Algoritmos en la Ciencia de Datos
Sección U

Eduardo Andrés Santizo Olivet
Carné: 21007361

**Laboratorio No. 2**

<u>Resolución Numérica de Ecuaciones no Lineales</u>

Como hemos visto en clase, los métodos numéricos para la resolución de ecuaciones no lineales pueden utilizarse para encontrar puntos en donde la primera derivada de una función $f$ es igual a 0, i.e. encontrar puntos estacionarios. De acuerdo al teorema de Fermat, estos puntos estacionarios son "candidatos" para determinar los extremos locales de una función $f$. Más que determinar puntos estacionarios de una función unidimensional, es importante mencionar que, estos algoritmos para encontrar raíces son útiles para determinar los step-sizes (o learning rates) óptimos en algoritmos de optimización para el caso multidimensional.

En este laboratorio, deberá agregar a su librería dos algoritmos para resolver una ecuación de la forma:

$$F(x) = 0$$

en donde, $F$ es una función no lineal de una variable real.

**Documentación Algoritmos**

<u>Código del Back-end</u>

*Función: Parsing de Ecuaciones (Strings)*

```python
def parseEquation(formula):
    """
    Función que toma una ecuación escrita y la transforma en una función lambda

    Parameters
    ----------
    formula : str
        Ecuación escrita en notación tradicional

    Returns
    -------
    f : lambda fun
        Función lambda creada a partir de la ecuación dada en la forma de un string
    lambda_str : str
        String final que fue traducido a una función lambda. Útil para cerciorarse
        que la función lambda tiene la estructura deseada.
    """

    # Se importan los paquetes necesarios
    import ast
    import numpy as np
    import re

    # Se reemplazan los signos de "^" por el equivalente en Python ("**")
    formula = formula.replace("^", "**")

    # Se agregan paréntesis alrededor de los términos "elevados"
    expRegex = r"(?P<Ast>\*\*)(?P<Exp>[a-zA-Z0-9+\-\*\/]+)"
    formula = addParenthesis(formula, expRegex)
```

```python
    # Se buscan sucesiones de letras y se separan por un "*"
    letterRegex = r"(?:[a-zA-Z])(?:[a-zA-Z]{1,})"
    formula = addCharBetweenMatch(formula, letterRegex, "*")

    # Se buscan números seguidos por letras y se separan por un "*"
    numberRegex = r"(?:[0-9])(?:[a-zA-Z])"
    formula = addCharBetweenMatch(formula, numberRegex, "*")

    # Se extraen todas las letras únicas presentes en la ecuación
    # Estas se consideran "variables" de la ecuación.
    variables = list(set(re.findall(r"[a-zA-Z]", formula)))

    # Se reemplaza la constante "e" por "np.exp"
    formula = formula.replace("e**", "np.exp")

    # Se elimina la constante "e" de las variables
    if "e" in variables:
        variables.remove("e")

    # Se construye el string de formula lambda
    # 1. "lambda"
    # 2. Todas las variables separadas por comas
    # 3. ":"
    # 4. La ecuación construida previamente
    lambda_str = "lambda " + ",".join(sorted(variables)) + " : " + formula

    # Se "parsea" la "lambda_str". El uso del parser "ast" evita la inyección de código malicioso
    code = ast.parse(lambda_str, mode="eval")

    # Se guarda el código evaluado en "f"
    f = eval(compile(code, "", mode="eval"))

    # Se devuelve la función lambda
    return f, lambda_str
```

*Función: Agregar un Carácter entre las Letras de un Match dado por un Regex*

```python
def addCharBetweenMatch(string, regex, char):
    """
    Agrega un caracter entre las letras de un match dado por un regex. El match
    es luego re-concatenado con el resto del texto original.

    Ejemplo:
        Input : "Maria tiene 10 años"
        Match : "10"
        Char  : "0"
        Output: "Maria tiene 100 años"


    Parameters
    ----------
    regex : str
        Expresión regular definiendo el patrón al que se le hará "match"
    string : str
        String sobre el que se realizará el procesamiento
    char : str
        Caracter o caracteres a insertar entre cada letra o elemento del match

    Returns
    -------
    string: str
        String procesado con los caracteres adicionados
    """

    import re

    # Obtiene el primer match del regex en el string
    match = re.search(regex, string)
```

```python
    while match:

        # Obtiene las diferentes partes del string
        # - strStart: Texto antes del match
        # - strMiddle: El match como tal
        # - strEnd: Texto luego del match
        strStart  = string[0:match.span()[0]]
        strMiddle = string[match.span()[0]:match.span()[1]]
        strEnd    = string[match.span()[1]:len(string)]

        # Se separa el "middle" en caracteres
        strMiddle = list(strMiddle)

        # Se agregan signos de multiplicación entre cada letra
        strMiddle = char.join(strMiddle)

        # Re-concatena cada parte del string
        string = strStart + strMiddle + strEnd

        # Vuelve a buscar matches luego del procesamiento
        match = re.search(regex, string)

    # Retorna el string una vez ya no encuentra más matches
    return string
```

