Centro universitario de ciencias exactas e ingenierías

Licenciatura en Física

REPORTE FINAL

CÓMPUTO CIENTÍFICO II

Jorge Iván Hernández Martínez

Índice

Capítulos **Página**

1. Evaluación y graficado de funciones

1.1. Evaluación de funciones

En MATLAB es posible asignar valores a ciertas variables, por ejemplo x,y,m entre muchísimas letras o combinaciones de ellas formando palabras. Asignar valores a las variables es lo mas importante de programar en MATLAB ya que todo lo que se haga en un futuro dependerá de como es que estas variables se definen. Por ejemplo nos permiten hacer pares ordenados, si le damos valores por ejemplo a x, que será la variable independiente, y a y que será la independiente. Con esos datos ya podemos definir una función de la forma $y = f(x)$, y podemos trabajar con ellos e incluso graficarlos.
La forma de asignar valores en MATLAB por ejemplo a x, comúnmente se escribe de la forma $x = 0 : 1{,}0 : 10$, lo que indica valores x desde 0 hasta 10 en intervalos de 1,0, es decir 10 valores, aunque también es posible darle los valores deseados siguiendo la forma $x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$.
Luego podemos definir una función $y = f(x)$, que por ejemplo puede ser $y = cos(x)$. Lo que quiere decir es que ahora y tomara los valores del coseno de cada x.
En la figura (1) se encuentra un ejemplo de como es que se puede definir una función, de las formas aquí vistas.

1.2. Graficado de funciones

La forma en la que MATLAB interpreta el graficado de funciones es por medio del comando plot, que se define de la forma:

$$plot(x1, y1, x2, y2, x3, y3, ..., 'opciones') \tag{1}$$

Donde los vectores x y y deben de ser de la misma longitud, esto para que sea posible relacionar el valor de uno con el otro.
Ademas después de definir cual será tu valor que estará en el eje x y cual estará en el eje y, MATLAB te da la opción de poder modificar cosas estéticas de tus gráficas, con las que se podrá mejorar o hacer mas visible partes del grafico que resultan importantes. Entre toda estas opciones nos encontramos con una que permite especificar el estilo de linea del marcador, color, grosor, etc.:
Para definir el color de la figura que se graficará, tenemos que hacer uso de algunas contracciones de colores desde el idioma ingles, en donde se toma la

primera letra que forma el nombre del color, por ejemplo; r (Red), g(Green), b(Blue), c(Cyan), m (Magenta), y (Yellow), k (blacK), w (White)
También podemos modificar el tipo de linea o estilo por ejemplo: - (solida), – (trazas), : puntos), -. (lineas y puntos)
Pero no solo eso, muchas veces es necesario remarcar algunos puntos en especifico, donde es necesario resaltar datos interesantes a analizar, para ello tenemos:$+, o, *, ., x, s$ (square), d(diamante), p (estrella 5 puntas, pentagram), h(estrella 6 puntas, hexagrama) En la figura (1) se puede observar el uso de uno de estos marcadores, específicamente el uso de círculos para resaltar puntos.

1.3. Nombre en los ejes

En la figura es posible agregarle un titulo en cada eje donde por ejemplo xlabel('Eje x','FontName','Times','Fontsize',12), coloca el titulo de Eje x en dicho eje, esto a un tamaño de letra 12, mientras que ylabel('Eje y','FontName' ,'Times','Fontsize',12) coloca el titulo del Eje y en dicho eje. También es posible colocar un titulo a la figura con el comando title, que se define con la siguiente froma: ('Titulo','FontName','Times','Fontsize',16), el que coloca el Titulo a un tamaño de letra 16. Cabe mencionar que estos tamaños se pueden cambiar según se desee.

1.4. Leyendas

Algo muy útil en MATLAB es colocar una leyenda o cuadro descriptivo en la figura, por ejemplo si se tienen tres funciones simplemente se escribe legend('uno','dos','tres') y se coloca un cuadro con la leyenda como el de la figura (1), la cual indica que es cada plot, colocando el nombre de cada uno.

