# Numerical Analysis Team 2.2 Project Documentation

## About

Our project uses Python 3.5+ code and the following dependencies (please note that the project will not run with Python 2.7):

* numpy
* scipy
* sympy
* matplotlib

These dependencies gave us access to robust mathematical algorithms and data structures, but this unfortunately comes at a cost… Installation kind of sucks. Detailed instructions for installation will follow.

A few things to note; we have built the application to either run through a GUI interface, or you can call the algorithms directly if you’re comfortable using Python in this way. Instructions will also follow.

## Installation

### The easy way

The easy way simply installing Anaconda, a Python distribution for data science:

<https://www.continuum.io/downloads>

This will install Python 3.5, Numpy, Scipy, Sympy, and Matplotlib all for you. You can see what packages it will install here:

<https://docs.continuum.io/anaconda/pkg-docs>

### The hard way

We have placed the dependencies for Windows You’ll need Python 3.5 and, run the following commands in the install directory in the following order:

**If you’re on Windows**

pip install numpy-1.11.3+mkl-cp35-cp35m-win\_amd64.whl

pip install scipy-0.19.0-cp35-cp35m-win\_amd64.whl

pip install matplotlib-2.0.0-cp35-cp35m-win\_amd64.whl

pip install mpmath

pip install sympy

**If you’re on a Mac/Linux OS**

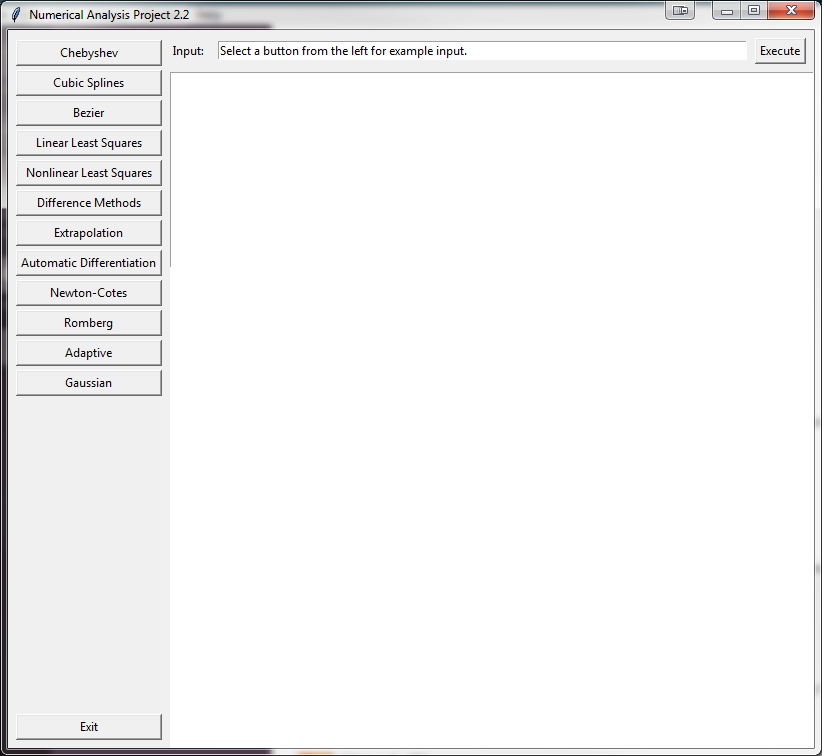
python setup.py install

If that still doesn’t work, reach out to one of us and we’ll try to help you get up and running.

## How to run the project

python project.py

You should get a nice window popup. If that doesn’t work, please reach out to one of us. The UI should look like this when it opens:



## Source Code

We used GitHub for version control and sharing the code. Please see the repository at:

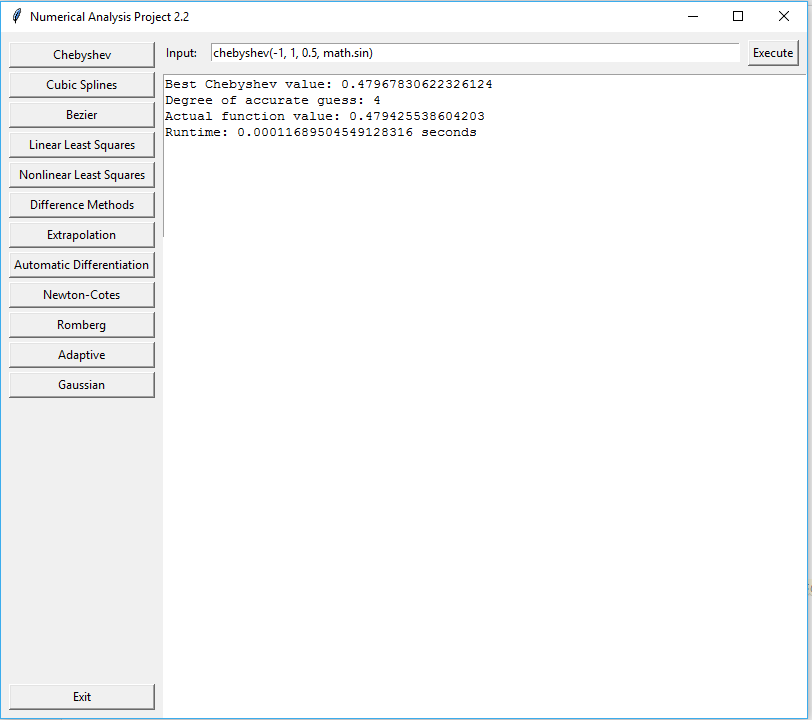
<https://github.com/protocol114/numerical-2.2>

## Algorithm Documentation- Here you are going to find the necessary documentation to run all our algorithms, along with various information on what will work and not work from the test questions. We will try to be as explicit as possible, but please do not hesitate to reach out to us if you have any questions about our program.

Section 1: Interpolation

We have implemented 3 separate algorithms in order to assist you in interpolating your data. Please understand that our functions will be limited in scope, and you must input data as prescribed and anything not explicitly stated here or in our GUI examples will not work.

## Chebyshev

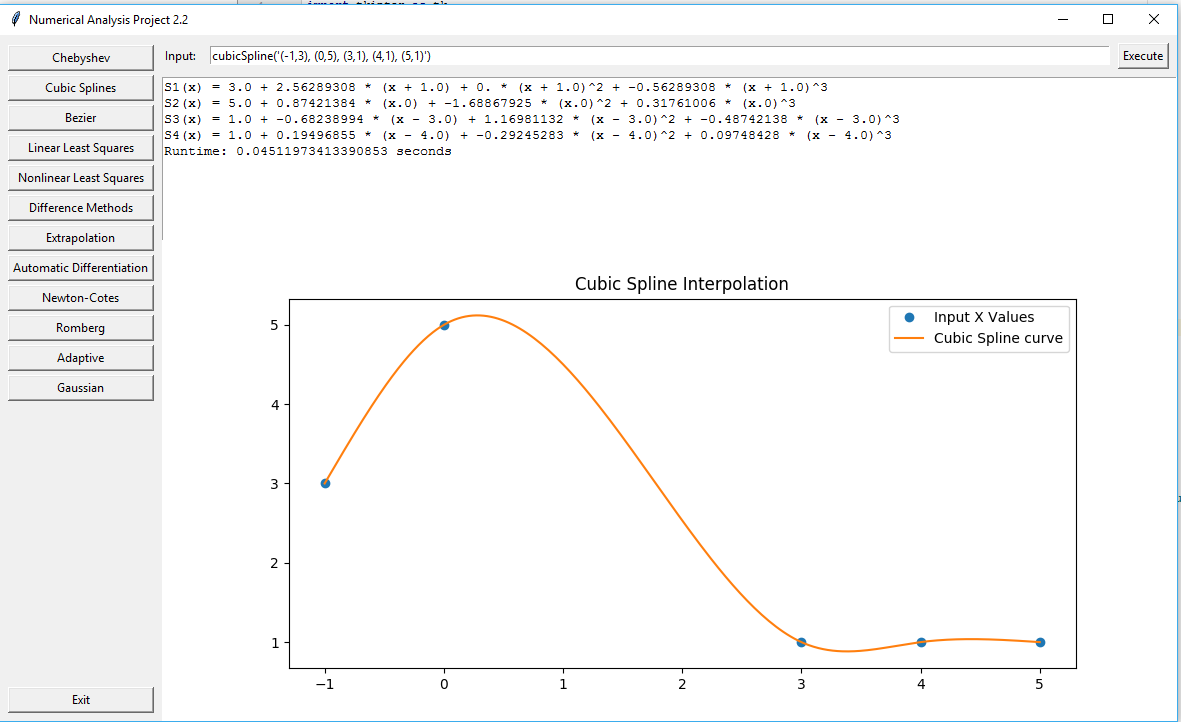


Runs the Chebyshev algorithm up to 30 times, increasing degree n until the guess is sufficiently close. Outputs the calculated Chebyshev value, the degree of the polynomial where the best guess was calculated and the actual value from the function. You MUST enter a function that can be evaluated, because func is not an optional argument.

