Proyecto Métodos Numéricos:
UNAM, Facultad De Estudios Superiores Aragón
Noviembre 29 de 2023

Calculadora de Métodos Numéricos Avanzados

Elaborada por: Correa Silva Diego Domingo

Bisección

Este método funciona iterativamente, comenzando con dos valores iniciales dentro del intervalo en el que se espera que se encuentre la raíz. En cada iteración, se calcula el punto medio del intervalo y se evalúa la función en ese punto. Si el valor de la función en el punto medio es cero, entonces se ha encontrado la raíz. De lo contrario, se selecciona el intervalo que contiene el signo de la función en el punto medio y se repite el proceso.

```
#private double Biseccion(string funcion, double a,
double b, int numeroIteraciones)
{
    // Implementación del método de bisección con el
    // cálculo específico del error
    int iteracion = 0;
    double c;

    do
    {
        // Calcular el punto medio
        c = (a + b) / 2;

        // Evaluar la función en los extremos y en el punto medio
        double fa = EvaluarFuncion(funcion, a);
        double fb = EvaluarFuncion(funcion, b);
        double fc = EvaluarFuncion(funcion, c);

        // Actualizar el intervalo [a, b]
        if (fa * fc < 0)
        {
            b = c;
        }
        else
        {
            a = c;
        }

        // Calcular el error y almacenarlo en la lista
```

```
        double error = Math.Abs((b - a) / b);
        errores.Add(error);

        // Incrementar el contador de iteraciones
        iteracion++;

    } while (iteracion < numeroIteraciones);

    return (a + b) / 2; // Devolver el valor medio como aproximación de la raíz
}
```

En esta implementación, el método recibe tres parámetros:

1. La función a resolver,
2. Los valores iniciales del intervalo
3. El número máximo de iteraciones.

En cada iteración, el método realiza los siguientes pasos:
Calcula el punto medio del intervalo.
Evalúa la función en los extremos y en el punto medio.
Actualiza el intervalo en función del signo de la función en el punto medio, calcula el error de la aproximación actual.
Incrementa el contador de iteraciones.
El método termina cuando se ha alcanzado el número máximo de iteraciones o cuando la función evaluada en el punto medio es cero. En este último caso, se ha encontrado la raíz exacta.

Punto Fijo

El método PuntoFijo() implementa el método de punto fijo para resolver ecuaciones no lineales.
El método funciona iterativamente, comenzando con un valor inicial x0. En cada iteración, se calcula el nuevo valor xi como el resultado de evaluar la función en el valor anterior. Este proceso se repite hasta que se alcanza la convergencia, es decir, hasta que el valor xi se aproxima a una raíz de la ecuación.
El método PuntoFijo() tiene los siguientes parámetros:

1. funcion: La función a resolver.
2. x0: El valor inicial.
3. numeroIteraciones: El número máximo de iteraciones.

El método devuelve el valor xi calculado en la última iteración.

```
double PuntoFijo(string funcion, double x0, int numeroIteraciones)
{
  int iteracion = 0;
```

```
  double xi = x0;

  do
  {
    double xi1 = EvaluarFuncion(funcion, xi);
    double error = Math.Abs((xi1 - xi) / xi1);
    errores.Add(error);
    xi = xi1;
    iteracion++;
  } while (iteracion < numeroIteraciones);

  return xi;
}
```

Explicación de la implementación;
La variable iteracion se utiliza para controlar el número de iteraciones. La variable xi almacena el valor actual de la aproximación. La variable xi1 almacena el valor de la aproximación en la siguiente iteración.
La instrucción do...while itera hasta que se alcanza la convergencia. En cada iteración, se realizan los siguientes pasos:
Se calcula el nuevo valor de la aproximación, xi1, evaluando la función en el valor actual de la aproximación, xi. Se calcula el error, error, entre el valor actual y el nuevo valor. Se almacena el error en la lista errores. Se actualiza el valor actual de la aproximación, xi, con el nuevo valor. Se incrementa el contador de iteraciones, iteracion. Una vez que se alcanza la convergencia, se devuelve el valor actual de la aproximación.