1.5. Código

```
x=0:.1:6
y=sin(x)
plot(x,y,'o--')
xlabel('eje x')
ylabel('eje y')
legend('sen(x)')
```

1.6. Ejemplo

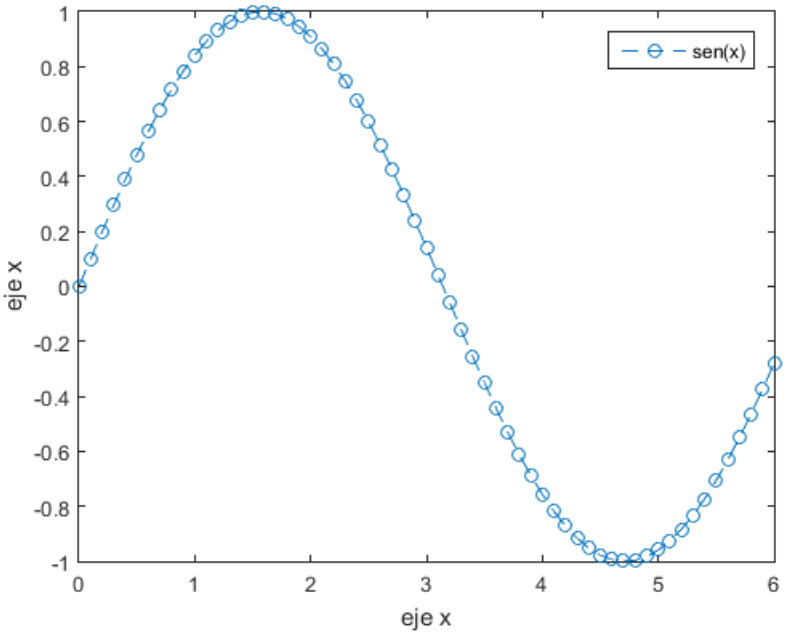

Figura 1: Ejemplo de graficado

2. Tiro parabólico

2.1. Introducción

En este apartado se presenta una simulación de tiro parabólico en MATLAB, en el que modificando las condiciones iniciales con las que se lanze el objeto, como la velocidad inicial en la dirección x (v_{0x}), la velocidad inicial en y (v_{0y}) y la altura inicial (y_0) , se es posible describir la trayectoria que tendrá el objeto en el tiempo, así como los puntos de mayor interés, que vendrían siendo la altura máxima y el tiempo que tarda en regresar a la misma altura en la que se lanzó.
Las ecuaciones que se pretenden utilizar, son descritas por la mecánica del movimiento, las cuales para nuestros intereses son las siguientes:[1]

Para el caso de la posición del objeto en el tiempo en las coordenadas x e y:

$$x = v_{0x}t$$

$$y = \frac{1}{2}gt^2 + v_{0y}t + y_0$$

Donde g es la aceleración de la gravedad y t es el tiempo.
Para la velocidad en las dos distintas coordenadas:

$$v_x = v_{0x}$$

$$v_y = v_{0y} + gt$$

2.2. Descripción del código

A continuación se realiza la descripción de todas las partes del código, con las que es posible obtener la figura(2), un gráfico que describe la trayectoria de un objeto con tiro parabólico y mediante el método de bisección resalta los puntos de interés (máximo y mínimo), así como dando el tiempo que tardo en caer a la misma posición de donde fue lanzado.

En la primera parte del código, es el lugar donde se definen las condiciones iniciales, y donde se pueden modificar a consideración, sin miedo a estropear el código.

```
clear all
global g v0y y0 t x
t0=0; tf=20; dt=0.1;
v0x =5;
v0y=10;
y0=10;
g=-9.81;
```

En la segunda parte se definen las funciones o ecuaciones de tiro parabólico que utilizaremos durante todo el proceso, de tal manera que las podamos llamar desde cualquier lugar.

```
t=t0:dt:tf;
y1='0.5*g*t.^2+v0y*t+y0';
vy1 ='g*t+v0y';
x1='v0x*t';
vx1='v0x';

y= eval(y1); vy= eval(vy1);
x= eval(x1); vx= eval(vx1);
```