*Función: Agregar paréntesis alrededor del segundo match de un REGEX*

```python
def addParenthesis(string, regex):
    """
    Agrega paréntesis alrededor del segundo match de un regex.

    Ejemplo:
        Input   : "Maria y Juan tienen 10 años"
        Match 1 : "Maria"
        Match 2 : "Juan"
        Output  : "Maria y (Juan) tienen 10 años"

    Parameters
    ----------
    regex : str
        Expresión regular definiendo el patrón al que se le hará "match". El regex
        debe de contener exactamente dos grupos nombrados.
    string : str
        String sobre el que se realizará el procesamiento


    Returns
    -------
    string: str
        String procesado con los paréntesis adicionados
    """

    import re

    # Se hace match para seleccionar el texto que se encerrará en paréntesis
    match = re.search(regex, string)

    # Si se encuentra un match, entonces se procesa. De lo contrario no
    if match:

        # Se separa el string en partes (Antes del match, el match y luego del match)
        strStart  = string[0:match.span()[0]]
        strMiddle = string[match.span()[0]:match.span()[1]]
        strEnd    = string[match.span()[1]:len(string)]

        # Se agregan paréntesis alrededor del segundo elemento
        strMiddle = match.groups()[0] + "(" + match.groups()[1] + ")"

        # Se concatenan nuevamente los strings
        string = strStart + strMiddle + strEnd

    else:
        pass

    return string
```

*Función: Diferencias Finitas Centradas 1*

```python
def DiferFinitaCentrada1(f, x0, h):
    """
    Función que calcula la derivada de una función "f" en el punto "x0" utilizando
    dos miembros de la aproximación de Taylor.

    Parameters
    ----------
    f : fun
        Función unidimensional a derivar numéricamente.
    x0 : float
        Punto sobre el que se evaluará la derivada numérica.
    h : float
        Valor que determina la precisión con la que se realizará la aproximación
        de la derivada. Mientras más pequeño, mayor precisión.

    Returns
    -------
    df : float
        Aproximación numérica de la derivada de la función unidimensional.
    """

    f_sup = f(x0 + h)
    f_inf = f(x0 - h)
    df = (f_sup - f_inf) / (2*h)

    return df
```

*Función: Diferencias Finitas Centradas 2*

```python
def DiferFinitaCentrada2(f, x0, h):
    """
    Función que calcula la derivada de una función "f" en el punto "x0" utlizando
    cuatro miembros de la aproximación de Taylor.

    Parameters
    ----------
    f : fun
        Función unidimensional a derivar numéricamente.
    x0 : float
        Punto sobre el que se evaluará la derivada numérica.
    h : float
        Valor que determina la precisión con la que se realizará la aproximación
        de la derivada. Mientras más pequeño, mayor precisión.

    Returns
    -------
    df : float
        Aproximación numérica de la derivada de la función unidimensional.
    """

    f_sup  = f(x0 + h)
    f_inf  = f(x0 - h)
    f_sup2 = f(x0 + 2*h)
    f_inf2 = f(x0 - 2*h)
    df = (f_inf2 - 8*f_inf + 8*f_sup - f_sup2) / (12*h)

    return df
```

*Función: Diferencias Finitas Progresivas*

```python
def DiferFinitaProgresiva(f, x0, h):
    """
    Función que calcula la derivada de una función "f" en el punto "x0" utlizando
    una diferencia finita progresiva.

    Parameters
    ----------
    f : fun
        Función unidimensional a derivar numéricamente.
    x0 : float
        Punto sobre el que se evaluará la derivada numérica.
    h : float
        Valor que determina la precisión con la que se realizará la aproximación
        de la derivada. Mientras más pequeño, mayor precisión.

    Returns
    -------
    df : float
        Aproximación numérica de la derivada de la función unidimensional.
    """

    f_x  = f(x0)
    f_h  = f(x0 + h)
    f_2h = f(x0 + 2*h)
    df = (-3*f_x + 4*f_h - f_2h) / (2*h)

    return df
```

