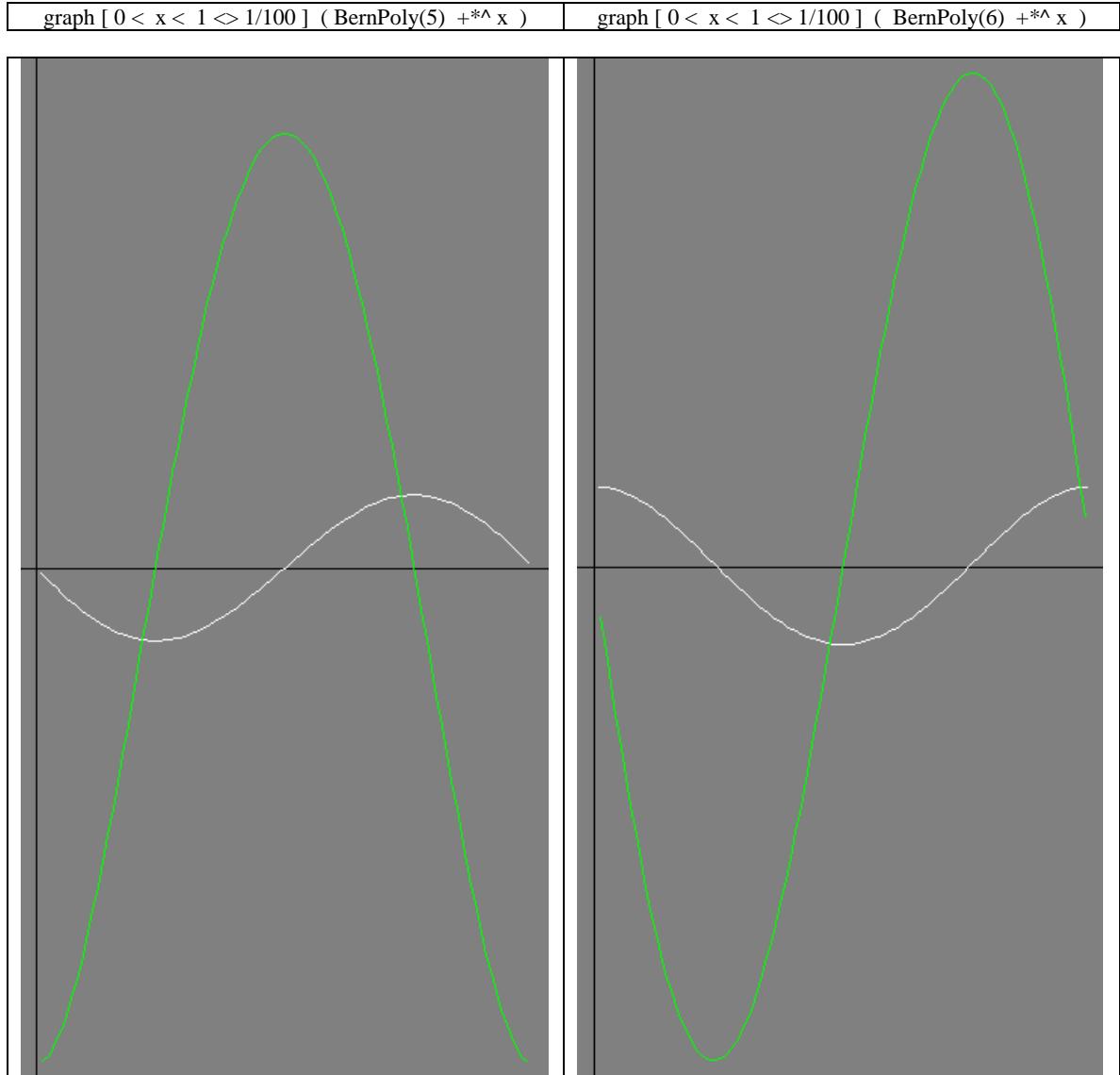
$$6192.123188405797 = 854513 / 138 = (11 * 131 * 593) / (2 * 3 * 23)$$

```
bn =  
0  
  
bn =  
-86580.25311355312 = -236364091 / 2730 = ( - 103 * 2294797 ) / ( 2 * 3 * 5 * 7 * 13 )  
  
bn =  
0  
  
bn =  
1425517.1666666667 = 8553103 / 6 = ( 13 * 657931 ) / ( 2 * 3 )  
  
bn =  
0  
  
bn =  
-2.7298231067816094E7 = -23749461029 / 870 = ( - 7 * 9349 * 362903 ) / ( 2 * 3 * 5 * 29 )  
  
bn =  
0  
  
bn =  
6.015808739006424E8 = 8615841276005 / 14322 = ( 5 * 1721 * 1001259881 ) / ( 2 * 3 * 7 * 11 * 31 )  
  
Iteration 30 has completed  
*** Maximum iteration count exceeded
```

// the correct values of the numerators and denominators of these ratios are published online:
// <https://oeis.org/> ([A027641](#) / [A027642](#) in [OEIS](#));

BernoulliPolynomialDerivation.txt

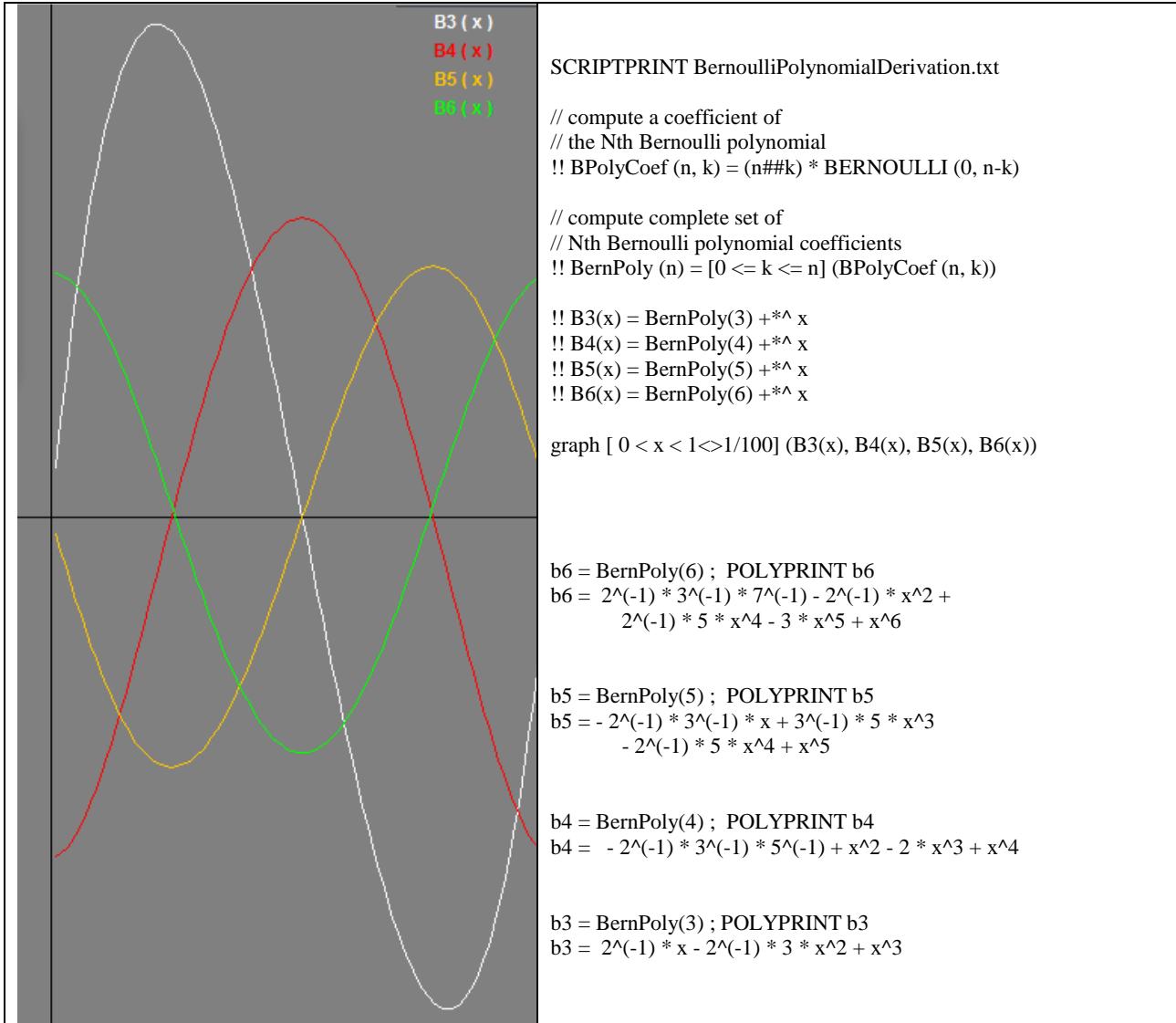
```
// compute a coefficient of the Nth Bernoulli polynomial  
!! BPolyCoef ( n, k ) = ( n ## k ) * BERNOULLI ( 0, n - k )  
  
// compute complete set of Nth Bernoulli polynomial coefficients  
!! BernPoly ( n ) = [0 <= k <= n] ( BPolyCoef ( n, k ) )
```



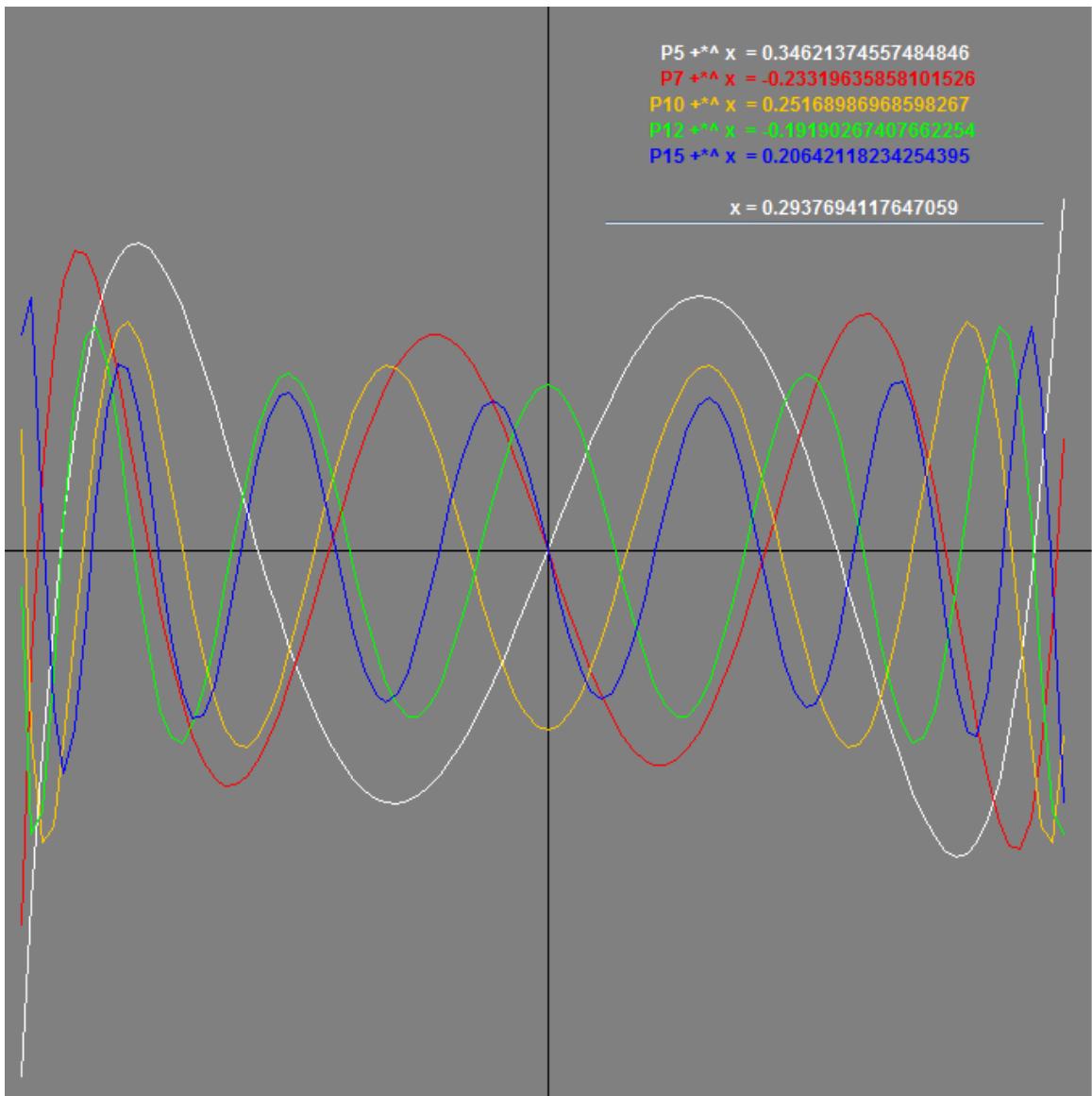
Functions

$$\begin{aligned}B6(x) &= 2^{-1} * 3^{-1} * 7^{-1} - 2^{-1} * x^2 + 2^{-1} * 5 * x^4 - 3 * x^5 + x^6 \\B5(x) &= -2^{-1} * 3^{-1} * x + 3^{-1} * 5 * x^3 - 2^{-1} * 5 * x^4 + x^5\end{aligned}$$

