Universidad Abierta Interamericana

Entrega Final - Cálculo e Informática

Análisis simbólico de integrales mediante AngouriMath en C#

Autores:
[Almiron Benjamin, Martelon Tomas, Rogel Esteban]

Fecha: 16/11/2025

Índice

1. Descripción General

El siguiente proyecto fue desarrollado en **C#** utilizando la biblioteca **AngouriMath** para la resolución simbólica de integrales. Se implementó una interfaz gráfica con **Windows Forms**, permitiendo al usuario seleccionar funciones, calcular su integral y visualizar el proceso paso a paso.

2. Código principal del formulario (Form1.cs)

El formulario principal se encarga de manejar la interfaz y la interacción con el usuario. A continuación se muestra el código comentado:

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;

namespace EntregaFinalCalculoInf
{
    public partial class Form1 : Form
    {
        // Constructor del formulario principal
        public Form1()
        {
            InitializeComponent();
            this.Load += Form1_Load; // Asocia el evento Load con el
                metodo Form1_Load
        }

        // --------------------------------
        // Campos de estado
        // --------------------------------
        List<string> pasosActuales = new List<string>();  // Lista que
            guarda los pasos de resolucion de la integral
        int indicePaso = 0;                                // Indice para
            controlar el paso actual mostrado

        // ----------------------------------------------------------------
        // Evento que se ejecuta al cargarse el formulario por primera
            vez
        // ----------------------------------------------------------------
        private void Form1_Load(object sender, EventArgs e)
        {
            // Carga de expresiones predefinidas en el ComboBox
            cbExpresiones.Items.Add("x * e^x");
            cbExpresiones.Items.Add("x * sin(x)");
            cbExpresiones.Items.Add("sin(x) * x");
            cbExpresiones.Items.Add("cos(x) * (2x + 5)");
            cbExpresiones.Items.Add("ln(x) * x^2");
            cbExpresiones.Items.Add("e^x * cos(x)");
            cbExpresiones.Items.Add("cos(x)");
            cbExpresiones.Items.Add("Esta no es una funcion");
            cbExpresiones.Items.Add("");

```

```
39              // Selecciona por defecto la primera expresion
40              cbExpresiones.SelectedIndex = 0;
41          }
42
43          // ------------------------------------------------------------
44          // Evento del boton "Calcular"
45          // Inicia el proceso de integracion y muestra el primer
                resultado
46          // ------------------------------------------------------------
47          private void btnCalcular_Click(object sender, EventArgs e)
48          {
49              // Crea un analizador con la expresion seleccionada por el
                    usuario
50              Analizador analizador = new Analizador(cbExpresiones.
                    SelectedItem.ToString());
51
52              // Obtiene la lista de pasos generados por el analizador
53              pasosActuales = analizador.getPasos();
54
55              // Reinicia el contador e interfaz
56              indicePaso = 0;
57              rtbSalida.Clear();
58
59              // Muestra la primera linea (la expresion original) si hay
                    pasos disponibles
60              if (pasosActuales.Count > 0)
61              {
62                  rtbSalida.AppendText(pasosActuales[0] + "\n\n");
63                  indicePaso = 1; // Apunta al siguiente paso
64              }
65          }
66
67          // ------------------------------------------------------------
68          // Evento del boton "Siguiente Paso"
69          // Muestra los pasos del procedimiento uno a uno
70          // ------------------------------------------------------------
71          private void btnSiguientePaso_Click(object sender, EventArgs e)
72          {
73              // Verifica que haya pasos pendientes
74              if (indicePaso < pasosActuales.Count)
75              {
76                  // Muestra el paso actual en el RichTextBox
77                  rtbSalida.AppendText(pasosActuales[indicePaso] + "\n\n")
                        ;
78                  indicePaso++; // Avanza al siguiente paso
79              }
80              else
```

```
81              {
82                  // Si no quedan mas pasos, se informa al usuario
83                  MessageBox.Show("No hay mas pasos para mostrar.",
84                      "Informacion", MessageBoxButtons.OK, MessageBoxIcon.
                            Information);
85              }
86          }
87
88          // ------------------------------------------------------------
89          // Evento del boton "Reset"
90          // Limpia la interfaz y reinicia las variables internas
91          // ------------------------------------------------------------
92          private void btnReset_Click(object sender, EventArgs e)
93          {
94              pasosActuales.Clear(); // Vacia la lista de pasos
95              indicePaso = 0;        // Reinicia el indice
96              rtbSalida.Clear();     // Limpia el cuadro de texto de
                    salida
97          }
98
99          // Evento vacio (reservado para uso futuro)
100         private void cbExpresiones_SelectedIndexChanged(object sender,
                EventArgs e)
101         {
102         }
103     }
104 }
```

Listing 1: Código fuente de Form1.cs

Descripción general

El formulario `Form1` cumple las siguientes funciones principales:

- Permite al usuario seleccionar una expresión matemática desde un `ComboBox`.
- Invoca la clase `Analizador`, la cual utiliza AngouriMath para calcular la integral simbólica.
- Muestra los pasos intermedios del proceso de resolución dentro de un `RichTextBox`.
- Ofrece botones de control para calcular, avanzar paso a paso o reiniciar el proceso.

Eventos destacados

- **Form1_Load:** Inicializa la interfaz cargando expresiones predefinidas.

- **btnCalcular_Click:** Llama al analizador e inicia el cálculo.
- **btnSiguientePaso_Click:** Muestra los pasos de la integral uno por uno.
- **btnReset_Click:** Limpia los resultados para comenzar nuevamente.

3. Clase Analizador (Analizador.cs)

La clase `Analizador` es el núcleo lógico del programa. Su función principal es aplicar las reglas de integración por partes de forma simbólica, usando la biblioteca **AngouriMath**, y devolver los pasos intermedios del proceso.

A continuación se muestra el código fuente completamente comentado:

