Documentación del módulo: Numéricos - Interactivo para Python 3.4

Versión 1.3

Semillero de Fisica Teorica y Computacional
Universidad EAFIT

Marzo 16, 2015

ÍNDICE GENERAL

CAPÍTULO

ONE

CAPITULO 01: INTRODUCCIÓN

Esta es la documentación oficial del proyecto cofinanciado por Colciencias desarrollado por el Semillero de Física Teórica y Computacional de la Universidad EAFIT, cuyo sitio oficial se encuentra en el hipervínculo: https://sites.google.com/site/fisicatyc/home. El objetivo principal del proyecto está en que un estudiante por medio de las herramientas habilidades, encuentre un camino más interactivo para el estudio del análisis numérico al igual que la programación e implementación de algoritmos, particularmente para la plataforma de *Python 3.4*https://www.python.org/.

El proyecto Numéricos - Interactivos es una implementación en *Python 3.4* de algunos de los métodos numéricos elementales de las categorías de Búsqueda de Raíces, Extrapolación, Interpolación, Diferenciación e Integración. Cada una de éstas se encuentra como submódulos de la librería principal, permitiendo cargar selectivamente los diferentes algoritmos. Cada algoritmo posee una descripción conceptual y características del programa para su correcta utilización. Los comentarios y notaciones de los programas se encuentra en *inglés* para favorecer el alcance los códigos y una mayor distribución. La información del documento se encuentra en *español*.

El repositorio vigente del proyecto puede encontrarse en *GitHub* en el vínculo:

https://github.com/fisicatyc/numericos-interactivo

El proyecto se encuentra como código abierto y está a disposición para ser mejorado, modificado y expandido sin ningún tipo de restricción bajo la licencia *GNU*. Los respectivos créditos han de hacerse al desarrollo original por parte del Semillero de Física Teórica - Universidad EAFIT. La contribución directa puede realizarse a través de la plataforma *GitHub*.

El repositorio, posee tanto los códigos fuentes, como los *Ipython Notebooks* de algunos de los algoritmos. En los *notebooks* se hace una revisión conceptual y explicación del funcionamiento, limitaciones de los métodos, posibles aplicaciones junto con una puesta en marcha de los algoritmos a través de la visualización y tabulación de datos dinámicos, a partir de los datos que ingrese el usuario.

Los *notebooks* están diseñados para instruir y cautivar estudiantes universitarios en las áreas de ciencias de la computación y ciencias matemáticas para favorecer el aprendizaje conceptual, junto con las habilidades de programación. Ésta material adicional no es necesario para ejecutar las librerías.

Requerimientos:

El proyecto fue desarrollado utilizando las funciones Built-In de *Python 3.4* por lo que no es necesario otro tipo paquetes para su funcionamiento. Recomendamos la distribución multiplataforma *Anaconda* para *Python* https://store.continuum.io/cshop/anaconda/, para obtener el paquete de utilidades básicos para poder utilizar *Ipython Notebook* y *Python 3.4*.

Adquisición del código

Para obtener acceso a los archivos del repositorio es posible descargar como *zip* directamente los archivos del proyecto y localizarlos en cualquier directorio. Asimismo se puede disponer acceso al registrarse gratuitamente en *GitHub* y crear un clon del repositorio. Otra manera está en cargar directamente las librerías con la función de *python* $\%load$ y la respectiva URL del código.

Cargar las librerías en *Python*

Para cargar las librerías desde cualquier consola de *python*, es necesario trabajar sobre el mismo directorio donde se encuentre el módulo *Numericos*. Una vez en éste directorio es posible ejecutar la linea de código:

```
from Numericos import Numericos
```

Sin embargo está sola secuencia no carga los archivos de las librerías, sino las características del paquete. Para acceder a los submódulos, la instrucción de importe ha de incluir el submódulo respectivo, por ejemplo para el paquete integradores, la secuencia de llamado es:

```
from Numericos.lib import integrators
```

Dentro de cada uno de los 5 submódulos están otros submódulos que definen los distintos métodos implementados. En el esquema siguiente se muestra el contenido del directorio principal.

```
numericos interactivo/                                  -> Carpeta Padre

            Notebook files/                -> Recursos Adicionales con Notebooks

                        Estructura y notas del programa.ipynb
                        Integración Numérica.ipynb

            project_2014_numerical_methods/     -> Carpeta de librerias de Python

                        Numericos/
```

En la carpeta de *Numericos* se encuentra contenido todo el código fuente en los diferentes submódulos, con la siguiente jerarquía que se presenta en el esquema siguiente:

```
Numericos/                          -> Módulo principal
          __init__.py
          doc/                      -> Documentación
          lib/                      ->Librería principal
                    __init__.py
                    differentiators/  -> Submódulo de derivación numérica
                                  __init__.py
                                  centraldiff/
                                              __init__.py
                                              FCDA.py
                                              SCDA.py
                                              TCDA.py
                                              QCDA.py
                                  ncentraldiff/
                                              __init__.py
                                              nfirstderv.py
                                              nsecondderv.py
                                              snfirstderv.py

                    extrapolators/  -> Submódulo de extrapoladores
                                  __init__.py
                                  richarexpol.py

                    integrators/    -> Submódulo de integración numérica
                                  __init__.py
                                  AdvanceIntegration
```

```
                        adapquad.py
                        romberginteg.py
                closednewtoncotes/
                        __init__.py
                        fournodescotes.py
                        simprule.py
                        tesimprule.py
                        trapzrule.py
                compositerules/
                        __init__.py
                        compmidtrule.py
                        compsimprule.py
                        comptraprule.py
                doubleIntegral/
                        __init__.py
                        gaussdblint.py
                        simpdblintg.py
                opennewtoncotes/
                        __init__.py
                        midpointrule.py
                        onenodeopn.py
                        thrnodeopn.py
                        twonodeopn.py
                tripleIntegral/
                        __init__.py
                        gausstpllint.py

        interpolators/  -> Submódulo de interpolación
                __init__.py
                beziercurve.py
                clampspline.py
                derivpol.py
                hermitpoly.py
                natspline.py
                nevilletable.py
                newtondivdef.py

        rootFinders/  -> Submódulo de búsqueda de raíces
                __init__.py
                bisection.py
                fixedpoint.py
                muller.py
                polyroot.py
                regulafalsi.py
                secant.py
```

En los capítulos siguientes se aborda una discusión sobre la formulación matemática y características generales de cada algoritmo.

CAPÍTULO

TWO

CAPITULO 02: MÉTODOS DE BÚSQUEDA DE RAÍCES

2.1 Métodos Cerrados

Los métodos cerrados son métodos que requieren de dos valores iniciales para la búsqueda de raíces, los cuales son los valores de los extremos de un intervalo que encierra una raíz. Estos métodos siempre son convergentes pero suelen ser algo lentos.

En los métodos cerrados la metodología se puede describir como una sucesión de pasos que cada vez acotan más el lugar de la raíz, es decir, disminuyen el tamaño del intervalo. Para poder asegurar que una raíz se halla al interior de este intervalo debemos cumplir con que:

$$f(a)f(b) \leq 0 \tag{2.1}$$

Siendo a y b los extremos del intervalo actual y f una función continua en el intervalo cerrado $[a, b]$. Si esta condición se cumple tendremos:

$$\exists c \in [a, b] \,/\, f(c) = 0$$

Este resultado se obtiene a partir del *teorema de Bolzano*. Dado que la función es continua recorrerá todos los valores comprendidos entre *f(a)* y *f(b)*, y como existe cambio de signo entre sus extremos uno de estos valores para la función será igual a cero, por ende, entre a y b existe una raíz.

Nótese que si la función presenta raíces de multiplicidad par (un numero par de raíces idénticas) los métodos cerrados no convergen.

$$\left(\lim_{a \to c^-} f(a)\right)\left(\lim_{b \to c^+} f(b)\right) > 0$$

La diferencia entre los métodos cerrados se basa en la forma de reducir el intervalo alrededor del cual se halla la raíz, o en otras palabras la aproximación del valor de la raíz. De ahí métodos como búsquedas incrementales, bisección y regla falsa.

2.1.1 Búsquedas Incrementales

Los métodos de búsquedas incrementales aprovechan la característica de la ec.2.1 para determinar el intervalo en el cuál se encuentra por lo menos una raíz, con un error igual a la mitad del tamaño de paso del avance en el intervalo. Para mejorar la precisión del método, una vez localizado el intervalo que presenta cambio de signo, este se puede subdividir en mas intervalos y nueva mente aplicar el método de búsquedas incrementales.

La formulación iterativa y el error absoluto son expresados a continuación, siendo n el número de iteraciones máxima o puntos totales ($n-1$ subintervalos explorados) e i el i-ésimo subintervalo (o iteración). a y b son los extremos del intervalo y c la aproximación de la raíz.

$$c = a + i\frac{b-a}{n} \qquad (2.2)$$
$$\epsilon = \frac{b-a}{2(n-1)}$$

2.1.2 Bisección

El método de bisección es un tipo de búsqueda incremental en el que el intervalo siempre se divide en 2 subintervalos. n es el número de iteraciones máxima (número de veces en que se subdividirá el intervalo).

$$c = \frac{a+b}{2} \qquad (2.3)$$
$$\epsilon = \frac{b-a}{2^{n+1}}$$

Este método al igual que búsquedas incrementales, solo realizan consideraciones del intervalo mas no del comportamiento de la función en el intervalo.

2.1.3 Regla Falsa

Este método introduce a la aproximación de la raíz un conocimiento del comportamiento de la función. El hecho que un valor de la función este próximo a cero puede indicar igualmente que la posición de la variable independiente este cerca de la raíz real. Para determinar la aproximación de la raíz se unen los puntos de los extremos por una línea recta, y la intersección de esta con el eje x constituye una muy buena aproximación de la raíz.

