# Mechanisms for performing Regression

*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

|  |  |
| --- | --- |
| FloatDemoRegression-LS | // also provided is a set of metrics  // describing the qualities of the  // regression  Linear regression correlation  r^2 = 0.9700834976712961  standard deviation  STD = 0.9750213672872332  covariance  COV = 0.0969206130504208  Mean square error  MSE = 0.9506666666666657  Sum squared residuals  SSR = 92.48  Sum squared error  SSE = 2.851999999999997  Sum square total (for series)  SST = 95.332  Pearson product moment  PC = 0.9849281687876015 |

// 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

// Ln Y = a + b \* X

// Solving for Y gives:

// Y = EXP ( a + b \* X ) = EXP(a) \* EXP ( b \* X )

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

|  |  |
| --- | --- |
| FloatDemoRegression-NL | r^2 = 0.9795614356940878  STD = 0.46072336115906243  COV = 0.40881363964246686  MSE = 0.21226601551770388  SSR = 30.519893637945838  SSE = 0.6367980465531117  SST = 31.15669168449896  PC = 0.9897279600344046 |

// 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

- 0.9083333333333333 \* x^3 + 0.3196428571428571 \* x^4

|  |  |
| --- | --- |
| vandervonde | // the computation of the analysis  // of the curve fit still applies  r^2 = 1.0  STD = 1.1659632469897484E-14  COV = 6.4063914669766386E-15  MSE = 1.3594702933308769E-28  SSR = 584.7879999999997  SSE = 4.078410879992631E-28  SST = 584.788 |

// 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

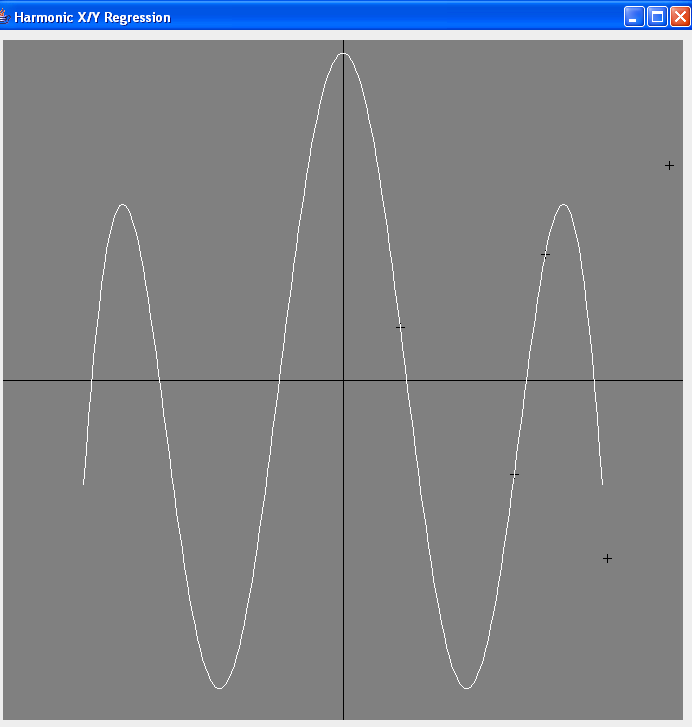
-478250.9996191938

84890.2593673543

]

// fitting the series Y = c0 + c1 cos wX + c2 cos 2wX + c3 cos 3wx + + +

// 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

- 0.9083333333333332 \* x^3 + 0.3196428571428571 \* x^4

|  |  |
| --- | --- |
| lagrange | r^2 = 1.0  STD = 5.02429586778808E-15  COV = 2.7606021251582853E-15  MSE = 2.5243548967072378E-29  SSR = 584.7879999999999  SSE = 7.573064690121713E-29  SST = 584.788 |

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) = c1 cos wx + c2 cos 2wx + c3 cos 3wx + … + s1 sin wx + s2 sin 2wx + …

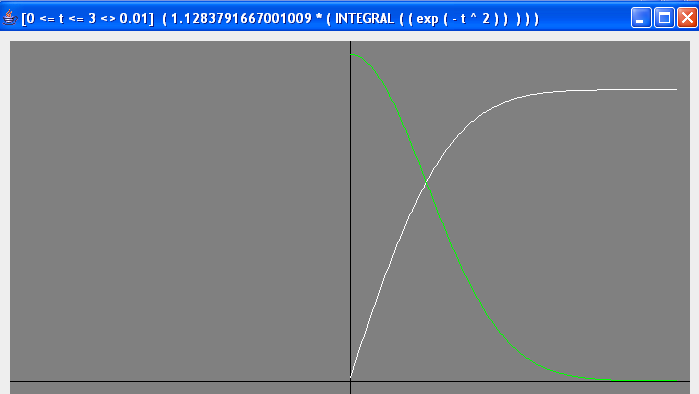
|  |  |
| --- | --- |
| harTransform | // coefficients of cos terms  cosco = (1,2,3)  // coefficients of sin terms  sinco = (4,5,6)  // period multiplier  omega = 2  // transform declared  !% f(x) +#\*= (cosco, sinco, omega)  graph [-4<=x<=4<>0.01] (f(x), f'(x))  // approximation error analysis  err = [0<=x<=4<>0.01] (abs (f'(x) - f'(x <> 0.000001)))  maxErr = MAX err; PRETTYPRINT maxErr  // 2.2E-08 is highest spike  // display of error  graph err |

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

|  |  |
| --- | --- |
| erf-raw | (continued next page) |

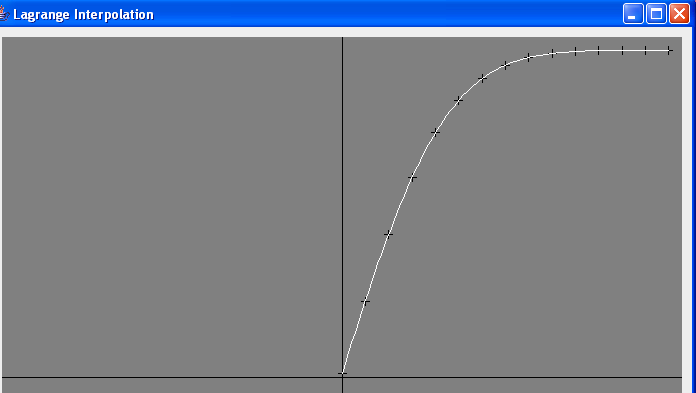


**Functions**

2 ÷ √ ( π ) \* ʃ [ 0 <= t <= 3 Δ 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  1 1.1316668575887787  2 -0.03166565082125494  3 -0.2578731233342175  4 -0.23932128353545323  5 0.3961193902536131 | 6 -0.19500366487283216  7 0.03985950459153642  8 -5.018255826056972E-4  9 -0.0010581619105540785  10 1.1998855568731703E-4 |

]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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) | where P(x) is the approximation function and f(x) is the correct value of the 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 | | erf-arrayint | erf-interp | erfx | | The definition of 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  erfc  Interpolation could also be done on erfc(x) and a polynomial approximation published for this function |

**Functions**

erfc (x) = 1 - ( erf @# x ) // array function evaluation based on the ‘erf’ array

erfx (x) = erfpoly +\*^ x // polynomial based on ‘erfpoly’ coefficients

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 = 0.9999999999999999 - 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

|  |  |
| --- | --- |
| bell-segmented | bell-err |
| 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));

}

}

|  |  |
| --- | --- |
| BELLX | // library import can be used allowing  // CALCLIB access to the compiled version  LIBRARY XPF a.sample.library.ExportedFunctions  !+ BELLX(x) = XPF.bell  // now BELLX is available as any other function in the system  GRAPH [0 <= x <= 3 <> 0.001] ( BELLX(x) )  // the result should be faster execution,  // improved organization and portability  Library  XPF a.sample.library.ExportedFunctions  Functions  **BELLX (x) XPF.bell (double)** |

*FloatDemoVanChe.txt*

Comparison of LaGrange polynomial interpolation against Chebyshev T function interpolation

|  |  |
| --- | --- |
| vanche-li | // 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)  !! lf(x) = x\_LAGRANGE\_jx +\*^ 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  // 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 | **OUTPUT:**  Reading... C:\Projects\CalcLib\scripts\FloatDemoVanChe.txt  Lagrange Interpolation  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  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  maxErf =  1.9089778279246516E-6 |
| jxc =  [  -40.95140998426814  77.09301545187009  -60.27713885060075  39.40757141249924  -21.317397422270005  9.256728076843729  -3.071351473209347  0.7052343269186707  -0.08795456729584572  ] | x\_LAGRANGE\_jx =  [  0.9917303000000143  0.21729748928555637  -2.4849073442426004  11.988874097151438  -37.1869706594598  69.12134722175688  -75.7669097214548  45.13501984081631  -11.258184523720729  ] |

|  |  |
| --- | --- |
| vanche-cf | // 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 |
| vanche-erf | // 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**

|  |  |
| --- | --- |
| vanche-bell-interp2 | 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 |
| vanche-bell-interp | bm = MATRIX (bell, 9, 1)  mm = MATMUL (vcInv, bm)  bcpc = mm|#1  !! bf(x) = bcpc @\*^ x  graph [-1 < x < 1 <> 0.01] ( bf (x) ) |
| vanche-bell | !! 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<=x<=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 <= x <= 1 <> 0.1]:

X = [-1 <= xi <= 1 <> 0.1] (xi)

VC21 = VANCHE X

This gives the constant Vandermonde Chebyshev matrix for T[n] over -1 <= x <= 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**

|  |  |
| --- | --- |
| BesFunOverlay | Calculations of Bessel equation for J0 imported from Colt library versus local interpolation of J0  WHITE  J0 as calculated  Using Colt library  RED  Local Interpolation  Chebyshev Polynomial  Using Vandermonde matrix  And LU decomposition  (outer-most red peaks should be considered likely Runge phenomenon)  (White is plotted first) |

|  |  |
| --- | --- |
| 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

1 1 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

1 0.8 0.64 1 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

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

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 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 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 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 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 0 0 0 0 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 0 0 0 0 0 0 0

1 0.75 0.75 0.97656 -0.0808 -0.44643 -0.37955 -0.35088 0.54945 -0.51471 0.23011 -0.329 1 0 0 0 0 0 0 0 0