Esta tercera parte, es la mas esencial, ya que encierra la parte del cálculo de los puntos máximos y mínimos, en donde dependiendo de la velocidad en el eje x escogida, se manda a llamar al programa "raiz2"que será el encargado del método de bisección.

```
n=length(x);
for k=2:n
 if  v0x==0
    if y(k-1)*y(k)<0
        xmn=raiz2(y1,t(k-1),t(k));
    end
    if vy(k-1)*vy(k)<0
        xmx=raiz2(vy1,t(k-1),t(k));
   end
 else
    %---------------------------------
     if y(k-1)*y(k)<0
         xmn=raiz2(y1,x(k-1),x(k));
     end
    if vy(k-1)*vy(k)<0
        xmx=raiz2(vy1,x(k-1),x(k));
    end
  end
    %-------------------------------------
end
```

Para la cuarta parte dependiendo de la velocidad inicial en el eje x que se haya escogido, se definirán las nuevas variables, en base a los resultados

obtenidos por el programa secundario “raiz2”. Esto para evitar una indeterminación. Estas variables son las que luego usaremos para gráficar los puntos máximos y mínimos.

```
if v0x==0
    t=xmx ; ymx=eval(y1); xmx=0;
    t=xmn ; ymn=eval(y1); xmn=0;
else
    t=(xmx)/v0x ; ymx=eval(y1);
    t=(xmn)/v0x ; ymn=eval(y1);
end
```

Esta última parte del código principal se dedica especialmente a las salidas de información. Primeramente se gráfica el recorrido que tendrá el objeto. Luego se gráfican ahí mismo los dos puntos correspondientes a la altura máxima y la posición de cuando vuelve el objeto a la misma altura de donde fue lanzado. Finalmente mediante el comando ”fprintf”de MATLAB al usuario se le brinda información en la ventana de comandos de la altura máxima y el tiempo que tardó en caer.

```
clf
plot(x,y,'k')
axis([-1 (xmx+10) (ymn-10) (ymx+10)])

hold on
plot(xmx,ymx,'ob')
plot(xmn,ymn,'or')
fprintf('la altura max es %4.2f m y la el tiempo en llegar
al piso es de%8.3f segundos \n',ymx,2*t)
```

El código que se muestra abajo corresponde al código secundario, llamado en la tercera parte del código principal. Encierra al método de bisección, en el que toma dos puntos donde ocurre un cambio de signo, toma un promedio entre ambos y el resultado lo evalúa en la función, verificando que el resultado sea menor a 0.01, valor que nosotros consideramos como el error. Si es así, encontramos el cero de la función y termina. Pero de no ser así, se re-definen los puntos donde ocurre el cambio de signo y vuelve a empezar.

```
 function r=raiz2 (y1, a, b)
global g v0y y0 v0x
    e=0.01;
    for k=1E+3
c=(a+b)/2; t=c;
        fc= eval(y1);
        if abs(fc)<e; r=c; return;end
        t=a;fa=eval(y1);
        t=b; fb=eval(y1);
        if fa*fc<0; b=c; end
        if fb*fc<0; a=c; end
    end

r=c;
```

2.3. Código

A continuación se muestra el código completo, escrito en MATLAB para el caso del tiro parabólico. Resaltando con círculos de colores los puntos donde el objeto alcanza la mayor altura, y donde regresa a la misma altura donde fue lanzado (figura (2)) . También en la ventana de comandos muestra la altura máxima y el tiempo que tardo en llegar a la misma altura de donde fue lanzado(figura(3)).

```
clear all
global g v0y y0 t x
t0=0; tf=20; dt=0.1;
v0x =5;
v0y=10;
y0=10;
g=-9.81;
%--------------------------------------
t=t0:dt:tf;
y1='0.5*g*t.^2+v0y*t+y0';
vy1 ='g*t+v0y';
x1='v0x*t';
vx1='v0x';

y= eval(y1); vy= eval(vy1);
x= eval(x1); vx= eval(vx1);
%--------------------------------------
n=length(x);
for k=2:n
 if  v0x==0
    if y(k-1)*y(k)<0
        xmn=raiz2(y1,t(k-1),t(k));
    end
    if vy(k-1)*vy(k)<0
        xmx=raiz2(vy1,t(k-1),t(k));
   end
 else
    %---------------------------------
     if y(k-1)*y(k)<0
         xmn=raiz2(y1,x(k-1),x(k));
     end
    if vy(k-1)*vy(k)<0
        xmx=raiz2(vy1,x(k-1),x(k));
    end
  end
end
%----------------------------------------
```

```
if v0x==0
    t=xmx ; ymx=eval(y1); xmx=0;
    t=xmn ; ymn=eval(y1); xmn=0;
else
    t=(xmx)/v0x ; ymx=eval(y1);
    t=(xmn)/v0x ; ymn=eval(y1);
end
%---------------------------------------
clf
 plot(x,y,'k')
 axis([-1 (xmx+10) (ymn-10) (ymx+10)])

 hold on
 plot(xmx,ymx,'ob')
 plot(xmn,ymn,'or')
 fprintf('la altura max es %4.2f m y la el tiempo en llegar al
 piso es de%8.3f segundos \n',ymx,2*t)
```