La obtención de formulación iterativa se obtiene a partir de triángulos semejantes.

$$\frac{f(a)}{c-a} = \frac{f(b)}{c-b} \qquad (2.4)$$

Dada la consideración del comportamiento de la función tenida en regla falsa, este método en general presenta una convergencia mayor a los otros métodos cerrados. En algunos casos, el método presenta puntos de estancamiento y debe plantearse un criterio de parada adicional al error y modificar la forma iterativa del método alterando la magnitud de uno de los extremos del intervalo.

2.2 Métodos Abiertos

Los métodos abiertos son métodos cuya formulación solo requiere de un valor inicial para estimar la raíz de la función. Estos métodos no necesariamente son convergentes, pero cuando lo logran generalmente son mas rápidos que los métodos cerrados. Esta situación puede darse gracias a que los métodos abiertos toman consideraciones acerca de la geometría de la función particular que se desea.

La forma general de estos métodos es dada por la expresión ec.2.5, donde a partir de una ecuación $f(x) = 0$ se obtiene:

$$x = g(x) \tag{2.5}$$

Para que el método sea convergente se debe cumplir que en ec.2.5

$$\left|g\prime(x)\right| < 1 \tag{2.6}$$

donde g es una función tal que $g: \mathbb{R} \to \mathbb{R}$

2.2.1 Método de Punto Fijo

La iteración de punto fijo exige llevar la ecuación de la forma $f(x) = 0$ a la forma ec.2.5. Una vez hecho esto, se realiza una sustitución sucesiva, teniendo la forma iterativa.

$$x_{i+1} = g(x_i) \tag{2.7}$$

En la cual la función g cumple con ec.2.6. El método de punto fijo posee convergencia lineal.

Ejemplo 2.1. Sea $f(x) = \exp^{-x} - x$, obtener la iteración de punto fijo.

$$e^{-x} - x = 0$$
$$x_{i+1} = e^{-x_i}$$

Ejemplo 2.2. Sea $f(x) = x^2 - 2x + 3$, obtener la iteración de punto fijo.

$$x^2 - 2x + 3 = 0$$
$$x_{i+1} = \frac{x_i^2 + 3}{2}$$

2.2.2 Método de Newton-Raphson

Dada una función $f(x)$ su expansión en series de Taylor alrededor de un punto x_0 puede ser expresada como:

$$f(x) = f(x_0)\ \frac{df(x)}{dx}\bigg|_{x=x_0}(x - x_0) + \frac{1}{2}\ \frac{d^2 f(x)}{dx^2}\bigg|_{x=\xi}(x - x_0)^2 \tag{2.8}$$

Tomando el orden 1 de aproximación en ec.2.8 y despejando para x:

$$x = \frac{f(x) - f(x_0)}{f\prime(x_0)} + x_0$$

finalmente, nos interesa la raíz de la función por lo que $f(x) = 0$.

$$x_{i+1} = x_i - \frac{f(x_i)}{f\prime(x_i)} \tag{2.9}$$

El método de Newton, cuya iteración es descrita por la ec.2.9 posee una convergencia cuadrática. El método de Newton puede tener modificaciones útiles para el caso de raíces múltiples. Así siendo m la multiplicidad puede definirse la iteración como:

$$x_{i+1} = x_i - m\frac{f(x_i)}{f\prime(x_i)} \tag{2.10}$$

Otra variación del método de Newton consiste en redefinir la función a la cual se buscaran las raíces. El método tiene la forma general:

$$x_{i+1} = x_i - \frac{u(x_i)}{u\prime(x_i)} \tag{2.11}$$

con $u(x) = f(x)$. Debemos proponer una función $u(x)$ tal que sus raíces sean las mismas que $f(x)$.
Si $f\prime(x) \neq 0$ las raíces de

$$u(x) = \frac{f(x)}{f\prime(x)}$$

serán las mismas de $f(x)$. Sustituyendo esta en ec.2.11 se obtiene la formulación de Newton modificada para raíces múltiples.

2.2.3 Método de Secante

El método de secante consiste en una alternativa al método de Newton para el calculo de la derivada de la función. Esto es ya que no siempre será posible calcular esta o dicho calculo puede constituye un costo computacional mayor, por lo que la expresión de ec.2.9 se modifica incorporando una forma discreta de la derivada.

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)} \tag{2.12}$$

Es posible considerar una variante del método de secante en la cual para la aproximación de la derivada se considera un pequeño cambio en la variable independiente de la función:

$$x_{i+1} = x_i - \frac{f(x_i)\Delta x}{f(x_i + \Delta x) - f(x_i)} \tag{2.13}$$

2.3 Implementación del método:

En este módulo se aplican los métodos de búsqueda de la raíces de una función, lo cual vendría a ser la búsqueda de todo elemento x que pertenezca al dominio de la función y que dicha función cumpla f(x) = 0.

Se pretende que el usuario aplique los métodos a la solución de problemas, para lograr una mejor compresión sobre la temática y se adquieran habilidades que faciliten el progreso hacia otros métodos de mayor complejidad.

Dentro del paquete rootFinders se encuentran los siguientes métodos para la búsqueda de raíces en una función:

> -Bisection Method -FixedPoint Method -Müller Method -PolyRoot Method -RegulaFalse Method -Secant Method

2.4 Secuencia de llamado:

Cada método corresponde a un archivo de extensión .py, que en el paquete se pueden encontrar como:

Note: bisection.py, fixedpoint.py, muller.py, polyroot.py, regulafalsi.py, secant.py.

Para ejecutar éstas funciones desde cualquier terminal de Python 3.4, es necesario cargar en memoria el paquete, con la línea:

Note: from Numericos.lib.rootFinders import *

Éste comando cargará simultáneamente los seis métodos del paquete. Para ejecutar un cálculo, es necesario agregar 'rootFinders.(método)', y sus respectivos argumentos. En el caso de bisección la secuencia sería:

Note: rootFinders.bisection(('6*x-6'),-5,6,200,0.005);

En el caso de cargar un único método, se utiliza el comando:

Note: from Numericos.lib.rootFinders import muller

De ésta manera solamente el método de Müller esta disponible para operar. Ésta forma simplifica la forma de llamado.

CAPÍTULO

THREE

CAPITULO 03: INTERPOLADORES

En general, para las aplicaciones de la ingeniería y de la investigación aplicada, sera necesario partir de información obtenida experimentalmente, lo que traduce en que la información que poseemos no es una función continua que describe el fenómeno sino de un conjunto de puntos cuya única relación inicial es que fueron obtenidos de un mismo experimento u observación de un fenómeno. El paso siguiente a partir de la obtención de estos puntos, es la obtención de una curva que describa adecuadamente los puntos (ya sea aproximándolos -curvas de regresión- o atravesándolos -curvas interpolantes-) y a partir de las cuales sea posible construir modelos matemáticos para la descripción del fenómeno y permitan realizar predicciones acerca del mismo (al igual que estudiar el comportamiento para las condiciones no obtenidas experimentalmente). En cuanto a la interpolación, se recurre generalmente a la interpolación polinomial.

Para un conjunto de $n+1$ puntos, existe un ubico polinomio de grado n que pasa por todos los puntos. Para construir este polinomio existen varias técnicas, como lo son Interpolación de Newton en diferencias divididas, Interpolación de Lagrange, calculo de los coeficientes directamente mediante un sistema de ecuaciones y otros. Igualmente, existen métodos de interpolación, que producen curvas suaves que pasan por todos los puntos, y que en general pueden acercarse mucho mas a la descripción del fenómeno, mas no son un polinomio, sino funciones por tramos que se constituyen de polinomios de grados bajos. Estos son los splines o trazadores.

3.1 Interpolación de Newton

Se puede formular de manera general como:

$$f_n(x) = \sum_{i=0}^{n} f(x_i) \prod_{j=0}^{i-1} (x - x_j) \tag{3.1}$$

Algorítmicamente es posible plantear una alternativa mediante la utilización de diferencias divididas (lo cual no se discutirá).

3.2 Interpolación de Lagrange

El método de interpolación de Lagrange es una reformulación del método de interpolación de Newton (ec.3.1), que evita el cálculo de las diferencias divididas. El polinomio resultante se puede expresar como:

$$f_n(x) = \sum_{i=0}^{n} L_i(x) f(x_i) \tag{3.2}$$

donde

$$L_i(x) = \prod_{j=0 j\neq,i}^{n} \frac{x - x_j}{x_i - x_j} \tag{3.3}$$

La formulación de Lagrange (ec.3.2) es posible comprenderse si se observa que el termino Li(x) a partir de ec.3.3 es 1 si x = xi y 0 para los demás puntos.

3.3 Splines

La utilización de la interpolación polinomial, al usar polinomios de grados grandes pueden propagar errores de redondeo significativos y conducir a resultados erróneos, además de que la existencia de puntos alejados entre si trae como consecuencia curvas no muy bien comportadas.

En general, es de esperar que el comportamiento de un sistema no posee cambios bruscos y se refleje por ende en curvas suaves. Para ello, se propone una alternativa de interpolación, los splines.

Los splines o trazadores, utilizan polinomios de grados menores para interpolar subconjuntos de puntos. Entre cada par de puntos se usa un polinomio interpolante, y cuyas condiciones para hallar los coeficientes se establecen a partir de condiciones de continuidad de la función y de sus primeras derivadas.

Es usual, el caso de los splines cúbicos, los cuales crean curvas suaves entre los puntos a interpolar con oscilaciones mínimas. Los splines cúbicos, son la forma resultante de poner una banda elástica a rodear los puntos (alfileres). Esto puede dar idea de un principio de mínima energía, y por ende de lo apropiados que pueden ser en la descripción de fenómenos naturales.

3.3.1 Splines de primer grado

Corresponde al caso mas sencillo de unión entre dos puntos, mediante el trazado de una linea recta.

$$f(x) = \begin{cases} f(x_0) + m_o(x - x_0) & x_0 \preceq, x \preceq, x_1 \\ f(x_1) + m_1(x - x_1) & x_1 \preceq, x \preceq, x_2 \\ . & . \\ . & . \\ . & . \\ f(x_{n-1}) + m_{n-1}(x - x_{n-1}) & x_{n-1} \preceq, x \preceq, x_n \end{cases} \tag{3.4}$$

Las pendientes indicadas en ec.3.4 corresponden a

$$m_i = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \tag{3.5}$$

3.3.2 Splines de segundo grado

Los trazadores de grado superior son usuales para asegurar la continuidad de las primeras derivadas de la función. Si se requieren m derivadas continuas, se requiere a lo sumo de usar trazadores de grado $m+1$. Un trazador cuadrático asegura que la primera derivada de la función sea también continua. Para hallar los coeficientes se requiere de imponer condiciones que creen el conjunto de ecuaciones iguales al número de coeficientes, de forma que el sistema sea soluble.