1 0.2 0.64 -1 0.99289 0.45714 -0.08637 -0.29942 0.10047 0.42353 -0.88364 -0.72193 0.33758 1 0 0 0 0 0 0 0

1 0.85 0.51 0.92969 0.07692 0.49583 0.51524 0.23815 -0.42621 0.79852 -0.119 0.21875 -0.76718 0.41319 1 0 0 0 0 0 0

1 0.55 0.99 0.25781 -0.10666 -0.39286 -0.18556 -0.34308 0.38374 0.35948 -0.375 0.22978 -0.48352 -0.52083 -0.67873 1 0 0 0 0 0

1 0.25 0.75 -0.97656 0.88882 0.81845 -0.07732 -0.25016 0.11193 0.41939 -0.8125 -0.49786 0.34921 0.94039 0.17507 -0.51587 1 0 0 0 0

1 0.7 0.84 0.875 -0.1448 -0.73333 -0.55421 -0.64042 0.93122 -0.43617 0.26 -0.31863 0.89397 0.24074 0.44818 -0.33016 -0.32 1 0 0 0

1 0.35 0.91 -0.71094 0.52941 0.975 0.09211 0.25544 -0.17143 -0.48176 0.78975 0.16131 -0.18857 -0.40625 -0.22689 0.55714 -0.72 0.5 1 0 0

1 0.6 0.96 0.5 -0.16548 -0.68571 -0.38866 -0.62877 0.77363 0.36235 -0.324 0.26471 -0.6189 -0.5 -0.74467 0.91429 0.29538 -0.71795 -0.41026 1 0

1 0.45 0.99 -0.25781 0.14932 0.4125 0.1169 0.27018 -0.24176 -0.45294 0.6075 -0.12408 0.20308 0.32812 0.30543 -0.6 0.38769 -0.53846 -0.76923 0.75 1

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

VC21U =

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 0 -2 -0 -0 0 -2 0 0 -0 -2 -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 -0.27424 -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 -2.0226

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 0.3078 0.23934 0.05676

0 0 0 0 0 0 -3.41453 -2.73162 0.13658 -1.47508 -1.36581 1.46633 -0.05966 -0.53356 -0.24321 -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 -1.40416 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 2.25608 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 -2.43151 0.52382 2.53907 0.01474 -0.19686

0 0 0 0 0 0 0 0 0 0 0 -2.96479 2.37183 1.77887 -2.94107 1.90221 0.32636 -0.04345 -0.76976 -1.21119 2.69323

0 0 0 0 0 0 0 0 0 0 0 0 2.5361 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 0 -1.88351 1.88351 2.26021 -3.45059 1.59722 0.31281 -2.93804 2.97948

0 0 0 0 0 0 0 0 0 0 0 0 0 0 -2.02347 -0.80939 3.31848 1.74827 0.845 -0.22922 -4.32225

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1.64807 -0.98884 4.15314 3.24341 -2.81623 -3.99334

0 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 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 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 0 0.16876 0.03375

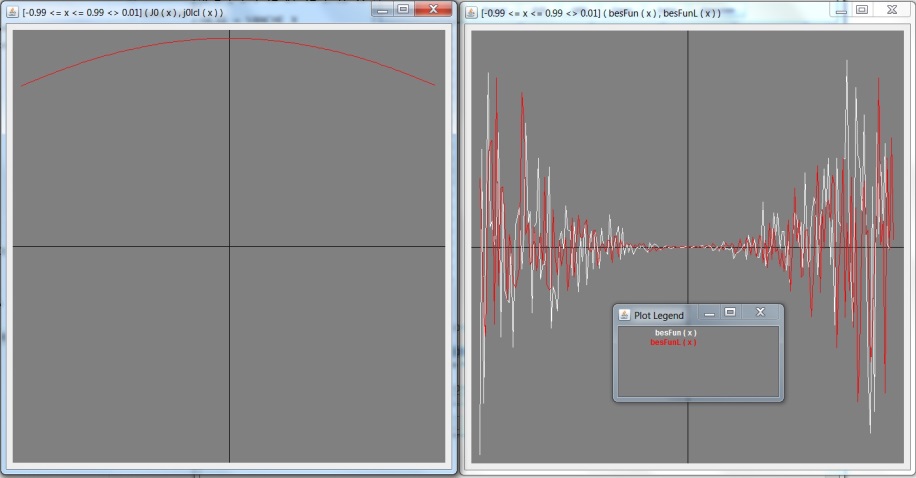
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.07594

VC31 Spline Organization



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 ½ leaving the other ½ 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 errL1.994155862661273E-6

err = [-1 < x < 1 <> 0.01] (abs(besFun(x))); CALC MAX err2.455309003113193E-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

|  |  |
| --- | --- |
| BesEQ-VC31-Plot | 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 |
| Bes-j0x-3 | 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)) |
| BesFunSpline | // 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))

|  |  |
| --- | --- |
| BesJ0 | READ VC31Prep.txt  READ BesselFunctions.txt  // 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)) |

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.

|  |  |
| --- | --- |
| // 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) ) | // 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)) |

**Mix and Match using High-Order Polynomials**

|  |  |  |
| --- | --- | --- |
| bell-interpolate | bell-curve-plot | bell-error-plot |
| **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.txtwhere 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.txtwhere 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.

|  |  |  |  |
| --- | --- | --- | --- |
| bell-seg3-console | bell-seg3-interpolate | bell-seg3-error | bell-seg3-curve |
| This follow-up uses 3 segments, first a high order Chebyshev Interpolation followed by smaller Lagrange polynomials | The Chebyshev Interpolation tracks very well in the 0-1 domain section, but changing the interpolation for 1-2 and 2-3 segments produces far better results on the error plot | The error curve shows peaks at the knots (stitch points), but outside these the error is far smaller. The highest peak is 6.7E-08, other than the stitch points the peaks are in 1E-10 to 1E-09, at 0 the error is 1E-16 | And of course the end product curve appears to have exceptionally smooth continuous shape |

**ChebCircle8th.txt**

Clenshaw 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

|  |  |  |
| --- | --- | --- |
| ChebCircle-interp-plot | ChebCircle-poly-plot | ChebCircle-error-plot |
| The scatter plot from the Chebyshev interpolation of the data | A plot of the Chebyshev polynomial generated | The error plot showing 4E-09 as the maximum value of the error over the interval |

|  |  |
| --- | --- |
| // 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 / 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 Clenshaw 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 |

|  |  |
| --- | --- |
| ChebCircle-data | 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**

|  |  |
| --- | --- |
| pi-approx-crop | pi-approx-crop-2 |
| 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 | An alternative is to break the area down and compute areas of sections using simple geometry and this leaves a smaller integral  **CrustLineSlope = - (1 – 2\2 / 2) / ( 2\2 / 2) = 1 – sqrt(2)** |
| pi-approx-smaller | // equation for a unit circle  !! f(x) = sqrt (1 - x^2)  // 45DEG is 1/8 of circle, sin=cos=sqrt(2)/2r2 = 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) |
| PRETTYPRINT areaCrust  areaCrust = 0.03914569119254674  CombinedAreaOfTriangles = r2 / 4  sliceArea = areaCrust + CombinedAreaOfTriangles | // LowTriArea = (r2 /2) ^ 2 / 2  // HiTriArea = [ (1 – r2) \* r2 / 2 ] / 2  // CombinedAreaOfTriangles = (r2 – 1) / 4 + 1 / 4 = r2 / 4 |
| PRETTYPRINT sliceArea  sliceArea = 0.3926990817858205 | piApprox = sliceArea\*8  piApprox = 3.14159265  // So the approximations are nearly identical |
| Compare the interpolated area approximation | To use of Tanh-Sinh quadrature built-in as $| function suffix |
| // 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 | // 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\2 /4)  3.141592653589793 |

# Differential Equations

|  |  |
| --- | --- |
|  | Select an equation “Run DiffEQ Solution Test”  From the Function drop-down menu |
|  | The “Solution Test” tool is shown The information about the equation is shown  The required parameters are shown with their values  The RK4 button starts the Runge-Kutta tool |
|  | The “Solution Approximation” screen shows the set values of the RK4 formula parameters Press the OK button to run the T0 approximation |

|  |  |
| --- | --- |
|  | 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 |
|  | Now the solution function “y” can be found in the functions list along with the derivatives y’ and y’’ |
|  | 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 know from the error plot |

# Test the Solution Quality

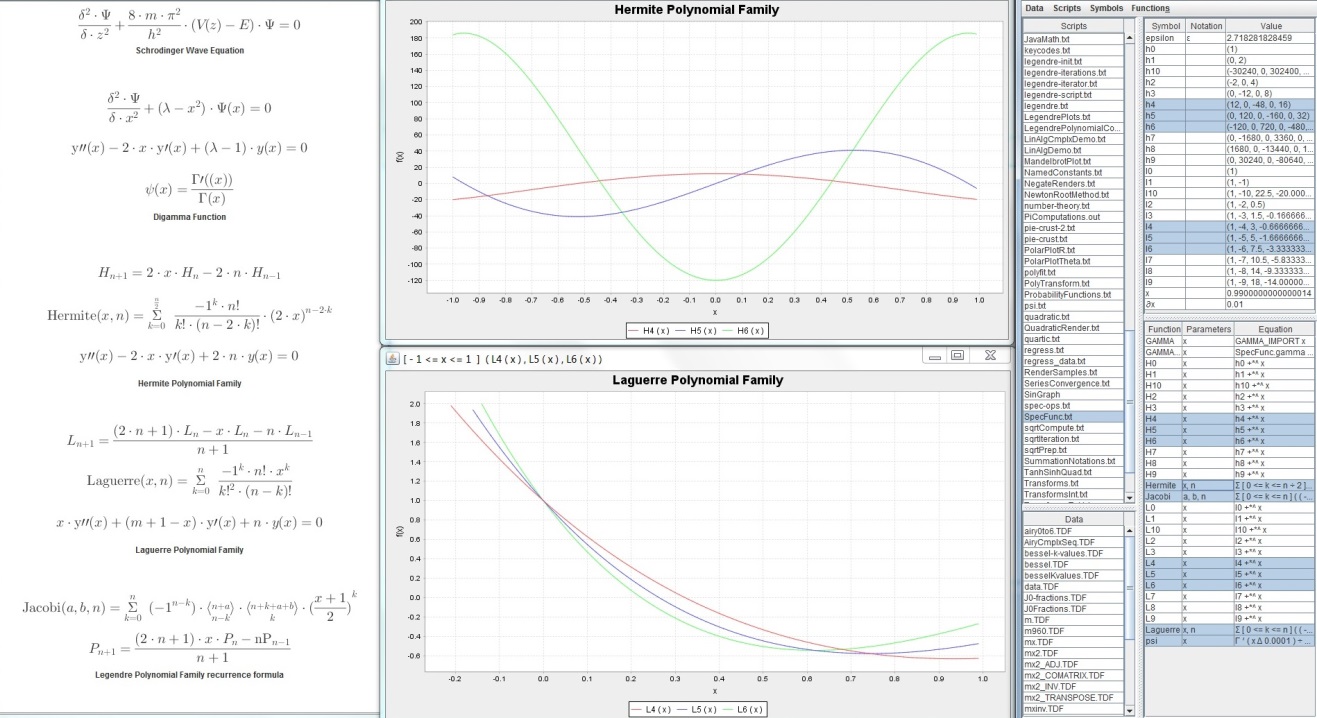
|  |  |
| --- | --- |
|  | The Solution Test screen now shows the solution “y” has a “Function Spline” available for test |
|  | The OK button show a form for entering domain parameters |
|  | A plot of the solution error for the domain is shown An error spike is typically seen at each knot |