Bernoulli Polynomials 3, 4, 5, and 6 overlaid



Other Overlaid Series Plots

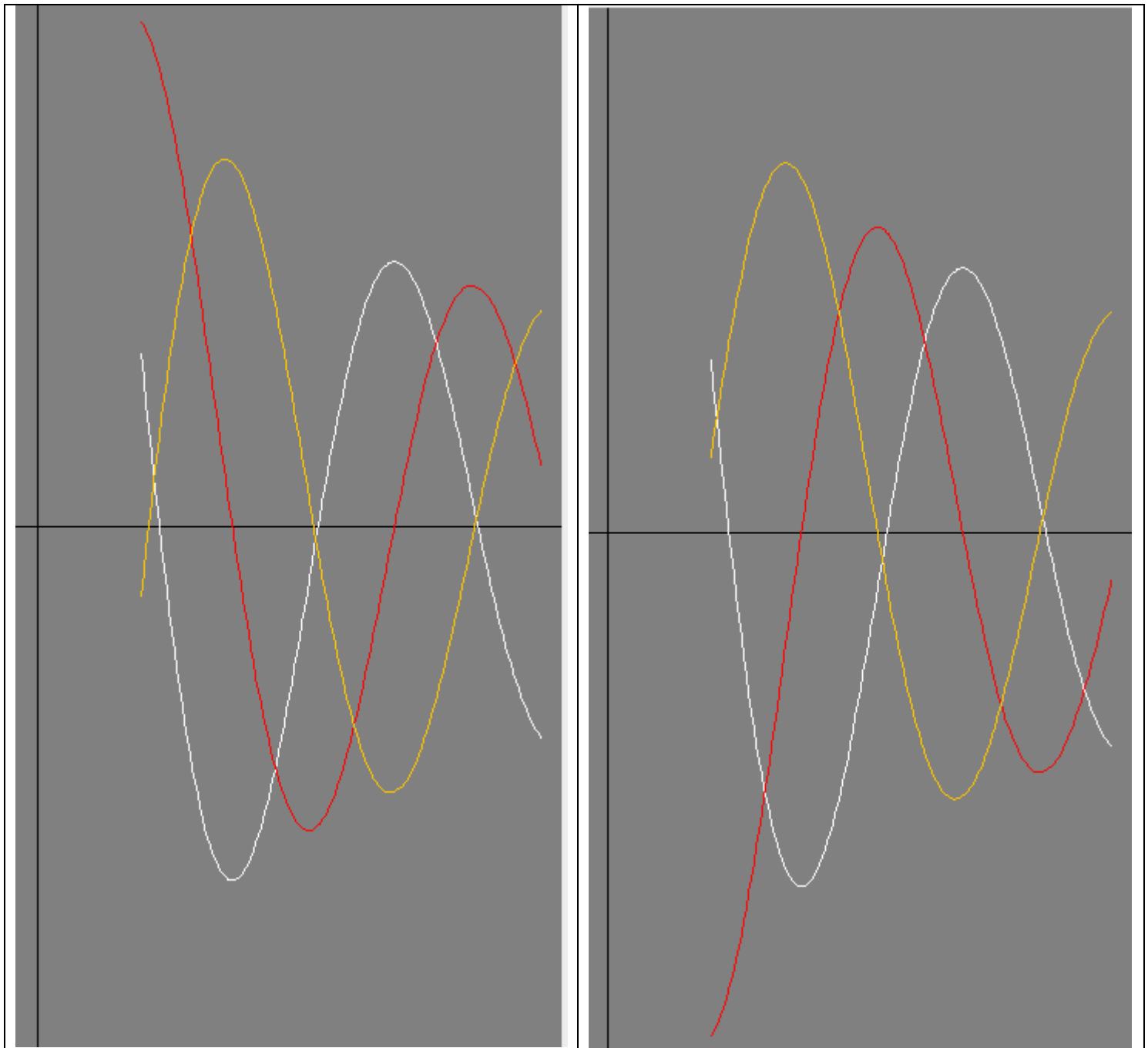


Legendre Polynomials 5, 7, 10, 12, and 15

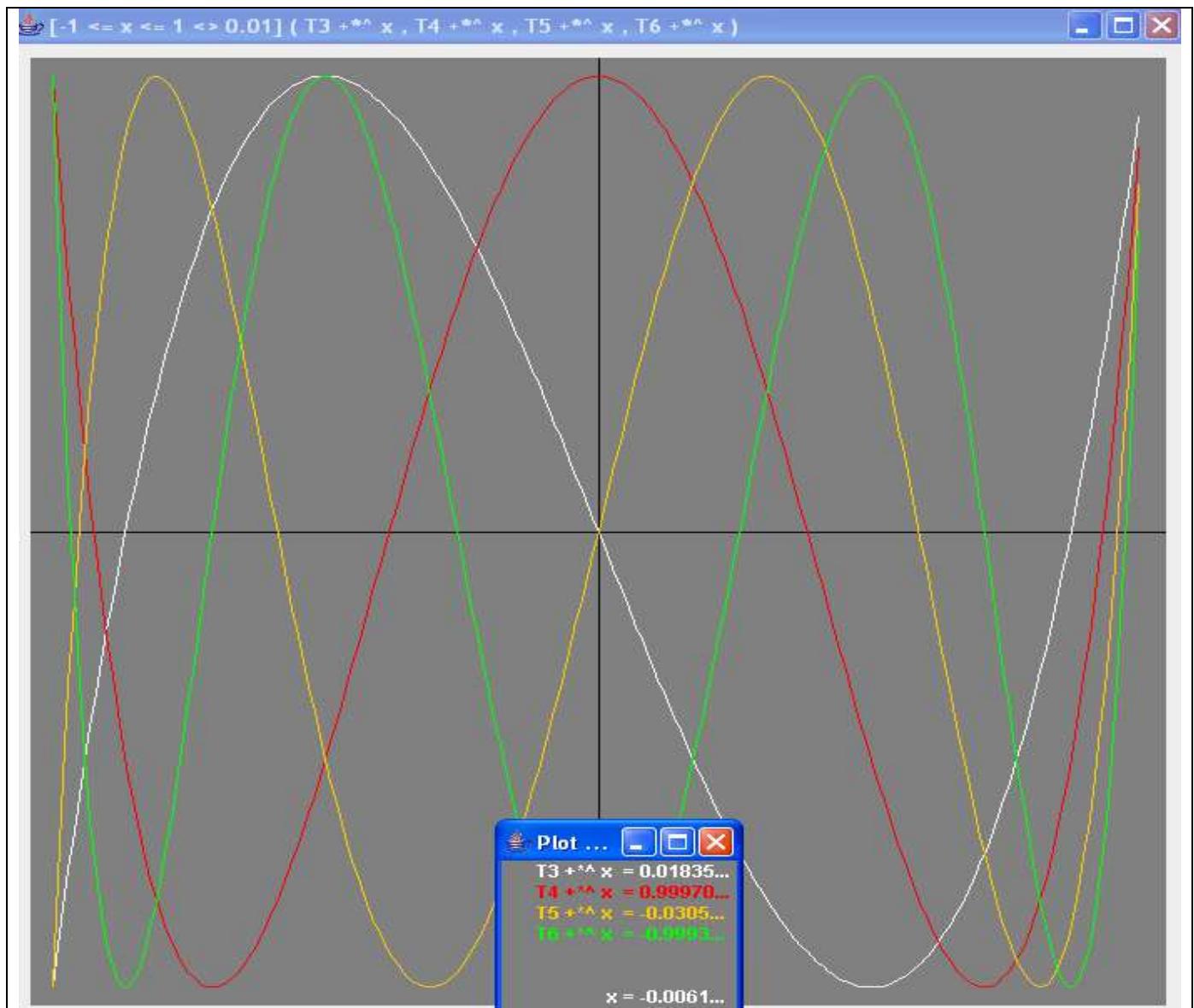
<pre>Legendre-Iterations.txt READ legendre-init.txt READ legendre-script.txt; P2 = pn READ legendre-script.txt; P3 = pn READ legendre-script.txt; P4 = pn READ legendre-script.txt; P5 = pn READ legendre-script.txt; P6 = pn READ legendre-script.txt; P7 = pn READ legendre-script.txt; P8 = pn</pre>	<pre>Legendre-Init.txt x = (0,1); p0 = (1); p1 = x; n = 1 pn = p1; pnm1 = p0 Legendre-Script.txt m = 2*n + 1; invNp1 = 1 / (n + 1); pnp1 = m*CONV(x,pn) pnm1 = APPEND (pnm1, 0, 0); pmm1 = n * pnm1 pnp1 = pnp1 - pnm1; pnp1 = pnp1 * invNp1 pnm1 = pn; pn = pnp1; n = n + 1 POLYPRT pn</pre>
---	--

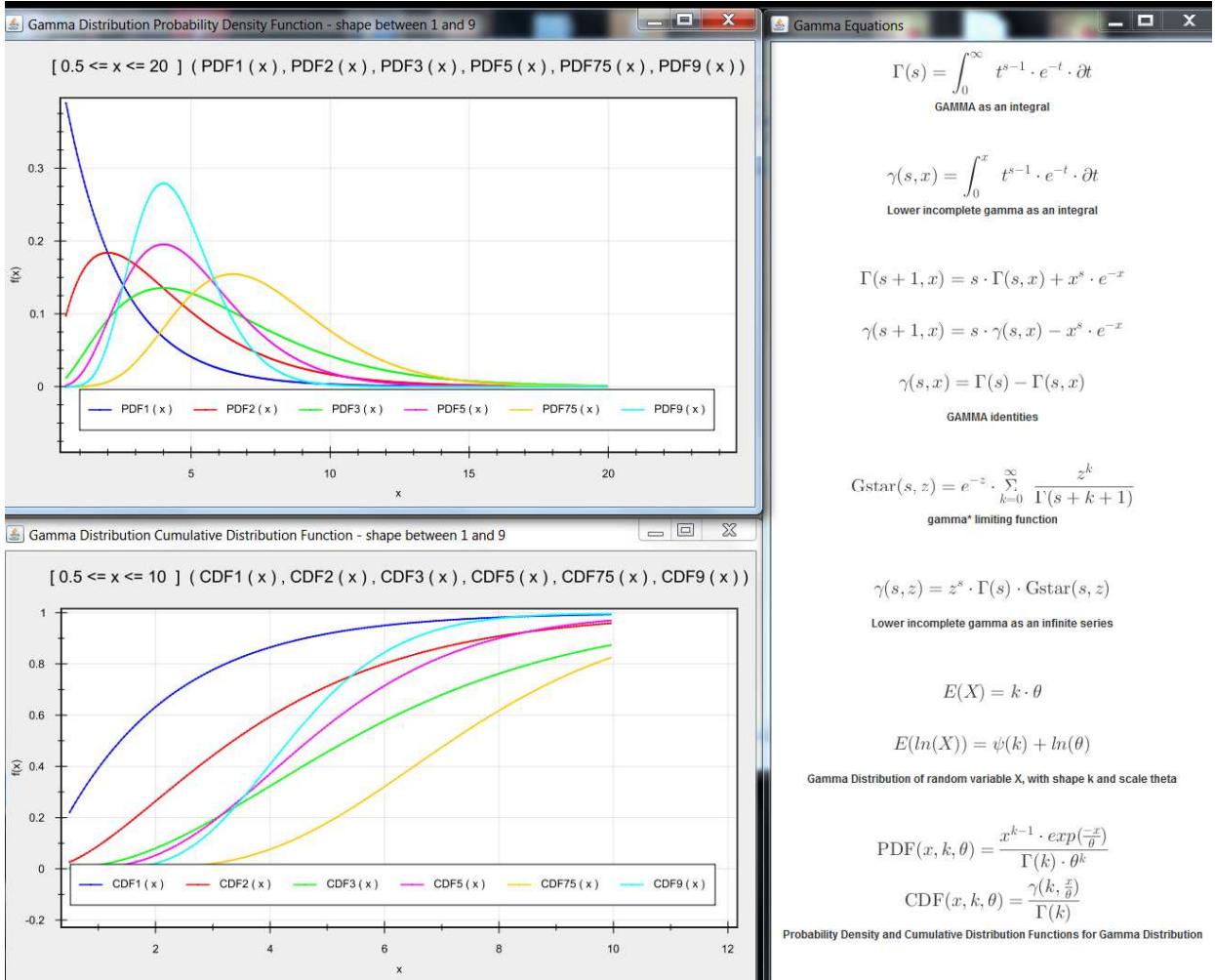
Bessel J0, J1, and Y1
(using imported Colt library functions)

graph [2<x<10 <> 0.05]
(J0(x), J0'(x <> 0.01), J0''(x <> 0.01))



Chebyshev Polynomials





Extended Function Libraries

```
READ ClmathTests.txt
Reading... C:\Users\Michael\workspace\CalcLib\scripts\ClmathTests.txt

loggammaIntegralExpected =
-0.04285374065101921

lgixhalf =
-0.0428597792507503

error =
-6.038599731091054E-6

lgix1 =
0.9183861223328741

error =
-5.52410871798581E-4
```

The Hurwitz zeta function generalizes the Riemann zeta function

$$\zeta(s, a) = \sum_{n=1}^{\infty} \frac{1}{(n + a)^s}$$

$\zeta(s, a) = \frac{1}{\Gamma(s)} \cdot \int_0^{\infty} \frac{x^{s-1} \cdot e^{-ax}}{1 - e^{-x}} \cdot dx$

$$\zeta(s, a) = \frac{1}{\Gamma(s)} \cdot \int_0^{\infty} \frac{x^{s-1} \cdot e^{(1-a)x}}{e^x - 1} \cdot dx$$

The Hurwitz zeta function represented as an integral

$$\text{loggammaIntegral}(x) = \int_0^1 \ln \Gamma(x) \cdot dx$$

$$\text{loggammaIntegral}(x) = \int_0^{0.5} \ln \Gamma(x+1) \cdot dx$$

Log Gamma identity integrals (per <https://mathworld.wolfram.com/LogGammaFunction.html>)

ClmathTests.txt

```
INFINITY = 1E4
dx = 1/INFINITY

!! zeta (s, a) = SIGMA [ 1 <= n <= INFINITY ] ( 1 / ( n + a ) ^ s )
RENDERF zeta
RENDER "The Hurwitz zeta function generalizes the Riemann zeta function" TOP

dx = 1 / INFINITY
!! zeta (s, a) = 1/( GAMMA s ) * INTEGRAL [ 0 <= x <= INFINITY <> dx ] ( ( ( x^(s-1) * exp (-a*x) ) / ( 1 - exp (-x) ) ) * <> x )
RENDERF zeta
!! zeta (s, a) = 1/( GAMMA s ) * INTEGRAL [ 0 <= x <= INFINITY <> dx ] ( ( ( x^(s-1) * exp ((1-a)*x) ) / ( exp (x) - 1 ) ) * <> x )
RENDERF zeta
RENDER "The Hurwitz zeta function represented as an integral" TOP

A = 1.2824271291
// per https://mathworld.wolfram.com/Glaisher-KinkelinConstant.html
loggammaIntegralExpected = -1/2 - 7/24*ln 2 + 1/4* ln pi + 3/2*ln A
prettyprint loggammaIntegralExpected

!! loggammaIntegral(x) = INTEGRAL [ 0 < x < 1 <> dx ] ( loggamma (x) * <> x )
!! loggammaIntegral(x) = INTEGRAL [ 0 <= x <= 0.5 <> dx ] ( loggamma (x+1) * <> x )
RENDERF loggammaIntegralx
RENDERF loggammaIntegral
RENDER "Log Gamma identity integrals (per https://mathworld.wolfram.com/LogGammaFunction.html)" TOP

lgixhalf = loggammaIntegral ()
prettyprint lgixhalf
error = lgixhalf - loggammaIntegralExpected
prettyprint error

lgix1 = loggammaIntegralx ()
prettyprint lgix1
lgix1Expected = ln (2*pi) / 2
error = lgix1 - lgix1Expected
prettyprint error
```

The Clmath Function Library

CLmath was a library constructed to be used as a demo for library extension mechanisms. Note that many of the functions being imported from CLmath were in other implementing classes. The CFG file identifies what libraries are to be included and selects the functions and/or operators to be referenced from that library. This allows multiple implementations to be available in the same session given different names. This could be used for instance to allow the user to check timing efficiency or precision of the two different implementations.

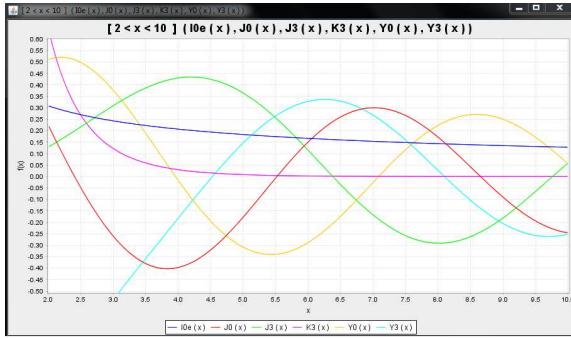
Note also that the same implementation can be instanced multiple times with different names. This allows the two instances to have different configurations. A good example is the configuration of the Jp function being constructed with p=2.5; this could be called Jp25 and another instance could be called Jp35 which could be configured with p=3.5

Configuration file importing CLMATH objects

- CLMATH extended library
- use version from CALCLIB original SF implementation
- E,clmath.net.myorb.math.expressions.algorithms.ClmathRealImplementations
- F,clmath,Unary conventional EXP function e<x>,Exp,exp
- O,clmath,Zeta function,Zeta,zeta,10,B6;,1000
- O,clmath,Zeta function,ZetaAnalytic,zeta,10,B6;,10000
- F,clmath,Gamma function,Gamma,GAMMA,Ɖ,1000
- F,clmath,Incomplete Gamma function,IncompleteGamma,gammainc
- F,clmath,Loggamma function,Loggamma,loggamma
- O,clmath,Euler Product function,EulerProduct,ep,10,B6;,1000
- O,clmath,Bernoulli Number,BernoulliNumber,B,10,B6;,1000
- O,clmath,Exponential Integral,ExponentialIntegral,Ei,10
- O,clmath,Logarithmic Integral,LogarithmicIntegral,Li,10
- O,clmath,E1 Exponential Integral,E1Integral,E1,10
- O,clmath,Bessel Jp,Bessel,Jp,10,,Jp;2.5;2
- F,clmath,Airy Ai,AiryAi,Ai
- F,clmath,Airy Bi,AiryBi,Bi

Colt Library

Colt is a set of [open-source](#) Libraries for High Performance Scientific and Technical Computing written in [Java](#) and developed at [CERN](#). Colt was developed with a focus on High Energy Physics, but is applicable to many other problems. Colt was last updated in 2004 (when Java 1.4 was the current release) and its code base has been incorporated into the [Parallel Colt](#) code base, which has received more recent development.



$$H^{(1)}_0(x) = J_0(x) + i \cdot Y_0(x)$$

$$H^{(2)}_0(x) = J_0(x) - i \cdot Y_0(x)$$

ColtLibTest.txt

```
!! J3(x) = Jn(3,x)
!! K3(x) = Kn(3,x)
!! Y3(x) = Yn(3,x)

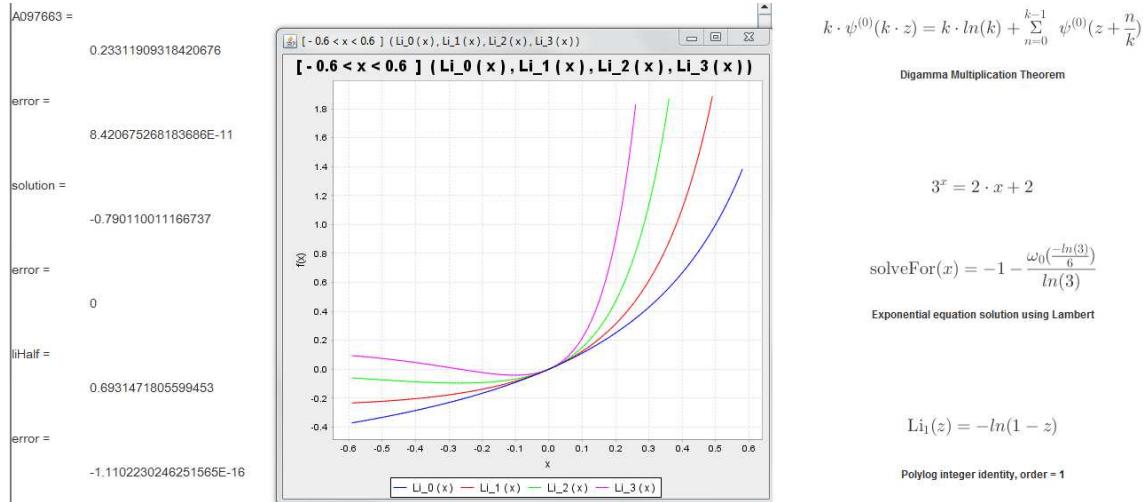
graph LIM 3 [ 2 < x < 10 <> 0.01 ] \
( I0e(x), J0(x), J3(x), K3(x), Y0(x), Y3(x) )

render H10(x)=J0(x)+i*Y0(x)
render H20(x)=J0(x)-i*Y0(x)
```

```
- Colt extended library
-
- use COLT library version
E,colt.net.myorb.math.coltlib.RealColtLibPrimitives
F,colt,Bessel Hankel function (first kind),H10,H10
F,colt,Bessel Hankel function (second kind),H20,H20
F,colt,Bessel I0 function,I0,I0
F,colt,Bessel I0e function,I0e,I0e
F,colt,Bessel II function,I1,I1
F,colt,Bessel I1e function,I1e,I1e
F,colt,Bessel J0 function,J0,J0
F,colt,Bessel K0 function,K0,K0
F,colt,Bessel Y0 function,Y0,Y0
F,colt,Bessel Jn function,Jn,Jn
F,colt,Bessel Kn function,Kn,Kn
F,colt,Bessel Yn function,Yn,Yn
```

Anaconda RPC Function Library

The Anaconda system is a Python platform that hosts an enormous mathematics library, notably the MPMATH and SciPy libraries are shipped with the default distribution of the system. The CalcTools system has been extended to access these library functions using client / server RPC concepts. The available functions list can be extended by adding to the Anaconda project and referring to the classes of that project in the configuration file as demonstrated below:



AnacondaTests.txt

```
INFINITY = 1E4
dx = 1/INFINITY

k = 0
RENDER k * psi0(k*z) = k * ln k + SIGMA [0 <= n <= k-1] ( psi0 ( z + n/k ) )
RENDER "Digamma Multiplication Theorem" TOP
gamma = 0.5772156649
A097663 = 3 * exp ( psi0 (1/3) + gamma )
prettyprint A097663
A097663_correct = 0.2331190931
error = A097663 - A097663_correct
prettyprint error

render 3^x = 2 * x + 2
!! solveFor (x) = -1 - omega0 ( - ln(3) / 6 ) / ln(3)
RENDERF solveFor
RENDER "Exponential equation solution using Lambert" TOP
correct_solution = -0.790110011166737
solution = solveFor (0)
prettyprint solution
error = solution - correct_solution
prettyprint error

RENDER Li1(z) = -ln (1 - z)
RENDER "Polylog integer identity, order = 1" TOP
liHalf = Li1 0.5
prettyprint liHalf
error = liHalf - ln 2
prettyprint error

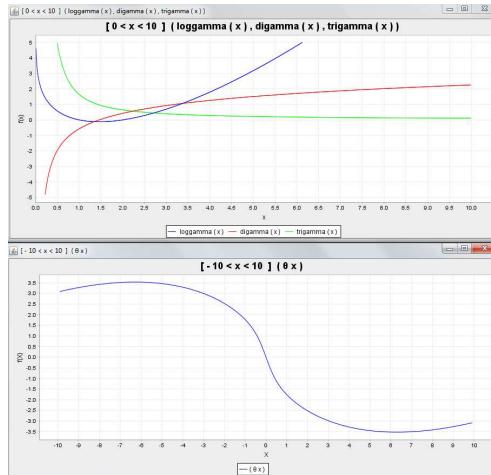
!! Li_0(x) = polylog ( 0, x)
!! Li_1(x) = polylog (-1, x)
!! Li_2(x) = polylog (-2, x)
!! Li_3(x) = polylog (-3, x)

GRAPH LIM 2 [ -0.6 < x < 0.6 <> 0.01 ] ( Li_0(x), Li_1(x), Li_2(x), Li_3(x) )
```