Condiciones

La forma general del trazador para $n+1$ datos ($i = 0..n$) es $f_i(x) = a_i x^2 + b_i x + c_i$. Estos polinomios aplican sobre los n intervalos existentes entre los puntos, y generan $3n$ coeficientes desconocidos que requieren de las siguientes condicione para formar las $3n$ ecuaciones para solucionar el sistema de ecuaciones.

1. Los valores de la función de polinomios adyacentes deben ser iguales en los nodos interiores. Esto es que para i = $2..n$ se da

$$f(x_{i-1}) = a_{i-1}x_{i-1}^2 + b_{i-1}x_{i-1} + c_{i-1} \tag{3.6}$$

$$f(x_{i-1}) = a_i x_{i-1}^2 + b_i x_i + c_i \tag{3.7}$$

   Obsérvese que por cada nodo interior se generar 2 ecuaciones, lo cual produce en total $2(n-2)$ ecuaciones.

2. La primera y última función deben pasar por los puntos extremos.

$$f(x_0) = a_i x_0^2 + b_1 x_0 + c_1 \tag{3.8}$$

$$f(x_n) = a_n x_n^2 + b_n x_n + c_n \tag{3.9}$$

   Esta condición aporta 2 ecuaciones más.

3. Las primeras derivadas en los nodos interiores deben ser iguales. La primera derivada en forma general se expresa como $f_i'(x) = 2a_i x_{i-1} + b_i$, y por lo tanto la condición se puede expresar como

$$2a_{i-1}x_{i-1} + b_{i-1} = 2a_i x_{i-1} + b_i \tag{3.10}$$

   Para $i = 2..n, aportando - 1 ecuaciones.$

4. La ecuación adicional se obtiene mediante una condición arbitrara, podemos suponer que en el primer punto la segunda derivada es cero, de donde se obtiene como ecuación, $a_1 = 0$.

El siguiente paso, es resolver el sistema de ecuaciones para obtener los coeficientes.

3.3.3 Splines de tercer grado

Para los trazadores cúbicos tenemos como expresión general $f_i(x) = a_i x^3 + b_i^2 + c_i x + d_i$. Igual que los casos anteriores, para $n+1$ datos, existen n intervalos en los cuales se trazara un polinomio interpolante, esta vez, de grado 3. Para los trazadores cúbicos, se tienen $4n$ coeficientes desconocidos, y por ende se requieren condiciones para formar el sistema de $4n$ ecuaciones necesarias para determinar los coeficientes.

Condiciones

1. Los valores de la función deben ser iguales en los nodos interiores. Aporta $2n-2$ ecuaciones.
2. La primera y última función deben pasar a través de los puntos extremos. Aporta 2 ecuaciones.
3. Las primeras derivadas en los nodos interiores deben ser iguales. Aporta $n-1$ecuaciones.
4. Las segundas derivadas en los nodos interiores deben ser iguales. Aporta $n-1$ ecuaciones.
5. Las segundas derivadas en los nodos extremos son cero. Aporta 2 ecuaciones.

La condición 5 representa que la función se vuelve una linea recta en los extremos, y esta condición particular hace que el trazador reciba la denominación de trazador ”natural”. Para los trazadores cúbicos es posible obtener una formulación que reduce el sistema de ecuaciones de $4n$ a $n-1$, representando una considerable reducción del costo computacional.

3.4 Descripción del método:

Este módulo se encarga de la implementación de los métodos de interpolación, que se basa en la problemática de tener $n+1$ puntos secuenciales, donde x0 < x1 < x2 < x3 < ... < xn, y a partir de estos puntos diferentes se va a definir una función polinómica que pase por esos puntos. Básicamente se está aproximando p(x)= y a una f(x)= y original, a la cual probablemente no se tiene acceso o conocimiento. Esto se hace con el fin de modelar el comportamiento de los puntos anteriores y obtener el "polinomio interpolante".

Se encuentran los siguientes métodos interpoladores:

-Beziercurve Method Página: 20

-Clampspline Method Página: 21

-Derivpol Method Página: 23

-Hermitpoly Method Página: 25

-Nevilletable Method Página: 27

-Newtondivdef Method Página: 28

-Natspline Method Página: 27

Cada uno de éstos métodos requieren variaciones en las entradas del algoritmo por lo que se sugiere verificar la documentación directa de las funciones, visitando los anteriores vínculos señalados en el texto.

CAPÍTULO

FOUR

CAPITULO 04: DIFERENCIADORES

4.1 Diferenciación numérica

En este módulo se aplican los métodos diferenciadores o también conocidos como diferencias finitas. Se van a categorizar como diferencias centradas y no centradas, donde la primera define un conjunto de métodos numéricos en diferencias finitas con paso centrado. Y en los métodos no centrados se define un conjunto de métodos numéricos basándose en diferencias finitas SIN paso centrado. Estas técnicas se emplean muy a menudo en análisis numérico, especialmente en ecuaciones numéricas ordinarias, ecuaciones en diferencias y ecuaciones. En derivadas parciales. Las aplicaciones habituales suelen ser en los campos de la computación, áreas de la ingeniería térmica o en la Mecánica de fluidos.

La diferenciación numérica retoma dos elementos principales, los cambios absolutos entre dos puntos de una función real y la expansión en series de Taylor. La diferencia absoluta entre dos puntos de una función real, determina el comportamiento de la pendiente de manera local. En el caso de una recta, la pendiente es determinada como el cociente entre las variaciones de la imagen de la función y su dominio, de la forma:

$$m = \frac{y_1 - y_0}{x_1 - x_0} = \frac{\Delta Y}{\Delta X} \tag{4.1}$$

El comportamiento de la recta es constante respecto a los cambios en todo el dominio. En el caso de una función que posea segunda derivada, implicará la existencia de concavidades y puntos de inflexión. En éste caso el cambio a lo largo de l dominio de la función no es constante, y varía punto a punto. En ésta consideración no se considera una recta tangente a un punto, sino una recta secante entre dos puntos. Considere el caso de la figura 4.1:

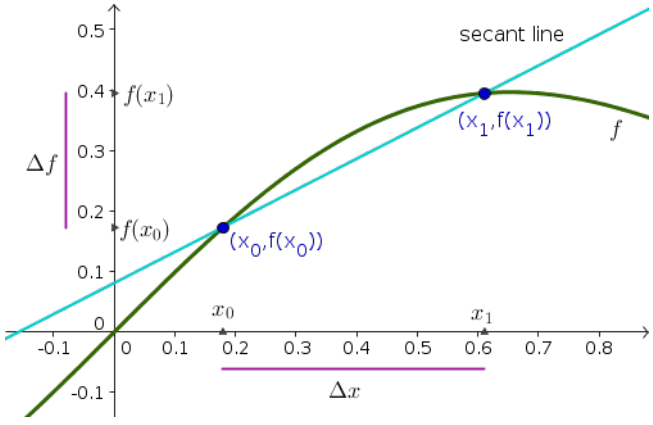

Figura 4.1: Recta secante entre dos puntos de una función $f(x)$.

Los puntos que forman la recta secante proveen la información de los cambio en ésta región de manera superficial, a razón que que hay muchos más cambios que no fueron mapeados. A medida que las diferencias absolutas de la imagen

$\Delta f(x)$ y el dominio ΔX se reducen, los cambios mapeos se aproximan a un mapeo de un punto. De tal forma que en lugar de formar una recta secante, vuelve a parecerse como una recta tangente. Por lo que otra vez se vuelve a tener la misma forma descrita en la ecuación de la recta, obteniendo información de la pendiente en esa región. Cuando $\Delta f(x) \approx 0$ y $\Delta X \approx 0$, se tiene información puntual de la pendiente también conocido como el cambio instantáneo, por lo que se establece la definición de derivada:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} \equiv \lim_{\Delta X \to 0} \frac{f(x+\Delta X) - f(X)}{\Delta X} = \lim_{\Delta X \to 0} \frac{\Delta Y}{\Delta X} \tag{4.2}$$

En el mundo discreto, no existe la noción de infinitud ni continuidad. Por tanto los limites no se evalúan completamente, implicando que los puntos de nuestra función son cercanos pero siempre $\Delta X \approx 0$ en lugar de $\Delta X = 0$. Ésta aproximación alude a lo que se tratará en ésta sección, las diferencias finitas.

El segundo elemento que se había mencionado en el inicio, corresponde a las series de Taylor. Una de las herramientas más poderosas que se tiene en los métodos numéricos, propiamente por simplificar conceptual y matemáticamente la noción de función, entre otras ventajas. La serie de Taylor es una aproximación a una función a través de una expansión alrededor de un punto α utilizando las derivadas de la función, definiendo:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(\alpha)}{n!}(x-\alpha)^n \tag{4.3}$$

donde n corresponde a la n-ésima derivada de la función, por tanto la función a expandir ha de ser del tipo suave, diferenciable en todo el dominio. Ésta expresión tiene el significado que una función es el resultado de la suma de todos los cambios posibles de ella misma, por tanto entre más cambios se tenga de la función, es posible generar una mejor reconstrucción de la función. Al observar el comportamiento de la expansión, a medida de que se toman las derivadas de orden superior, el coeficiente que acompaña la derivada es un número que cada vez es mucho menor, por tanto los aportes a la suma son menores en magnitud para las derivadas de orden superior. Por ejemplo para la sexta derivada de la función, 6! = 720 por tanto el aporte de ésta derivada es 720 veces más pequeño.