# 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 |
|  | 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 |
|  | Enter domain description |
|  | The error plot is displayed |

# 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

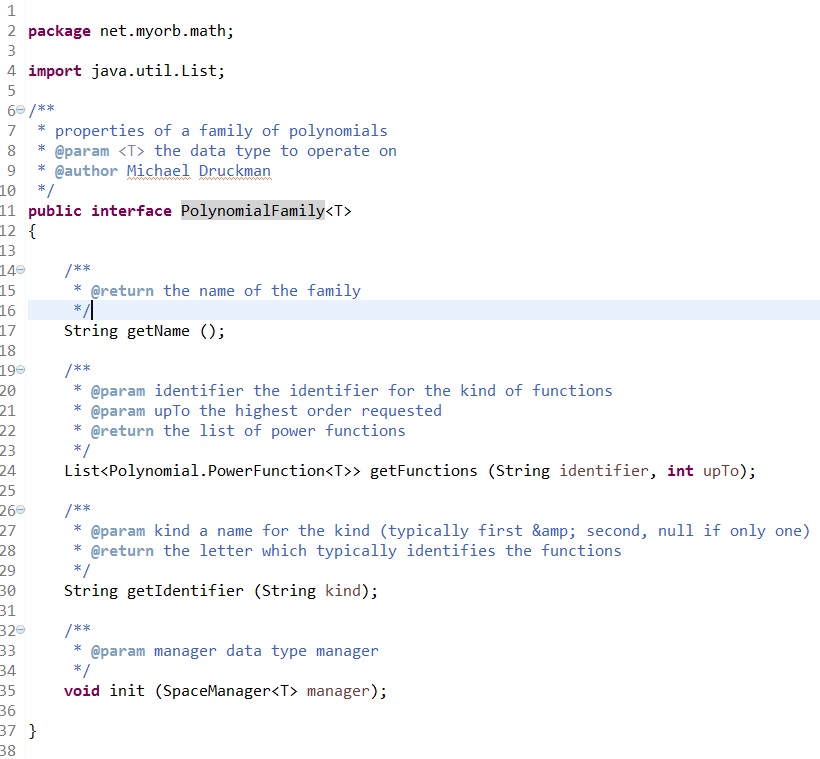
****

**The SpecFunc script demonstrates this functionality**

|  |  |
| --- | --- |
| // Schrodinger  render delta^2 \* PSI / (delta\*z^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/2 ] ( (-1)^k \* n! /  ( k! \* (n - 2\*k)! ) \* (2\*x)^(n - 2\*k) )  RENDER H#(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! \* x^k / ( (k!)^2 \* (n-k)! ) )  RENDER L#(n+1) = ( (2\*n+1)\*L#n - x\*L#n - n\*L#(n-1) ) / (n+1)  RENDERF Laguerre  render x\*y''(x) + (m + 1 - x)\*y'(x) + n\*y(x) = 0  FAMILY Laguerre 10  GRAPH [ -1 <= x <= 1 <> 0.01 ] ( L4(x), L5(x), L6(x) )  RENDER "Laguerre Polynomial Family" TOP  // Jacobi Polynomials  !! Jacobi(a,b,n) = SIGMA [ 0 <= k <= n ] ( (-1)^(n-k) \*  ((n+a)##(n-k)) \* ((n+k+a+b)##k) \* ((x+1)/2)^k )  RENDERF Jacobi  // Legendre Polynomials  render P#(n+1) = ( (2\*n+1)\*x\*P#n - nP#(n-1) ) / (n+1)  RENDER "Legendre Polynomial Family recurrence formula" TOP |

# Polynomial Families

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



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

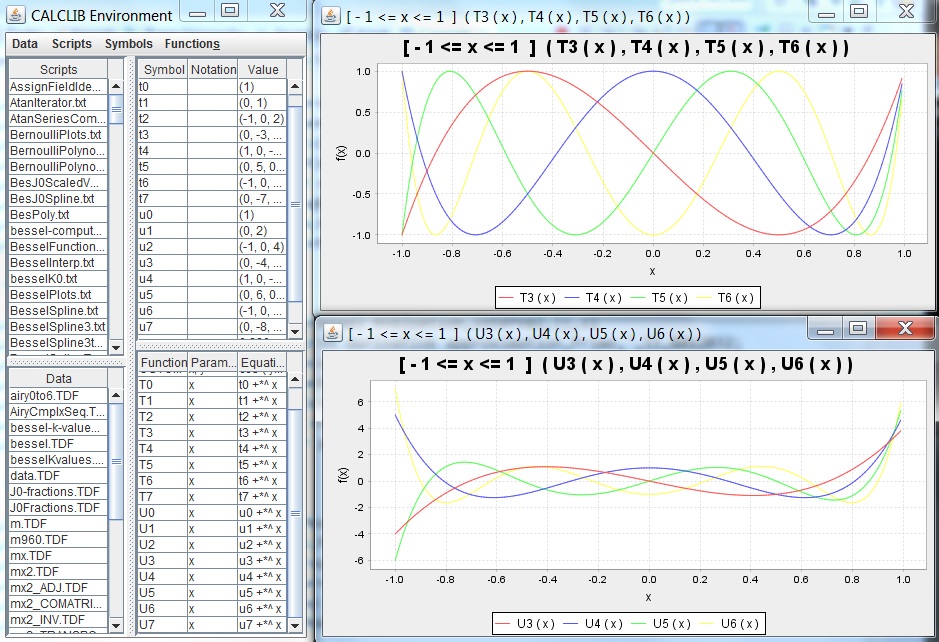
// ChebyshevFamily.txt

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

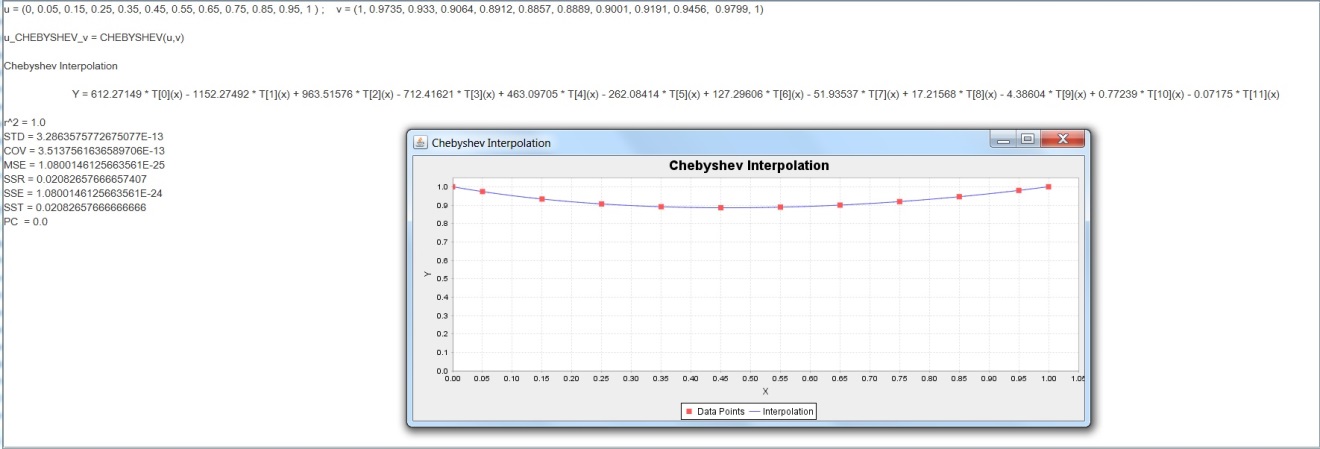
CalcLib was used to build a spline for the Gamma Function. The core interval is the domain 1 to 2. Special treatment is given to x=1, x=1.5, and x=2. Other parts of the domain are calculated by the generating functions

GAMMA (x + 1) = x \* GAMMA (x)