**Example usage**: chebyshev(-1, 1, 0.5, math.sin)

Advanced functions can be input as example: lambda x: (math.sin(x) - math.cos(x))

## Cubic Splines



Takes a string of points in the string form: '(-1,3), (0,5), (3,1), (4,1), (5,1)' and optionally, the graph resolution. Prints the cubic spline functions and displays an interpolated line plot below.

Example usage: cubicSpline('(-1,3), (0,5), (3,1), (4,1), (5,1)')

or cubicSpline('(-1,3), (0,5), (3,1), (4,1), (5,1)', resolution=2) for a low resolution graph.

## Bezier



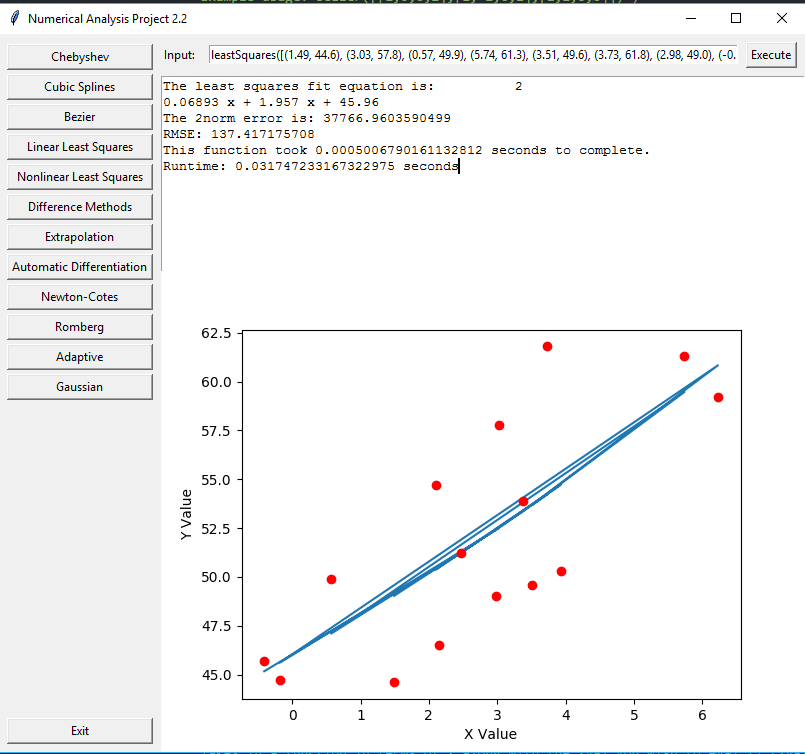
Takes a series of points in the form: [[1,0,6,2],[1,-1,0,1],[1,1,6,0]] and outputs the Bezier spline's knots and control points based on the input coordinates. It can also take in a series of **2-dimensional** points in order to get the Bezier interpolation equations. If you input points it will also output a graph.

Example usage: bezier([[1,0,6,2],[1,-1,0,1],[1,1,6,0]], 4), bezier([(1,0),(2,0),(0,3),(0,1)],4)

## Section 2: Least Squares

In Section 2 we have implemented two separate functions, evaluating a set of points either linearly or nonlinearly. This section is straightforward.