Como se había mencionado anteriormente, ésta definición aplica para infinitos términos, pero en el mundo discreto ha de existir un número discreto de términos suficientes que permita aproximar la función de manera parcial. En ese caso la función se exige que sea diferenciable en el intervalo donde se realiza la expansión. A partir de ésta idea, también es posible generar aproximaciones no solamente a la función, sino también de las derivadas de la función.

Éstas aproximaciones se realizan al realizar la expansión en series de Taylor hasta un cierto término y en un punto α que pertenece a un intervalo $[A-B]$. Dependiendo si la serie se realiza hacia la izquierda o a la derecha de α se define que la expansión es hacia atrás o hacia adelante, respectivamente. Ésto define el signo en los términos de la expansión. Adicionalmente el hecho que se incluya los límites del intervalo, implica que se está tratando de con una expansión centrada. A continuación, se presenta algunas de las aproximaciones a las derivadas de una función $f(x)$.

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}\frac{\mathrm{d}^3 f(x)}{\mathrm{d}x^3} \tag{4.4}$$

$$\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \frac{h^2}{12}\frac{\mathrm{d}^4 f(x)}{\mathrm{d}x^4} \tag{4.5}$$

$$\frac{\mathrm{d}^3 f(x)}{\mathrm{d}x^3} \approx \frac{f(x+2h) - 2f(x+h) + 2f(x-h) - f(x-2h)}{2h^3} + TOS(h^2) \tag{4.6}$$

$$\frac{\mathrm{d}^4 f(x)}{\mathrm{d}x^4} \approx \frac{f(x+2h) - 4f(x+h) + 6f(x) - 4f(x-h) + f(x-2h)}{h^4} + TOS(h^2) \tag{4.7}$$

donde $TOS(h^2)$ representa las contribuciones de los Términos de Orden Superior (TOS). Los términos con las derivas, dan indicio del orden de la expansión y del error analítico involucrado. En éste punto se puede observar lo fundamental

que es el parámetro h para generar las aproximaciones. Nótese que los ordenes superiores de las derivadas incluyen un factor de h^2, por tanto el error es menor.

Otro tipo de aproximación de las derivadas se puede realizar en el caso en que se considera un intervalo $(A-B)$, en el que no se toma los extremos del intervalo no se evalúan. Ésta aproximación se define como expansión no centrada y es de particular uso en el caso en que el intervalo a mapear es un conjunto de datos interpolados. En el caso de las dos primeras derivadas, la expansión se tiene la forma:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \frac{f(x+h)-f(x)}{h} + TOS(h) \tag{4.8}$$

$$\frac{\mathrm{d}^2f(x)}{\mathrm{d}x^2} = \frac{f(x+2h)-2f(x+h)+f(x)}{h^2} + TOS(h) \tag{4.9}$$

Nótese que éstas aproximaciones son más sencillas que la expansión centrada. Sin embargo el término de orden superior depende ahora de un h con potencia de uno, implicando que hay mayor propagación de error en el truncamiento. Por tanto se sugiere que al utiliza éste tipo de expansión, utilizar valores no muy distantes para definir un adecuado valor para h.

La diferenciación numérica se encarga de evidenciar los cambios locales de una sección de la función definido entre dos puntos A y B. A medida que éstos puntos se acerquen entre sí, el cambio se vuelve más local. Mientras que al alejarse éstos valores o al crecer el intervalo de diferenciación, los cambios son por secciones, pero al ser tan grandes los cambios no se mapea correctamente y por tanto la precisión de la aproximación disminuye. Ésto se puede observar en un número pequeño al elevarlo a una potencia entera, entre más grande la potencia, mucho menor se vuelve el número, pero si el número es grande, la potencia crece la magnitud. Éste número corresponde a la diferencia que hay entre éstos, definiendo $h = |A-B|$ entre más pequeño sea h, los errores de truncamiento y el aporte de los ordenes superiores disminuyen, en el caso contrario, el error crece exponencialmente.

4.2 Descripción del método:

Respecto a la implementación realizada los siguientes diferenciadores, utilizan la formulación de la sección previa, para encontrar las derivadas locales de una función. La implementación también articula la expansión centrada y no centrada. En ambos casos una función analítica es ingresada como objeto lambda de *python*. Ésta función definida en el intervalo de derivación entre A y B se le asocia un valor de h para generar un mapeo entre éstos puntos y calcular punto a punto las derivadas numéricas. Por tanto el resultado será una lista con éstos datos.

Adicionalmente para el caso en que se tenga los datos de una función interpolada, estos datos pueden ser utilizados para ser calcular las derivadas de éstos. Sin embargo tanto en el caso de la función como en el caso de los datos, se requiere tener en cuenta que para una mejor precisión, el intervalo de diferenciación no ha de ser muy grande y asimismo, tener presente que las derivadas de orden superior expandidas son mucho más propensas en la pérdida de precisión, que las primeras dos derivadas. Para ésto se puede utilizar valores muy cercanos o en su defecto una expansión de derivas de contengan mayores términos para ganar mayor precisión, sin embargo ésto último no es una práctica común.

A continuación los nombres de los programas:

-FCDA o First Centered Difference Aproximation: Calcula la primera derivada central de una función en el intervalo dado.

-SCDA o Second Centered Difference Aproximation: Calcula la segunda derivada central de una función en el intervalo dado.

-TCDA o Third Centered Difference Aproximation: Calcula la tercera derivada central de una función en el intervalo dado.

-QCDA o Fourth Centered Difference Aproximation: Calcula la cuarta derivada central de una función en el intervalo dado.

Se encuentran los siguientes métodos diferenciadores no centrados:

-nfirstderv

-nsecondderv

-snfirstderv

Para los ordenes superiores de diferenciación, no se recomienda utilizar ésta versión no centrada por la propagación del error, ya que en éste caso el factor h de potencia uno otorga más peso en las contribuciones de los TOS, aumentando el error de truncamentiento.

CAPÍTULO

FIVE

CAPITULO 05: INTEGRADORES

5.1 Integración Numérica

La integración numérica es una de las herramientas más utilizadas eb distintas disciplinas de las ciencias exactas y la ingeniería. Los distintos métodos existentes parten de la idea de discretizar el proceso de integración como la suma de contribuciones n-ésimas partes de un intervalo. La idea de éstas aproximaciones pueden evidenciarse con el desarrollo de las *Sumas de Riemann*, al descomponer el área de cualquier curva como la suma de múltiples áreas de diferentes rectángulos.

$$\int_a^b \mathrm{f}(x)\mathrm{d}x = \sum_{n=1}^{\infty} \lim_{x_n - x_{n-1} \to 0} f(x_n)(x_n - x_{n-1}); x \in [b-a] \tag{5.1}$$

El proceso de discretización permite seleccionar el tipo de área conocida para mapear el área de una función, de ésta manera es posible obtener una aproximación que dependerá del número de elementos que mapean la función y el área conocida. Bajo ésta idea los métodos de integración básica pueden presentarse en las formulas de Newton - Cotes.

Las formulas de Newton - Cotes cerradas son un amplio conjunto de técnicas de integración utilizando la idea de descomponer el dominio de integración $x \in [a, b]$ en n puntos equidistantes a una distancia h. La imagen de éstos puntos en la función representan el comportamiento de algunas de las partes más relevantes. La idea descomponer el dominio y su imagen, también puede ser vista como la interpolación de la función a través de un polinomio de grado n. El área entonces se calcula al sumar la contribución ponderada (las imágenes de los puntos por un factor de escala) multiplicado por el factor ponderado de espaciamiento h, estableciendo:

$$\int_a^b \mathrm{f}(x)\mathrm{d}x \approx \sum_{i=1}^{n-1} w_i f(x_i) \tag{5.2}$$

donde w_i es el factor de ponderación. Éste proceso define qué elementos del interpolador contribuyen más al momento re realizar la suma. El factor de h, el tipo de ponderación y el grado del interpolador están definidos por la figura geométrica que mapea la función. De la misma manera al tener un interpolador de mayor grado, la precisión del método es mayor. Para cada interpolador, existe un término de error analítico que dependerá de las derivadas de la función a evaluar. De manera similar las formulas abiertas de Newton - Cotes, poseen la misma estructura, pero difiere en la forma de tratar el primer elemento del interpolador, cual no es considerado en la contribución del área.

Como defecto del método está la limitación del grado de la función a evaluar. Cuando el grado del interpolador aumenta, el número de oscilaciones en los bordes aumenta también; éste comportamiento es conocido como el fenómeno de Runge. Este fenómeno ocasiona que las áreas debajo la curva en los puntos de mayor oscilación posean un área mucho mayor, generando errores exponenciales.

5.2 Descripción del método:

En este módulo se encuentran los métodos integradores que aproximan la función $f(x)$ que se va a integrar por medio de polinomios interpolantes de diferentes grados, donde los métodos varían según el número de puntos o intervalo que se utilice para dicha aproximación permitiendo calcular integrales definidas. Existen varios métodos que permiten realizar el proceso de hallar una integral defina y se dividen en este proyecto en las siguientes categorías:

-ClosedNewtonCotes: *Fournodescotes *simprule *tesimprule *trapzrule

-CompositeRules: *Compmidtrule *compsimrule *comptraprule

-DoubleIntegral: *Gaussblint *Simpbdlintg

-OpenNewtonCotes: *Midpointrule *Onenodeopn *Thrnodeopn *twonodeopn

-Tripleintegral: *Gausstpllint

-AdvancedIntegration: *Adapquad *Romberginted

CAPÍTULO

SIX

CAPITULO 06: EXTRAPOLADORES

6.1 Descripción del método:

Este módulo se encarga de los métodos de extrapolación, son métodos científico lógicos, que se basan en suponer que el curso de los acontecimientos continuara, es decir, afirma que existen unos axiomas y que estos son extrapolables a una nueva situación. La base de la extrapolación son observaciones secuenciales realizados en periodos conocidos de tiempo, estas observaciones son luego registradas como variables cuantitativas, medidas con algún tipo de escala.