Complex.cfg

```
- Anaconda extended library
-
- use Anaconda project version with RPC
E,anaconda,net.myorb.math.anacoda.ComplexAnacondaRPC,LOCALHOST:8080
F,anaconda,Lambert_W (omega) function,Omega,omega0,&#03A9;,0
F,anaconda,Gamma function,Gamma,GAMMA,10,&#0393;
F,anaconda,Loggamma function,Loggamma,loggamma,10
F,anaconda,Polygamma (psi) function,Polygamma,psi0,&#03A8;,0
F,anaconda,Jonquiere polylog function,Polylog,Li1,,1
F,anaconda,Jonquiere polylog function,Polylog,polylog,,0
F,anaconda,Riemann Zeta function,Zeta,zeta,&#03B6;,1
```

Polygamma Study



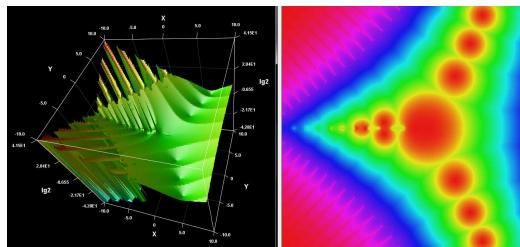
PolygammaReal.txt

```
!! loggamma(x) = ln (GAMMA x)
!! digamma(x) = loggamma' ( x <> 0.001 )
!! trigamma(x) = digamma' ( x <> 0.001 )

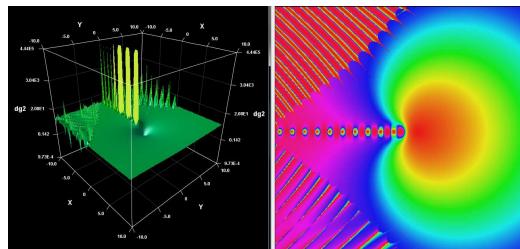
graph LIM 5 [0 < x < 10 <> 0.01]
( loggamma (x), digamma (x), trigamma (x) )
```

[real axis plot computed using complex arithmetic]

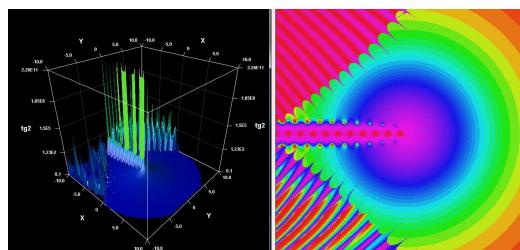
```
graph [0 < x < 10 <> 0.01] ( THETA x )
```



```
// Loggamma
!! lg2(x,y) = loggamma(x + i*y)
plotc lg2 (-10,-10) 20 100
plot3d lg2 (-10,-10) 20 20
```

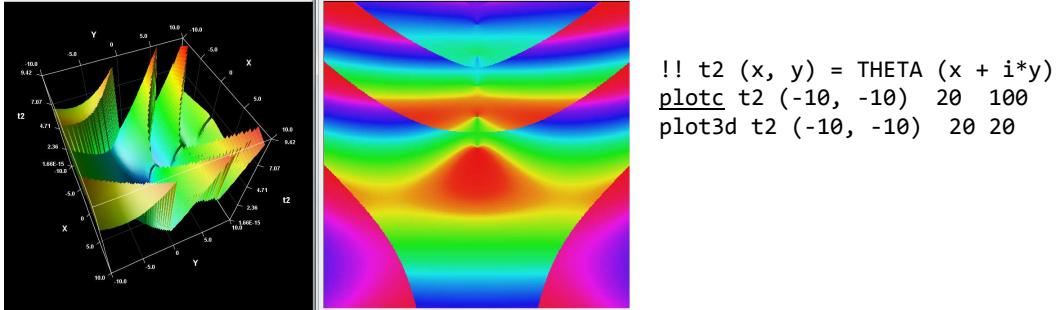


```
// Digamma
!! dg2(x,y) = digamma(x + i*y)
plotc dg2 (-10, -10) 20 100
plot3d dg2 (-10,-10) 20 20
```



```
// Trigamma
!! tg2(x,y) = trigamma(x + i*y)
plotc tg2 (-10, -10) 20 100
plot3d tg2 (-10,-10) 20 20
```

Theta Study

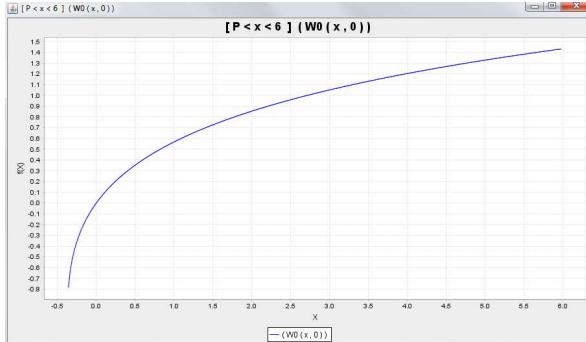


```
Complex u = CV (t.times (i).times (HALF).plus (C (Q)));
Complex v = CV (u.conjugate ());

if (d == 0)
{
    if (t.Im () != 0)
    {
        // # XXX: cancellation occurs
        Complex a = CV (polygamma.loggamma (u));           // (0.25+0.5j*t);
        Complex b = CV (polygamma.loggamma (v));           // (0.25-0.5j*t)
        Complex aMb = CV (a.plus (b.negate ())));

        return CV
        (
            t.times (HALF_LN_PI).plus
            (
                aMb.times (i).times (- HALF)
            )
        );
        // -ctx.ln(ctx.pi)/2*t - 0.5j*(a-b)
    }
    else
    {
        return CV
        (
            C (polygamma.loggamma (u).Im ()).plus
            (
                t.times (HALF_LN_PI)
            )
        );
        // ctx._im(ctx.loggamma(0.25+0.5j*t)) - ctx.ln(ctx.pi)/2*t
    }
}
```

Lambert Omega

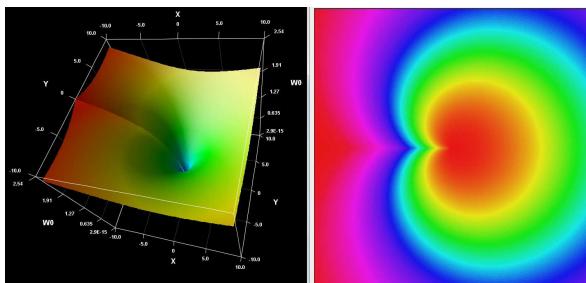


$$P = -1/e$$

```
!! W0 (x,y) = OMEGA ( x + y*i, 0 )
!! W0r (x,y) = Re ( OMEGA ( x + y*i, 0 ) )
!! W0i (x,y) = Im ( OMEGA ( x + y*i, 0 ) )

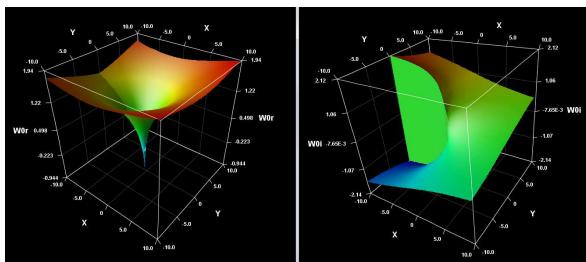
!! W1 (x,y) = OMEGA ( x + y*i, 1 )

// Plot W0 from pole at -1/e
graph [P < x < 6 <> 0.01] ( W0 (x, 0) )
```



Contour and 3D magnitude plots

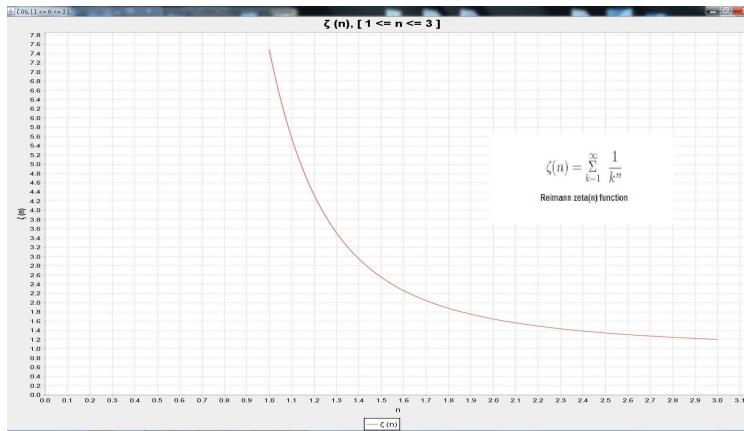
```
plotc W0 (-10, -10) 20 100
plot3d W0 (-10, -10) 20 20
```



```
// 3D plots of Re and Im parts
// downward spiral characteristic of
// imaginary part of OMEGA seen clearly
```

```
plot3d W0r (-10, -10) 20 20
plot3d W0i (-10, -10) 20 20
```

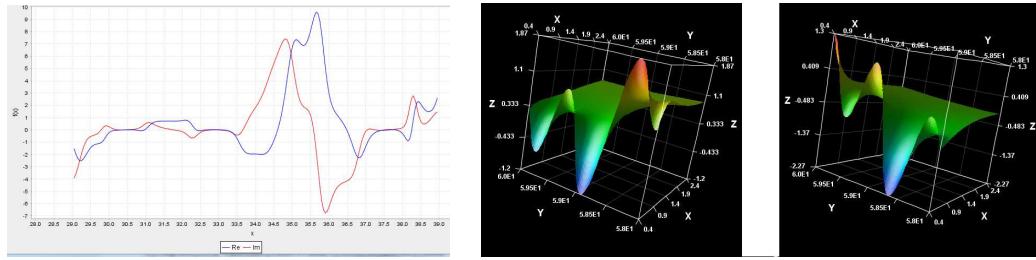
Zeta Function Study



Simple plot of Zeta on Real numbers domain

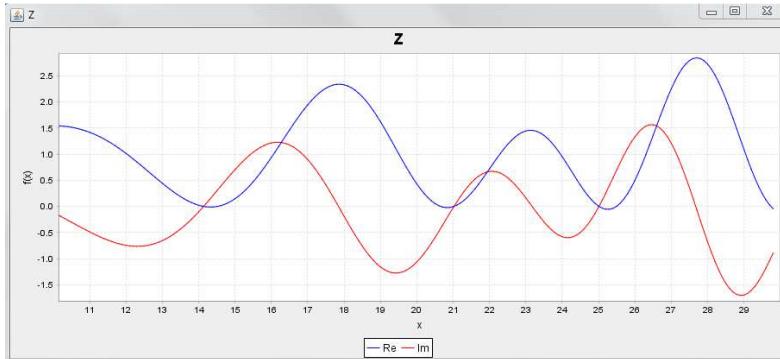
<pre>// ZetaGeneric.txt N = 1000 // log of first N integers // saves repeated calculations logN = [0 < n <= N] (ln n) // implementation of exp comes from configuration file !! GenericZeta (z) = SIGMA [0 <= n < N] (exp (- z * logN#n)) // ZetaGenericPlot.txt READ ZetaGeneric.txt // simple plot of specified domain // graph [1.1 < t < 5 < 0.1] (GenericZeta (t))</pre>	<pre>// ZetaComplex.txt READ ZetaGeneric.txt // domain of Reimann hypothesis // !! dom (t) = 1/2 + t*i // // use zeta implemented in generic script // rng = [0 <= x <= 30 <> 0.1] (GenericZeta (dom (x))) // // use code from CalcLib build // rng = [-10 <= x <= 10 <> 0.1] (zeta (dom (x)))</pre>	<pre>// // real and imaginary parts // separated into arrays // RE = [0 <= n < LENGTH rng] (Re (rng#n)) IM = [0 <= n < LENGTH rng] (Im (rng#n)) // // dump vectors of real and imag parts // calc RE calc IM // // plot both parts // graph [0 <= n < LENGTH rng] (RE#n,IM#n)</pre>
--	--	---

Plots of Zeta on Complex Plane



RI plot over domain
 $(1/2 + x*i)$ [29< x <39]

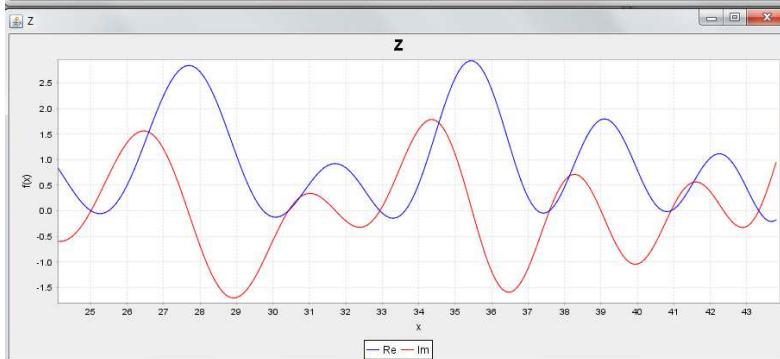
3D contour plots of
 $\operatorname{Re}(\zeta(x + i^*y))$ $\operatorname{Im}(\zeta(x + i^*y))$



Plots of the critical line

Zeta ($0.5 + i^*x$)

Zeros seen at 14, 21, & 25

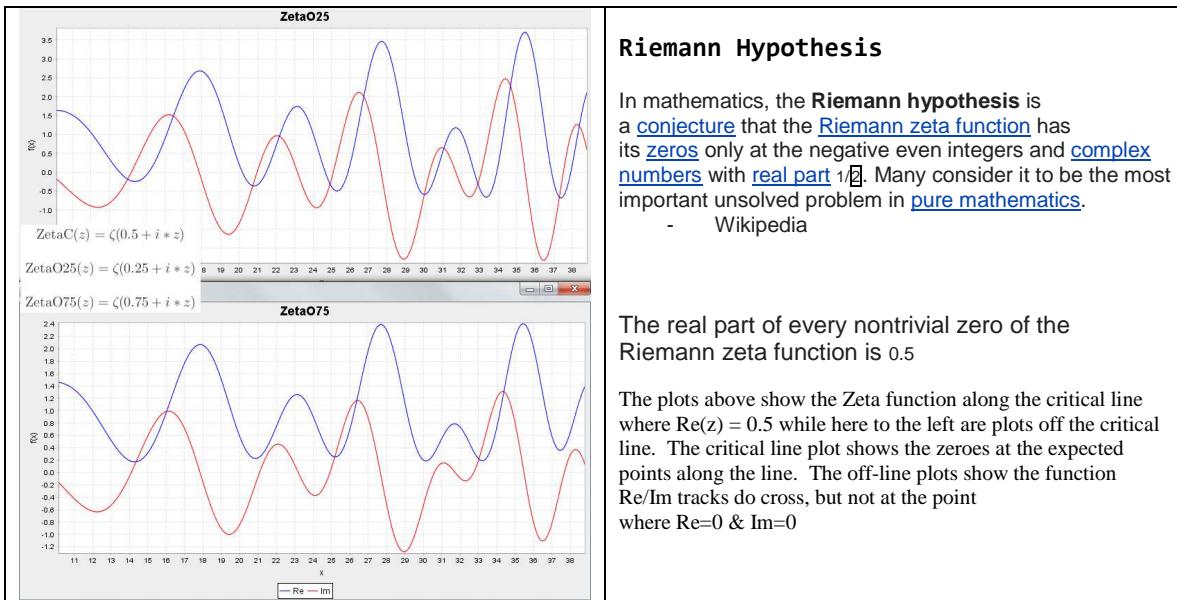


Additional zeros:

At 30, 32, 37, 40, & 43

More accurately:

14.134725142, 21.022039639, 25.010857580, 30.424876126,
 32.935061588, 37.586178159, 40.918719012, 43.327073281, 48.005150881
 (per Mpmath source)



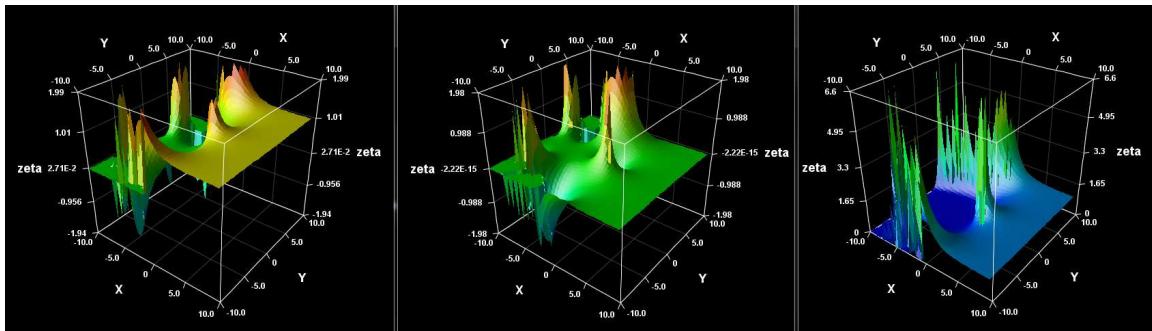
Riemann Hypothesis

In mathematics, the **Riemann hypothesis** is a [conjecture](#) that the [Riemann zeta function](#) has its [zeros](#) only at the negative even integers and [complex numbers](#) with [real part](#) $1/2$. Many consider it to be the most important unsolved problem in [pure mathematics](#).