Or

GAMMA (x) = GAMMA (x + 1) / x

Chebyshev interpolation provided the coefficients for the polynomial

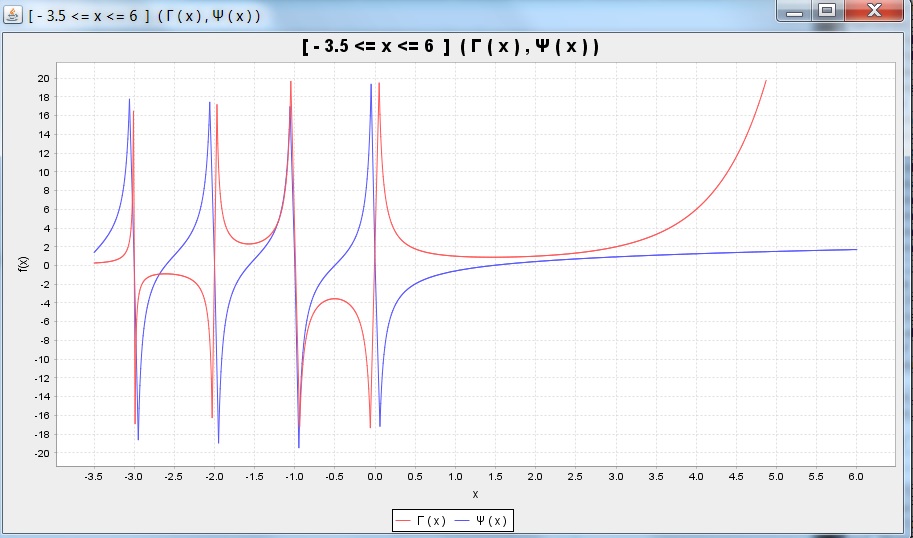


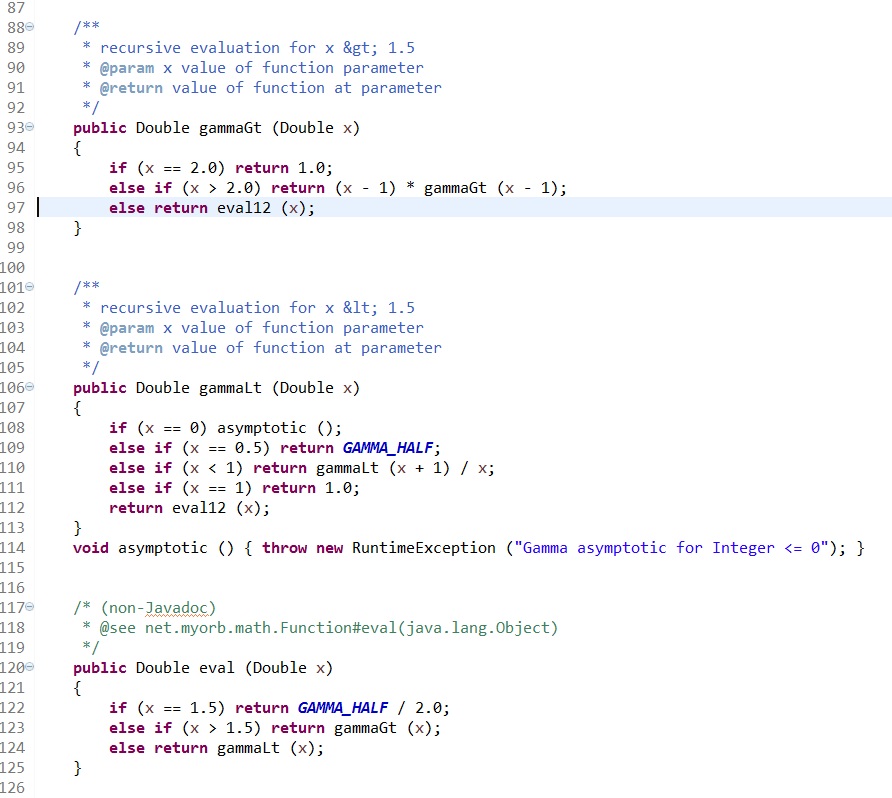
LIBRARY SpecFunc net.myorb.math.specialfunctions.Library

!+ GAMMA(x) = SpecFunc.gamma

!! PSI(x) = GAMMA'(x <> 0.0001)/GAMMA(x)

GRAPH [ -3.5 <= x <= 6 <> 0.01 ] ( GAMMA(x), PSI(x) )





# Anti Derivative Spline

Working with improper integrals can generate time costly algorithms which are painfully slow to execute. Analysis techniques for improper integrals call for breaking the equation into sums of segments of the overall domain interval. Translation of this concept into computer science techniques of numerical analysis suggest the use of spline concepts. By breaking the integral into its component segments and calculating the result of each segment, a description of the full equation can be seen. The sum of segment contributions can be used to evaluate much of an interval and the correct evaluation over the interval of interest can be computed as a difference of the computed evaluation of a smaller integral.

This is the full source of ExponentialIntegral class which calculates Ei(x) using the AntiDerivativeSpline library features:

**package** net.myorb.math.specialfunctions;

**import** net.myorb.math.computational.ADSplineRealSegmentManager;

**import** net.myorb.math.Function;

/\*\*

\* general support for Exponential Integral Functions

\* **@author** Michael Druckman

\*/

**public** **class** ExponentialIntegral

{

/\*\*

\* describe the Anti Derivative as segments covering portions of the domain

\*/

**public** **static** **class** Segments **extends** ADSplineRealSegmentManager

{

**public** **static** **final** **double**[]

***UP\_TO*** = **new** **double**[]{-10, -1, -0.1, -0.000001, -0.0000000001, 0.000001, 0.1, 1, 10},

***AREA*** = **new** **double**[]

{

-4.15696892968E-6,

-0.2193797774266,

-1.603540023,

-11.4153,

-9.21,

9.21,

11.615481,

3.51793063,

2490.333858425

};

**public** Segments () { **super** (***UP\_TO***, ***AREA***); }

/\* (non-Javadoc)

\* @see net.myorb.math.computational.ADSplineRealSegmentManager#getFirstSegmentBase()

\*/

**public** Double getFirstSegmentBase () { **return** -100d; }

/\* (non-Javadoc)

\* @see net.myorb.math.computational.ADSplineRealSegmentManager#eval(java.lang.Double)

\*/

**public** Double eval (Double x) { **return** Math.*exp* (x) / x; }

}

/\*

\* Ei(x) = exp x INTEGRAL [0<t<INF] ((exp -xt) / (1-t)) dt

\*/

/\*\*

\* **@param** x parameter value

\* **@return** INTEGRAL [-INFINITY:x] exp(t)/t dt

\*/

**public** **static** **double** Ei (**double** x)

{

**if** (*spline* == **null**)

{ *spline* = **new** Segments ().newSplineInstance (); }

**return** *spline*.eval (x);

}

**static** Function<Double> *spline* = **null**;

/\*

\* E1(x) = -gamma -ln x + INTEGRAL [0<t<x] (1 - exp -t)/t dt

\* E1(x) = x \* INTEGRAL [1<t<INFINITY] (exp -xt \* ln t) dt

\*/

/\*\*

\* **@param** x parameter value

\* **@return** INTEGRAL [x:INFINITY] exp(-t)/t dt

\*/

**public** **static** **double** E1 (**double** x)

{

**return** -*Ei* (-x);

}

/\*\*

\* **@param** x parameter value

\* **@return** Ei ( ln x )

\*/

**public** **static** **double** li (**double** x)

{

**return** *Ei* (Math.*log* (x));

}

}

Graphs below show Ei, E1, and li plots generated using this code.

The constants in the above code were generated with a tool class AnalysisTool shown referenced in code segment below:

**public** **class** ExpInt

{

**static** String *lo* = "1", *hi* = "10", *inc* = "0.1";

**public** **static** **void** main (String[] args)

{

PrimitiveRangeDescription range = **new** PrimitiveRangeDescription (*lo*, *hi*, *inc*);

AnalysisTool.*display* (range, **new** Ei (), 25);

}

**static** **class** Ei **implements** Function<Double>

{

/\* (non-Javadoc)

\* @see net.myorb.math.Function#eval(java.lang.Object)

\*/

**public** Double eval (Double x) { **return** Math.*exp* (x) / x; }

/\* (non-Javadoc)

\* @see net.myorb.math.Function#getSpaceManager()

\*/

**public** SpaceManager<Double> getSpaceManager() { **return** sm; }

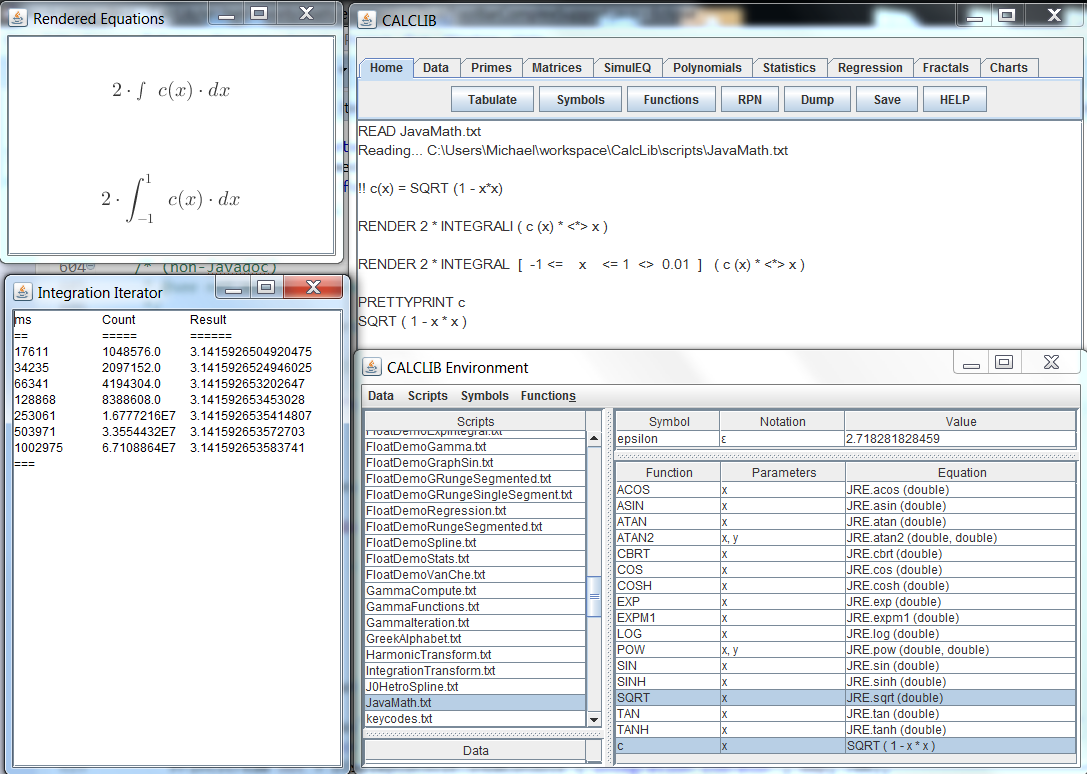
SpaceManager<Double> sm = **new** ExpressionFloatingFieldManager ();