Se usan para buscar soluciones a problemas lógicos o enseñar la misma pedagogía, lo cual los convierte en una herramienta muy utilizada en el marco profesional y de enseñanza, no son necesariamente exclusivos de los métodos de interpolación y tampoco se pueden considerar únicos.

En el marco de los métodos numéricos, los extrapoladores permiten recalcular, a partir de dos aproximaciones, un valor más preciso. Ésta herramienta es la piedra angular de los métodos de cuadratura. Al estar integrado en un método particular, los valores que arroja cada dos iteraciones son refinados. En la próxima iteración se obtiene una nueva aproximación que se refina con el valor extrapolado generando un valor aún más preciso. Se cuenta con el siguiente método:

-Richarexpol o Extrapolación de Richardson

En las técnicas de cuadratura, éste método es ampliamente utilizado. En el caso de la integración numérica se presenta en la integración de Romberg para el refinamiento del n-ésimo ciclo del cálculo, para daca elemento de la aproximación.

CAPÍTULO

SEVEN

DOCUMENTACIÓN E ÍNDICE DE LOS MÉTODOS - ENGLISH VER.

7.1 adapquad.py

Description: function adapquad is an adaptive quadrature method based on Simpson's rule for integration. It can also can be expanded into another integration technique by replacing those lines. It provides more accurate results by dividing the integration interval [a,b] into n subintervals, and again applying integration rule until achiving desired tolerance in each subinterval. If not met, the subinterval is divided again and the process is done again.

Inputs: lowerlimit - float or integer - integration interval element. upperlimit - float or integer - integration interval element. tol_val - float - desired error tolerance in calculated value. n - integer - numbers of levels or subintervals to generate. function - function type object - evaluates f(x) at x.

Output: app - float - defined integral aproximation.

Example line: adapquad(-2, 4, 0.005, 10, (lambda x: 4*sin(x)**2));

Dependencies: None.

Version: 1.1 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical differentiation and integration". Cengage Learning. 2010. pp: 220 - 226.

Date: 29/12/2014.

7.2 beziercurve.py

Description: function beziercurve generates an interpolation parametric curve using a third order Hermite polynomial and a pair of nodes and guidelines. Endpoint slope is calculated with the slope of lines tangent to them. Therefore no second or first analytic derivative forms of curve is needed a priori. The fast computation of this interpolant allows an interactive way to generate curves. Spatial coordinates are the only requiremnt for calculation.

Inputs: endpoints - list object - includes 4 float values for nodes. leftguide - list object - includes 4 float values for left node. rightguide - list object - includes 4 float values for right node. n - integer - optional parameter defines number n+1 elements of list.

Output: beziercoeff - list object - Bezier cofficients of polynomial.

Example code: endpoints = [[3.2, 2.4], [3.6, 0.4]]; leftguide = [[2.8, 7.5], [5.3, 6.4]]; rightguide = [[8.3, 7.5], [1.6, 2.4]]; beziercurve (endpoints, leftguide, rightguide, n=10);

Dependencies: None.

Version: 1.3 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 3 - Interpolation and Polynomial Approximation". Cengage Learning. 2010. pp: 166 - 169.

Date: 27/12/2014.

7.3 bisection.py

bisection.py is a function that calculates the roots of a polinomial function using the bisection numerical method.

inputs= rf ; raw function, string of the form a*x+b a ; left range interval b ; right range interval n ; number of iterations t ; tolerance of calculation

output= x ; the aproximated value of x such that f(x)=0

version: 1.01 for python 2.7

fast_Code: bisection((‘6*x-6’),-5,6,200,0.005);

author: JJ.Cadavid - SFTC - EAFIT University date: 24/08/2014.

7.4 clampspline.py

Description: function clampspline generates an interpolant of third order polynomial. Uses knot information of first derivatives in the first and last knot. The coefficients are solved using the tridiagonal linear system vectorial equation AX = B by recursion in the strictly diagonally dominant matrix A.

Inputs: domainpnt - list object - list whose elements are x domain data. imagepnt - list object - list whose elements are y domain data. fstderpnt - list object - first derivative knot information, 2 elements.

Outputs: a, b, c, d - Automatic tuple of lists - interpolant coefficients.

Example code: domainpnt = [0 + (x*(10-0)/5)*x2 for x in range(6)];** imagepnt = [0 + (x*(10-0)/5)*x**(2*3**0.5) for x in range(6)]; fstderpnt = [];

Dependencies: None.

Version: 1.1 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 3 - Interpolation and Polynomial Approximation". Cengage Learning. 2010. pp: 153 - 156.

Date: 27/12/2014.

7.5 compmidtrule.py

Description: function compmidtrule is part of the Composite Integration Techniques and can be thought as an upgrated version of midpoint rule integration technique that aproximates a defined integal evaluated in [a,b] with better presicion in larger intervals. It's truncation error is of order h^2 and defined by a second order derivative.

Inputs: lowerlimit - float - defines first limit of integration. upperlimit - float - defines last limit of integration. redc - integer - even integer that defines subintervals. function - function type object - evaluates f(x) at x.

Outputs: integ - float - defined integral aproximation

Example line: compmidtrule(-3.15, 6.2, 500, (lambda x: 2*x**3 + 4.3));

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 203 - 209.

Date: 28/12/2014.

7.6 compsimprule.py

Description: function compsimprule is part of the Composite Integration Techniques and can be thought as an upgrated version of Simpson rule integration technique that aproximates a defined integal evaluated in [a,b] with better presicion in larger intervals. It's truncation error is of order h^4 and defined by a fourth order derivative. Presicion is slighly better than Composite midpoint rule and so is the best option when minimizing number of computations. Is also known to be an all-purpose quadrature algorithm.

Inputs: lowerlimit - float - defines first limit of integration. upperlimit - float - defines last limit of integration. redc - integer - even integer that defines subintervals function - function type object - evaluates f(x) at x.

Outputs: integ - float - defined integral aproximation

Example line: integ = compsimprule(0, 2, 1000, (lambda x: 4*x**3));

Dependencies: None.

Version: 1.2 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 203 - 209.

Date: 28/12/2014.

7.7 comptraprule.py

Description: function comptraprule is part of the Composite Integration Techniques and can be thought as an upgrated version of Trapezoidal rule integration technique that aproximates a defined integal evaluated in [a,b] with better presicion in larger intervals. It's truncation error is of order h^2 and defined by a second order derivative. Similar to compmidtrule but request more operations Presicion is similar to Composite midpoint rule. Difference from the other methods is the need of only one integration interval therefore the number of subintervals can be even or odd.

Inputs: lowerlimit - float - defines first limit of integration. upperlimit - float - defines last limit of integration. redc - integer - integer that reduces space grid in domain. function - function type object - evaluates f(x) at x.

Outputs: integ - float - defined integral aproximation

Example line: integ = comptraprule(-2, 2, 40, (lambda x: 3*x**4));

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 153 - 156.

Date: 28/12/2014.

7.8 derivpol.py

Description: function derivpol calculates the derivatives of interpolated data. A good choice to create interpolant are the cubic splines they are used in this program. Clamped and Natural Splines used information of first and second derivatives in knots, by applying algebraic manipulation expressions to calculate derivates are obtained.

Inputs: domainlist - list object - list whose elements are x domain data. imagelist - list object - list whose elements are y domain data. evalpnt - list object - point to evaluate the derivatives.

Outputs: firstderiv, secndderiv - Automatic tuple of lists of derivatives.

Example code: domainlist = [0 + (x*(10-0)/5)*x2 for x in range(6)];** imagelist = [0 + (x*(10-0)/5)*x**4 for x in range(6)]; evalpnt = 5.4; fd, sd = derivpol(domainlist, imagelist, evalpnt);

Dependencies: cubicSpline.py - Jaan Kiusalaasr - 2013.

Version: 1.1 for Python 3.4

Definition and structure were taken from: Jaan Kiusalaasr. "Numerical Methods in Engineering With Python 3". 3th ed. "Chapter 5 - Numerical Differentiation". Cambridge University Press. 2013. PP. 191 - 195.

Date: 28/12/2014.

7.9 FCDA.py

Description: class centraldiff defines a set of numerical methods based on finite differences of central step. Capable of aproximating derivatives at a given point inside an interval. Central step does not evaluate derivatives at endpoints. Precision is highly dependant on interval size, bigger intervals lead to highter truncation errors. Minor error types are of h^2 order. Analitical errors can be calculated using fourth derivatives. Highter derivative orders can lead to high truncation errors, so results are likely to differ a lot from analytical results, especially with interpolated data.

FCDA or First Centered Difference Aproximation, calculates the first derivative in the interval. Dependencies: testlambda, getarray.

Method inputs: n - integer - Defines number of n+1 elements in list. lowerlimit - float - first term of interval. upperlimit - float - last term of interval. function - function type object - evaluates f(x) at x.

Method output: diff_array - list - list of derivated values in (a,b).

call sequence example: centraldiff.FCDA(4, 0, 2, (lambda x: 4*sin(x)**2));

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Jaan Kiusalaasr. "Numerical Methods in Engineering With Python 3". 3th ed. "Chapter 5 - Numerical Differentiation". Cambridge University Press. 2013. PP. 183 - 185.

Date: 28/12/2014.

7.10 fixedpoint.py

fixedpont.py finds the root of a function using the fixed point method given the initial aproximations p0.

Inputs: g - Fixed point function with the form x=g(x) obtained from f(x)=0 p0 - Initial aproximation tol - Tolerance n - Maximum number of itarations

Outputs: Aproximate solution X or message of failure

Quick Code: fixedpoint('(1.0/2.0)*((10.0-x**3)**(1.0/2.0))',1.5,3,10**-8)

Version: 1.0

Author: Sebastián Castaño y Felipe Lopez - SFTC - EAFIT University

7.11 fournodescotes.py

Description: class numecinteg is a set of numerical techniques applied to Numerical Integration, based on the elemental Closed Newton-Cotes and Open Newton-Cotes Formulas. This set holds the fundamental technique for highter ones like Composite Numerical Integration and Adapatative Quadrature. So for that extent, small integration intervals are required to achive certain presicion. The methods described here works in a similar way by taking an integration interval and reduced it in equally spaced elements or more subintervals and sum each of them. The success of this techniques is based on parameter h, which is the space grid size. Lower spacing will grand better presicion, also truncation and round-off error will increase by minor changes in h, every technique differs on how it's analitical error increases.