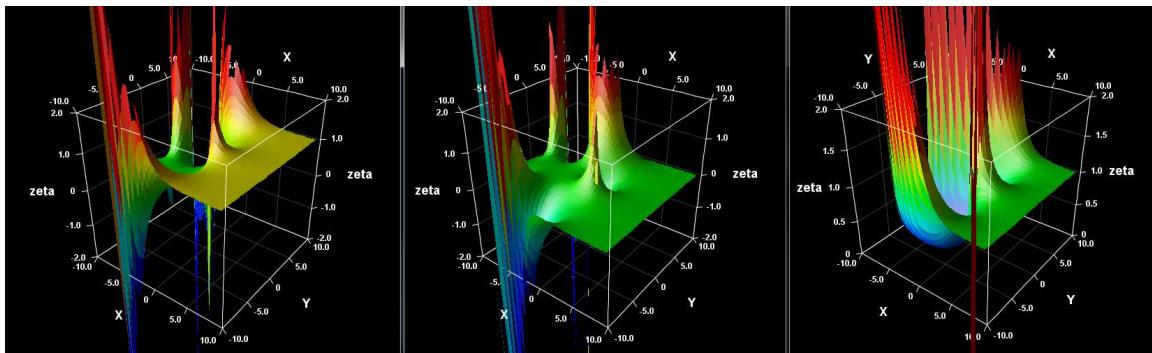
- Wikipedia

The real part of every nontrivial zero of the Riemann zeta function is 0.5

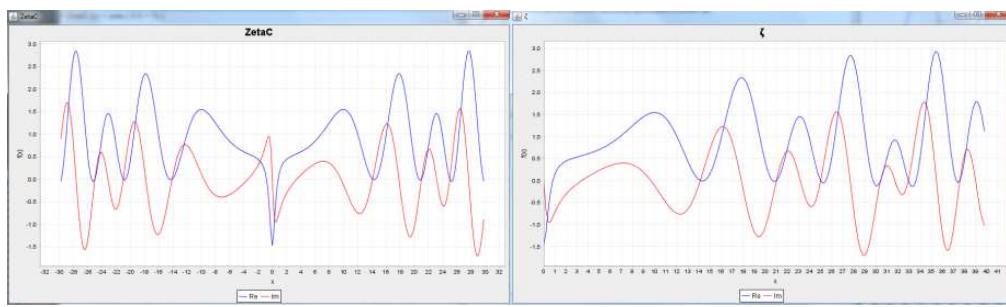
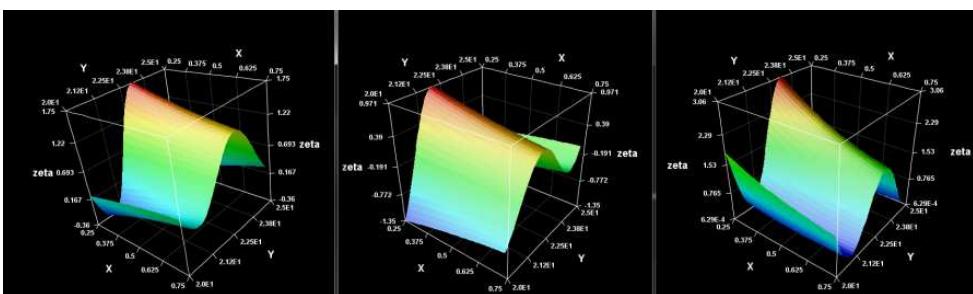
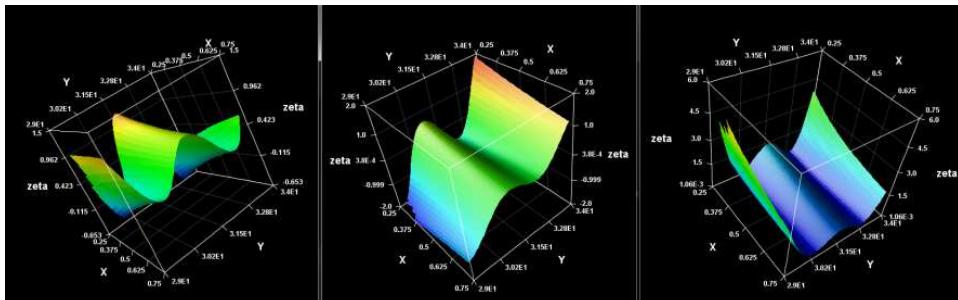
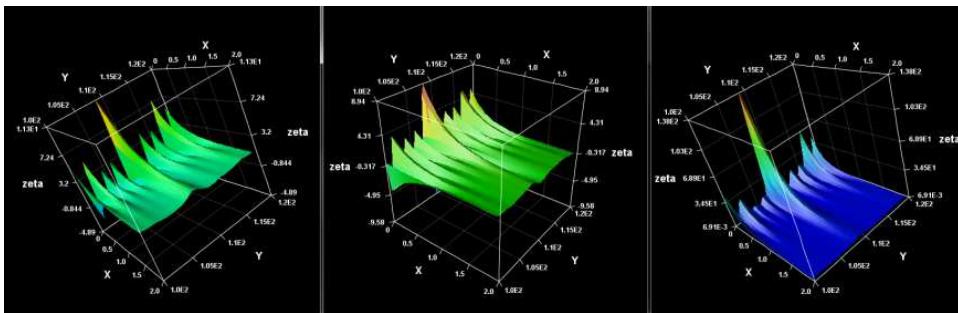
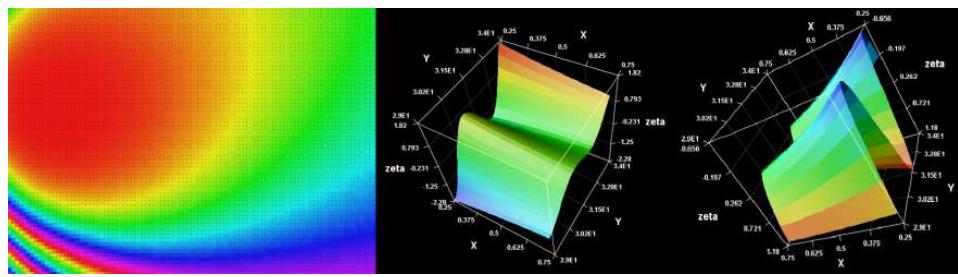
The plots above show the Zeta function along the critical line where $\text{Re}(z) = 0.5$ while here to the left are plots off the critical line. The critical line plot shows the zeroes at the expected points along the line. The off-line plots show the function Re/Im tracks do cross, but not at the point where $\text{Re}=0$ & $\text{Im}=0$

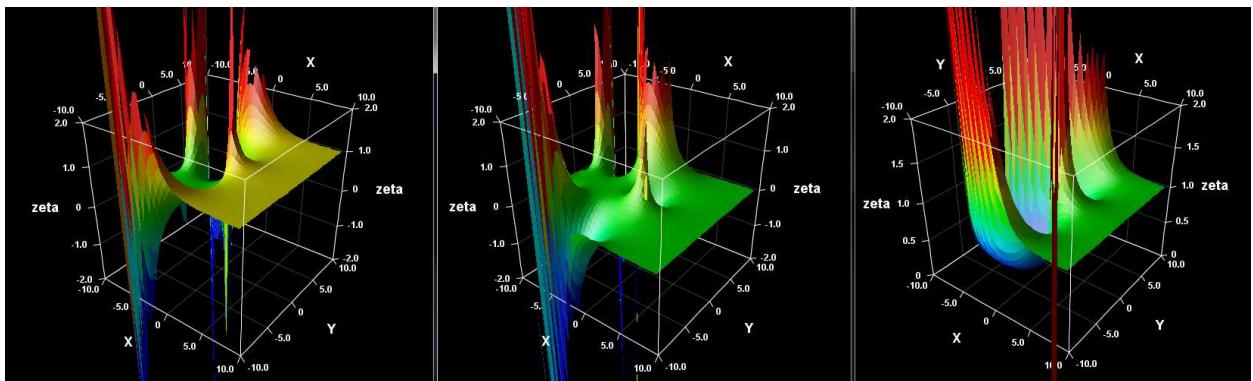
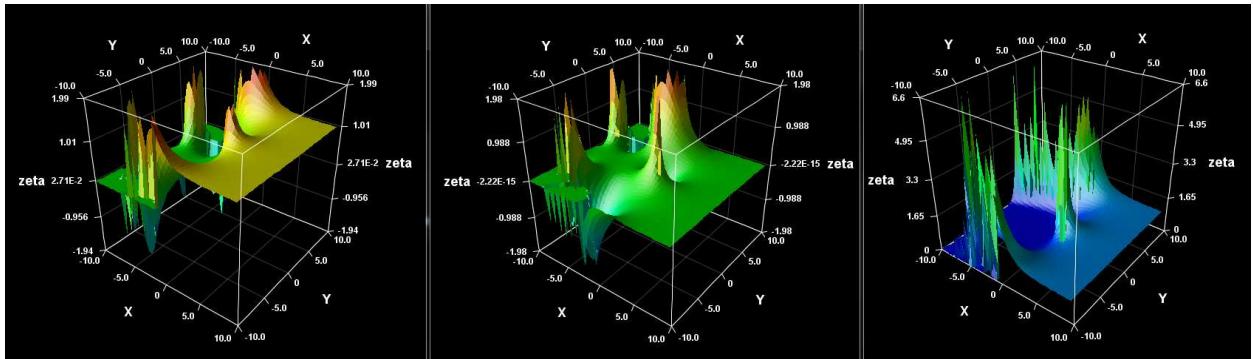
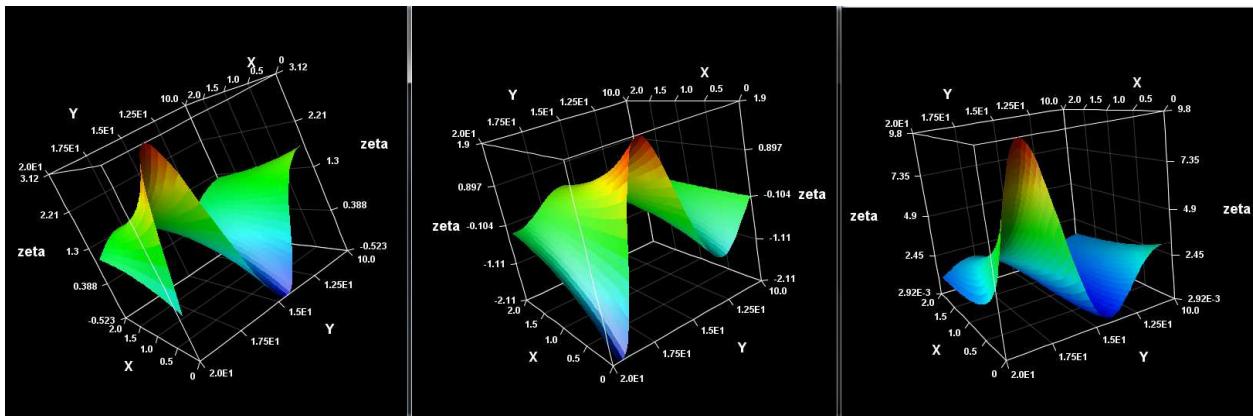
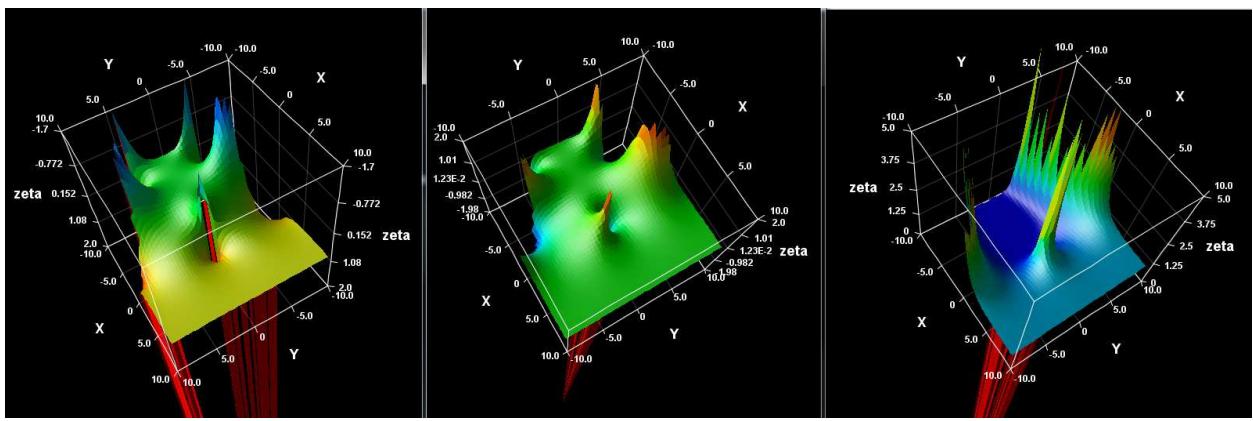


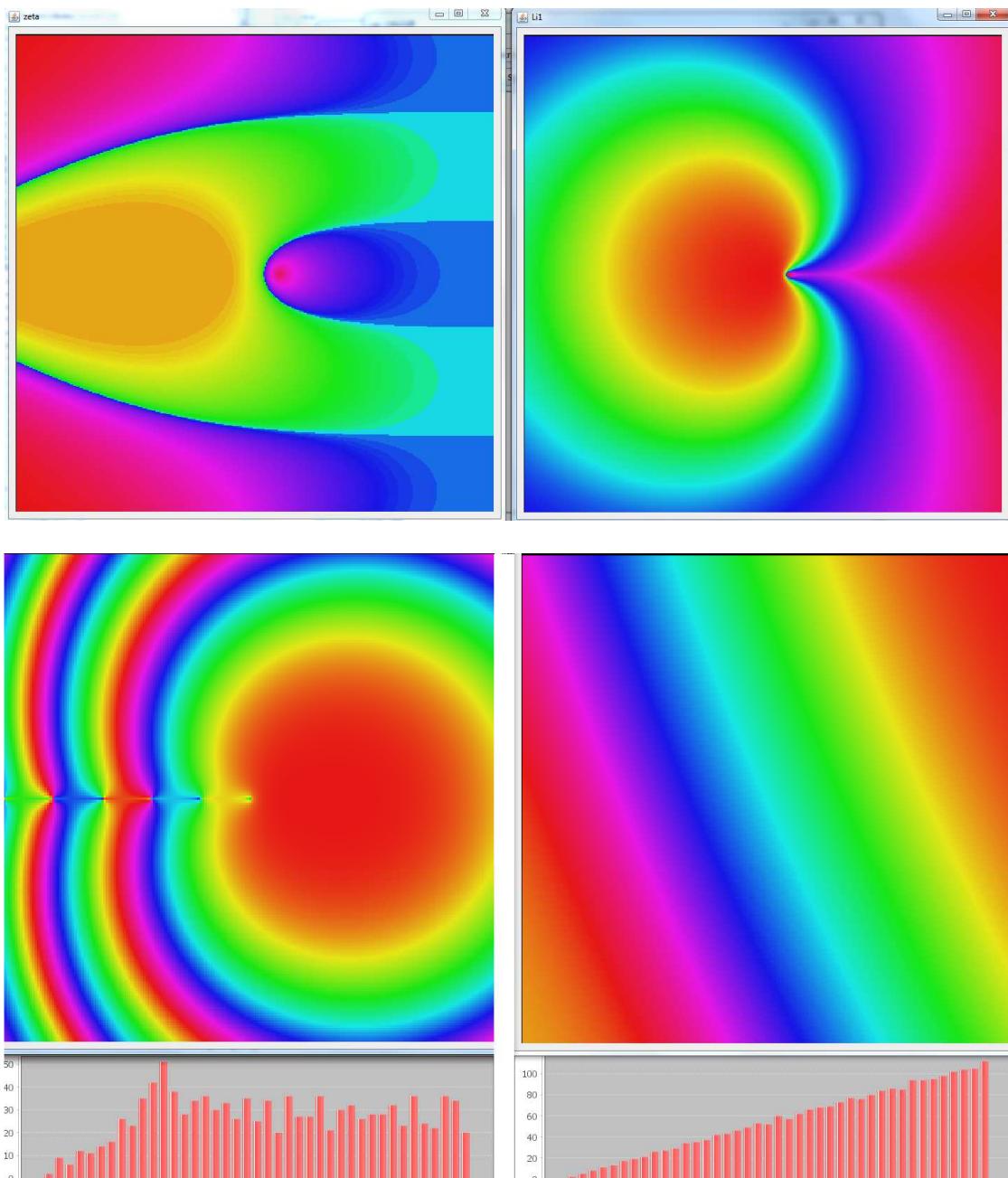
Real, Imaginary, and Magnitude (respectively) plots with limit of 2 assigned on calculation

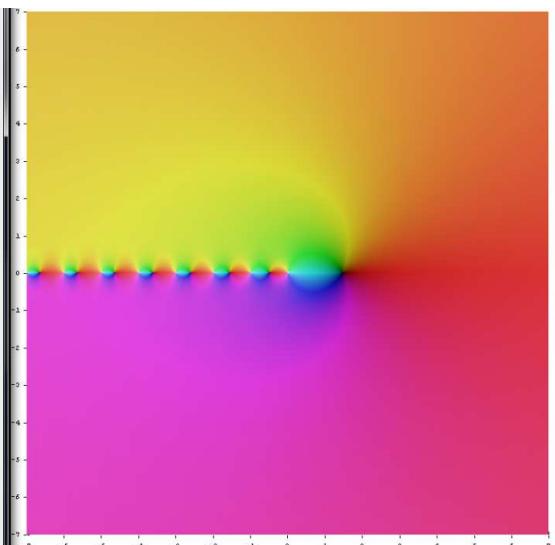
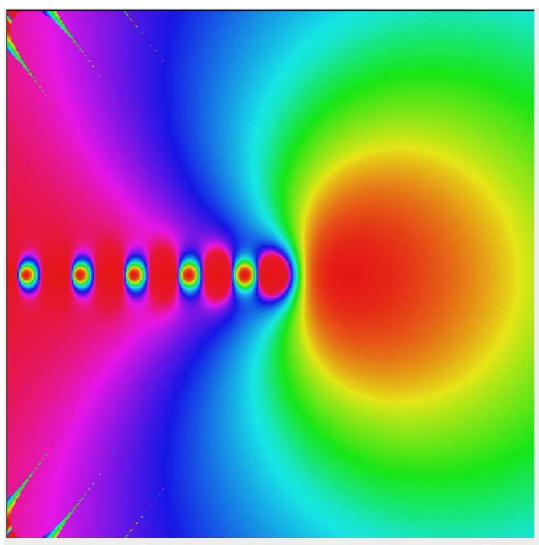
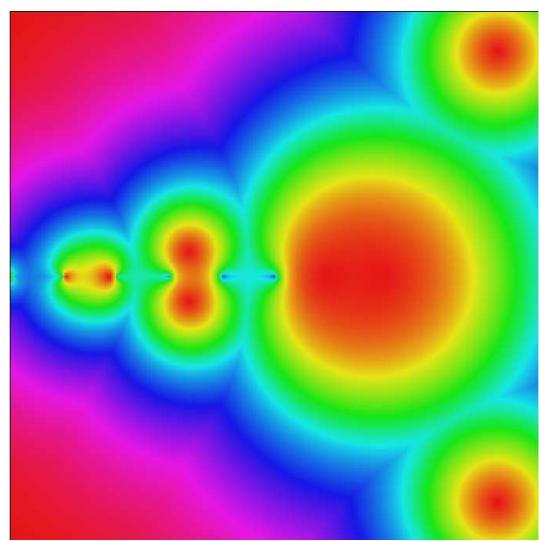
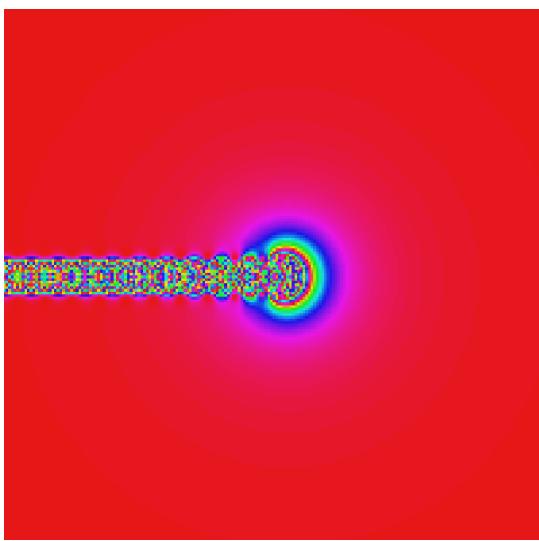