}

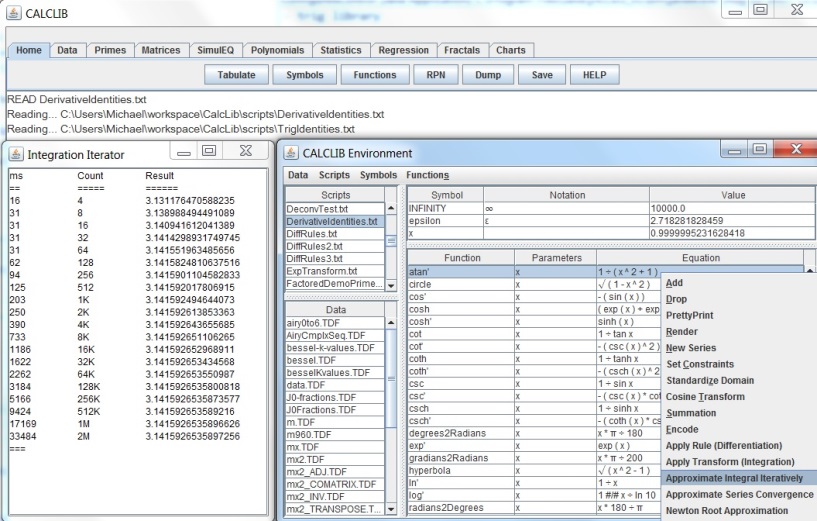
}

# 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.



# 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

// = 4946041176255201878775086487573351061418968498177 / 3497379255757941172020851852070562919437964212608

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

radical2 = sqrtx

// 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

// = 38724300764502680644567946090543611814594121481337355363539093593852710780984930941516221071647971468276373108204166977

206077712680271927759102018748518486070845634112299488263564859436406918411999470275

// / 351081594038816320929297562816137461689144827667834786009789239684083346499308205283520216578632712876164962142367521057398

56126714936857216329479003182021378136916689419412298749856707187360789430272

// = ( 5^2 \* 7^3 \* 13 \* 23 \* 29 \* 31 \* 37 \* 41 \* 43 \* 47 \* 53 \* 59 \* 61 \* 67 \* 71 \* 73 \* 79 \* 83 \* 191 \* 1451 \*

4552468928477187574050463918838755530501788164295356574114208913461935568005590671309567043

4403864087806034470850422555296422129968289083320181488913830968277491841459 )

// / ( 2^154 \* 3^153 \* 11^77 )

PRETTYPRINT s 50

c = 2 \* radical2 / 9801

// c = 0.00028858556522254770917287801738796002011420709630

// = 4946041176255201878775086487573351061418968498177 / 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

// = 106350882988564395570252386882913300824112521658619763797244208822275879453109659023010481430065976159254416616124366527517669023328669747469931773952

// / 33852537459635572260728729160404365911842247830713137420558614376094550354462810792014557657118680669891956744732872439287244601612731402376166467685

// = ( 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 \*

83854987397673428314269118937324505750737151609756676749664030413280866345417661675621 )

// 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 =

-0.5000000000000000 = -1 / 2 = ( - 1 ) / ( 2 )

sqrtx\_squared =

2.2500000000000000 = 9 / 4 = ( 3^2 ) / ( 2^2 )

sqrtx =

1.5000000000000000 = 3 / 2 = ( 3 ) / ( 2 )

sqrt\_iteration =

0.0833333333333334 = 1 / 12 = ( 1 ) / ( 2^2 \* 3 )

sqrtx\_squared =

2.0069444444444445 = 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 =

2.0000000000045110 = 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 =

2.0000000000000001 = 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 =

8.9929283216504540E-25 = 1 / 1111984844349868137938112 = ( 1 ) / ( 2^6 \* 3 \* 17 \* 257 \* 577 \* 1409 \* 665857 \* 2448769 )

sqrtx\_squared =

2.0000000000000001 = 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 =

2.8592838433339520E-49 = 1 / 3497379255757941172020851852070562919437964212608 =

( 1 ) / ( 2^7 \* 3 \* 17 \* 257 \* 577 \* 1409 \* 11777 \* 665857 \* 2393857 \* 2448769 \* 55780318173953 )

sqrtx\_squared =

2.0000000000000001 =

24463323317211940977293404419928347279246091129334268572773850449273611455484253634058690852323329 / 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 =

116354295547844200479625540962705305445031010498388307062857519290687784871920308555177681218916232885 / 105488932654005296215338934589125951145247950655030308540653845613856673595635654924408113432887296 =

( 5 \* 7^2 \* 13 \* 17 \* 19 \* 23 \* 29 \* 31 \* 37 \* 41 \* 43 \* 83854987397673428314269118937324505750737151609756676749664030413280866345417661675621 ) / ( 2^74 \* 3^78 \* 11^37 )

c =

0.00028858556522254770917287801738796002011420709630 = 4946041176255201878775086487573351061418968498177 /

17138907042841790713488184501071793586705743623885504 =

( 7681 \* 1492993 \* 431302713980890947612633357964569696769 ) / ( 2^6 \* 3^5 \* 11^2 \* 17 \* 257 \* 577 \* 1409 \* 11777 \* 665857 \* 2393857 \* 2448769 \* 55780318173953 )

piApproximation =

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862804 =

106350882988564395570252386882913300824112521658619763797244208822275879453109659023010481430065976159254416616124366527517669023328669747469931773952 /

33852537459635572260728729160404365911842247830713137420558614376094550354462810792014557657118680669891956744732872439287244601612731402376166467685 =

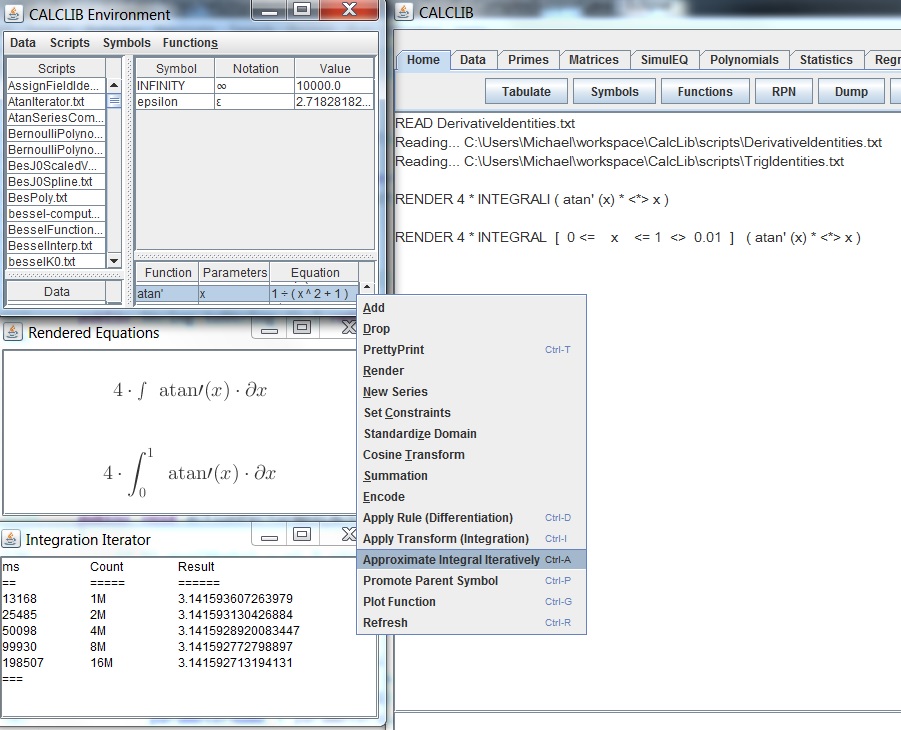
( 2^80 \* 3^83 \* 11^39 \* 257 \* 577 \* 1409 \* 11777 \* 665857 \* 2393857 \* 2448769 \* 55780318173953 ) /

( 5 \* 7^2 \* 13 \* 19 \* 23 \* 29 \* 31 \* 37 \* 41 \* 43 \* 7681 \* 1492993 \* 431302713980890947612633357964569696769 \*

83854987397673428314269118937324505750737151609756676749664030413280866345417661675621 )

# 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.

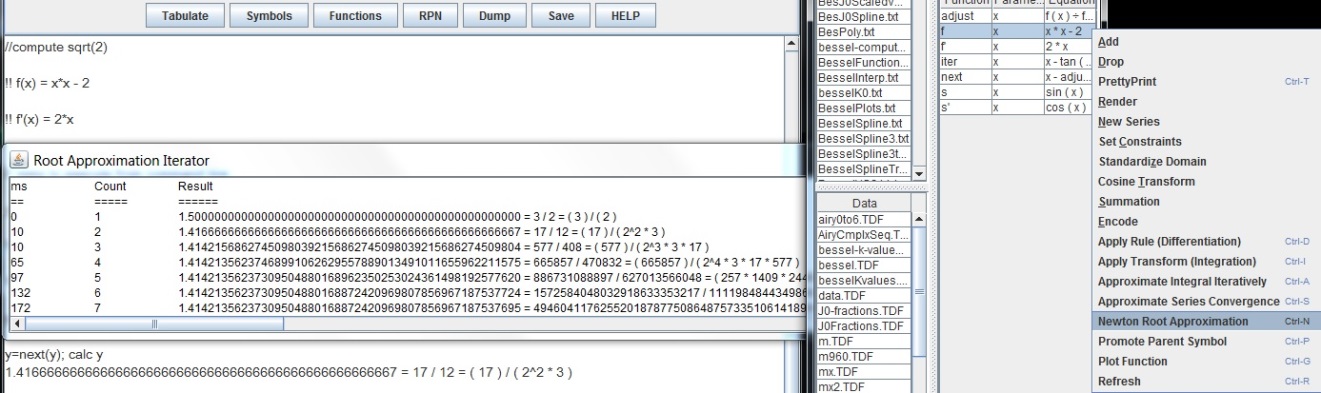


# Approximate Series Convergence

|  |  |
| --- | --- |
| SeriesIteration | 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. |
| atanSeries | Another Example  This one shows the James Gregory ATAN series being used with the Leonhard Euler PI computation from integer ratios  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.