## Linear Least Squares



**Function Usage**:

leastSquares(points [Array])

leastSquares(points [Array], n [Integer])

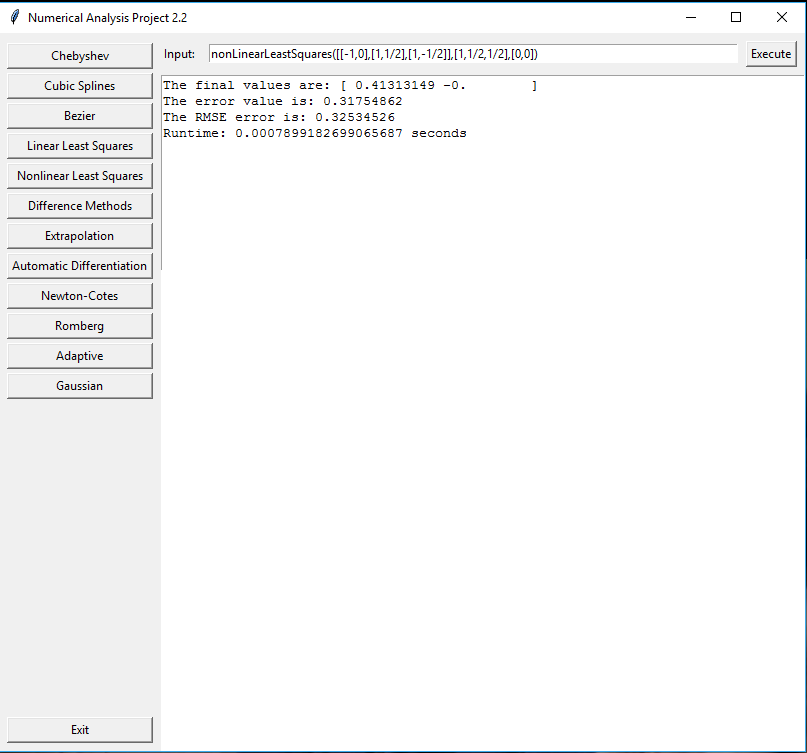
points = Series of points in the form: [[0, 1], [1, 2], [2, 3]] or [(0, 1), (1, 2), (2, 3)]

n = degree (only used when points are provided using parentheses

Takes either a series of coordinate points or a series of A and B matrices in bracket form. If coordinates are provided, will output least squares fit function and graph.

If an A and B matrix is provided, it will output the coefficient, residual, and rank.

## Nonlinear Least Squares



**Function Usage**:

nonLinearLeastSquares(points [array], radii [array], initial[array])

points = series of points in the form [[-1,0],[1,1/2],[1,-1/2]]

radii = array of the radius lengths of each circle in the form [1,1/2,1/2]

initial = set of points for your initial guess in the form [0,0]

It will take in the above values, ALL OF WHICH ARE REQUIRED, and will output

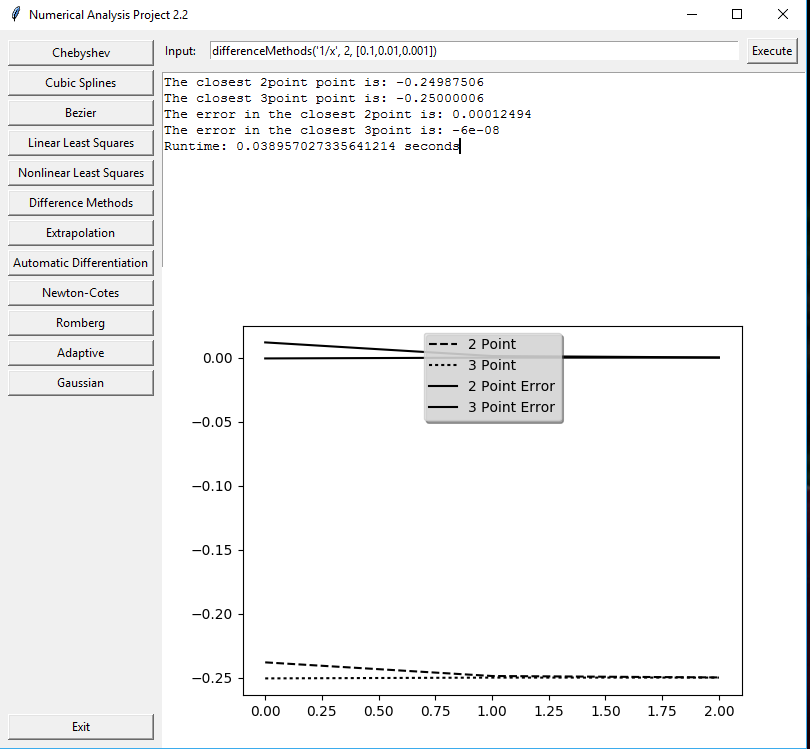
the best fit points, the standard error, and the root mean squared error RMSE

No graph outputting on this one as it is just a series of points

## Section 3: Differentiation and Integration

This section is by far the largest, and don’t expect to many graphs in this one as most of the functions are just straight evaluations.