Newton-Raphson

El método NewtonRaphson() implementa el método de Newton-Raphson para resolver ecuaciones no lineales.
El método funciona iterativamente, comenzando con un valor inicial x0. En cada iteración, se calcula el nuevo valor xi como la solución de la ecuación f(xi) = 0. Este proceso se repite hasta que se alcanza la convergencia, es decir, hasta que el valor xi se aproxima a una raíz de la ecuación.
El método NewtonRaphson() tiene los siguientes parámetros:

1. funcion: La función a resolver.
2. x0: El valor inicial.
3. El método devuelve el valor xi calculado en la última iteración.

```
double NewtonRaphson(string funcion, double x0)
{
  int iteracion = 0;
  double xi = x0;
  double error;

  do
```

```
  {
    // Calcular f(xi)
    double fx = EvaluarFuncion(funcion, xi);

    // Calcular f'(xi)
    double fpx = CalcularDerivada(funcion, xi);

    // Calcular el nuevo valor según la fórmula de Newton-Raphson
    double xi1 = xi - fx / fpx;

    // Calcular el error en cada iteración
    error = Math.Abs((xi1 - xi) / xi1);
    errores.Add(error);

    // Actualizar el valor para la siguiente iteración
    xi = xi1;

    // Incrementar el contador de iteraciones
    iteracion++;

  } while (error > 0.0001 && iteracion < 1000);

  return xi;
}
```

Explicación de la implementación:
La variable iteracion se utiliza para controlar el número de iteraciones. La variable xi almacena el valor actual de la aproximación. La variable xi1 almacena el valor de la aproximación en la siguiente iteración.
La instrucción do...while itera hasta que se alcanza la convergencia. En cada iteración, se realizan los siguientes pasos:

1. Se calcula el valor de la función en el valor actual de la aproximación, xi.
2. Se calcula la derivada de la función en el valor actual de la aproximación, xi.
3. Se calcula el nuevo valor de la aproximación según la fórmula de Newton-Raphson.
4. Se calcula el error entre el valor actual y el nuevo valor.
5. Se almacena el error en la lista errores.
6. Se actualiza el valor actual de la aproximación con el nuevo valor.
7. Se incrementa el contador de iteraciones.

Interpolación Lineal

Este método numérico implementa la interpolación lineal y extrapolación lineal simple para estimar el valor de y correspondiente a un valor de x dado, utilizando un conjunto de puntos dados por el usuario. Aquí está la documentación de la lógica del método:

Atributos

- **xValues (List¡double¿):** Lista que almacena las coordenadas x de los puntos proporcionados por el usuario.
- **yValues (List¡double¿):** Lista que almacena las coordenadas y correspondientes a los puntos proporcionados por el usuario.

Métodos

1. **frmInterpol():** Constructor de la clase que inicializa el formulario y asigna los manejadores de eventos.
2. **frmInterpol_Load(object sender, EventArgs e):** Configuración inicial del DataGridView en el formulario.
3. **btnCalcular_Click(object sender, EventArgs e):**
   - Limpia las listas `xValues` y `yValues`.
   - Obtiene los datos ingresados por el usuario desde el DataGridView y los almacena en las listas correspondientes.
   - Solicita al usuario que ingrese el valor de x para interpolar/extrapolar.
   - Calcula la pendiente (m) y la ordenada al origen (b):
     - Si x es menor que el primer valor de x, utiliza la función `calcularPendienteExtrapolacion(0)`.
     - Si x es mayor que el último valor de x, utiliza la función `calcularPendienteExtrapolacion(xValues.Count - 1)`.
     - En otros casos, utiliza la función `calcularPendiente()` para la interpolación.
   - Calcula el valor de y correspondiente al x proporcionado.
   - Muestra el resultado al usuario mediante un MessageBox.
4. **promptForX():**
   - Muestra un cuadro de diálogo para que el usuario ingrese el valor de x para interpolar/extrapolar.
   - Devuelve el valor ingresado como una cadena.
5. **calcularPendiente():**
   - Calcula la pendiente (m) entre el primer y segundo punto.
   - Utiliza la fórmula: $m = \frac{y_1 - y_0}{x_1 - x_0}$.
   - Devuelve la pendiente calculada.
6. **calcularPendienteExtrapolacion(int indice):**
   - Calcula la pendiente (m) utilizando el primer o último punto para extrapolación.

- Utiliza la fórmula: $m = \frac{y_{\text{índice}} - y_0}{x_{\text{índice}} - x_0}$.
- Devuelve la pendiente calculada.

```
    private void btnCalcular_Click(object sender, EventArgs e)
{
    // Limpiar listas
    xValues.Clear();
    yValues.Clear();

    // Obtener datos del DataGridView
    for (int i = 0; i < dgvPuntos2.Rows.Count - 1; i++)
    {
        if (double.TryParse(dgvPuntos2.Rows[i].Cells[0].Value.ToString(), out dou
            double.TryParse(dgvPuntos2.Rows[i].Cells[1].Value.ToString(), out dou
        {
            xValues.Add(x);
            yValues.Add(y);
        }
        else
        {
            MessageBox.Show("Por favor, ingrese valores numéricos en todas las ce
            return;
        }
    }

    // Preguntar al usuario qué valor de x desea interpolar/extrapolar
    if (double.TryParse(promptForX(), out double xInterpolar))
    {
        // Calcular la pendiente m y la ordenada al origen b
        double m, b;

        if (xInterpolar < xValues[0]) // Extrapolación usando el primer polo
        {
            m = calcularPendienteExtrapolacion(0);
            b = yValues[0] - m * xValues[0];
        }
        else if (xInterpolar > xValues[xValues.Count - 1]) // Extrapolación usand
        {
            m = calcularPendienteExtrapolacion(xValues.Count - 1);
            b = yValues[yValues.Count - 1] - m * xValues[xValues.Count - 1];
        }
        else // Interpolación
        {
            m = calcularPendiente();
            b = yValues[0] - m * xValues[0];
        }

        // Calcular y para el valor de x interpolar/extrapolar
```

```
            double yInterpolado = m * xInterpolar + b;

            // Mostrar el resultado
            MessageBox.Show($"Para x = {xInterpolar}, y = {yInterpolado}");
        }
    }
```