```
using AngouriMath;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static AngouriMath.Entity;
using static AngouriMath.MathS;

namespace EntregaFinalCalculoInf
{
    public class Analizador
    {
        // ----------------------------
        // Variables internas
        // ----------------------------
        Entity funcion;          // Almacena la expresion original
        bool hayError = false;   // Indica si hubo un error en la
           expresion
        bool hayFuncion = false; // Indica si la funcion es valida
        string mensajeError = ""; // Mensaje descriptivo del error

        // Variables auxiliares para el metodo ILATE
        int cantidadfactores = 0;
        Entity[] factores;
        int[] prioridades;

        string tipo_u;
        Entity u, du;            // Parte u y su derivada
        string tipo_dv;
        Entity v, dv;            // Parte dv y su integral

        // Resultados intermedios
```

```
33          Entity uv, vdu, segunda, final, simplificada;
34
35          // ------------------------------------------------------------
36          // Constructor: recibe la expresion en formato string
37          // ------------------------------------------------------------
38          public Analizador(string funcion)
39          {
40              // Verifica que la entrada no este vacia
41              if (string.IsNullOrEmpty(funcion))
42              {
43                  this.mensajeError = "ERROR: La expresion no es una
                        funcion valida";
44                  this.hayError = true;
45                  return;
46              }
47
48              // Convierte el texto a un objeto simbolico de AngouriMath
49              this.funcion = funcion;
50              this.hayFuncion = true;
51
52              // Validaciones basicas
53              if (!this.funcion.Vars.Contains("x"))
54              {
55                  this.mensajeError = "ERROR: La funcion debe contener la
                        variable 'x'";
56                  this.hayError = true;
57              }
58              else if (!(this.funcion is Entity.Mulf))
59              {
60                  this.mensajeError = "ERROR: La expresion debe ser una
                        multiplicacion";
61                  this.hayError = true;
62              }
63              else
64              {
65                  // Si pasa las validaciones, se aplica ILATE
66                  SeleccionarPartesPorILATE();
67                  this.du = this.u.Differentiate("x").Simplify(); //
                        Derivamos u
68                  this.v = this.dv.Integrate("x");                //
                        Integramos dv
69                  this.uv = (this.u * this.v);                    //
                        Producto u*v
70                  this.vdu = (this.v * this.du);                  // v *
                        du
71                  this.segunda = this.vdu.Simplify().Integrate("x"); //
                        Segunda integral
```

```
72                  this.final = (this.uv - this.segunda);          //
                        Aplicamos la formula
73                  this.simplificada = this.final.Simplify();      //
                        Simplificamos el resultado
74              }
75          }
76
77          // ------------------------------------------------------------
78          // Metodo principal: genera los pasos explicativos
79          // ------------------------------------------------------------
80          public List<string> getPasos()
81          {
82              var pasos = new List<string>();
83
84              // Mostrar la integral original
85              if (hayFuncion)
86              {
87                  if (this.hayError)
88                      pasos.Add("Expresion original:" + this.funcion.
                            ToString());
89                  else
90                      pasos.Add("Integral original:" + this.funcion.
                            ToString() + " dx");
91              }
92
93              // Si hubo errores, se muestran y se detiene el analisis
94              if (this.hayError)
95              {
96                  pasos.Add(this.mensajeError);
97                  return pasos;
98              }
99
100             // Mostrar factores detectados y su clasificacion ILATE
101             string p = "La integral consta de " + this.cantidadfactores
                    + " factores";
102             for (int f = 0; f < this.cantidadfactores; f++)
103             {
104                 p += "\n- El factor " + this.factores[f].ToString() +
105                      " es una expresion " + this.ObtenerNombreTipo(this.
                             prioridades[f]) +
106                      " (Prioridad ILATE " + this.prioridades[f] + ")";
107             }
108             pasos.Add(p);
109
110             // Mostrar eleccion de partes
111             pasos.Add("Eleccion de u (factor de mayor prioridad): " +
                    this.u.ToString());
```