*Función: Método de Newton-Raphson*

```python
def SolverNewton(F, x0, k_max, epsilon):
    """
    Función para obtener los puntos estacionarios de una función F(x) = 0 a través del
    método de Newton-Raphson.

    ...

    Parameters
    ----------
    f : function
        Función unidimensional diferenciable
    x0 : float
        Solución inicial a la ecuación.
    k_max : int
        Número máximo de iteraciones para el algoritmo
    epsilon : float
        Tolerancia de error empleada para detener el algoritmo en caso alcance una
        cierta precisión deseada.


    Returns
    -------
    xk : float
        Aproximación a la raíz de la ecuación F(x) = 0
    table : pandas dataframe
        Dataframe conteniendo un resumen de la aproximación xk y el error en cada
        iteración "k".

    """
    # Se inicializan las iteraciones
    k = 0

    # Se inicializa la aproximación de la raíz de la ecuación
    xk = x0

    # Se inicializa el dataframe que almacenará los datos de cada iteración
    table = pd.DataFrame(columns = ["Iter", "Xk", "Error"])

    while (k < k_max) and (abs(F(xk)) > epsilon):
```

```python
        # Se calcula la derivada de la función en el punto xk
        dF = DiferFinitaCentrada2(F, xk, 0.00001)

        # Para evitar divisiones entre 0, si la derivada es igual a cero se
        # reemplaza por un valor muy muy pequeño
        if dF == 0:
            dF = 0.000000001

        # Se actualiza xk
        xk = xk - (F(xk) / dF)

        # Se incrementan las iteraciones
        k += 1

        # Se agrega la información actual al dataframe
        table = table.append({"Iter": k, "Xk": xk, "Error": np.abs(F(xk))}, ignore_index=True)

    return xk, table
```

*Función: Método de Bisección*

```python
def SolverBiseccion(F, lim_inf, lim_sup, k_max, epsilon):
    """
    Función para obtener los puntos estacionarios de una función F(x) = 0 a través del
    método de Bisección. Si, dadas las condiciones iniciales proporcionadas, se determina
    que el problema divergerá, se retorna una excepción.

    ...

    Parameters
    ----------
    F : function
        Función unidimensional continua en un "intervalo" y que cambia de signo dentro
        de dicho intervalo.
    intervalo : list
        Intervalo en el que la función es continua y cambia de signo. Consiste de una
        lista con la siguiente forma [límite_inf, límite_sup] o (a, b)
    k_max : int
        Número máximo de iteraciones para el algoritmo
    epsilon : float
        Tolerancia de error empleada para detener el algoritmo en caso alcance una
        cierta precisión deseada.


    Returns
    -------
    xk : float
        Aproximación a una raíz de la ecuación F(x) = 0 en el intervalo (a, b)
    df : pandas dataframe
        Dataframe conteniendo un resumen de la aproximación xk y el error en cada
        iteración "k".

    """


    # Se obtiene el límite superior e inferior del intervalo
    a, b = lim_inf, lim_sup

    # Si la multiplicación de las dos evaluaciones es mayor o igual a 0 entonces
    # el método fallará desde un inicio.
    if F(a)*F(b) >= 0:
        raise Exception("El método de bisección divergerá al utilizar este intervalo")

    # Se inicializan las iteraciones
    k = 0

    # Se inicializa la aproximación de la raíz de la ecuación
    xk = (a + b) / 2
```

```python
    # Se inicializa el dataframe que almacenará los datos de cada iteración
    df = pd.DataFrame(columns = ["Iter", "Xk", "Error"])

    while (k < k_max) and (abs(F(xk)) > epsilon):

        if (F(a)*F(xk)) < 0:
            b = xk
        else:
            a = xk

        # Se incrementan las iteraciones y
        k += 1
        xk = (a + b) / 2

        # Se agrega la información actual al dataframe
        df = df.append({"Iter": k, "Xk": xk, "Error": np.abs(F(xk))}, ignore_index=True)

    return xk, df
```