A continuación se muestra el código secundario, encargado de realizar el cálculo de la altura máxima y el punto donde vuelve a regresar a la misma altura. Todo mediante el método de bisección.

```
 function r=raiz2 (y1, a, b)
global g v0y y0 v0x
    e=0.01;
    for k=1E+3
c=(a+b)/2; t=c;
        fc= eval(y1);
        if abs(fc)<e; r=c; return;end
        t=a;fa=eval(y1);
        t=b; fb=eval(y1);
        if fa*fc<0; b=c; end
        if fb*fc<0; a=c; end
    end

r=c;
```

2.4. Simulación

Para la primera simulación se utilizaron cómo condiciones iniciales $y_0 = 10m, v_{0x} = 7m/s, v_{0y} = 10m/s$

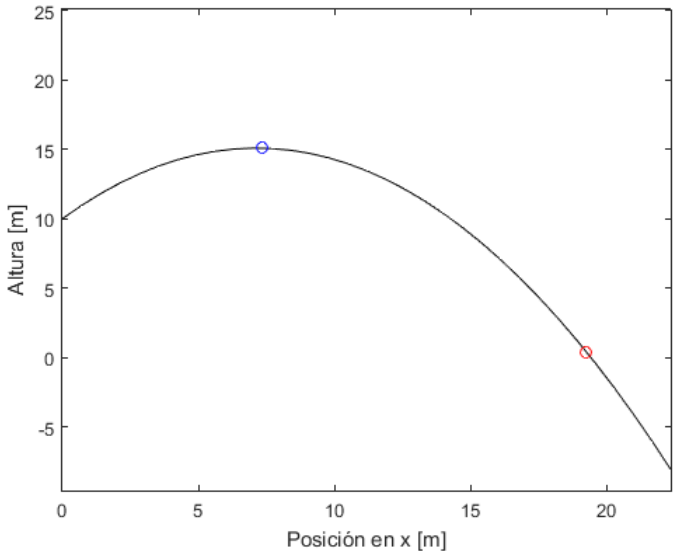

Figura 2: Gráfica generada por MATLAB para primera simulación

```
>> caida_libre
la altura max es 15.09 m y la el tiempo en llegar al piso es de   5.500 segundos
>>
```

Figura 3: Resultados mostrados en la ventana de comandos de MATLAB

Para la segunda simulación se utilizaron como condiciones iniciales $y_0 = 10, v_{0x} = 0, v_{0y} = 10,$

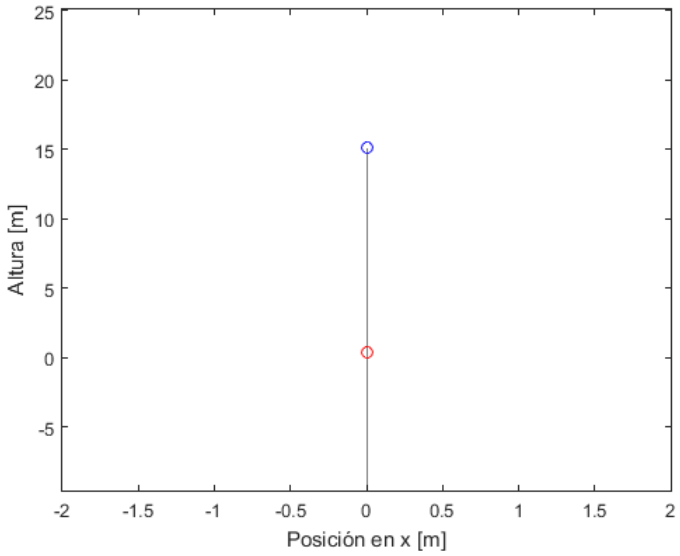

Figura 4: Gráfica generada por MATLAB segunda simulación

```
>> caida_libre
la altura max es 15.09 m y la el tiempo en llegar al piso es de   5.500 segundos
>>
```

Figura 5: Resultados mostrados en la ventana de comandos de MATLAB

Cómo se pudo observar comparando la figura (2) y la figura (4), solo se modificó la velocidad en el eje x con la que salia el objeto, y se encontró que aun así el tiempo y la altura máxima no cambian. Un resultado esperado correctamente.