## Difference Methods



differenceMethods(func function, x int, h [array])

function: must be entered as a string. CANNOT BE ENTERED OUTSIDE string

x: the point at which you are evaluating your derivative

h: an array of values of initial step sizes. The one on the right is the one evaluated

It will, using Two-point centered-difference and Three-point centered difference calculate

with the given inputs, output the best approximation for both and the error for both.

## Extrapolation

## 

extrapolation(f [Function], xval [Number], n [Number], hval [Number])

f = function to approximate; xval = value to extrapolate from; n = levels of of extrapolation; hval = step size

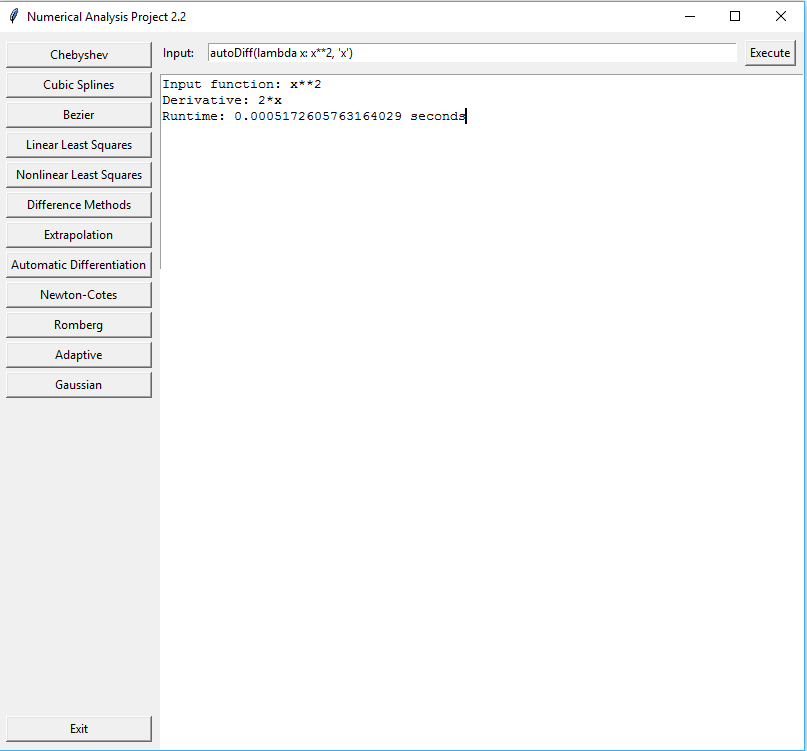
Takes in a function in terms of x along with the xval you wish to eval f'(xval) at

.You must provide the value for n, which is the number of levels of extrapolation, and

you must provide the initial stepsize h. This will return the most accurate value for

f'(xval) along with the actual value and the error. You MUST enter a single variable function

## Automatic Differentiation

. autoDiff(f [Function])

f = function to differentiate

Takes a function and calculates the derivative via automatic differentiation.

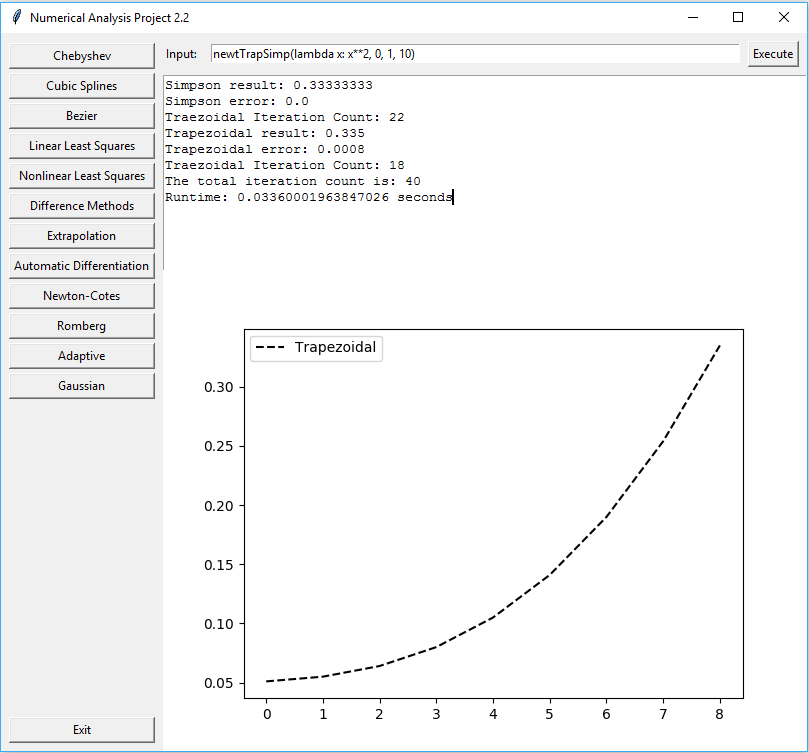
Functions can be input as Pythonic lambda functions or basic sin/cos/tan et cetera.