The program set allows to input interpolated data or an analytical expression as a lambda object. More restriction is applied to interpolated data, since not all orders can be integrated properly with every technique. For high interval integration, comptechrule.py or adapquad.py set is recommended.

Method Inputs: lowerlimit - float - First element in integration interval upperlimit - float - Last element in integration interval function - lambda object type - evaluates f(x) at x – or

- list - coefficients of interpolated data

flag - integer - optional - defines 0: lambda type 1: list type

Method Output: integ - float - Integration aproximation

Example code: integ = newtoncotesf(0, 6, lambda x: 2*x, flag = 0);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 193 - 201.

Date: 28/12/2014.

7.12 gaussdblint.py

Description: function gaussdblint obtains a defined double integral aproximation using a nested gaussian quarature. The strategy of calculation is using the Legendre polynomials roots and coeficients. So that high grade polynomials can be evaluated with accuracy. Adventage of this is the no need of equally spaced elements in domain. The function allows that the limits of second integral can be variable as a function of the independent variable or can be defined values. The second strategy of calculation is the continuous calculation of the second variable space grid. Since it can change, new grid size is created for every y value.

Inputs: x_limits - list object - list with x first integral limits y_limits - list object - list with y second integral limits or functions evalpnt - list object - point to evaluate the derivatives. m - integer - number of subintervals in y domain. n - integer - number of subintervals in x domain. function - function object type - a double variable lambda object. roots - nested list - list of n, m order roots of Legendre Polynomials. coefflist - nested list - list of n, m coefficients of L. Polynomials. flag - integer - 0: defined y limits, 1: variable y limits. Default 0.

Outputs: J - float - double integration aproximation.

Example code: x_limits = [0, 5]; y_limits = [0, 5]; m = 3; n = 2; function = lambda x,y: y*x*3 + y**2 - x; roots = []; Here goes the Bessel roots of first kind. coefflist = []; Here goes the pre-calculated Bessel coefficients.

Dependencies: None.

Version: 1.3 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 240 - 244.

Date: 29/12/2014.

7.13 gausstpllint.py

Description: function gausstpllint obtains a defined triple integral aproximation using a nested gaussian quarature. The strategy of calculation is using the Legendre polynomials roots and coeficients. So that high grade polynomials can be evaluated with accuracy. Adventage of this is the no need of equally spaced elements in domain. The function allows that the limits of second or third integral can be variable as a function of the independent variable or can be defined values. The second strategy of calculation is the continuous calculation of the second variable space grid. Since it can change, new grid size is created for every y value and z value.

Inputs: x_limits - list object - list with x first integral limits y_limits - list object - list with y second integral limits or functions z_limits - list object - list with z second integral limits or functions evalpnt - list object - point to evaluate the derivatives. m - integer - number of subintervals in y domain. n - integer - number of subintervals in x domain. p - integer - number of subintervals in x domain. function - function object type - a double variable lambda object. roots - nested list - list of n, m order roots of Legendre Polynomials. coefflist - nested list - list of n, m coefficients of L. Polynomials. flag - integer - 0: defined y limits, 1: variable y limits. Default 0.

Outputs: J - float - double integration aproximation.

Example code: x_limits = [0, 5]; y_limits = [0, 5]; m = 3; n = 2; function = lambda x,y: y*x*3 + y**2 - x; roots = []; Here goes the Bessel roots of first kind. coefflist = []; Here goes the pre-calculated Bessel coefficients.

Dependencies: None.

Version: 1.3 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 245 - 246.

Date: 29/12/2014.

7.14 hermitpoly.py

Description: function hermitpoly calculates an interpolant based on Hermit interpolation aided with devided difference. Using the first derivative along the domain.

Inputs: domainlist - list object - list whose elements are x domain data. imagelist - list object - list whose elements are y domain data. derivalist - list object - list with first derivative data.

Outputs: hermitcoeff - list object- Hermit interpolant coefficients.

Example code: domainlist = [0 + (x*(10-0)/5)*x2 for x in range(6)];** imagelist = [0 + (x*(10-0)/5)*x**4 for x in range(6)]; derivalist = []; Precomputed derivative list for the function - Use the differentiators submodule.

Dependencies: None.

Version: 1.3 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 3 - Interpolation and Polynomial Approximation". Cengage Learning. 2010. pp: 133 - 141.

Date: 27/12/2014.

7.15 midpointrule.py

Description: class numecinteg is a set of numerical techniques applied to Numerical Integration, based on the elemental Closed Newton-Cotes and Open Newton-Cotes Formulas. This set holds the fundamental technique for highter ones like Composite Numerical Integration and Adapatative Quadrature. So for that extent, small integration intervals are required to achive certain presicion. The methods described here works in a similar way by taking an integration interval and reduced it in equally spaced elements or more subintervals and sum each of them. The success of this techniques is based on parameter h, which is the space grid size. Lower spacing will grand better presicion, also truncation and round-off error will increase by minor changes in h, every technique differs on how it's analitical error increases.

The program set allows to input interpolated data or an analytical expression as a lambda object. More restriction is applied to interpolated data, since not all orders can be integrated properly with every technique. For high interval integration, comptechrule.py or adapquad.py set is recommended.

Method Inputs: lowerlimit - float - First element in integration interval upperlimit - float - Last element in integration interval function - lambda object type - evaluates f(x) at x – or

- list - coefficients of interpolated data

flag - integer - optional - defines 0: lambda type 1: list type

Method Output: integ - float - Integration aproximation

Example code: lowerlimit = 0; upperlimit = 5; function = [(x*(10-0)/5)*x-x for x in range(4)]; flag = 1; integ = numecinteg.midpointrule(lowerlimit, upperlimit, function, flag = 0);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 193 - 201.

Date: 28/12/2014.

7.16 muller.py

Description: function muller is a multiple polynomial root finding algorithm. Due to method limitations functions with roots that are not simple (Multiplicity > 1), Müller's technique is limitated. For that purpose use high order quadrature or adaptative root finding algorithms. Since polynomial roots are unique, all functions of this type, even those with complex roots, can be easily obtained with Müller's algorithm. This program request three seeds in which the roots are located, from there all roots there are obatained. This program will be traslated to polyroot class to make a selection between Müller's and Horner's technique whether or not there are complex roots. If tolerance is not met the program raises an exception.

Inputs: polynm - lambda object type - polynomial function to be evaluated. aprx_list - list object - list containing three seeds. tol - float - error tolerance in roots. iters - integer - Positive value providing the number of loops.

Outputs: p - float - Polynomial root obtained from seeds.

Example code: [];

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 2 - Solutions of Equations in One Variable". Cengage Learning. 2010. pp: 97 - 98.

Date: 24/12/2014.

7.17 natspline.py

Description: function clampspline generates an interpolant of third order polynomial. Uses knot information of second derivatives in the first and last knot. The coefficients are solved using the tridiagonal linear system vectorial equation AX = B by recursion in the strictly diagonally dominant matrix A.

Inputs: domainpnt - list object - list whose elements are x domain data. imagepnt - list object - list whose elements are y domain data. fstderpnt - list object - second derivative knot information, 2 elements.

Outputs: a, b, c, d - Automatic tuple of lists - interpolant coefficients.

Example code: domainpnt = [(x*(10-0)/5)*x2 for x in range(6)];** imagepnt = [(x*(10-0)/5)*x**(2*3**0.5) for x in range(6)]; fstderpnt = [];

Dependencies: None.

Version: 1.1 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 3 - Interpolation and Polynomial Approximation". Cengage Learning. 2010. pp: 145 - 149.

Date: 27/12/2014.

7.18 nevilletable.py

Description: function nevilletable computes a data interpolant in a recursive way. While Lagrange interpolation request an a priori knowledge of the n order of interpolant, a way of reducing the computation needed to test which polynomial order fits better is done by computing the Neville's Table. Interpolation nodes are created in a similar way such as Newton's divided difference. As an initial aproximation, nodes equal the f(x) data and are slowly force to fit function behaviour by comparing the difference in first x data element with the following element in a forward way. Interval is set dividing the first data with the following element, until is been completed x data reading. When tolerance is not met, new nodes must be include or if so increase tolerance error.

Inputs: domainpnt - list - Elements that compose data x-domain. imagepnt - list - Elements that compose data y-domain. tol - float - Criteria to check whether or not data fits with tolerance

Output: intrpnt - list - Polynomial coefficients of n-th grade

Example code: domainpnt = [(x*(10-0)/5)+x2 for x in range(6)];** imagepnt = [(x*(10-0)/5)*x-x for x in range(6)]; tol = 0.05; intrpnt = nevilletable(domainpnt, imagepnt, tol);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 3 - Interpolation and Polynomial Approximation". Cengage Learning. 2010. pp: 117 - 123.

Date: 28/12/2014.

7.19 newtondivdef.py

Description: function newtondivdef computes a data interpolant in a recursive way. In a similar way like Neville Table, Newton Divided difference makes a similar approach to fit data by evaluating the function in a certain shifted subset in domain and compare it in the non shifted place - Forward y element respect actual position y element. In a certain way it might look like this approach is similar to the limit definition of first derivative. Coefficients are store in a similar 1-row way done in nevilletable. Tolerance criteria is placed here as well, increase the number of nodes for better fitting or increace tolerance error.

Inputs: domainpnt - list - Elements that compose data x-domain. imagepnt - list - Elements that compose data y-domain. tol - float - Criteria to check whether or not data fits with tolerance

Output: intrpnt - list - Polynomial coefficients of n-th grade

Example code: domainpnt = [(x*(10-0)/5)+x2 for x in range(6)];** imagepnt = [(x*(10-0)/5)*x-x for x in range(6)]; tol = 0.05; intrpnt = nevilletable(domainpnt, imagepnt, tol);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 3 - Interpolation and Polynomial Approximation". Cengage Learning. 2010. pp: 124 - 126.