# Imported Libraries

**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 =

0.9999999999999998 -3.000000000000001 2

7.9999999999999964 5.999999999999999 -2.9999999999999973

-1.999999999999998 1.0000000000000042 7.0000000000000036

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

]

# Script Execution Samples

**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 =

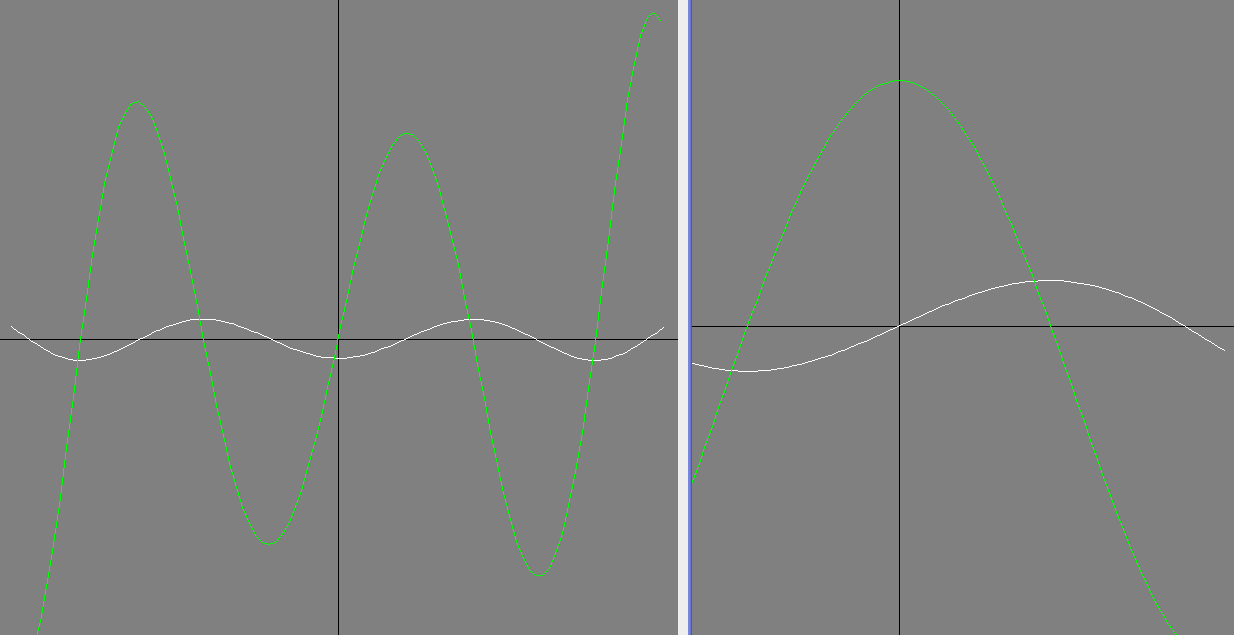
1.875 \* x - 8.75 \* x^3 + 7.875 \* x^5

pn =

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

Iteration 5 has completed

\*\*\* Maximum iteration count exceeded

**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).

|  |  |
| --- | --- |
| ChebEqTst2 | // ChebEQExec.txt:  !% TnD1(x) <>= Tn  !% TnD2(x) <>= TnD1  // display of plots of Tn, Tn', & Tn''  GRAPH [-1 <= x <= 1 <> 0.01] (Tn(x), TnD1(x), TnD2(x))  !% UnD1(x) <>= Un  !% UnD2(x) <>= UnD1  // display of plots of Un, Un', & Un''  GRAPH [-1 <= x <= 1 <> 0.01] (Un(x), UnD1(x), UnD2(x))  READ ChebEQDef.txt  // display MAX error and graph of error of first kind  GRAPH [-1 <= x <= 1 <> 0.01] (abs(chebeq1(x)))  err = [-1 <= x <= 1 <> 0.001] (abs(chebeq1(x)))  errT = MAX err; PRETTYPRINT errT  // display MAX error and graph of error of second kind  GRAPH [-1 <= x <= 1 <> 0.01] (abs(chebeq2(x)))  err = [-1 <= x <= 1 <> 0.001] (abs(chebeq2(x)))  errU = MAX err; PRETTYPRINT errU |
| ChebEqT7Dn | Graphs of Un and Tn overlaid with their first and second derivative functions |

**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:

(-7 - 11\*i) (-11 + 10\*i) (14 - 5\*i)

(-11 - 16\*i) (-13 + 11\*i) (19 - 4\*i)

(18 + 12\*i) (12 - 18\*i) (-15 + 12\*i)

=EOD=

mx2 =

(-7 - 11\*i) (-11 + 10\*i) (14 - 5\*i)

(-11 - 16\*i) (-13 + 11\*i) (19 - 4\*i)

(18 + 12\*i) (12 - 18\*i) (-15 + 12\*i)

mx2\_DET =

(-369 + 108\*i)

mx2\_TRANSPOSE =

(-7 - 11\*i) (-11 - 16\*i) (18 + 12\*i)

(-11 + 10\*i) (-13 + 11\*i) (12 - 18\*i)

(14 - 5\*i) (19 - 4\*i) (-15 + 12\*i)

mx2\_COMATRIX =

(-93 + 69\*i) (33 + 48\*i) (-54 - 36\*i)

(33 - 30\*i) (-75 + 3\*i) (-36 + 54\*i)

(-42 + 15\*i) (-57 + 12\*i) -69

mx2\_ADJ =

(-93 + 69\*i) (33 - 30\*i) (-42 + 15\*i)

(33 + 48\*i) (-75 + 3\*i) (-57 + 12\*i)

(-54 - 36\*i) (-36 + 54\*i) -69

mx2\_INV =

(0.28256 - 0.10429\*i) (-0.10429 + 0.05078\*i) (0.1158 - 0.00676\*i)

(-0.04731 - 0.14393\*i) (0.18941 + 0.04731\*i) (0.15105 + 0.01169\*i)

(0.10849 + 0.12932\*i) (0.12932 - 0.10849\*i) (0.17224 + 0.05041\*i)

mx2\_TR =

(-35 + 12\*i)

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

Von Mises dominant Eigen-Pair

Dominant Eigenvalue is: 42.720018726587654

Dominant Eigenvector is:

[

(0.45054741724475555 + 0.06575787393774088\*i),

(0.5946656941881084 + 0.005590057346825491\*i),

(-0.6128582806028051 - 0.25185138105731264\*i)

]

**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 =

5.4444444444444455

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 =

5.000000000000843

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)

**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 =

-0.0333333333333334 = -1 / 30 = ( - 1 ) / ( 2 \* 3 \* 5 )

bn =

0

bn =

0.0238095238095239 = 1 / 42 = ( 1 ) / ( 2 \* 3 \* 7 )

bn =

0

bn =

-0.0333333333333334 = -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

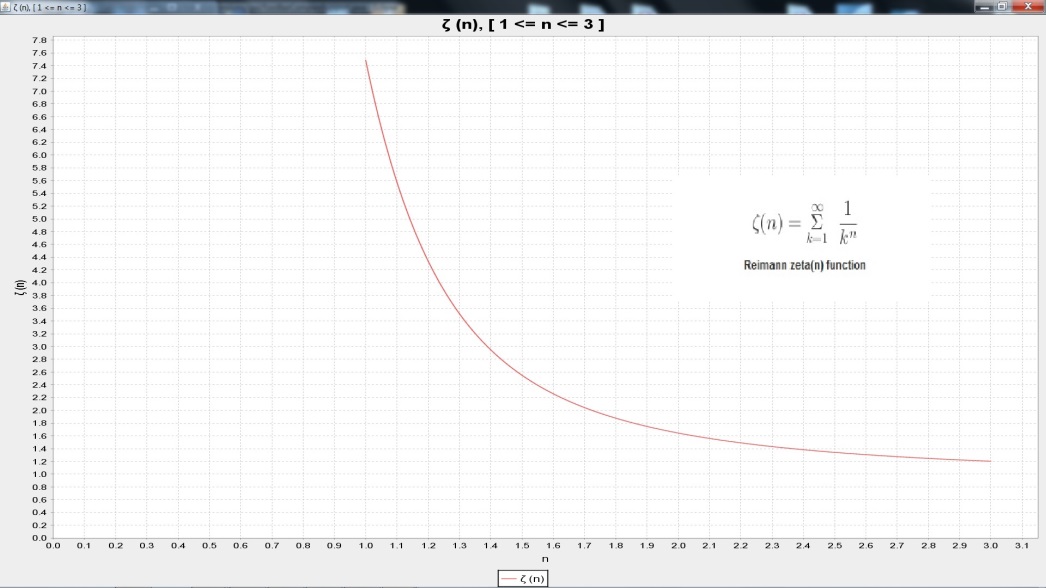
\*\*\* Maximum iteration count exceeded

// the correct values of the numerators and denominators of these ratios are published online:

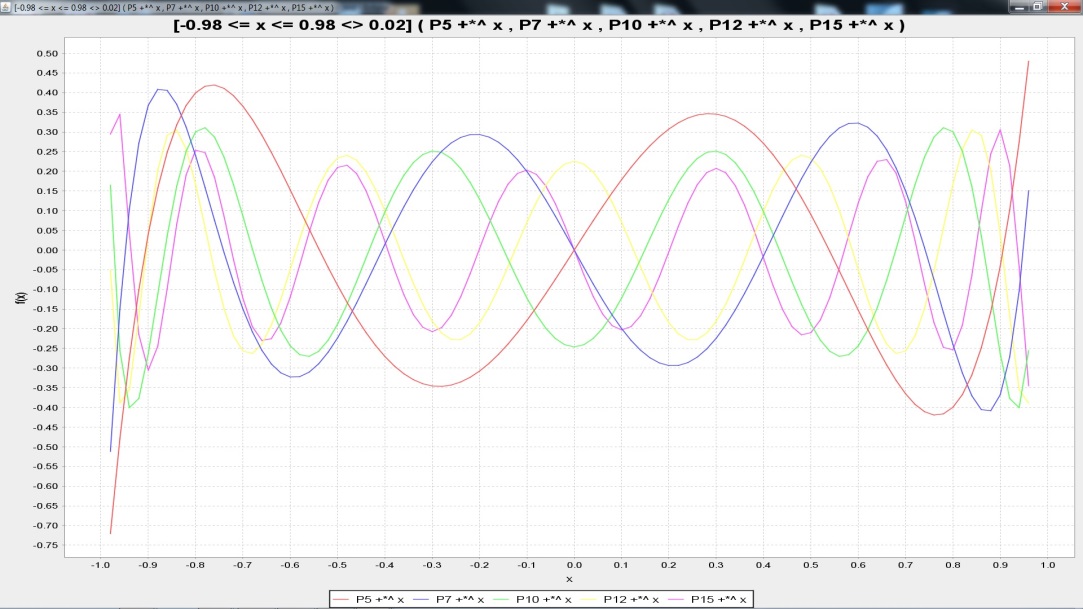
// <https://oeis.org/>  ([A027641](https://oeis.org/A027641) / [A027642](https://oeis.org/A027642) in [OEIS](https://en.wikipedia.org/wiki/OEIS));