Interpolación Cuadrática

Implementación en Csharp

1. **frmInterCuadratica():** Constructor de la clase que inicializa el formulario y asigna los manejadores de eventos específicos para la interpolación cuadrática.

2. **frmInterCuadratica_Load(object sender, EventArgs e):** Configuración inicial del DataGridView en el formulario de interpolación cuadrática.

3. **btnCalcular_Click(object sender, EventArgs e):**

   - **Obtención de Datos:** Se obtienen los puntos (x, y) del DataGridView y se almacenan en los arreglos `xValues` y `yValues`.
   - **Verificación de Datos:** Se verifica que todos los datos ingresados en el DataGridView sean valores numéricos. En caso contrario, se muestra un mensaje de error y se detiene la operación.
   - **Construcción del Sistema de Ecuaciones:** Se construye un sistema de ecuaciones lineales utilizando los puntos proporcionados. Se crea una matriz de coeficientes y un vector de constantes para representar el sistema lineal.
   - **Resolución del Sistema de Ecuaciones:** Se utiliza el paquete MathNet.Numerics para resolver el sistema de ecuaciones lineales y encontrar los coeficientes (a, b, c) de la ecuación cuadrática $y = ax^2 + bx + c$.
   - **Mostrar Resultados:** Se muestran los resultados al usuario mediante un MessageBox. Se incluyen los valores de los coeficientes (a, b, c) y la fórmula de la ecuación cuadrática resultante.

```
    private void btnCalcular_Click(object sender, EventArgs e)
{

    // Obtener datos del DataGridView
    double[] xValues = new double[3];
    double[] yValues = new double[3];

    for (int i = 0; i < 3; i++)
    {
        if (!double.TryParse(dgvPuntos.Rows[i].Cells[0].Value.ToString(), out xValues
            !double.TryParse(dgvPuntos.Rows[i].Cells[1].Value.ToString(), out yValues
        {
            MessageBox.Show("Por favor, ingrese valores numéricos en todas las celdas
            return;
        }
```

```
    }

    // Construir el sistema de ecuaciones
    var coefficients = Matrix<double>.Build.DenseOfArray(new double[,]
    {
        { Math.Pow(xValues[0], 2), xValues[0], 1 },
        { Math.Pow(xValues[1], 2), xValues[1], 1 },
        { Math.Pow(xValues[2], 2), xValues[2], 1 }
    });

    var constants = Vector<double>.Build.Dense(yValues);

    // Resolver el sistema de ecuaciones
    var solution = coefficients.Solve(constants);

    // Mostrar resultados
    double a = solution[0];
    double b = solution[1];
    double c = solution[2];

    MessageBox.Show($"Resultados:\na = {a}\nb = {b}\nc = {c}\n\nFórmula: y = {a}x^2 +
}
```

Polinomio Interpolador de Lagrange

1. **frmLagrange():** Constructor de la clase que inicializa el formulario y asigna los manejadores de eventos específicos para el Polinomio Interpolador de Lagrange.

2. **frmLagrange_Load(object sender, EventArgs e):** Configuración inicial del DataGridView en el formulario de interpolación de Lagrange.

3. **btnCalcular_Click(object sender, EventArgs e):**

   - **Obtención de Datos:** Se obtienen los puntos (x, y) del DataGridView y se almacenan en las listas `xValues` y `yValues`.
   - **Verificación de Datos:** Se verifica que todos los datos ingresados en el DataGridView sean valores numéricos. En caso contrario, se muestra un mensaje de error y se detiene la operación.
   - **Cálculo del Polinomio de Lagrange:** Se utiliza el método `CalcularPolinomioLagrange()` para calcular el polinomio interpolador de Lagrange. Se obtiene una expresión simbólica usando el paquete MathNet.Symbolics.
   - **Generación de Puntos para la Gráfica:** Se utiliza el método `GenerarPuntosGrafica()` para generar puntos a lo largo del polinomio de Lagrange, que se utilizarán para la representación gráfica.
   - **Mostrar Resultado en WebView:** Se utiliza el método `MostrarResultadoWebView()` para mostrar la expresión simbólica del polinomio de Lagrange en un WebView, utilizando MathJax para la representación matemática en HTML.