Date: 25/12/2014.

7.20 nfirstderv.py

Description: class ncentraldiff defines a set of numerical methods based on finite differences with no central step - forward or backward. Capable of aproximating derivatives at a given point inside an interval including endpoints. Precision is highly dependant on interval size, bigger intervals lead to highter truncation errors. Minor error types are of h^2 order. First non centered aproximations for derivatives are a fast way to obtain a good aproximation, but second non centerd aproximations provides more accurate results but increase computation demand. Other aproximations use high order Taylor series expansion.

Method inputs: n - integer - Defines number of n+1 elements in list. lowerlimit - float - first term of interval. upperlimit - float - last term of interval. function - function type object - evaluates f(x) at x.

Method output: diff_array - list - list of derivated values in (a,b).

call sequence example: function = lambda x: 0.5*x3+5*x*2+6*x+6.33;** nfirstderv(4, 0, 2, function);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Jaan Kiusalaasr. "Numerical Methods in Engineering With Python 3". 3th ed. "Chapter 5 - Numerical Differentiation". Cambridge University Press. 2013. PP. 185 - 186.

Date: 28/12/2014.

7.21 nsecondderv.py

Description: class ncentraldiff defines a set of numerical methods based on finite differences with no central step - forward or backward. Capable of aproximating derivatives at a given point inside an interval including endpoints. Precision is highly dependant on interval size, bigger intervals lead to highter truncation errors. Minor error types are of h^2 order. First non centered aproximations for derivatives are a fast way to obtain a good aproximation, but second non centerd aproximations provides more accurate results but increase computation demand. Other aproximations use high order Taylor series expansion.

Method inputs: n - integer - Defines number of n+1 elements in list. lowerlimit - float - first term of interval. upperlimit - float - last term of interval. function - function type object - evaluates f(x) at x.

Method output: diff_array - list - list of derivated values in (a,b).

call sequence example: function = lambda x: 0.5*x3+5*x*2+6*x+6.33;** nsecondderv(4, 0, 2, function);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Jaan Kiusalaasr. "Numerical Methods in Engineering With Python 3". 3th ed. "Chapter 5 - Numerical Differentiation". Cambridge University Press. 2013. PP. 185 - 186.

Date: 28/12/2014.

7.22 onenodeopn.py

Description: class numecinteg is a set of numerical techniques applied to Numerical Integration, based on the elemental Closed Newton-Cotes and Open Newton-Cotes Formulas. This set holds the fundamental technique for highter ones like Composite Numerical Integration and Adapatative Quadrature. So for that extent, small integration intervals are required to achive certain presicion. The methods described here works in a similar way by taking an integration interval and reduced it in equally spaced elements or more subintervals and sum each of them. The success of this techniques is based on parameter h, which is the space grid size. Lower spacing will grand better presicion, also truncation and round-off error will increase by minor changes in h, every technique differs on how it's analitical error increases.

The program set allows to input interpolated data or an analytical expression as a lambda object. More restriction is applied to interpolated data, since not all orders can be integrated properly with every technique. For high interval integration, comptechrule.py or adapquad.py set is recommended.

Method Inputs: lowerlimit - float - First element in integration interval upperlimit - float - Last element in integration interval function - lambda object type - evaluates f(x) at x – or

- list - coefficients of interpolated data

flag - integer - optional - defines 0: lambda type 1: list type

Method Output: integ - float - Integration aproximation

Example code: lowerlimit = 0; upperlimit = 5; function = [(x*(10-0)/5)*x-x for x in range(4)]; flag = 1; integ = numecinteg.onenodeopn(lowerlimit, upperlimit, function, flag = 0);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 193 - 201.

Date: 28/12/2014.

7.23 polyroot.py

Description: class numecinteg is a small class type python program used as a prototipe algorithm. It's actual function is to represent a polynomial from a set of coefficients. The second feature is finding it's roots by applying Horner's algorithm.

Method Inputs: None - Input built-in function takes data from user

Method Outputs: Polinomial visualization py,pz,b0 - automatic tuple - Provides information of roots.

Example code: [];

Dependencies: None.

Version: 0.8 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 2 - Solutions of Equations in One Variable". Cengage Learning. 2010. pp: 92 - 96.

Date: 24/12/2014.

7.24 QCDA.py

Description: class centraldiff defines a set of numerical methods based on finite differences of central step. Capable of aproximating derivatives at a given point inside an interval. Central step does not evaluate derivatives at endpoints. Precision is highly dependant on interval size, bigger intervals lead to highter truncation errors. Minor error types are of h^2 order. Analitical errors can be calculated using fourth derivatives. Highter derivative orders can lead to high truncation errors, so results are likely to differ a lot from analytical results, especially with interpolated data.

QCDA or Fourth Centered Difference Aproximation, calculates the fourth derivative in the interval. Dependencies: testlambda, getarray.

Method inputs: n - integer - Defines number of n+1 elements in list. lowerlimit - float - first term of interval. upperlimit - float - last term of interval. function - function type object - evaluates f(x) at x.

Method output: diff_array - list - list of derivated values in (a,b).

call sequence example: centraldiff.FCDA(4, 0, 2, (lambda x: 4*sin(x)**2));

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Jaan Kiusalaasr. "Numerical Methods in Engineering With Python 3". 3th ed. "Chapter 5 - Numerical Differentiation". Cambridge University Press. 2013. PP. 183 - 185.

Date: 28/12/2014.

7.25 regulafalsi.py

regulafalsi.py finds the root of a function continuous in an interval [p0,p1] where f(p0) and f(p1) have opposite signs

Inputs: f- Function in terms of x p0,p1 - Interval tol - Tolerance n - Maximum number of itarations

Outputs: Aproximate solution X or message of failure

Quick Code: fixedpoint('(1.0/2.0)*((10.0-x**3)**(1.0/2.0))',1.5,3,10**-8)

Version: 1.0

Author: Sebastián Castaño y Felipe Lopez - SFTC - EAFIT University

7.26 richarexpol.py

Description: function richarexpol is a small method that applies Richardson Extrapolation. This technique is one of the keystones in Advance Numerical Methods. Extrapolation allows to increase the presicion of a numerical result whitout increasing data nodes, whenever space grid size, h, is involve. For this program, extrapolation receives 2 approximations of a value dependent on h, and by setting the correct order p from analytical truncation error high accurate results can be obtained.

Inputs: P - integer - Non negative integer setting the order of correction term firstaprx - float - first approximation of an h dependent value; secondprx - float - second approximation of an h dependent value;

Method Outputs: G - float - High accurate result

Example code: [];

Dependencies: None.

Version: 1.0 for Python 3.4

Definitions are taken from: Jaan Kiusalaasr. "Numerical Methods in Engineering With Python 3". 3th ed. "Chapter 5 - Numerical Differentiation". Cambridge University Press. 2013. PP. 188 - 189.

Date: 28/12/2014.

7.27 romberginteg.py

Description: function romberginteg computes an approximation of a defined integral in [a,b]. Romberg Integration uses other minor techniques to produce accurate results. First aproximations are obtain from one of the Newton-Cotes Formulas and then results are improve with extrapolation techniques. Then the process is done all over again on the subinterval section. Every loop grid size is half for better presicion improvement.

Inputs: lowerlimit - float - First element in integration interval upperlimit - float - Last element in integration interval subinterv - positive integer - Number of subintervals function - lambda object type - evaluates f(x) at x

Outputs: R - list object - Romberg table - integration aproximation.

Example code: R = romberginteg(-3.15, 6.2, 10, (lambda x: 2*x**3 + 4.3));

Dependencies: None.

Version: 1.5 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 213 - 218.

Date: 19/02/2015.

7.28 SCDA.py

Description: class centraldiff defines a set of numerical methods based on finite differences of central step. Capable of aproximating derivatives at a given point inside an interval. Central step does not evaluate derivatives at endpoints.

Precision is highly dependant on interval size, bigger intervals lead to highter truncation errors. Minor error types are of h^2 order. Analitical errors can be calculated using fourth derivatives. Highter derivative orders can lead to high truncation errors, so results are likely to differ a lot from analytical results, especially with interpolated data.

SCDA or Second Centered Difference Aproximation, calculates the second derivative in the interval. Dependencies: testlambda, getarray.

Method inputs: n - integer - Defines number of n+1 elements in list. lowerlimit - float - first term of interval. upperlimit - float - last term of interval. function - function type object - evaluates f(x) at x.

Method output: diff_array - list - list of derivated values in (a,b).

call sequence example: centraldiff.FCDA(4, 0, 2, (lambda x: 4*sin(x)**2));

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Jaan Kiusalaasr. "Numerical Methods in Engineering With Python 3". 3th ed. "Chapter 5 - Numerical Differentiation". Cambridge University Press. 2013. PP. 183 - 185.

Date: 28/12/2014.

7.29 secant.py

secant.py finds the root of a function using the Secant method given the initial aproximations p1 and p0 so that the root is a number between them

Inputs: f - Function in terms of 'x' p0,p1 - Initial aproximations tol - Tolerance n - maximum number of iterations

Outputs: Aproximate solution X or message of failure

Quick Code: secant('math.cos(x)-x',0.5,math.pi/4,5*10**-8,20)

Version: 1.0

Author: Sebastián Castaño - SFTC - EAFIT University

7.30 simpdblintg.py

Description: function simpdblintg obtains a defined double integral aproximation using Simpson Integration Rule by reducing intervals into subintervals. Similar to Double Legendre-Gauss Quadrature, calculations are done from inside to outsie, using outside domain to calculate inside integration domain. The sum process in this method takes the results in three stages, the endpoints of the interval, and subinterval results, this last one is devided on even and odd intervals. The heavy process is done within the subintervals by continuously modifying space grid size and by applying Simpson Rule. There is no need for use roots or coeficients from Legendre Polynomials for approximation, making it a bit faster than the Gauss-Legendre Quadrature, yet presicion might decrease if compare with it.