# Alternative Graphics Engines

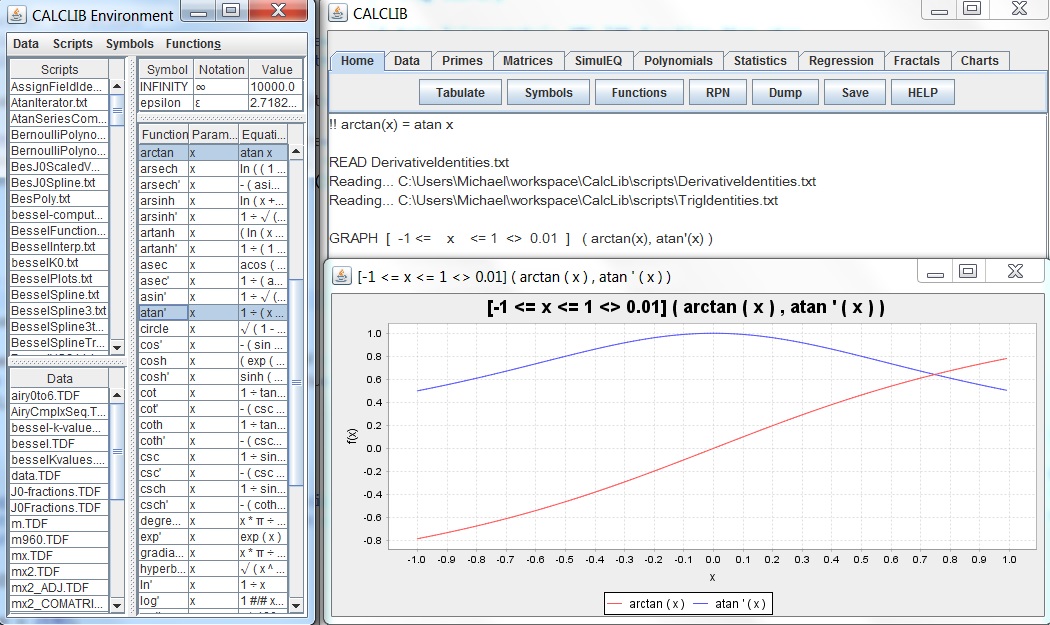
The CalcLib engine has public interfaces for external calculation libraries, MathML rendering engines, and for graphics libraries. A library that can generate plots can be referenced via the application configuration and the interface to provide for substitution of the plot engine. The plot of Zeta below 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 theCalcLib configuration set to use the JFreeChart implementation. The override for each type of chart is available in the interface, implementations can choose to override some and optionally not others.



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

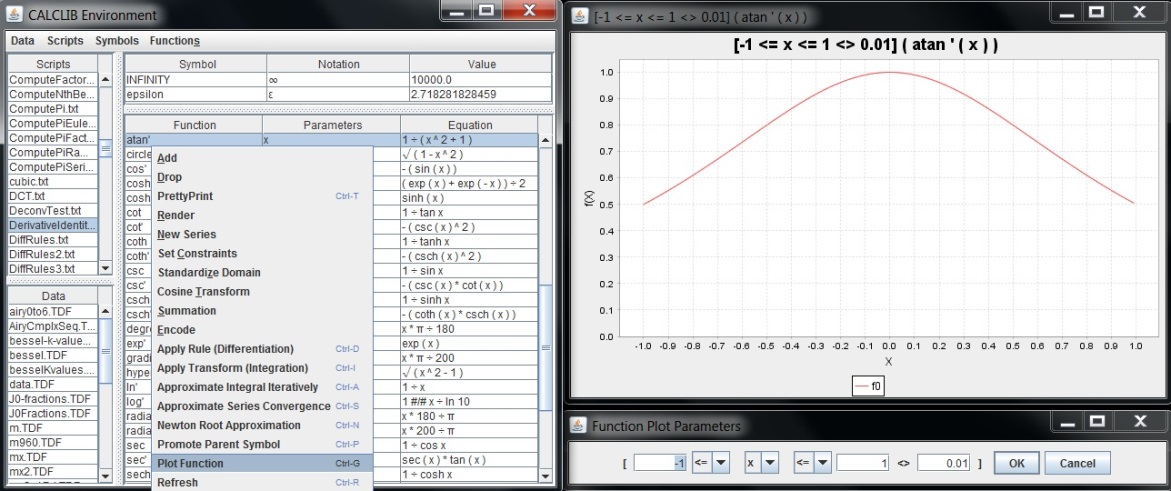


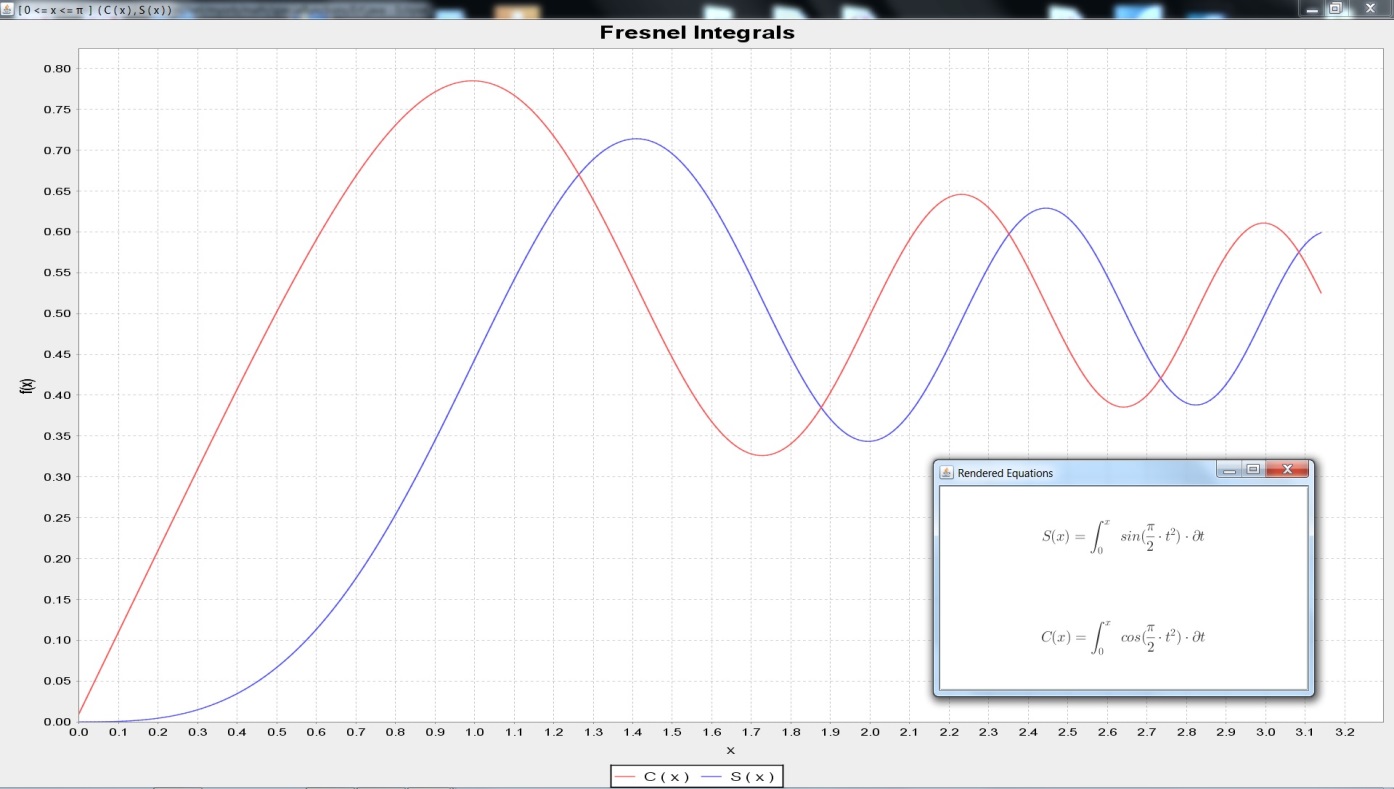
Using Plot Function menu item causes the Function Plot Parameters form to be displayed.

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.