## Código del Front-end

### *Server Script*

```r
library(shiny)
library(reticulate)

# Se especifica la versión de python a usar
use_python("~/AppData/Local/Programs/Python/Python39")

# Se define el archivo del que vienen las funciones de Python
source_python("algoritmos.py")


shinyServer(function(input, output) {

    # ================================
    # EVALUACIÓN DE EVENTOS
    # ================================

    # ----------------
    # Cálculo de Ceros
    # ----------------

    # Método de Newton-Raphson
    newtonCalculate = eventReactive(input$nwtSolver, {

        # Se convierten todos los inputs en string a números
        in_EcuacionStr = input$ecuacionNew[1]
        in_X0 = as.numeric(input$x0New[1])
        in_MaxIter = as.numeric(input$maxIterNew[1])
        in_Epsilon = as.numeric(input$epsilonNew[1])

        # Se convierte la ecuación en string en una función lambda
        # "proc_Ecuacion" consiste de un objeto con dos elementos. El primero contiene la función lambda
        proc_Ecuacion = parseEquation(in_EcuacionStr)
        in_Ecuacion = proc_Ecuacion[[1]]

        # Se imprime la ecuación convertida a formato de Python
        print("Método de Newton-Raphson:")
        print(proc_Ecuacion[[2]])

        # Se ejecuta el método de Newton.
        # El output de la función consiste de la solución "xk" y la tabla con record de iteraciones
        solOutput = SolverNewton(in_Ecuacion, in_X0, in_MaxIter, in_Epsilon)

        # Se retorna la tabla con el record de iteraciones
        return(solOutput[[2]])
    })

    # Método de Bisección
    biseccionCalculate = eventReactive(input$bisSolver, {

        # Se convierten todos los inputs en string a números
        in_EcuacionStr = input$ecuacionBis[1]
        in_limInf = as.numeric(input$lim_infBis[1])
        in_limSup = as.numeric(input$lim_supBis[1])
        in_MaxIter = as.numeric(input$maxIterBis[1])
        in_Epsilon = as.numeric(input$epsilonBis[1])
```

```r
    # Se convierte la ecuación en string en una función lambda
    # "proc_Ecuacion" consiste de un objeto con dos elementos. El primero contiene la función lambda
    proc_Ecuacion = parseEquation(in_EcuacionStr)
    in_Ecuacion = proc_Ecuacion[[1]]

    # Se imprime la ecuación convertida a formato de Python
    print("Método de Bisección:")
    print(proc_Ecuacion[[2]])

    # Se ejecuta el método de Newton.
    # El output de la función consiste de la solución "xk" y la tabla con record de iteraciones
    solOutput = SolverBiseccion(in_Ecuacion, in_limInf, in_limSup, in_MaxIter, in_Epsilon)

    # Se retorna la tabla con el record de iteraciones
    return(solOutput[[2]])
})

# ----------------
# Diferenciación
# ----------------

# Diferencias Finitas Centradas 1
diferFinit1Calculate = eventReactive(input$DiferFinCentr1_Solver, {

    # Se convierten todos los inputs en string a números (menos la ecuación)
    in_EcuacionStr = input$ecuacion_Dif1[1]
    in_x0 = as.numeric(input$x0_Dif1[1])