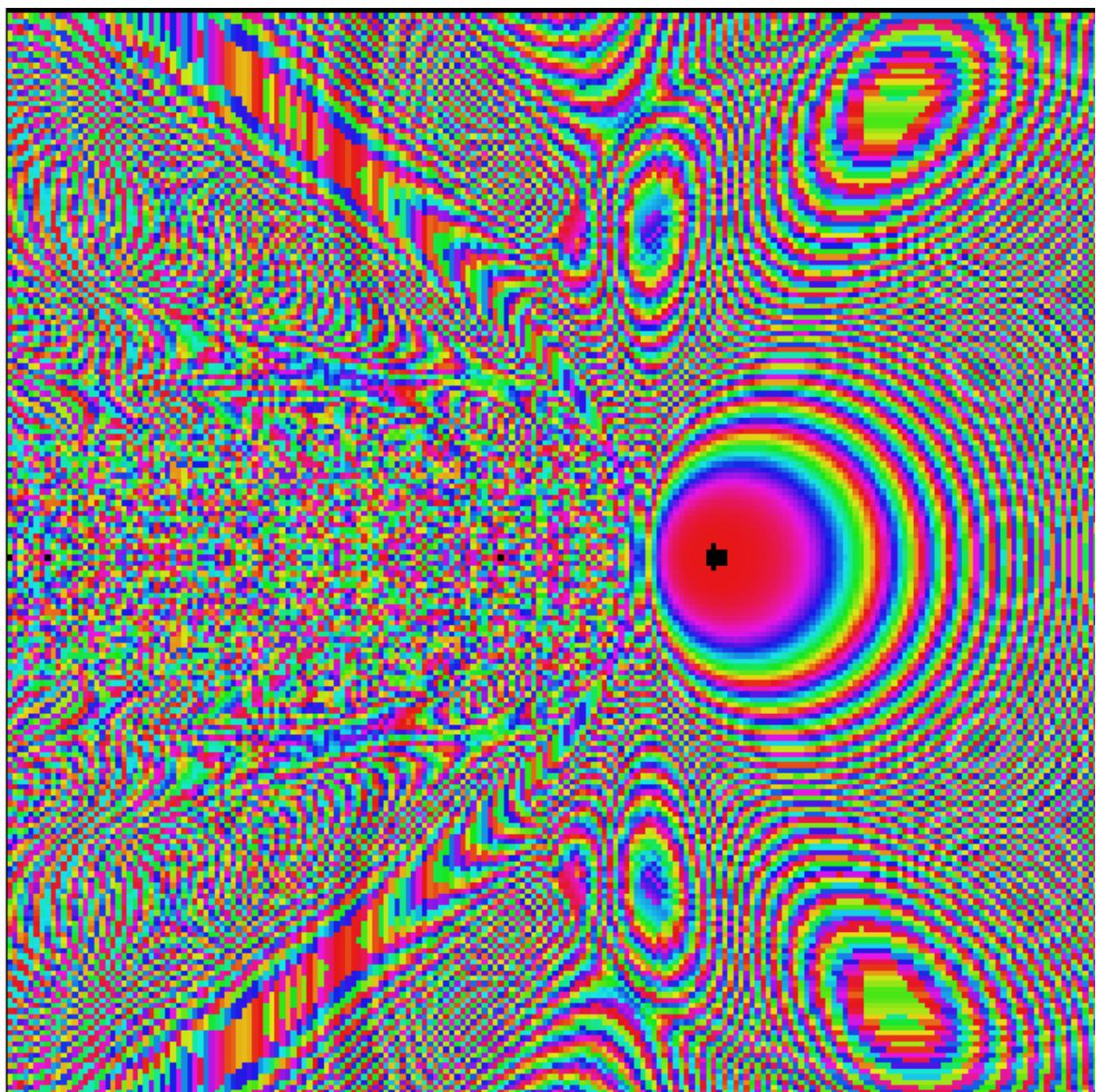
Same plots with no limit assigned on calculation (range restriction added in plot display)

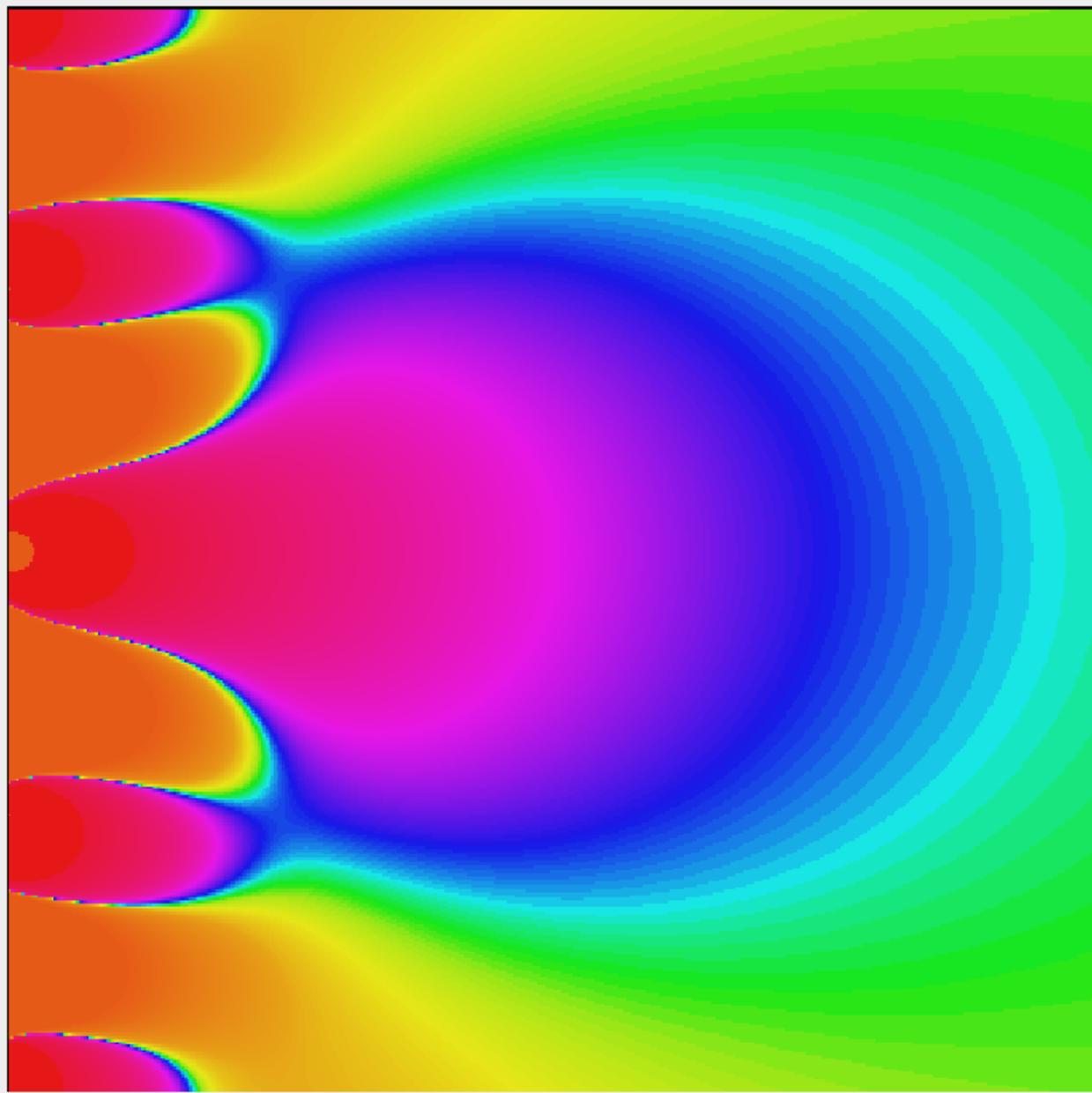


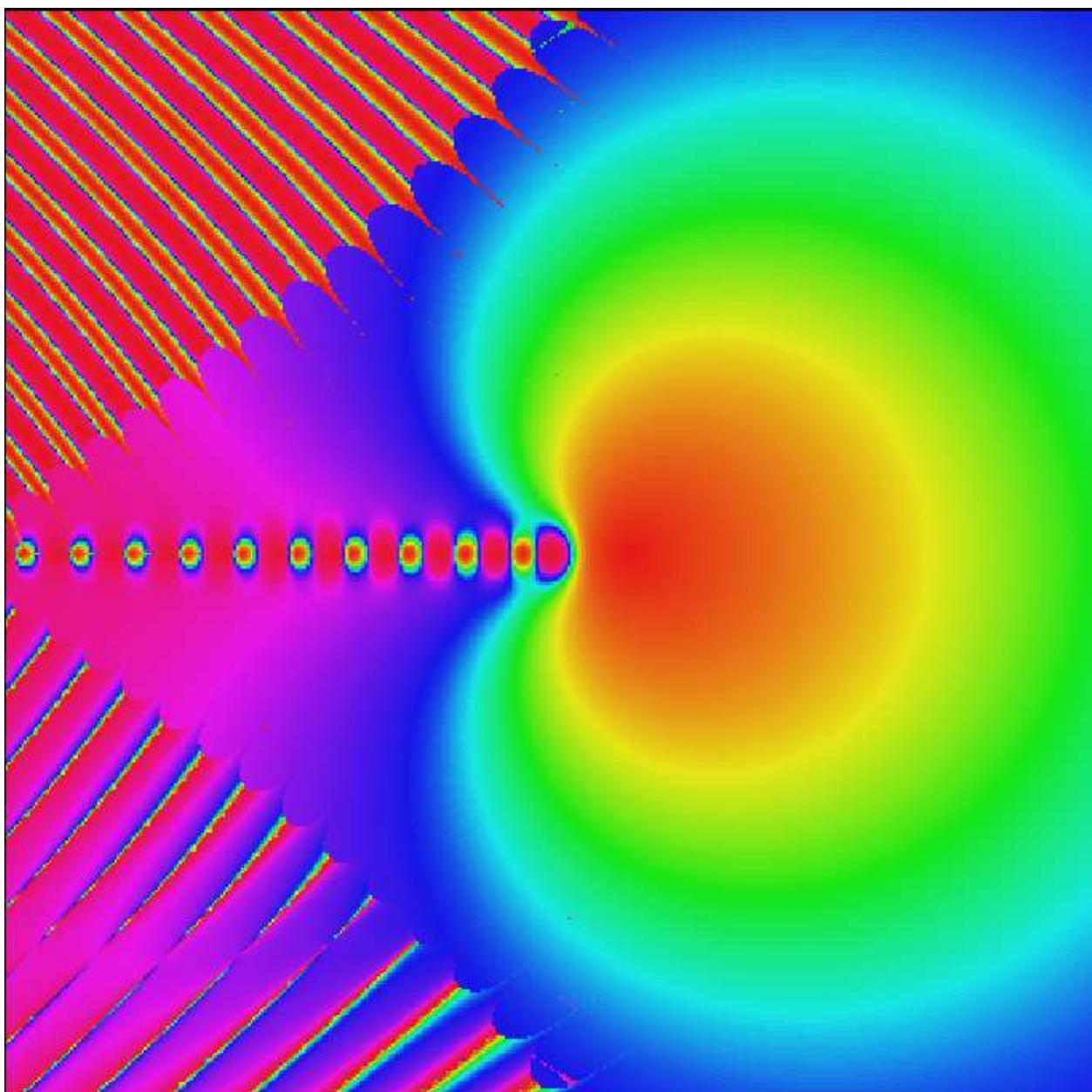


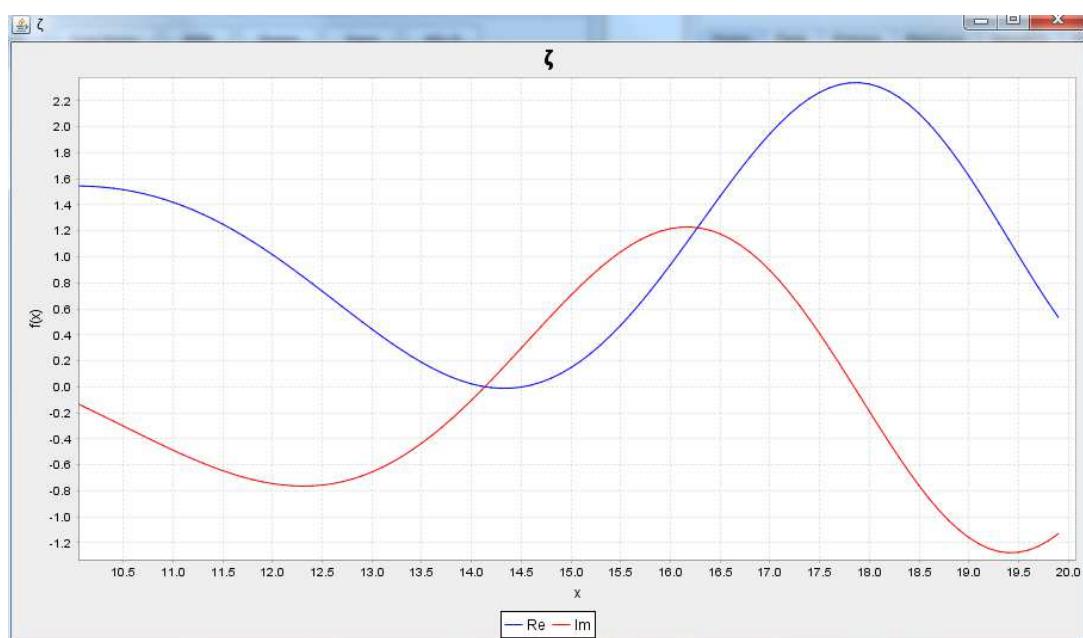
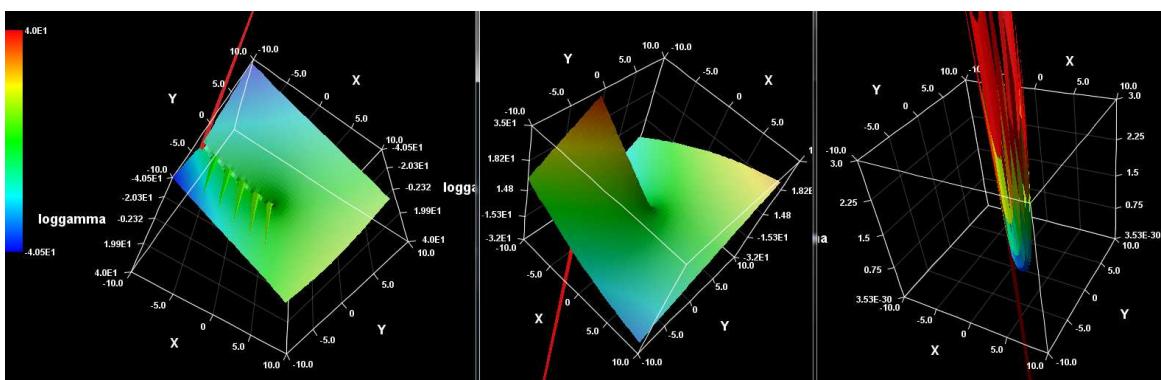
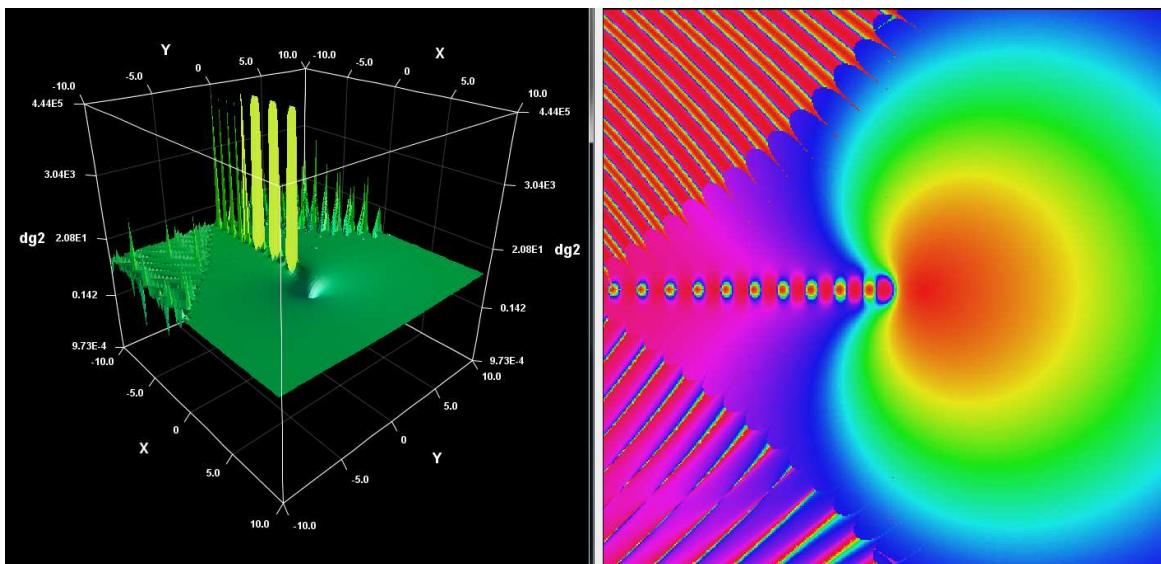