```
112             pasos.Add("Calculo de du: " + this.du.ToString());
113             pasos.Add("Eleccion de dv: " + this.dv.ToString());
114             pasos.Add("Calculo de v: " + this.v.ToString());
115 
116             // Aplicar la formula por partes
117             pasos.Add("Aplicamos la formula: " + this.uv.ToString() +
118                       " -  " + this.vdu.ToString() + "dx");
119 
120             // Segunda integral y resultados
121             pasos.Add("Resolucion de la segunda integral: " + this.
                    segunda.ToString());
122             pasos.Add("Resultado final: " + this.final.ToString() + " +
                    C");
123             pasos.Add("Resultado final simplificado: " + this.
                    simplificada.ToString() + " + C");
124             pasos.Add("***Fin***");
125 
126             return pasos;
127         }
128 
129         // ------------------------------------------------------------------
130         // Clasificacion ILATE
131         // ------------------------------------------------------------------
132         int ClasificarILATE(Entity expr)
133         {
134             if (expr is Entity.Arcsinf || expr is Entity.Arccosf || expr
                    is Entity.Arctanf)
135                 return 1; // Inversa
136 
137             if (expr is Entity.Logf)
138                 return 2; // Logaritmica
139 
140             if (expr is Entity.Sinf || expr is Entity.Cosf || expr is
                    Entity.Tanf)
141                 return 4; // Trigonometrica
142 
143             if (expr is Entity.Powf)
144             {
145                 Entity.Powf potencia = (Entity.Powf)expr;
146                 string baseStr = potencia.Base.ToString();
147                 if (baseStr == "e" || baseStr == "2.71828182845905")
148                     return 5; // Exponencial
149             }
150 
151             return 3; // Algebraica (por defecto)
152         }
153 
```

```
154         // Devuelve el nombre textual de cada tipo ILATE
155         string ObtenerNombreTipo(int prioridad)
156         {
157             if (prioridad == 1) return "Inversa";
158             if (prioridad == 2) return "Logaritmica";
159             if (prioridad == 3) return "Algebraica";
160             if (prioridad == 4) return "Trigonometrica";
161             if (prioridad == 5) return "Exponencial";
162             return "Desconocida";
163         }
164 
165         // ------------------------------------------------------------
166         // Agrupa los factores en un solo producto (excepto uno)
167         // ------------------------------------------------------------
168         Entity AgruparFactores(Entity[] factores, int numFactores, int
                indiceSaltear)
169         {
170             Entity resultado = null;
171             for (int i = 0; i < numFactores; i++)
172             {
173                 if (i == indiceSaltear) continue;
174                 resultado = (resultado == null) ? factores[i] :
                        resultado * factores[i];
175             }
176             return resultado;
177         }
178 
179         // ------------------------------------------------------------
180         // Selecciona u y dv segun la regla ILATE
181         // ------------------------------------------------------------
182         void SeleccionarPartesPorILATE()
183         {
184             this.u = null;
185             this.dv = null;
186 
187             Entity.Mulf mult = (Entity.Mulf)this.funcion;
188 
189             // Contar factores
190             this.cantidadfactores = mult.DirectChildren.Count();
191 
192             // Crear arrays
193             this.factores = new Entity[this.cantidadfactores];
194             this.prioridades = new int[this.cantidadfactores];
195 
196             // Rellenar los arrays con cada factor y su prioridad
197             int index = 0;
198             foreach (var factor in mult.DirectChildren)
```

```
199             {
200                 this.factores[index] = factor;
201                 this.prioridades[index] = ClasificarILATE(factor);
202                 index++;
203             }
204
205             // Determinar cual tiene mayor prioridad (menor numero)
206             int indiceMasPrioritario = 0;
207             int menorPrioridad = prioridades[0];
208             for (int i = 1; i < this.cantidadfactores; i++)
209             {
210                 if (this.prioridades[i] < menorPrioridad)
211                 {
212                     menorPrioridad = this.prioridades[i];
213                     indiceMasPrioritario = i;
214                 }
215             }
216
217             // Asignar u y dv
218             u = factores[indiceMasPrioritario];
219             if (this.cantidadfactores == 2)
220                 dv = this.factores[1 - indiceMasPrioritario];
221             else if (this.cantidadfactores > 2)
222                 dv = AgruparFactores(this.factores, this.
                        cantidadfactores, indiceMasPrioritario);
223             else
224                 return;
225         }
226     }
227 }
```

Listing 2: Código fuente de Analizador.cs

Resumen del funcionamiento

- La clase recibe una expresión y valida que contenga la variable x y sea una multiplicación.
- Se clasifican los factores según la regla **ILATE** (Inversa, Logarítmica, Algebraica, Trigonométrica, Exponencial).
- Se elige el factor de mayor prioridad como u y el resto como dv.
- Se aplican las operaciones simbólicas: derivación, integración y simplificación final.
- El método `getPasos()` devuelve una lista de cadenas con cada paso detallado para mostrar en la interfaz.