3. Taylor

3.1. Introducción

El análisis completo de una función puede resultar muy difícil en algunas ocasiones. Una forma de abordar este problema es aproximar la función por una más sencilla.

En este caso vamos aproximar las funciones por polinomios. Dicha aproximación se hace cerca de un valor concreto y solo servirá para valores cercanos. A medida que nos alejemos, la aproximación será menos confiable y es posible que el polinomio se aleje mucho de la función bajo estudio.
Utilizaremos un método llamado polinomios de Taylor que permite aproximar una función derivable en el entorno reducido alrededor de un punto $x \in (a, d)$ mediante un polinomio cuyos coeficientes dependen de las derivadas de la función en ese punto.
Podemos expresarlo más formalmente como: si $n \geq 0$ es un entero y f una función que es derivable n veces en el intervalo cerrado $[\ a,\ x]$ y $n+1$ veces en el intervalo abierto $(\ a,\ x)$, entonces se cumple que:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + ... + \frac{f^n(a)}{n!}(x-a)^n \qquad (4)$$

Podemos ver que entre más valores calculemos de la ecuación, va creciendo el grado del polinomio que obtendremos, y por lo tanto este va tomando la forma de la función , en un pequeño intervalo alrededor del punto que se esta considerando.[2]

3.2. Descripción del código

A continuación se realiza la descripción de todas las partes del código escrito en MATLAB, con las que es posible mediante el método de polinomios de Taylor, obtener aproximaciones de una función alrededor de un punto.
Con este mismo código, al final se obtienen la figura (6) y la figura (7), con las diferentes aproximaciones dependiendo del orden del polinomio de taylor que se elija. Resaltando cada una de diferente color y forma, para su fácil apreciación.

En la primera parte del código se comienza definiendo a x como simbólica para poder trabajar con ella en funciones. *xo* representa el punto alrededor del que trabajáremos; f es la función que utilizáremos; *x1* como todos los valores de x que serán usados en la función; n es el grado máximo que tendrá el polinomio de Taylor; *col* y *style* son los colores y tipo de linea que tendrá cada plot de cada aproximación, para así poder diferenciarlas entre sí. También toda esta parte esta dedicada a definir todas las variables que el usuario puede modificar sin estropear el código.

```
clear all
syms x
%Entradas ------------------------------------
x0=3*pi/4;
x1=0:pi/50:2*pi;
f=sin(x);
n=3;
col=['g';'b';'k';'y'];
style=['--';'-.';' :';'--'];
```

En la segunda parte se le da a x (que era simbólica) el valor de *x1* , para así evaluarla en la función f

```
%Evaluar función-------------------------------
x=x1;
y1=eval(f);
```

Para la tercera parte se dedica a solo graficar la función de color rojo. Esto con la finalidad de poder observar como se van realizando las aproximaciones a nuestra función.

```
%Gráficar función-----------------------------------
clf
plot(x1,y1,'r')
```

En esta parte a x se le vuelve a dar un valor definido, esta vez será el valor alrededor del que trabajáremos *x0*. Para luego evaluar en la función.

```
%Evaluar x0 ------------------------------------
x=x0;
y0=eval(f);
```

Luego solamente se ubica este punto en la gráfica con un asterisco color rojo.

```
%Resaltar x0 -----------------------------------
hold on
plot(x0,y0,'*r')
```

La sexta parte encierra todo el calculo del polinomio de Taylor. Primeramente se le vuelve a dar a x el valor *x0*, para luego evaluar la función en ese punto. Teniendo así el primer termino del polinomio de Taylor. Luego mediante un ciclo que iniciará en 1 hasta el n que hayamos puesto. Se comenzará por obtener la derivada de la función, para luego ser evaluada en el punto *x0*.
Por último se construye el polinomio de Taylor como en la ecuación (4).

```
%Polinomio de Taylor------------------------------
x=x0;
F=eval(f);

for k=1:n
    syms x
    fp=diff(f,x,k);
    x=x0;
    fp0=eval(fp);

    x=x1;

    F=F+(fp0*(x-x0).^(k))/factorial(k);
```