*FresnelIntegrals.txt*

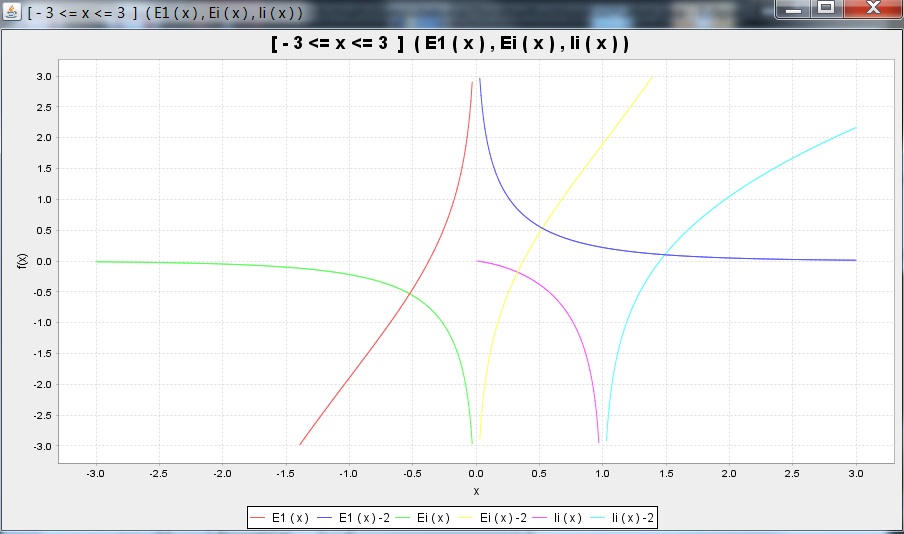
!! S(x) = INTEGRAL [0 <= t <= x <> 0.01] ( sin (pi / 2 \* t^2 ) \* <\*> t )

!! C(x) = INTEGRAL [0 <= t <= x <> 0.01] ( cos (pi / 2 \* t^2 ) \* <\*> t )

RENDERF S

RENDERF C

GRAPH [ 0 <= x <= pi <> 0.01 ] ( C(x), S(x) )



*ExpInt.txt*

LIBRARY SpecFunc net.myorb.math.specialfunctions.Library

!+ EI\_IMPORT(x) = SpecFunc.ei

!! Ei(x) = EI\_IMPORT x

!+ E1\_IMPORT(x) = SpecFunc.e1

!! E1(x) = E1\_IMPORT x

!+ LI\_IMPORT(x) = SpecFunc.li

!! li(x) = LI\_IMPORT x

GRAPH LIM 3 [ -3 <= x <= 3 <> 0.01 ] ( Ei(x), li(x), E1(x) )

Note: The range limit is set to 3 so the plots truncate at the point just before/after f (x) > 3

|  |  |
| --- | --- |
| C:\Users\Michael\Desktop\software backup\plots\P.jpg | LegendrePlots*.txt*  FAMILY Legendre 20 First  graph [ -1 <= x < 1.02 <> 0.01 ]  ( P5(x), P4(x), P3(x), P2(x), P1(x), P0(x) )  ENTITLED Legendre Polynomial Family Pn(x),  0 <= n <= 5, -1 <= x <= 1 |
| C:\Users\Michael\Desktop\software backup\plots\legendreQ.jpg | LegendreQ*.txt*  LIBRARY Legendre net.myorb.math.specialfunctions.Legendre  !+ Q0\_IMPORT(x) = Legendre.Q0  !+ Q1\_IMPORT(x) = Legendre.Q1  !+ Q2\_IMPORT(x) = Legendre.Q2  !+ Q3\_IMPORT(x) = Legendre.Q3  !+ Q4\_IMPORT(x) = Legendre.Q4  !+ Q5\_IMPORT(x) = Legendre.Q5  !! Q0(x) = Q0\_IMPORT x  !! Q1(x) = Q1\_IMPORT x  !! Q2(x) = Q2\_IMPORT x  !! Q3(x) = Q3\_IMPORT x  !! Q4(x) = Q4\_IMPORT x  !! Q5(x) = Q5\_IMPORT x  GRAPH LIM 3 [ -1 <= x <= 1 <> 0.01 ]  ( Q0(x), Q1(x), Q2(x), Q3(x), Q4(x), Q5(x) )  ENTITLED Legendre Q0 - Q5 [-1,1] |
| C:\Users\Michael\Desktop\software backup\plots\LegQ.png | This is a plot of Q0 .. Q5 over interval [-1,3]  The graph zoom feature was then used to reduce the displayed domain to [0.6,1.4] to focus on the asymptote which occurs at x=1. The range limit is set to 3 so the plots truncate at the point just before/after Q(x) > 3 |

Other Plot Types

Fractal Plots

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| madelbrot | julia285 | julia-dendrite | julia-i156-dragon | julia-rabbit | newtonz3m1 |
| Mandelbrot | Julia - Clown | Julia - Dendrite | Julia - Dragon | Julia - Rabbit | Newton z^3-1 |

Polar Plots

Angular – with radial distance held constant the angle moves from –pi to pi

|  |  |  |
| --- | --- | --- |
| plot21 | CalcLogo2-200x200 | polar1 |
| polar3 | polar5 | polar4 |

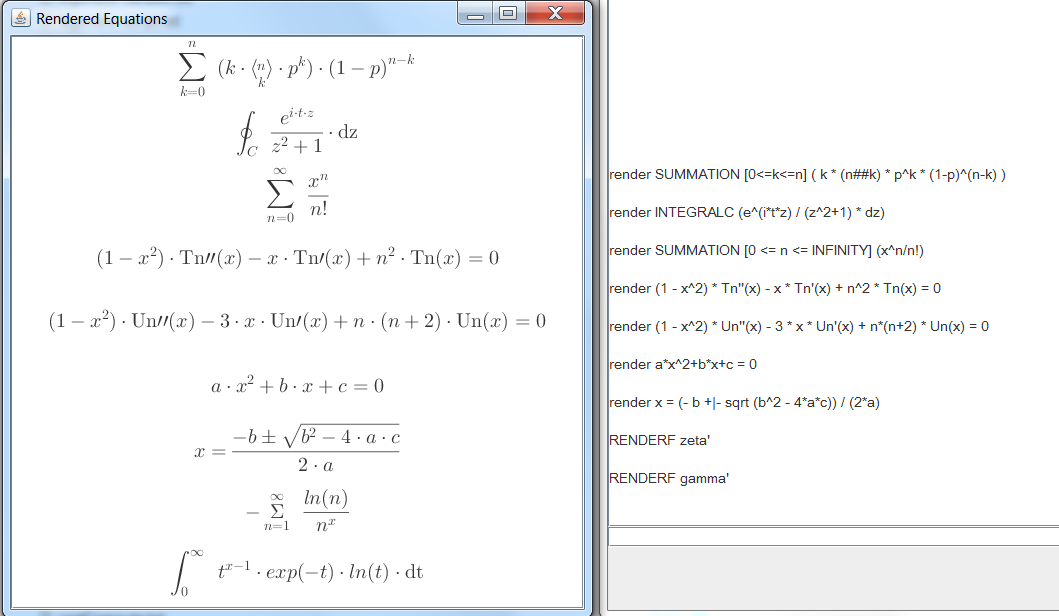
Radial – with the angle held constant the distance extends from -2 to 2

# Contour Plots

|  |  |
| --- | --- |
| C:\Users\Michael\Desktop\software backup\plots\beta.jpg | Beta*.txt*  LIBRARY SpecFunc net.myorb.math.specialfunctions.Library  !+ GAMMA\_IMPORT(x) = SpecFunc.gamma  !! GAMMA(x) = GAMMA\_IMPORT x  // BETA function  !! BETA(x,y) = GAMMA(x) \* GAMMA(y) / GAMMA (x + y)  plot3d BETA ( -1.9, -1.9 ) 3.8 10  RENDER B(x,y) = GAMMA(x) \* GAMMA(y) / GAMMA (x + y)  RENDER "BETA Function (computed using GAMMA)" TOP |
| C:\Users\Michael\Desktop\software backup\plots\Octopus.png | ContourPlot*.txt*  !! OCTOPUS (x,y) = cos (y \* exp (x-y) / x)  plot3d OCTOPUS (-1.9,-1.9) 3.8 100  RENDER cos (y \* exp (x-y) / x) |

# 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

# 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.

|  |  |
| --- | --- |
| sample1 | 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  render SIGMA [0 <= n <= INFINITY] (x^n/n!)  RENDER "The SIGMA notation for series of terms (Taylor series for e^x)" TOP  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  render PI [j+1 <= i <= n+1] (x#i)  RENDER "The PI notation for series of factors" TOP  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 |

|  |  |
| --- | --- |
| sample2 | render INTEGRALI (asin(theta) \* <\*> theta)  render INTEGRALD (y \* asin(x) \* <\*> x \* <\*> y)  RENDER "Simple indefinite integrals" TOP  render INTEGRAL [-1 <= x <= 1 <> dx] (asin(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''(x) - x \* Tn'(x) + n^2 \* Tn(x) = 0  render (1 - x^2) \* Un''(x) - 3 \* x \* Un'(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-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 |

|  |  |
| --- | --- |
| history1 | !!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 = 1.6449340668482262, sumInvSq(100000) = 1.6449240668982423  // pi^2/6 - sumInvSq(100000) = 9.999949983852119E-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] (asn'(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\*x)  // or the circular segment equation  // circle'(x) = SQRT (x - x\*x) is shown to have circular segment  // pi = 3\*SQRT(3)/4 + 24 \* area => area = 0.07677310616304730284654410946459  RENDER SegmentArea = INTEGRAL [0 <= x <= 0.25 <> dx] ( sqrt (x - x^2) ) \* <\*>x  areaOfCircularSegment = 0.07677310616304730284654410946459  // = INTEGRAL [0 <= x <= 0.25 <> dx] (circle'(x)) \* dx  RENDER pi = 3 #/# 4 \* sqrt(3) + 24 \* SegmentArea  RENDER "Off center circle segment area used to compute pi" TOP  !!circle'(x) = SQRT (x - x\*x)  // James Gregory atan series  // atan x = x - x^3/3 + x^5/5 - x^7/7 + ...  !!atn(x,n) = SIGMA [1 <= i <= n <> 4] (x^i/i - x^(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 |

|  |  |
| --- | --- |
| history2 | LIM = 1/INFINITY  !!atn'(x) = 1 / (1 + x\*x)  // 4 \* INTEGRAL [0 <= x <= 1 <> dx] (atn'(x)) \* dx  atnIntRez = INTEGRAL [0 <= x <= 1 <> LIM] (4 \* atn'(x) \* <\*> x)  PRETTYPRINT LIM  PRETTYPRINT invSqRez  PRETTYPRINT atnIntRez  // 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  RENDER pi = 4 \* ( 4 \* atan (1 / 5) - atan (1 / 239) )  RENDER "Arc Tangent series of John Machin" TOP  RENDER pi = 20 \* atan (1 / 7) + 8 \* atan (3 / 79)  RENDER "Arc Tangent series of Leonhard Euler" TOP  RENDER 1/pi = 2 \* sqrt(2) / 9801 \* ( SUMMATION [0 <= k <= INFINITY] ( (4\*k)! \* (1103 + 26390\*k) / ((k!)^4 \* 396 ^ (4\*k)) ) )  RENDER "Series of Srinivasa Ramanujan" TOP |

# CalcLib Notation, MathML, and rendered expression

|  |  |
| --- | --- |
| **MathML**  **Generated by CalcLib**  mmlSample | **CalcLib Notation**  **render x = (- b +|- sqrt (b^2 - 4\*a\*c)) / (2\*a)**  **Renders As**  quadratic  Using the public interface implementation connected to the JMTeX library |