Supports up to three variables (x, y, z) in space-delimited string form: 'x y z'

Example usage: autoDiff(lambda x: x\*\*2, 'x')

autoDiff(lambda x, y, z: 2 \* x \*\* 2 + 4 \* y \*\* 3 + 8 \* z \*\* 4, 'x y z')

## Newton-Cotes



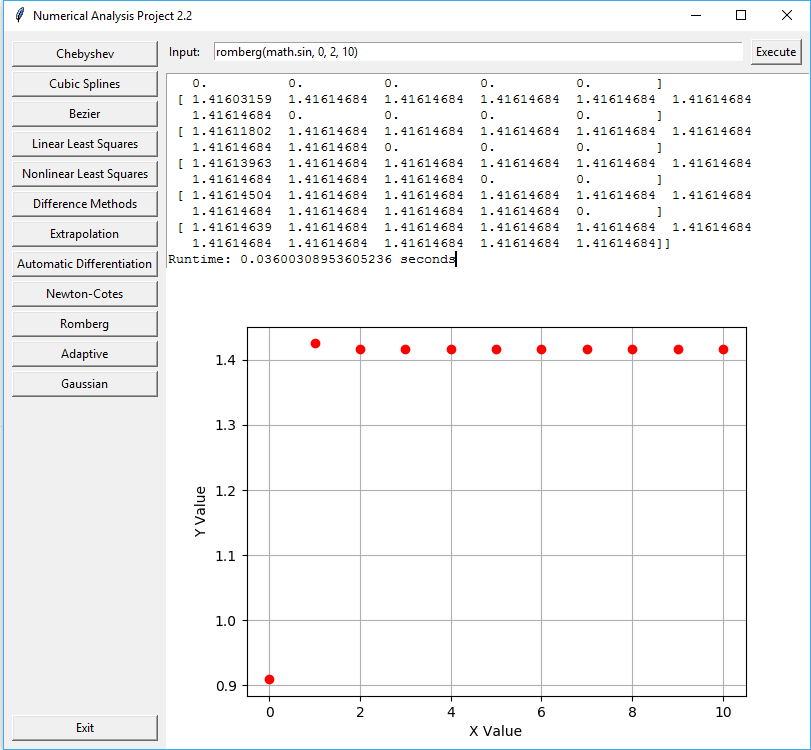
newtTrapSimp(f [Function], a [Number], b[Number], n[Number])

f = function to approximate; a, b = interval [a,b]; n = # of steps to take

Calculates the best guess for the Newton-Cotes Trapezoidal/Newton-Cotes Simpson result value, and plots the graph below.

Example usage: newtTrapSimp(lambda x: x\*\*2, 0, 1, 10)

## Romberg



romberg(f [Function], a [Number], b[Number], n[Number])