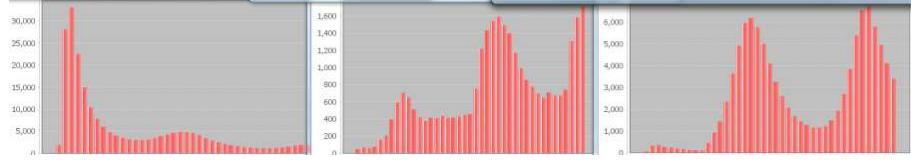
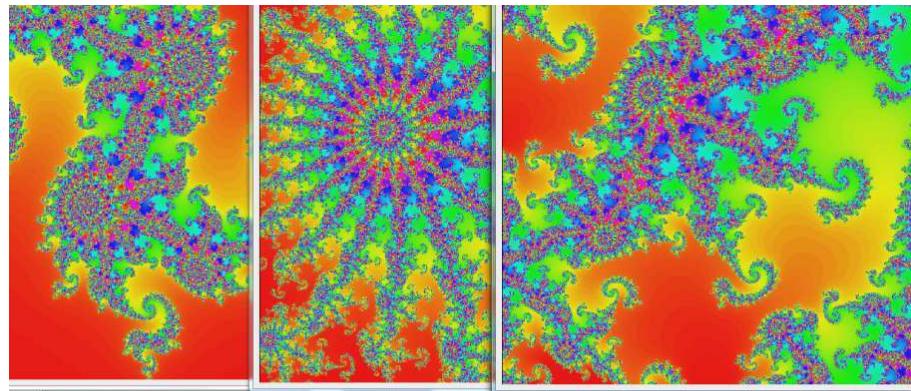
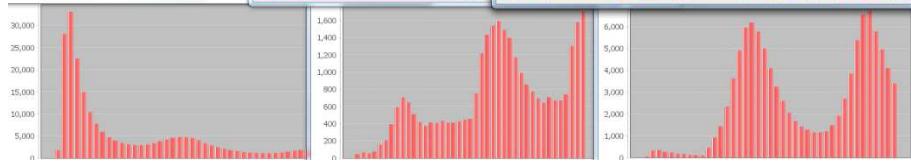
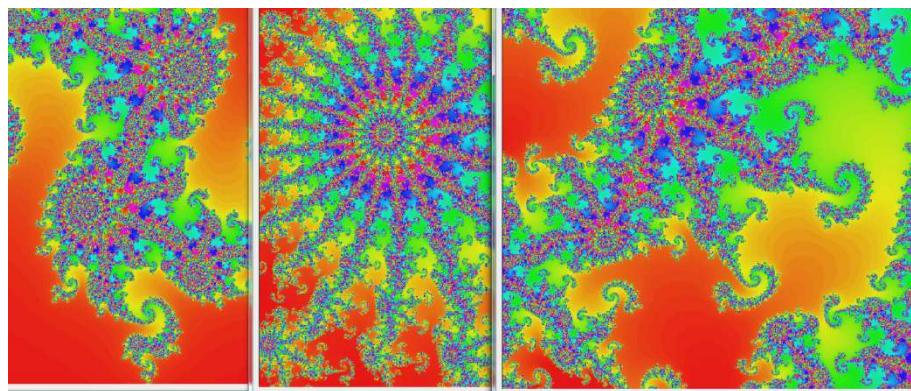


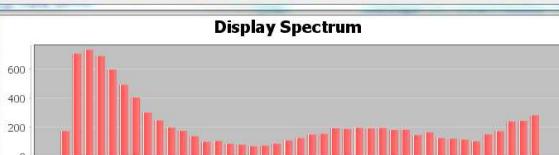
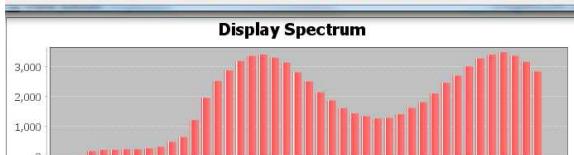
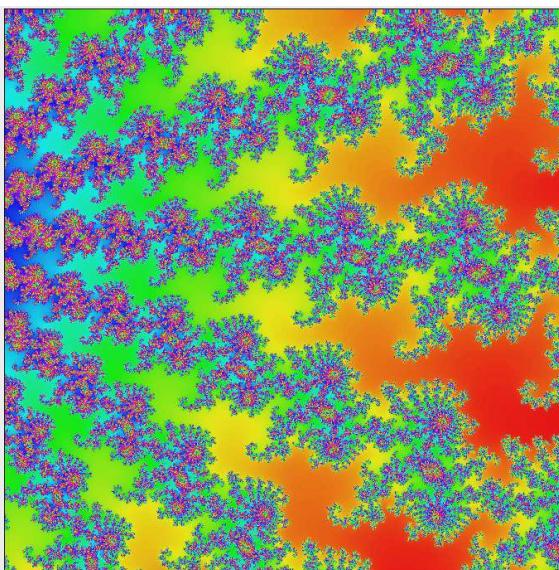
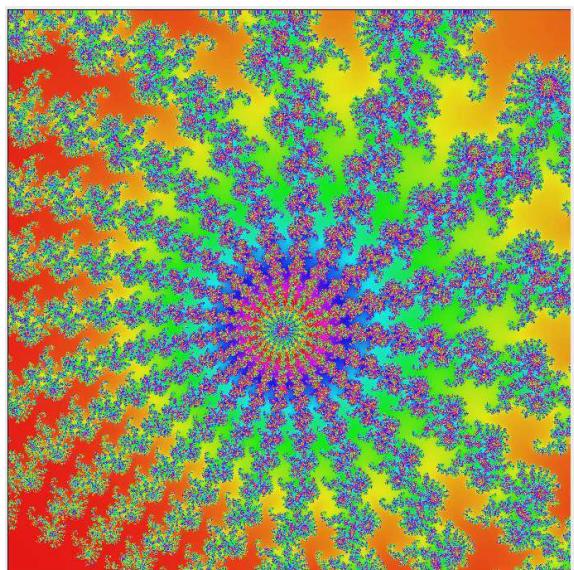
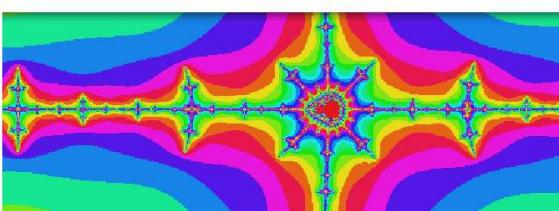
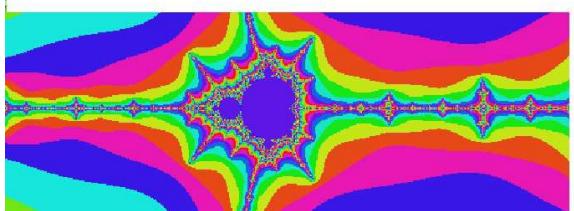
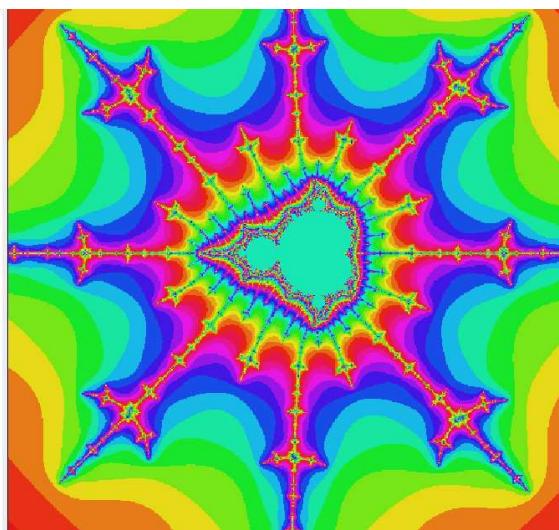
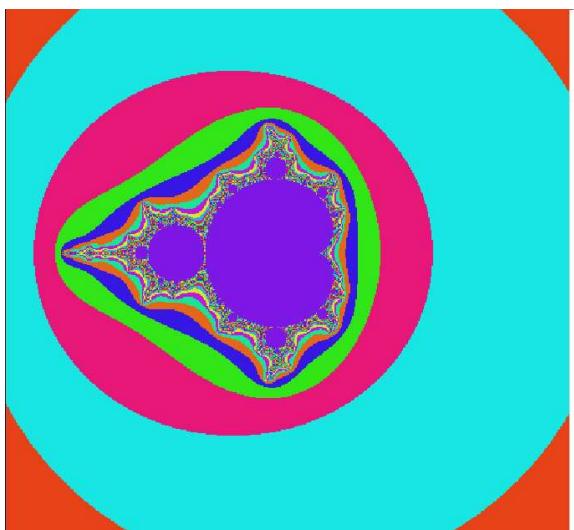




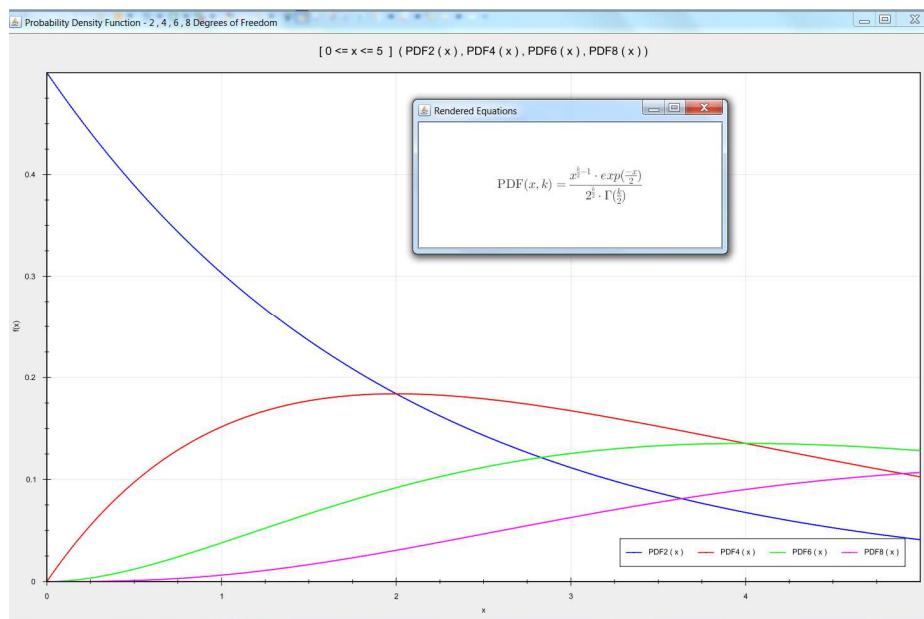
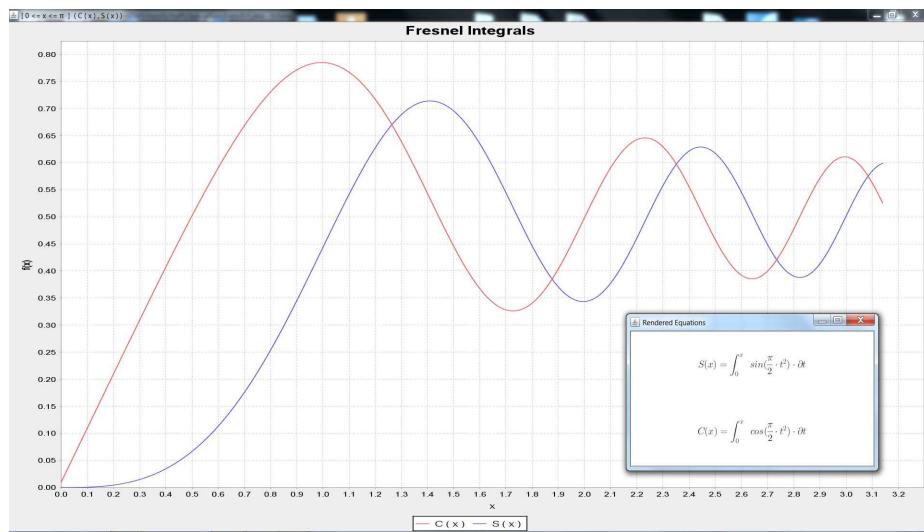




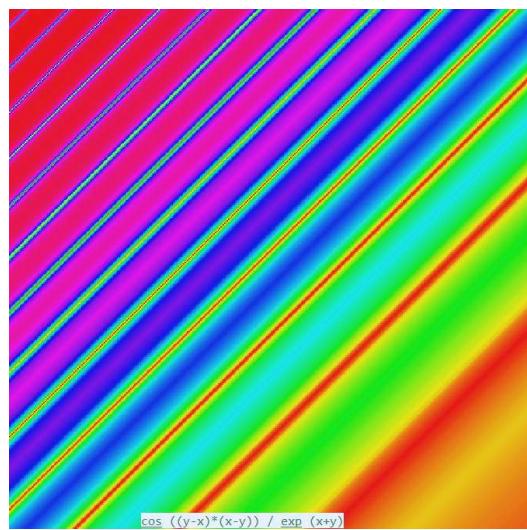
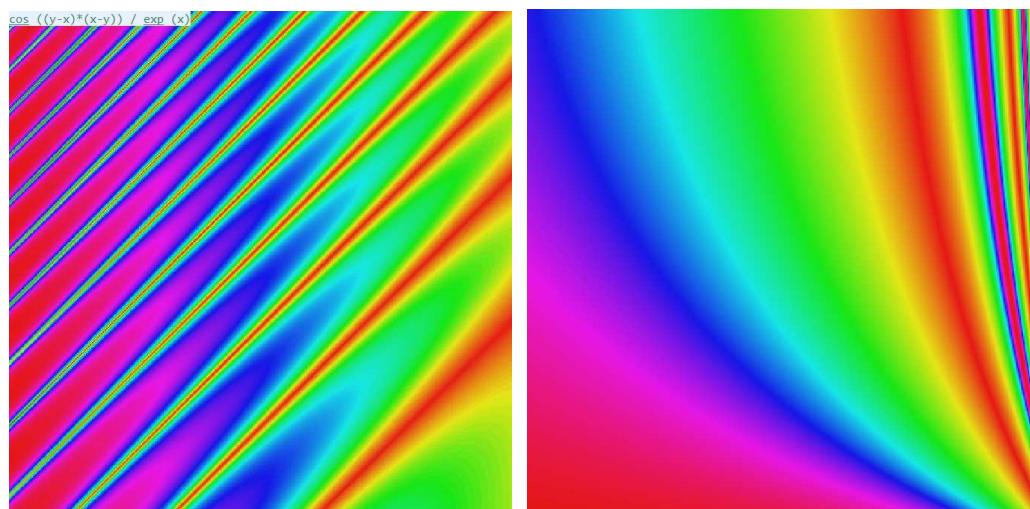
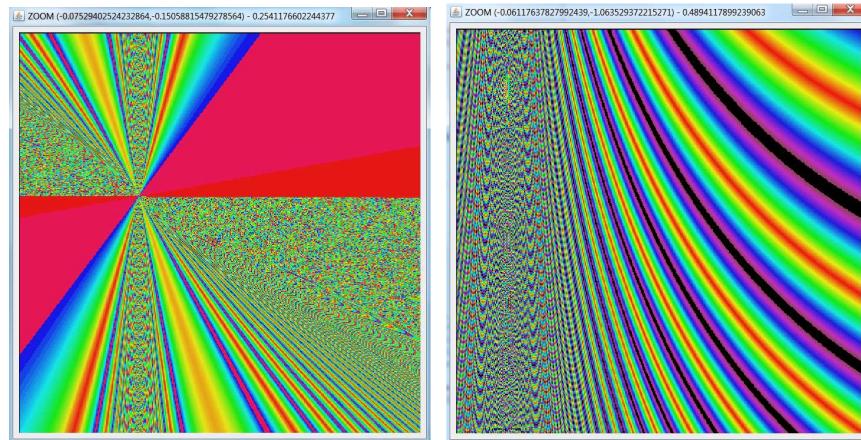