- **Actualizar la Gráfica:** Se utiliza el método `ActualizarGrafica()` para actualizar la gráfica con los puntos generados a lo largo del polinomio de Lagrange.

4. **GenerarPuntosGrafica(SymbolicExpression polinomio):**
   - Borra la lista de puntos de la gráfica (`puntosGrafica`).
   - Agrega puntos a lo largo del polinomio para la gráfica, variando x desde -10 hasta 10 con un paso de 0.1.
5. **ActualizarGrafica():**
   - Crea un modelo de gráfica (`plotModel`) y una serie de líneas (`lineSeries`).
   - Agrega los puntos generados a lo largo del polinomio a la serie de líneas.
   - Agrega la serie de líneas al modelo de gráfica.
   - Asigna el modelo de gráfica al control `plotView1`.
6. **CalcularPolinomioLagrange():**
   - Obtiene el número de puntos (n).
   - Inicializa un polinomio simbólico a cero.
   - Utiliza dos bucles para construir el polinomio de Lagrange, multiplicando cada término por la función de Lagrange correspondiente.
   - Retorna el polinomio de Lagrange como una expresión simbólica.
7. **MostrarResultadoWebView(SymbolicExpression polinomio):**
   - Formatea la expresión simbólica para mostrar las fracciones correctamente.
   - Construye una página HTML con MathJax y la muestra en el control WebView (`webView1`).

```
    private void btnCalcular_Click(object sender, EventArgs e)
{
    // Limpiar listas
    xValues.Clear();
    yValues.Clear();

    // Obtener datos del DataGridView
    for (int i = 0; i < dgvPuntos.Rows.Count - 1; i++)
    {
        try
        {
            var x = BigRational.Parse(dgvPuntos.Rows[i].Cells[0].Value.ToString());
            var y = BigRational.Parse(dgvPuntos.Rows[i].Cells[1].Value.ToString());

            xValues.Add(x);
            yValues.Add(y);
        }
```

```
        catch (FormatException)
        {
            MessageBox.Show("Por favor, ingrese valores numéricos en todas las celdas
            return;
        }
    }

    // Calcular el polinomio interpolador de Lagrange
    var polinomio = CalcularPolinomioLagrange();

    // Generar puntos para la gráfica
    GenerarPuntosGrafica(polinomio);

    // Mostrar el resultado en el WebView con MathJax
    MostrarResultadoWebView(polinomio);

    // Actualizar la gráfica
    ActualizarGrafica();
    private SymbolicExpression CalcularPolinomioLagrange()
{
    int n = xValues.Count;
    var polinomio = SymbolicExpression.Zero;

    for (int i = 0; i < n; i++)
    {
        var termino = new SymbolicExpression(yValues[i]);

        for (int j = 0; j < n; j++)
        {
            if (j != i)
            {
                var denominador = xValues[i] - xValues[j];
                termino *=
                (SymbolicExpression.Variable("x") - xValues[j]) / denominador;
            }
        }

        polinomio += termino;
    }

    return polinomio;
}
```

Referencias

[1] C. Verschoor, M. Cuda, and C. van de Berg. *Math.NET Numerics*. https://numerics.mathdotnet.com/, 2023.

[2] C. Verschoor, M. Cuda, and C. van de Berg. *Math.NET Symbolics*. https://symbolics.mathdotnet.com/, 2023.

[3] Microsoft. *Microsoft WebView2*. https://docs.microsoft.com/en-us/microsoft-edge/webview2/, 2023.

[4] OxyPlot. *OxyPlot*. https://oxyplot.org/, 2023.

[5] Burden, Richard L., Faires, J. Douglas, Burden, M. Annete (2016). *Numerical Analysis, 10th edition*. Cengage Learning Editores S.A. ISBN-13: 978-1305253667, ISBN-10: 1305253663.

[6] Burden, Richard L., Faires, J. Douglas (2011). *Análisis Numérico (Spanish Edition), 9th Revised edition*. Cengage Learning Editores S.A. ISBN: 978-6074816631.

[7] Chapra, S., Canale, R. (2010). *Métodos Numéricos para Ingenieros, Sexta Edición*. Editorial Mc-Graw Hill. México.

[8] Rodriguez, L. (2016). *ANÁLISIS NUMÉRICO BÁSICO: Un enfoque algorítmico con el soporte de Python*. Libro digital, Versión 4.4 – 2016. Departamento de Matemáticas, Facultad de Ciencias Naturales y Matemáticas (FCNM), ESPOL.

[9] MathJax. *MathJax - Beautiful math in all browsers*. https://www.mathjax.org/, 2023.