    in_h = as.numeric(input$h_Dif1[1])

    # Se convierte la ecuación en string en una función lambda
    # "proc_Ecuacion" consiste de un objeto con dos elementos. El primero contiene la función lambda
    proc_Ecuacion = parseEquation(in_EcuacionStr)
    in_Ecuacion = proc_Ecuacion[[1]]

    # Se imprime la ecuación convertida a formato de Python
    print("Diferencia Finita Centrada 1:")
    print(proc_Ecuacion[[2]])

    # Se calcula la derivada
    df = DiferFinitaCentrada1(in_Ecuacion, in_x0, in_h)

    # Se retorna la tabla de resultados
    return(df)

})

# Diferencias Finitas Centradas 2
diferFinit2Calculate = eventReactive(input$DiferFinCentr2_Solver, {

    # Se convierten todos los inputs en string a números (menos la ecuación)
    in_EcuacionStr = input$ecuacion_Dif2[1]
    in_x0 = as.numeric(input$x0_Dif2[1])
    in_h = as.numeric(input$h_Dif2[1])
```

```r
    # Se convierte la ecuación en string en una función lambda
    # "proc_Ecuacion" consiste de un objeto con dos elementos. El primero contiene la función lambda
    proc_Ecuacion = parseEquation(in_EcuacionStr)
    in_Ecuacion = proc_Ecuacion[[1]]

    # Se imprime la ecuación convertida a formato de Python
    print("Diferencia Finita Centrada 2:")
    print(proc_Ecuacion[[2]])

    # Se calcula la derivada
    df = DiferFinitaCentrada2(in_Ecuacion, in_x0, in_h)

    # Se retorna la tabla de resultados
    return(df)

})

# Diferencias Finitas Progresivas
diferFinit3Calculate = eventReactive(input$DiferFinProg_Solver, {

    # Se convierten todos los inputs en string a números (menos la ecuación)
    in_EcuacionStr = input$ecuacion_Dif3[1]
    in_x0 = as.numeric(input$x0_Dif3[1])
    in_h = as.numeric(input$h_Dif3[1])

    # Se convierte la ecuación en string en una función lambda
    # "proc_Ecuacion" consiste de un objeto con dos elementos. El primero contiene la función lambda
    proc_Ecuacion = parseEquation(in_EcuacionStr)
    in_Ecuacion = proc_Ecuacion[[1]]

    # Se imprime la ecuación convertida a formato de Python
    print("Diferencia Finita Progresiva:")
    print(proc_Ecuacion[[2]])

    # Se calcula la derivada
    df = DiferFinitaProgresiva(in_Ecuacion, in_x0, in_h)

    # Se retorna la tabla de resultados
    return(df)

})
```

```r
    # ===============================
    # RENDER DE SALIDAS A UI
    # ===============================

    # Render Newton-Raphson
    output$salidaNewton = renderTable({
        newtonCalculate()
    }, digits = 8)

    # Render Bisección
    output$salidaBiseccion = renderTable({
        biseccionCalculate()
    }, digits = 8)

    # Render Diferencia Finita Centrada 1
    output$DiferFinCentr1_Out = renderText({
        diferFinit1Calculate()
    })

    # Render Diferencia Finita Centrada 2
    output$DiferFinCentr2_Out = renderText({
        diferFinit2Calculate()
    })

    # Render Diferencia Finita Progresiva
    output$DiferFinProg_Out = renderText({
        diferFinit3Calculate()
    })


})
```