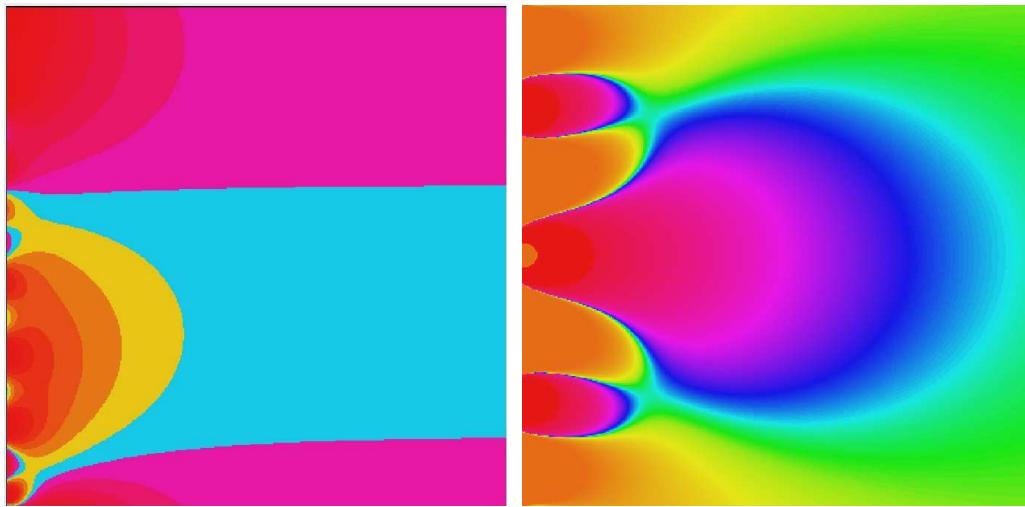
Additional Samples of Line Plots



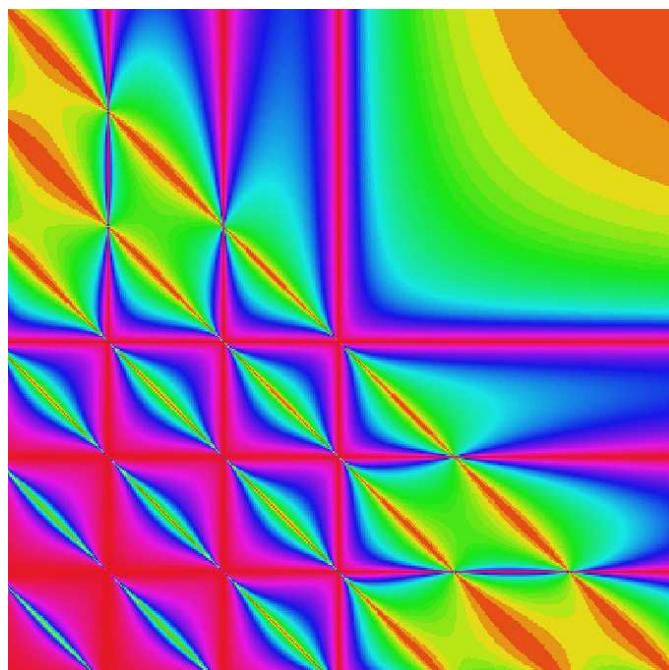
Contour Plots



Zeta Function Translation of Complex plane

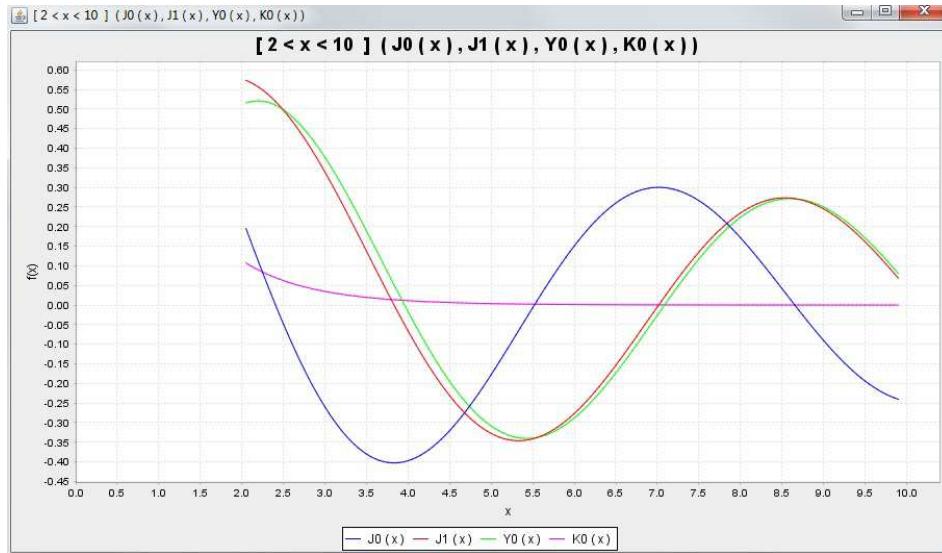


Beta Function Contour Plot

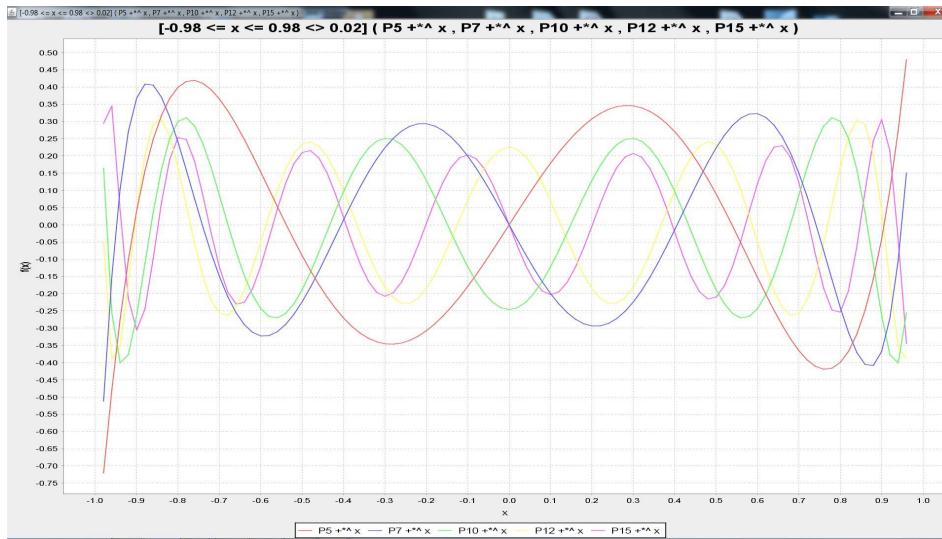


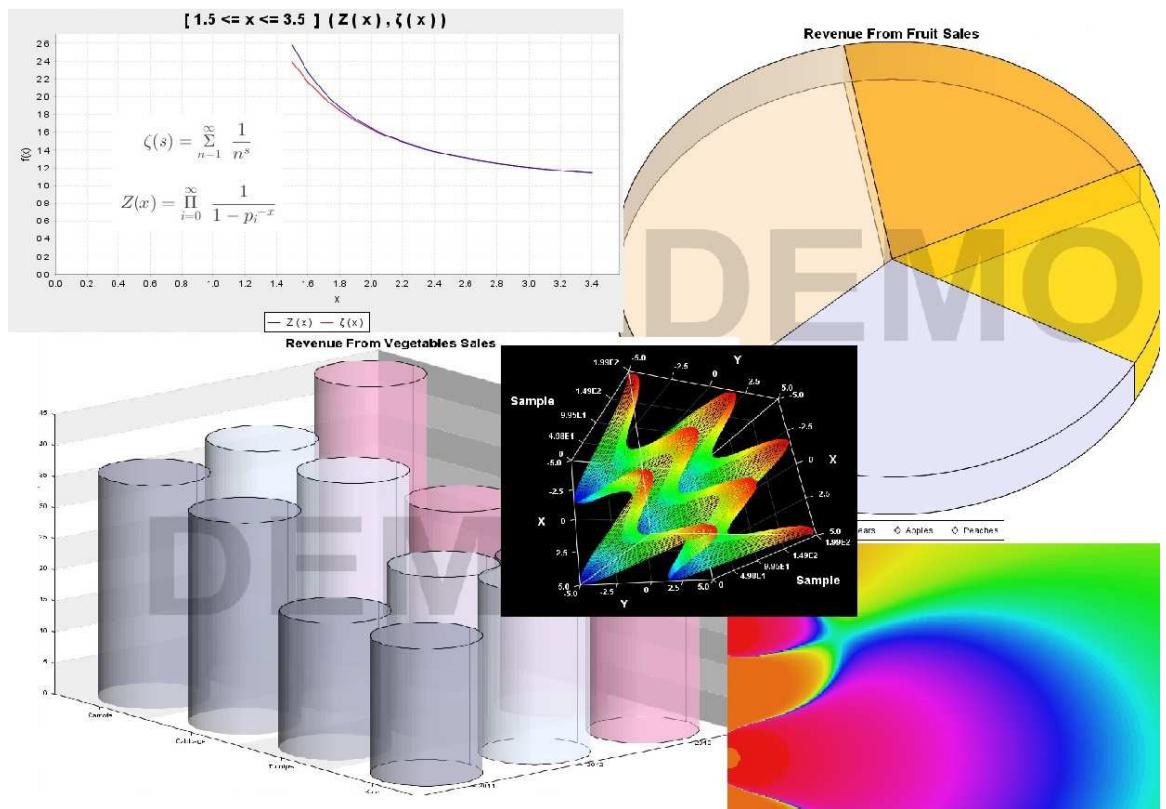
Alternative Graphics Engines

The CalcLib engine has public interfaces for external calculation libraries, MathML rendering engines, and for graphics libraries. A library that can generate plots can be referenced via the application configuration and the interface to provide for substitution of the plot engine. The plot of the Bessel functions below was generated by JFreeChart using the CalcLib graphics library interface.



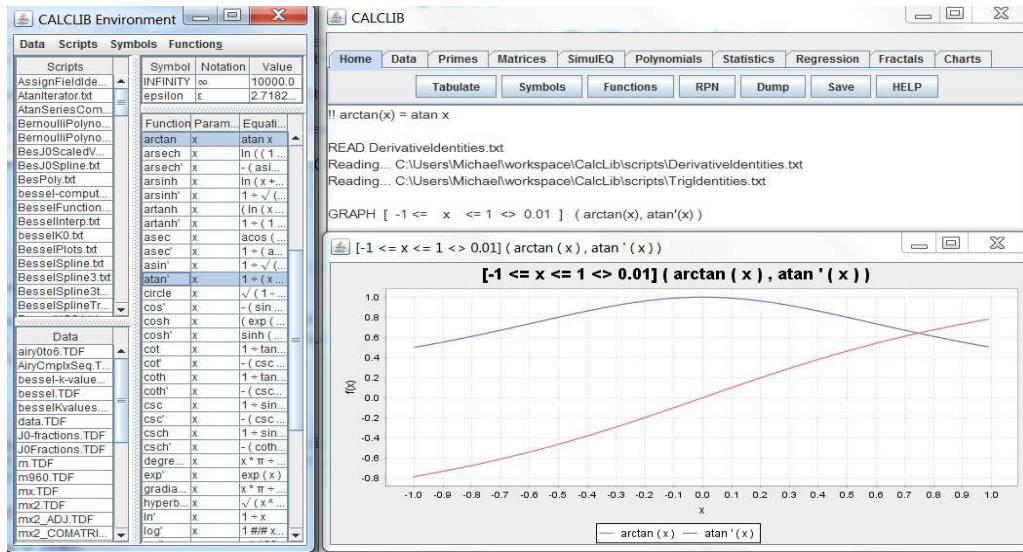
This is the same plot request as shown above ([Legendre Polynomials 5, 7, 10, 12, and 15](#)) with the CalcLib configuration set to use the JFreeChart implementation. The override for each type of chart is available in the charting interface, implementations can choose to override some and optionally not others.





Diverse forms of sophisticated graphics engines

The plotting engine can be invoked from the Function menu or directly from the command line

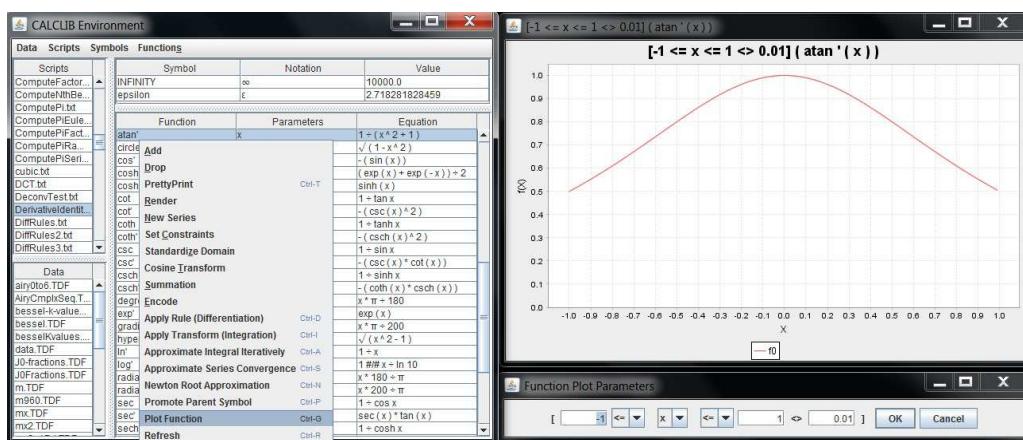


Using Plot Function menu item causes the Function Plot Parameters form to be displayed.

The Low bound, Hi bound, and Increment are taken from the form and used to parameterize the function plot.

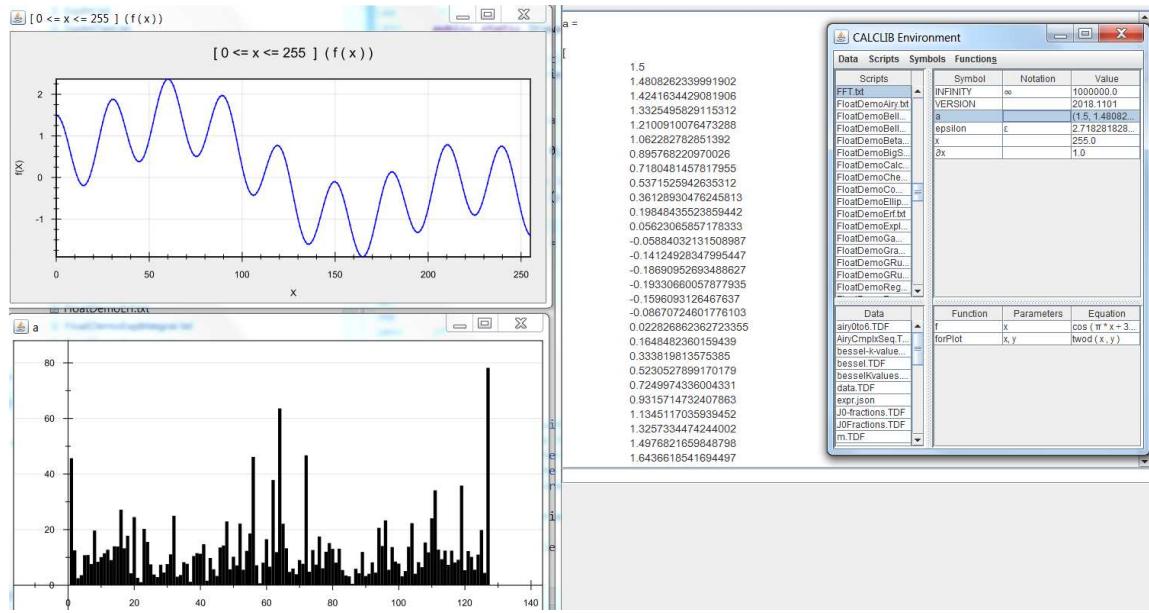
The Increment value will determine the count of points calculated for the display, and hence the smoothness.

Typical trade-offs are seen between plot smoothness versus memory usage and calculation time.



FFT – Fast Fourier Transform

$$f(x) = \cos\left(\frac{\pi \cdot x}{300}\right) + 0.3 \cdot \cos\left(\frac{\pi \cdot x}{150}\right) - 0.7 \cdot \cos\left(\frac{\pi \cdot x}{75}\right) + 0.9 \cdot \cos\left(\frac{\pi \cdot x}{15}\right)$$



FFT.txt

```
!!f(x) = cos (pi*x/300) + 0.3 * cos(pi*x/150)
      - 0.7 * cos (pi*x/75) + 0.9 * cos (pi*x/15)
```

```
RENDERF f
```

```
calc f 1
```

```
a = [0 <= x <= 255] (f(x))
```

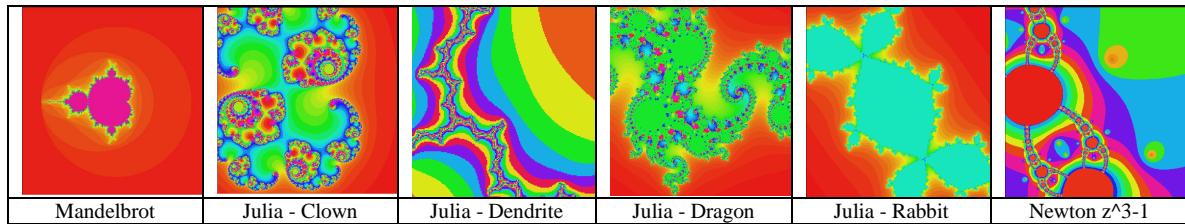
```
PRETTYPRINT a
```

```
GRAPH a
```

```
FFT a
```

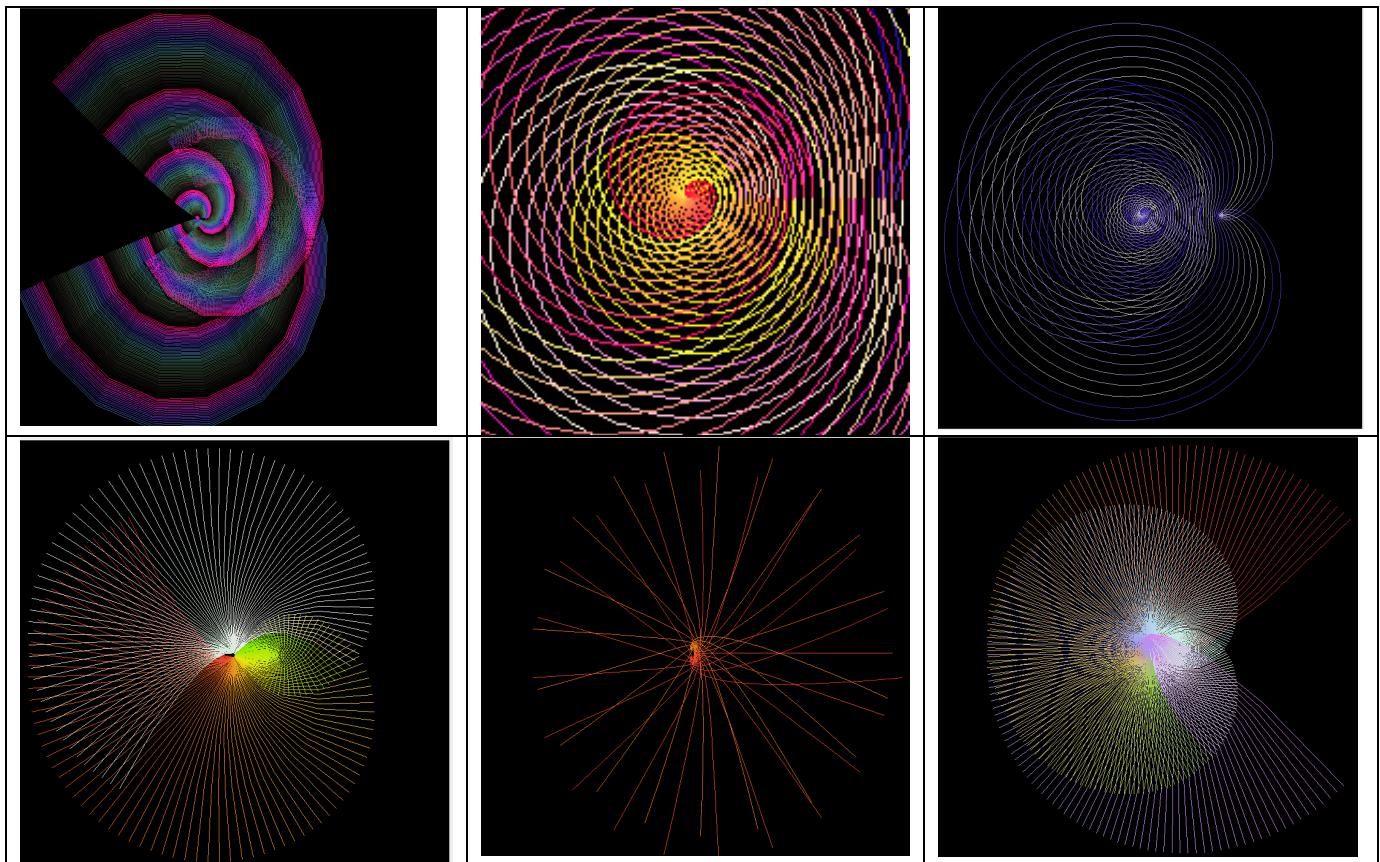
Other Plot Types

Fractal Plots



Polar Plots

Angular – with radial distance held constant the angle moves from $-\pi$ to π



Radial – with the angle held constant the distance extends from -2 to 2

Rendering Expression Notations

The CalcLib engine provides for generation of MathML from command line expression notations. A public interface is provided which enables the graphic rendering of the MathML to common display formats. The examples below were generated with the JMTeX library running from an implementation of the public interface.