Dentro del mismo ciclo se encuentra el código encargado de graficar cada aproximación, de un color y lineado diferente, utilizando los comandos *color* y *linestyle*, y haciendo uso de las matrices de colores y lineado que se definieron al principio.
Para darle mejor presentación a la gráfica, se utilizan los comandos *axis* que muestra las dimensiones de la gráfica; *grid on* pone una cuadricula en el gráfico. Y por último el comando *legend* que es el encargado de mostrar al usuario que representa cada linea o punto del gráfico.

```
%Gráfica cada aproximación-----------------------------------
  plot(x,F,'color',col(k,:),'linestyle',style(k,:))
  axis([0 6 -5 5])
  grid on
  legend('Función','x0','1° aproximación','2° aproximación',
  '3° aproximación','4° aproximación','Location','northeast')

end
```

3.3. Código

A continuación se muestra el código completo para encontrar el polinomio de Taylor de grado n, mostrando al final un gráfico como la figura (6) y la figura(7), con todas las aproximaciones alrededor del punto seleccionado.

```
clear all
syms x
%Entradas ----------------------------------------------
x0=3*pi/4;
x1=0:pi/50:2*pi;
f=sin(x);
n=3;
col=['g';'b';'k';'y'];
style=['--';'-.';' :';'--'];
%Evaluar función-----------------------------------------
x=x1;
y1=eval(f);
%Gráficar función----------------------------------------
clf
plot(x1,y1,'r')
%Evaluar x0 ----------------------------------------------
x=x0;
y0=eval(f);
%Resaltar x0 ---------------------------------------------
hold on
plot(x0,y0,'*r')
%Polinomio de Taylor-------------------------------------
x=x0;
F=eval(f);

for k=1:n
   syms x
   fp=diff(f,x,k);
   x=x0;
   fp0=eval(fp);
   x=x1;
   F=F+(fp0*(x-x0).^(k))/factorial(k);
%Gráfica cada aproximación-------------------------------
   plot(x,F,'color',col(k,:),'linestyle',style(k,:))
   axis([0 6 -5 5])
   hold on
   ylabel('eje y')
   xlabel('eje x')
   grid on
   legend('Función','x0','1° aproximación','2° aproximación',
   '3° aproximación','4° aproximación','Location','northeast')
end
```

3.4. Simulación

Para la primera simulación se utilizó n=2 para la función sin(x) y $x0$=3*pi/4; $x1$=0:pi/50:2*pi

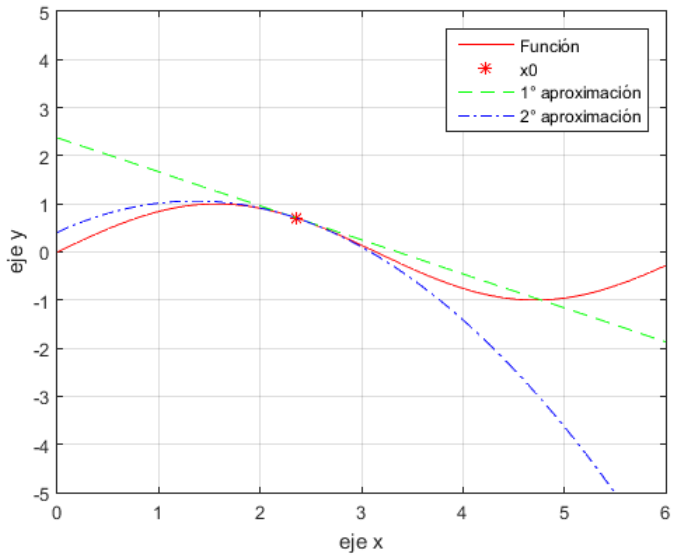

Figura 6: Gráfica generada por MATLAB para primera simulación

Para la segunda simulación se utilizó n=4 para la función sin(x) y $x0$=3*pi/4; $x1$=0:pi/50:2*pi