Inputs: x_limits - list object - list with x first integral limits y_limits - list object - list with y second integral limits or functions evalpnt - list object - point to evaluate the derivatives. m - integer - number of subintervals in y domain. n - integer - number of subintervals in x domain. function - function object type - a double variable lambda object. flag - integer - 0: defined y limits, 1: variable y limits. Default 0.

Outputs: J - float - double integration aproximation.

Example code: x_limits = [0, 5]; y_limits = [0, 5]; m = 3; n = 2; function = lambda x,y: y*x**3 + y**2 - x; roots = []; coefflist = [];

Dependencies: None.

Version: 1.3 for Python 3.4

Definition and structure were taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 240 - 244.

Date: 29/12/2014.

7.31 simprule.py

Description: class numecinteg is a set of numerical techniques applied to Numerical Integration, based on the elemental Closed Newton-Cotes and Open Newton-Cotes Formulas. This set holds the fundamental technique for highter ones like Composite Numerical Integration and Adapatative Quadrature. So for that extent, small integration intervals are required to achive certain presicion. The methods described here works in a similar way by taking an integration interval and reduced it in equally spaced elements or more subintervals and sum each of them. The success of this techniques is based on parameter h, which is the space grid size. Lower spacing will grand better presicion, also truncation and round-off error will increase by minor changes in h, every technique differs on how it's analitical error increases.

The program set allows to input interpolated data or an analytical expression as a lambda object. More restriction is applied to interpolated data, since not all orders can be integrated properly with every technique. For high interval integration, comptechrule.py or adapquad.py set is recommended.

Method Inputs: lowerlimit - float - First element in integration interval upperlimit - float - Last element in integration interval function - lambda object type - evaluates f(x) at x – or

- list - coefficients of interpolated data

flag - integer - optional - defines 0: lambda type 1: list type

Method Output: integ - float - Integration aproximation

Example code: lowerlimit = 0; upperlimit = 5; function = [(x*(10-0)/5)*x-x for x in range(4)]; flag = 1; integ = numecinteg.simprule(lowerlimit, upperlimit, function, flag = 0);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 193 - 201.

Date: 28/12/2014.

7.32 snfirstderv.py

Description: class ncentraldiff defines a set of numerical methods based on finite differences with no central step - forward or backward. Capable of aproximating derivatives at a given point inside an interval including endpoints. Precision is highly dependant on interval size, bigger intervals lead to highter truncation errors. Minor error types are of h^2 order. First non centered aproximations for derivatives are a fast way to obtain a good aproximation, but second non centerd aproximations provides more accurate results but increase computation demand. Other aproximations use high order Taylor series expansion.

Method inputs: n - integer - Defines number of n+1 elements in list. lowerlimit - float - first term of interval. upperlimit - float - last term of interval. function - function type object - evaluates f(x) at x.

Method output: diff_array - list - list of derivated values in (a,b).

call sequence example: function = lambda x: 0.5*x3+5*x*2+6*x+6.33;** ncentraldiff.snfirstderv(4, 0, 2, function);

call sequence example: ncentraldiff.nfirstderv(4, 0, 2, (lambda x: 4*sin(x)**2));

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Jaan Kiusalaasr. "Numerical Methods in Engineering With Python 3". 3th ed. "Chapter 5 - Numerical Differentiation". Cambridge University Press. 2013. PP. 185 - 186.

Date: 28/12/2014.

7.33 TCDA.py

Description: class centraldiff defines a set of numerical methods based on finite differences of central step. Capable of aproximating derivatives at a given point inside an interval. Central step does not evaluate derivatives at endpoints. Precision is highly dependant on interval size, bigger intervals lead to highter truncation errors. Minor error types are of h^2 order. Analitical errors can be calculated using fourth derivatives. Highter derivative orders can lead to high truncation errors, so results are likely to differ a lot from analytical results, especially with interpolated data.

TCDA or Third Centered Difference Aproximation, calculates the third derivative in the interval. Dependencies: testlambda, getarray.

Method inputs: n - integer - Defines number of n+1 elements in list. lowerlimit - float - first term of interval. upperlimit - float - last term of interval. function - function type object - evaluates f(x) at x.

Method output: diff_array - list - list of derivated values in (a,b).

call sequence example: centraldiff.FCDA(4, 0, 2, (lambda x: 4*sin(x)**2));

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Jaan Kiusalaasr. "Numerical Methods in Engineering With Python 3". 3th ed. "Chapter 5 - Numerical Differentiation". Cambridge University Press. 2013. PP. 183 - 185.

Date: 28/12/2014.

7.34 tesimprule.py

Description: class numecinteg is a set of numerical techniques applied to Numerical Integration, based on the elemental Closed Newton-Cotes and Open Newton-Cotes Formulas. This set holds the fundamental technique for highter ones like Composite Numerical Integration and Adapatative Quadrature. So for that extent, small integration intervals are required to achive certain presicion. The methods described here works in a similar way by taking an integration interval and reduced it in equally spaced elements or more subintervals and sum each of them. The success of this techniques is based on parameter h, which is the space grid size. Lower spacing will grand better presicion, also truncation and round-off error will increase by minor changes in h, every technique differs on how it's analitical error increases.

The program set allows to input interpolated data or an analytical expression as a lambda object. More restriction is applied to interpolated data, since not all orders can be integrated properly with every technique. For high interval integration, comptechrule.py or adapquad.py set is recommended.

Method Inputs: lowerlimit - float - First element in integration interval upperlimit - float - Last element in integration interval function - lambda object type - evaluates f(x) at x – or

- list - coefficients of interpolated data

flag - integer - optional - defines 0: lambda type 1: list type

Method Output: integ - float - Integration aproximation

Example code: integ = tesimprule(0, 2, lambda x: x, 3, flag = 0);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 193 - 201.

Date: 28/12/2014.

7.35 thrnodeopn.py

Description: class numecinteg is a set of numerical techniques applied to Numerical Integration, based on the elemental Closed Newton-Cotes and Open Newton-Cotes Formulas. This set holds the fundamental technique for highter ones like Composite Numerical Integration and Adapatative Quadrature. So for that extent, small integration intervals are required to achive certain presicion. The methods described here works in a similar way by taking an integration interval and reduced it in equally spaced elements or more subintervals and sum each of them. The success of this techniques is based on parameter h, which is the space grid size. Lower spacing will grand better presicion, also truncation and round-off error will increase by minor changes in h, every technique differs on how it's analitical error increases.

The program set allows to input interpolated data or an analytical expression as a lambda object. More restriction is applied to interpolated data, since not all orders can be integrated properly with every technique. For high interval integration, comptechrule.py or adapquad.py set is recommended.

Method Inputs: lowerlimit - float - First element in integration interval upperlimit - float - Last element in integration interval function - lambda object type - evaluates f(x) at x – or

- list - coefficients of interpolated data

flag - integer - optional - defines 0: lambda type 1: list type

Method Output: integ - float - Integration aproximation

Example code: lowerlimit = 0; upperlimit = 5; function = [(x*(10-0)/5)*x-x for x in range(4)]; flag = 1; integ = numecinteg.onenodeopn(lowerlimit, upperlimit, function, flag = 0);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 193 - 201.

Date: 28/12/2014.

7.36 trapzrule.py

Description: class numecinteg is a set of numerical techniques applied to Numerical Integration, based on the elemental Closed Newton-Cotes and Open Newton-Cotes Formulas. This set holds the fundamental technique for highter ones like Composite Numerical Integration and Adapatative Quadrature. So for that extent, small integration intervals are required to achive certain presicion. The methods described here works in a similar way by taking an integration

interval and reduced it in equally spaced elements or more subintervals and sum each of them. The success of this techniques is based on parameter h, which is the space grid size. Lower spacing will grand better presicion, also truncation and round-off error will increase by minor changes in h, every technique differs on how it's analitical error increases.

The program set allows to input interpolated data or an analytical expression as a lambda object. More restriction is applied to interpolated data, since not all orders can be integrated properly with every technique. For high interval integration, comptechrule.py or adapquad.py set is recommended.

Method Inputs: lowerlimit - float - First element in integration interval upperlimit - float - Last element in integration interval function - lambda object type - evaluates f(x) at x – or

- list - coefficients of interpolated data

flag - integer - optional - defines 0: lambda type 1: list type

Method Output: integ - float - Integration aproximation

Example code: integ = trapzrule(0, 2, lambda x: 1.5*x, flag = 0);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 193 - 201.

Date: 28/12/2014.

7.37 twonodeopn.py

Description: class numecinteg is a set of numerical techniques applied to Numerical Integration, based on the elemental Closed Newton-Cotes and Open Newton-Cotes Formulas. This set holds the fundamental technique for highter ones like Composite Numerical Integration and Adapatative Quadrature. So for that extent, small integration intervals are required to achive certain presicion. The methods described here works in a similar way by taking an integration interval and reduced it in equally spaced elements or more subintervals and sum each of them. The success of this techniques is based on parameter h, which is the space grid size. Lower spacing will grand better presicion, also truncation and round-off error will increase by minor changes in h, every technique differs on how it's analitical error increases.

The program set allows to input interpolated data or an analytical expression as a lambda object. More restriction is applied to interpolated data, since not all orders can be integrated properly with every technique. For high interval integration, comptechrule.py or adapquad.py set is recommended.

Method Inputs: lowerlimit - float - First element in integration interval upperlimit - float - Last element in integration interval function - lambda object type - evaluates f(x) at x – or

- list - coefficients of interpolated data

flag - integer - optional - defines 0: lambda type 1: list type

Method Output: integ - float - Integration aproximation

Example code: lowerlimit = 0; upperlimit = 5; function = [(x*(10-0)/5)*x-x for x in range(4)]; flag = 1; integ = numecinteg.onenodeopn(lowerlimit, upperlimit, function, flag = 0);

Dependencies: None.

Version: 1.2 for Python 3.4

Definitions are taken from: Richard L. Burden, J. Douglas Faires. "Numerical Analysis" 9th ed. "Chapter 4 - Numerical Differentiation and Integration". Cengage Learning. 2010. pp: 193 - 201.

Date: 28/12/2014.