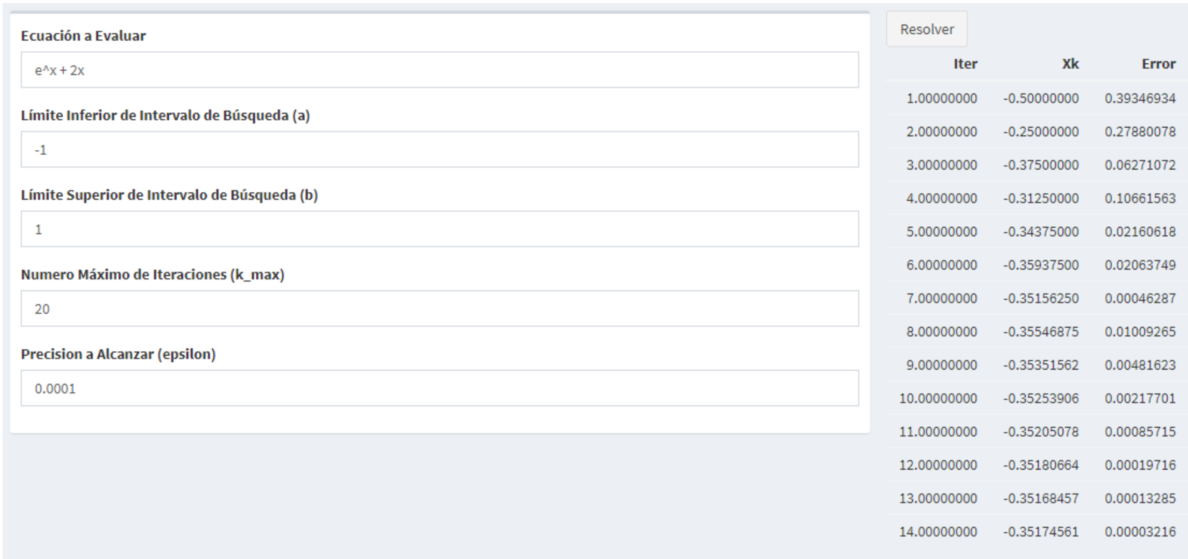
*UI Script*

```r
library(shiny)
library(shinydashboard)

# Define UI for application that draws a histogram
dashboardPage(
    dashboardHeader(title = "Algoritmos en DS"),
    dashboardSidebar(
        sidebarMenu(
            menuItem("Ceros", tabName = "Ceros",
                     menuSubItem("Bisección", tabName = "Biseccion"),
                     menuSubItem("Newton-Raphson", tabName = "Newton")),
            menuItem("Derivación", tabName = "Derivacion",
                     menuSubItem("Diferencia Finita Centrada 1", tabName = "DiferFinCentr1"),
                     menuSubItem("Diferencia Finita Centrada 2", tabName = "DiferFinCentr2"),
                     menuSubItem("Diferencia Finita Progresiva", tabName = "DiferFinProg"))
        )
    ),

    dashboardBody(
        tabItems(
            tabItem("Newton",
                    h1("Método de Newton-Raphson"),
                    h3("Rutina para obtener los puntos estacionarios de una ecuación con la forma F(x) = 0 a través del método de Newton-Raphson."),
                    box(textInput("ecuacionNew", "Ecuación a Evaluar"),
                        textInput("x0New", "Solucion Inicial de Ecuación (X0)"),
                        textInput("maxIterNew", "Numero Máximo de Iteraciones (k_max)"),
                        textInput("epsilonNew", "Precision a Alcanzar (epsilon)")),
                    actionButton("nwtSolver", "Resolver"),
                    tableOutput("salidaNewton")),

            tabItem("Biseccion",
                    h1("Método de Bisección"),
                    h3("Función para obtener los puntos estacionarios de una función F(x) = 0 a través del método de Bisección"),
                    box(textInput("ecuacionBis", "Ecuación a Evaluar"),
                        textInput("lim_infBis", "Límite Inferior de Intervalo de Búsqueda (a)"),
                        textInput("lim_supBis", "Límite Superior de Intervalo de Búsqueda (b)"),
                        textInput("maxIterBis", "Numero Máximo de Iteraciones (k_max)"),
                        textInput("epsilonBis", "Precision a Alcanzar (epsilon)")),
                    actionButton("bisSolver", "Resolver"),
                    tableOutput("salidaBiseccion")),

            tabItem("DiferFinCentr1",
                    h1("Diferencia Finita Centrada 1"),
                    h3("Función que calcula la derivada de una función 'f' en el punto 'x0' utilizando dos miembros de la aproximación de Taylor."),
                    box(textInput("ecuacion_Dif1", "Ecuación a Evaluar"),
                        textInput("x0_Dif1", "Punto para el que se Calculará la Derivada Numérica (x0)"),
                        textInput("h_Dif1", "Precisión a Utilizar (h)")),
                    actionButton("DiferFinCentr1_Solver", "Calcular Derivada"),
                    textOutput("DiferFinCentr1_Out")),
```

```r
            tabItem("DiferFinCentr2",
                    h1("Diferencia Finita Centrada 2"),
                    h3("Función que calcula la derivada de una función 'f' en el punto 'x0' utilizando cuatro miembros de la aproximación de Taylor."),
                    box(textInput("ecuacion_Dif2", "Ecuación a Evaluar"),
                        textInput("x0_Dif2", "Punto para el que se Calculará la Derivada Numérica (x0)"),
                        textInput("h_Dif2", "Precisión a Utilizar (h)")),
                    actionButton("DiferFinCentr2_Solver", "Calcular Derivada"),
                    textOutput("DiferFinCentr2_Out")),

            tabItem("DiferFinProg",
                    h1("Diferencia Finita Progresiva"),
                    h3("Función que calcula la derivada de una función 'f' en el punto 'x0' utilizando una diferencia finita progresiva."),
                    box(textInput("ecuacion_Dif3", "Ecuación a Evaluar"),
                        textInput("x0_Dif3", "Punto para el que se Calculará la Derivada Numérica (x0)"),
                        textInput("h_Dif3", "Precisión a Utilizar (h)")),
                    actionButton("DiferFinProg_Solver", "Calcular Derivada"),
                    textOutput("DiferFinProg_Out"))
        )
    )
)
```