Samples of CalcLib RENDER displays	
$a \cdot x^2 + b \cdot x + c = 0$ $x = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$ <p>Order 2 polynomial and the Quadratic equation</p>	<pre>render a*x^2+b*x+c = 0 render x = (- b + - sqrt (b^2 - 4*a*c)) / (2*a) (Source: QuadraticRender.txt) RENDER "Order 2 polynomial and the Quadratic equation" TOP</pre>
$\sum_{n=0}^{\infty} \frac{x^n}{n!}$ <p>The SIGMA notation for series of terms (Taylor series for e^x)</p>	<pre>render SIGMA [0 <= n <= INFINITY] (x^n/n!) RENDER "The SIGMA notation for series of terms (Taylor series for e^x)" TOP</pre>
$\sum_{x=0}^{1\Delta 0.1} \sqrt{1 - x^2}$ <p>An added convention for non-standard increment (delta other than standard 1 default)</p>	<pre>render SIGMA [0 <= x <= 1 > 0.1] (sqrt (1-x^2)) RENDER "An added convention for non-standard increment (delta other than standard 1 default)" TOP</pre>
$\prod_{i=j+1}^{n+1} x_i$ <p>The PI notation for series of factors</p>	<pre>render PI [j+1 <= i <= n+1] (x#i) RENDER "The PI notation for series of factors" TOP</pre>
$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$ $\sum_{k=0}^n (k \cdot \binom{n}{k} \cdot p^k) \cdot (1-p)^{n-k}$ <p>Use of Binomial Coefficient expressed as N choose K</p>	<pre>render n ## k = n! / (k! * (n-k) !) render SIGMA [0<=k<=n] (k * (n##k) * p^k * (1-p)^(n-k)) RENDER "Use of Binomial Coefficient expressed as N choose K" TOP</pre>

$\int \sin(\theta) \cdot d\theta$ $\int \int y \cdot \sin(x) \cdot dx \cdot dy$ Simple indefinite integrals $\int_{-1}^1 \sin(x) \cdot dx$ Simple definite integral $\oint_C \frac{e^{iz}}{z^2 + 1} \cdot dz$ Cauchy's Formula $(1 - x^2) \cdot Tn\theta(x) - x \cdot Tn\theta(x) + n^2 \cdot Tn(x) = 0$ $(1 - x^2) \cdot Un\theta(x) - 3 \cdot x \cdot Un\theta(x) + n \cdot (n + 2) \cdot Un(x) = 0$ Differential Equations (in this case Chebyshev first and second order) $\Gamma(t) = \int_0^\infty t^{x-1} \cdot \exp(-t) \cdot \ln(t) \cdot dt$ Derivative of the Gamma function $\zeta(t) = - \sum_{n=1}^{\infty} \frac{\ln(n)}{n^x}$ Derivative of the Zeta function	render INTEGRALI ($\sin(\theta) * <*> \theta$) render INTEGRALD ($y * \sin(x) * <*> x * <*> y$) RENDER "Simple indefinite integrals" TOP render INTEGRAL [-1 <= x <= 1 > dx] ($\sin(x) * <*> x$) RENDER "Simple definite integral" TOP render INTEGRALC ($e^{i*t*z} / (z^{2+1}) * <*> z$) RENDER "Cauchy's Formula" TOP render ($1 - x^2) * Tn\theta(x) - x * Tn\theta(x) + n^2 * Tn(x) = 0$ render ($1 - x^2) * Un\theta(x) - 3 * x * Un\theta(x) + n * (n + 2) * Un(x) = 0$ RENDER "Differential Equations (in this case Chebyshev first and second order)" TOP !! GAMMA'(x) = INTEGRAL [0 <= t <= INFINITY > dt] ($t^x(x-1) * \exp(-t) * \ln(t) * <*> t$) RENDERF GAMMA' RENDER "Derivative of the Gamma function" TOP !! zeta'(x) = - (SIGMA [1 <= n <= INFINITY] ($\ln(n) / n^x$)) RENDERF zeta' RENDER "Derivative of the Zeta function" TOP
--	---

<p>Historical methods for computation of the value of pi</p> $\zeta(n) = \sum_{k=1}^{\infty} \frac{1}{k^n}$ <p>Reimann zeta(n) function</p> $\pi = \sqrt{6 \cdot \zeta(2)}$ <p>Basel problem computing the convergence of zeta(2)</p> $\text{asin}'(x) = \frac{1}{\sqrt{1-x^2}}$ $\pi = 6 \cdot \int_0^{0.5} \text{asin}'(x) \cdot dx$ <p>Isaac Newton asin series</p> $\text{SegmentArea} = \int_0^{0.25} \sqrt{x - x^2} \cdot dx$ $\pi = 3/4 \cdot \sqrt{3} + 24 \cdot \text{SegmentArea}$ <p>Off center circle segment area used to compute pi</p> $\text{atan}'(x) = \frac{1}{1+x^2}$ $\pi = 4 \cdot \int_0^1 \text{atan}'(x) \cdot dx$ <p>integration of arc tangent derivative to compute pi</p>	<pre>!!zeta(n) = SIGMA [1 <= k <= INFINITY] (1 / k^n) RENDERF zeta; RENDER "Reimann zeta(n) function" TOP // zeta uses parameter for exponent which is 2 // instead using a function which parameterizes the term count // sum of inverted squares gets small fast but all positive terms // pi^2/6 = 1.644934066848226, sumInvSq(100000) = 1.6449240668982423 // pi^2/6 - sumInvSq(100000) = 9.99994983852119E-6, large error for 100,000 terms !sumInvSq(n) = SIGMA [1 <= k <= n] (1 / k^2) invSqRez = SQRT (6 * sumInvSq(INFINITY)) // 10000 terms = 3.141497 // very slow convergence RENDER pi = sqrt (6 * zeta(2)) RENDER "Basel problem computing the convergence of zeta(2)" TOP // declaration of symbols that can be used with integral approximation RENDER asin'(x) = 1 / sqrt (1 - x^2) // 6 * INTEGRAL [0 <= x <= 0.5 <> dx] (asin'(x)) * dx RENDER pi = 6 * INTEGRAL [0 <= x <= 0.5 <> dx] (asin'(x)) * <*>x RENDER "Isaac Newton asin series" TOP !asn'(x) = 1 / SQRT (1 - x^2*x) // or the circular segment equation // circle'(x) = SQRT (x - x^2*x) is shown to have circular segment // pi = 3*SQRT(3)/4 + 24 * area => area = 0.07677310616304730284654410946459 // = INTEGRAL [0 <= x <= 0.25 <> dx] (circle'(x)) * dx RENDER SegmentArea = INTEGRAL [0 <= x <= 0.25 <> dx] (sqrt(x - x^2*x)) * <*>x areaOfCircularSegment = 0.07677310616304730284654410946459 RENDER pi = 3 ## 4 * sqrt(3) + 24 * SegmentArea RENDER "Off center circle segment area used to compute pi" TOP !circle'(x) = SQRT (x - x^2*x) // James Gregory atan series // atan x = x - x^3/3 + x^5/5 - x^7/7 + ... !atan(x,n) = SIGMA [1 <= i <= n <> 4] (x^(4*i) - x^(4*(i+2)))/(i+2)) RENDER atan'(x) = 1 / (1 + x^2) RENDER pi = 4 * INTEGRAL [0 <= x <= 1 <> LIM] (atan'(x)) * <*>x RENDER "integration of arc tangent derivative to compute pi" TOP</pre>
--	---

<p>$\text{atan}(x) = \sum_{i=1}^{\infty} \frac{x^{4i-3}}{4 \cdot i - 3} - \frac{x^{4i-1}}{4 \cdot i - 1}$</p> <p>James Gregory atan series : atan x = x - x^3/3 + x^5/5 - x^7/7 + ...</p> $\pi = 4 \cdot (4 \cdot \text{atan}(\frac{1}{5}) - \text{atan}(\frac{1}{239}))$ <p>Arc Tangent series of John Machin</p> $\pi = 20 \cdot \text{atan}(\frac{1}{7}) + 8 \cdot \text{atan}(\frac{3}{79})$ <p>Arc Tangent series of Leonhard Euler</p> $\frac{1}{\pi} = \frac{2 \cdot \sqrt{2}}{9801} \cdot \sum_{k=0}^{\infty} \frac{(4 \cdot k)! \cdot (1103 + 26390 \cdot k)}{k!^4 \cdot 396^{4k}}$ <p>Series of Srinivasa Ramanujan</p>	<p>LIM = 1/INFINITY !!tan'(x) = 1 / (1 + x^2*x)</p> $/ 4 * INTEGRAL [0 <= x <= 1 <> dx] (tan'(x)) * dx$ $\text{atnIntRez} = \text{INTEGRAL} [0 <= x <= 1 <> LIM] (4 * tan'(x)) * <*>x$ <p>PRETTYPRINT LIM PRETTYPRINT invSqRez PRETTYPRINT atnIntRez</p> <p>/ arc tangent function used to compute pi polyTerms = 8; polyDegree = polyTerms * 2 - 1; countMax = polyDegree - 2 RENDER atan(x) = SUMMATION [1 <= i <= INFINITY] (((x^(4*i) - 3) / (4*i - 3)) MINUS (x^(4*i - 1) / (4*i - 1)))) RENDER "James Gregory atan series : atan x = x - x^3/3 + x^5/5 - x^7/7 + ..." TOP</p> $\text{RENDER pi} = 4 * (4 * \text{atan}(1/5) - \text{atan}(1/239))$ <p>RENDER "Arc Tangent series of John Machin" TOP</p> $\text{RENDER pi} = 20 * \text{atan}(1/7) + 8 * \text{atan}(3/79)$ <p>RENDER "Arc Tangent series of Leonhard Euler" TOP</p> $\text{RENDER } 1/\pi = 2 * \text{sqrt}(2) / 9801 * (\text{SUMMATION} [0 <= k <= INFINITY] ((4*k)! * (1103 + 26390*k) / ((k!)^4 * 396^(4*k))))$ <p>RENDER "Series of Srinivasa Ramanujan" TOP</p>
--	---

CalcLib Notation, MathML, and rendered expression

<u>MathML</u>	<u>CalcLib Notation</u>
<p>Generated by CalcLib</p> <pre><?xml version="1.0"?> <math> <mrow> <mi>x</mi> <mo>=</mo> <mfrac> <mrow> <mo>-</mo> <mrow> <mi>b</mi> </mrow> </mrow> <mo>\pm</mo> <msqrt> <mrow> <mo>-</mo> <msup> <mn>2</mn> </msup> <mo>\cdot</mo> <mn>4</mn> <mo>\cdot</mo> <mi>a</mi> <mo>\cdot</mo> <mi>c</mi> </mrow> </msqrt> </mrow> </math></pre>	<p>render x = (- b + - sqrt (b^2 - 4*a*c)) / (2*a)</p> <p>Renders As</p> $x = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$ <p>Using the public interface implementation connected to the JMTeX library</p>

$DTR(x) = \text{degrees2Radians}(x)$ $\Gamma(t) = \int_0^{\infty} t^{x-1} \cdot \varepsilon^{-t} \cdot \ln(t) \cdot \partial t$ $GTR(x) = \text{gradians2Radians}(x)$ $OCTOPUS(x, y) = \cos\left(\frac{y \cdot \varepsilon^{x-y}}{x}\right)$ $RTD(x) = \text{radians2Degrees}(x)$ $RTG(x) = \text{radians2Gradians}(x)$ $\text{acos}(x) = \frac{\pi}{2} - \text{asin}(x)$ $\text{acos}'(x) = -\text{asin}(x)$ $\text{acot}(x) = \frac{\pi}{2} - \text{atan}(x)$ $\text{acot}'(x) = -\text{atan}(x)$ $\text{acsc}(x) = \frac{\pi}{2} - \text{asec}(x)$ $\text{acsc}'(x) = -\text{asec}(x)$	$\text{arcosh}(x) = \ln(x + \text{hyperbola}(x))$ $\text{arcosh}'(x) = \frac{1}{\sqrt{x^2 - 1}}$ $\text{arcoth}(x) = \left(\frac{1}{2}\right) \cdot \ln\left(\frac{x+1}{x-1}\right)$ $\text{arcoth}'(x) = \frac{1}{1-x^2}$ $\text{arcsch}(x) = \ln\left(\frac{1}{x} + \sqrt{\frac{1}{x^2} + 1}\right)$ $\text{arcsch}'(x) = -\frac{\text{arsinh}'(x)}{ x }$ $\text{arsech}(x) = \ln\left(\frac{1 + \text{circle}(x)}{x}\right)$ $\text{arsech}'(x) = -\frac{\text{asin}'(x)}{x}$ $\text{arsinh}(x) = \ln(x + \sqrt{x^2 + 1})$ $\text{arsinh}'(x) = \frac{1}{\sqrt{x^2 + 1}}$ $\text{artanh}(x) = \left(\frac{1}{2}\right) \cdot \ln\left(\frac{1+x}{1-x}\right)$ $\text{artanh}'(x) = \frac{1}{1-x^2}$
$\text{asec}(x) = \text{acos}\left(\frac{1}{x}\right)$ $\text{asec}'(x) = \frac{1}{ x \cdot \sqrt{x^2 - 1}}$ $\text{asin}(x) = \frac{1}{\sqrt{1 - x^2}}$ $\text{atan}(x) = \frac{1}{x^2 + 1}$ $\text{circle}(x) = \sqrt{1 - x^2}$ $\text{cos}(x) = -\sin(x)$ $\text{cosh}(x) = \frac{\varepsilon^x + \varepsilon^{-x}}{2}$ $\text{cosh}'(x) = \sinh(x)$ $\text{cot}(x) = \frac{1}{\tan(x)}$ $\text{cot}'(x) = -\csc^2(x)$ $\text{coth}(x) = \frac{1}{\tanh(x)}$ $\text{coth}'(x) = -\text{csch}^2(x)$	$\csc(x) = \frac{1}{\sin(x)}$ $\csc'(x) = -\csc(x) \cdot \cot(x)$ $\text{csch}(x) = \frac{1}{\sinh(x)}$ $\text{csch}'(x) = -\coth(x) \cdot \text{csch}(x)$ $\text{degrees2Radians}(x) = \frac{x \cdot \pi}{180}$ $\text{exp}(x) = \varepsilon^x$ $f(x) = \sin^2(2 \cdot x) - \cos\left(\frac{\pi}{4} \cdot x\right) \cdot \sec^3\left(\frac{x}{4}\right)$ $\text{gradians2Radians}(x) = \frac{x \cdot \pi}{200}$ $\text{hyperbola}(x) = \sqrt{x^2 - 1}$ $\ln(x) = \frac{1}{x}$ $\log(x) = \frac{1/x}{\ln(10)}$ $\text{radians2Degrees}(x) = \frac{x \cdot 180}{\pi}$ $\text{radians2Gradians}(x) = \frac{x \cdot 200}{\pi}$
$\sec(x) = \frac{1}{\cos(x)}$ $\sec'(x) = \sec(x) \cdot \tan(x)$ $\text{sech}(x) = \frac{1}{\cosh(x)}$ $\text{sech}'(x) = -\tanh(x) \cdot \text{sech}(x)$ $\sin(x) = \cos(x)$ $\sinh(x) = \frac{\varepsilon^x - \varepsilon^{-x}}{2}$	$\sinh(x) = \cosh(x)$ $\tan(x) = 1 + \tan^2(x)$ $\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$ $\tanh'(x) = \text{sech}^2(x)$ $\text{tenToX}(x) = 10^x \cdot \ln(10)$

DomainDefs.py

```
# -*- coding: utf-8 -*-
"""
Created on Sat Aug 14 18:45:18 2021
Methods that provide domain production for evaluation
@author: Michael Druckman
"""

import io
import json

def domain (p, delta, count):
    # construct domain from point and delta*count
    x = [ p + i * delta for i in range ( 0, count ) ]
    return x

def cdomain (pr, pi, deltar, deltai, count):
    # P and delta are converted to complex values
    x = domain ( complex(pr,pi), complex(deltar,deltai), count )
    return x
```

```

def runF (f, pr, pi, dr, di, count):
    # domain is built and n-tuple is made from iterator
    x = cdomain (pr, pi, dr, di, count)
    y = [ getFhndld(f,z) for z in x ]
    return y

def getFtuples (f, pr, pi, dr, di, count):
    # function is called for each domain value
    y = runF (f, pr, pi, dr, di, count)
    # complex values are split into re/im tuples
    tups = [ ( float (a.real), float (a.imag) ) for a in y ]
    # JSON text is formatted to represent value list
    sbuf = io.StringIO ()
    json.dump (tups, sbuf)
    return sbuf.getvalue ()

def getFhndld (f, x):
    try:                      # catch exceptions thrown by EVAL call
        y = f(x)              # each call to function could cause raise
    except:
        y = complex(9E99,9E99) # 9E99 will be recognized as invalid
    return y;                  # return a value as a function result

```

Networking.py

```
# -*- coding: utf-8 -*-
"""


```

```
Created on Sat Aug 14 18:45:18 2021
```

```
Methods that provide socket services for eval processing
```

```
@author: Michael Druckman
```

```
"""


```

```
tracing_accept = 0
```

```
tracing_request = 1
```

```
# symbol imports for networking implementation
```

```
import socket
```

```
from socket import SOCK_STREAM
```

```
from socket import AF_INET
```

```
def serve_request (socket_served, buffer_size, request_processor):
    # accept socket connection and recv request
    client, addr = socket_served.accept ()
    request_recd = client.recv ( buffer_size )
    req_string = request_recd.decode ('utf-8')
    if tracing_accept: print ('Request accepted from ', addr)
    if tracing_request: print (req_string)
    # check for STOP request and raise if found
    if req_string.startswith('STOP'):
        message = b'Stopping...'
        client.send (message)
        raise RuntimeError ('STOP requested')
    # process request and encode response for sending
    rsp_string = request_processor (req_string)
    encd_response = rsp_string.encode ('utf-8')
    # send response and closee connection
    client.send ( encd_response )
    client.close ()
    return
```

```
def serve_eval_until_stop (socket_used):
    # loop until an exception is raised
    try:
        while 1:      # looping forever
            serve_eval_request (socket_used)
    except:        # close connection when done
        socket_used.close()

def serve_eval_on (port, instances):
    # boiler-plate socket creation for bind/listen
    bound_socket = socket ( AF_INET, SOCK_STREAM )
    bound_socket.bind ( ("port") )
    bound_socket.listen (instances)
    # start server loop listening on bound_socket
    serve_eval_until_stop (bound_socket)

def serve_eval_8080 ():
    # default service start on port 8080
    serve_eval_on (8080, 5)
    return
```

Client.py

```
def request_eval (source, host, port, buffer_size):
    # send encoded request / decode response from server
    socket_used = socket ( AF_INET, SOCK_STREAM )
    socket_used.connect ( (host,port) )
    coded_request = source.encode ('utf-8')
    socket_used.send ( coded_request )
    byte_buffer = socket_used.recv ( buffer_size )
    response_buffer = byte_buffer.decode ('utf-8')
    socket_used.close ()
    return response_buffer
```

```
def request_stop (host, port):
    # STOP request is recognized as termination request
    socket_used = socket ( AF_INET, SOCK_STREAM )
    socket_used.connect ( (host,port) )
    coded_request = b'STOP'
    socket_used.send ( coded_request )
    socket_used.close ()
    return

def serve_eval_request (socket_used):
    # simple service test with default parameters
    return serve_request (socket_used, 4096, lambda x: eval (x) )
```

socket_server.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Aug 15 09:31:42 2021
Start services for EVAL requests coming over TCP sockets
@author: Michael Druckman
"""

tracing_request = 0

# import library symbols for EVAL
import scipy
import mpmath

# import code references for interface implementations
from socket import *
import json
import io

# start EVAL service
serve_eval_8080()
# will loop until STOP is seen
```