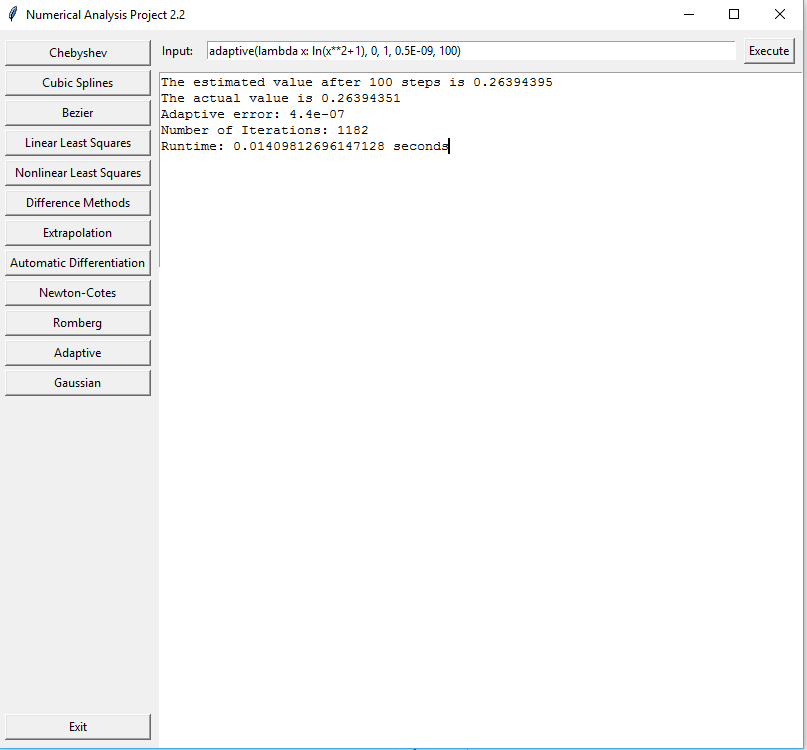
f = function to approximate; a, b = interval [a,b]; n = # of steps to take

Plots the Romberg output and also outputs the associated array.

Example usage: romberg(math.sin, 0, 2, 10)

Advanced functions can be input as example: lambda x: (math.sin(x) - math.cos(x))

## Adaptive



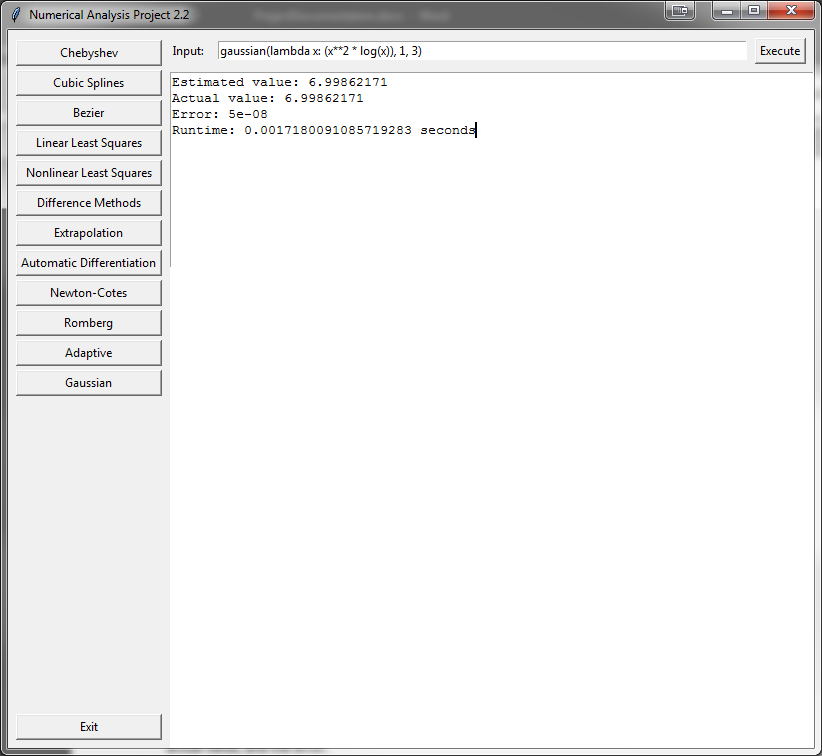
adaptive(f [Function], a [Number], tolerance[Number], steps[Number])

a, b = interval [a,b]; tolerance = guess tolerance; steps = # of steps to take

Takes a function, a - b interval, tolerance, and number of steps and outputs the integrated function value, the adaptive error, and the number of iterations necessary to find the integrated value.

Example usage: adaptive(lambda x: ln(x\*\*2+1), 0, 1, 0.5E-09, 100)

## Gaussian Integration



Function usage:

gaussian(f [Function], a [Number], b[Number], y[Number - Default None])

a, b = interval [a,b]; f = function to approximate; y = Gaussian Y value

Takes a function, a and b interval, and optionally, an extra Y value.Outputs the estimated value, the actual value, and the error.

Example usage: gaussian(lambda x: (x\*\*2 \* log(x)), 1, 3)