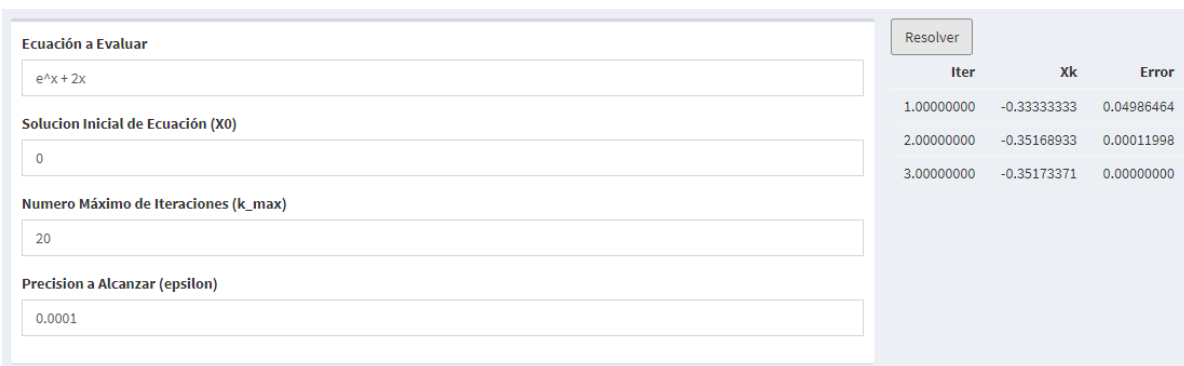
## Experimentación

### Método de Bisección




| Iter | Xk | Error |
|---|---|---|
| 1.00000000 | -0.50000000 | 0.39346934 |
| 2.00000000 | -0.25000000 | 0.27880078 |
| 3.00000000 | -0.37500000 | 0.06271072 |
| 4.00000000 | -0.31250000 | 0.10661563 |
| 5.00000000 | -0.34375000 | 0.02160618 |
| 6.00000000 | -0.35937500 | 0.02063749 |
| 7.00000000 | -0.35156250 | 0.00046287 |
| 8.00000000 | -0.35546875 | 0.01009265 |
| 9.00000000 | -0.35351562 | 0.00481623 |
| 10.00000000 | -0.35253906 | 0.00217701 |
| 11.00000000 | -0.35205078 | 0.00085715 |
| 12.00000000 | -0.35180664 | 0.00019716 |
| 13.00000000 | -0.35168457 | 0.00013285 |
| 14.00000000 | -0.35174561 | 0.00003216 |

### Método de Newton-Raphson




| Iter | Xk | Error |
|---|---|---|
| 1.00000000 | -0.33333333 | 0.04986464 |
| 2.00000000 | -0.35168933 | 0.00011998 |
| 3.00000000 | -0.35173371 | 0.00000000 |

## Conclusiones

Finalmente, aplique los dos algoritmos anteriores para resolver el siguiente problema de optimización convexo:

$$\min_{x \in \mathbb{R}} e^x + x^2$$

¿Qué algoritmo converge más rápidamente? ¿Qué ventajas y desventajas tiene el método de la bisección? ¿Qué ventajas y desventajas tiene el método de Newton-Raphson?

- Luego de realizar las pruebas dadas, se llegó a determinar que el algoritmo que converge más rápido es el método de Newton-Raphson. El método de Newton consiguió alcanzar la precisión buscada luego de apenas 3 iteraciones, mientras que el de Bisección alcanzó la precisión deseada en 13. Además de esto, el método de Bisección tiene la desventaja que requiere que el usuario le proporcione un "intervalo de búsqueda". Esto no es ventajoso, ya que el usuario debe conocer vagamente la ubicación del mínimo para que este método resulte útil. En funciones donde el mínimo no es fácilmente observable, se debe de utilizar un intervalo muy amplio que incluso puede no llegar a contener la solución requerida. El método de Newton no cuenta con esta desventaja, al únicamente requerir de un valor inicial del que partirá la búsqueda.

A pesar de las ventajas de Newton por sobre la Bisección, cabe mencionar que ambos métodos son altamente susceptibles a mínimos locales, ya que estos tienden a converger al primer mínimo local que encuentren. Esta es la razón por la que es tan importante que la función a minimizar consista de una función convexa, porque en este ámbito, cualquier mínimo local encontrado puede llegar a clasificarse como un minimizador global.

Otra desventaja importante que puede llegar a mencionarse para el método de Newton es que el mismo depende de un método adicional de diferenciación. Esto puede no afectar para problemas simples unidimensionales, pero si la función se tratara de un problema multi-dimensional, la capacidad computacional requerida probablemente incrementará ligeramente para el método de Newton a comparación del método de Bisección.