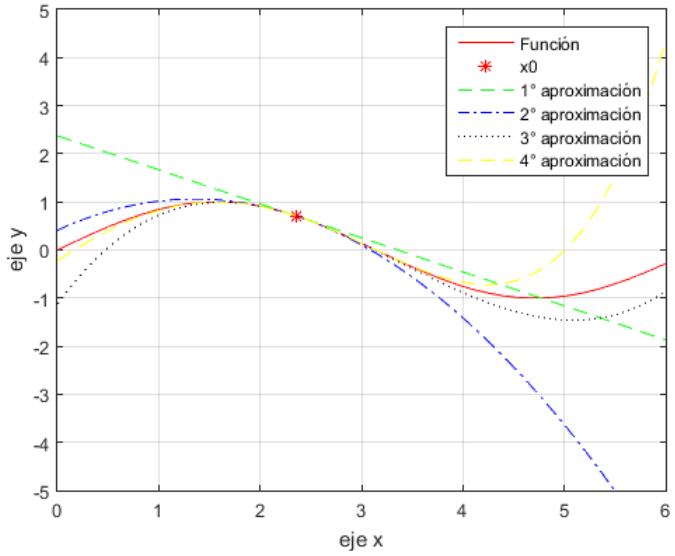

Figura 7: Gráfica generada por MATLAB para segunda simulación

Cómo se pudo observar entre la figura (6) y la figura (7), solo se modificó el valor de n (grado de aproximación) para las dos simulaciones, mostrando que entre mayor es el grado del polinomio, se obtiene una mayor aproximación a la función alrededor de un punto arbitrario dado.

4. Interpolación (interp1)

4.1. Introducción

El comando interp1 se emplea para interpolar una serie de datos. El formato de este comando es:

$$yi = interp1(x, y, xi, metodo) \tag{5}$$

Donde:
x : abscisa de los puntos a interpolar, expresada como vector fila.
y : ordenada de los puntos a interpolar, expresada como vector fila.
xi : abscisas para construir la función de interpolación, expresada como vector fila. Si es un solo valor, calculará el valor interpolando con la función

declarada en métodos.
metodo: determina el método de interpolación, entre:
nearest -interpolación asignado el valor del vecino más cercano.
linear -interpolación lineal (default)
spline -interpolación con spline cúbica
pchip -interpolación con polinomios de Hermite
cubic -(igual que 'pchip')
v5cubic -interpolación Cúbica usada in MATLAB 5 [3]

4.2. Código

A continuación el código para realizar interpolación con método *spline*, para datos discontinuos en t y p. en donde el ajuste se realizara desde $x = 1$ hasta $x = 6$

```
t = [1 2 3 4 5];
p = [3 5 7 5 6];
x = 1:0.1:6;
y = interp1 (t, p, x, 'spline');
plot (t, p,'o',x, y)
ylabel('eje y')
xlabel('eje x')
```

4.3. Simulación

Utilizando el código anterior es posible interpolar los datos usando método *splines*

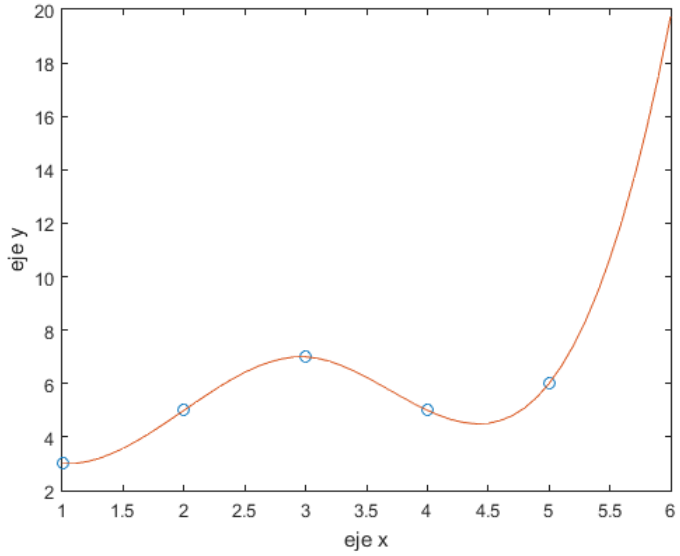

Figura 8: Gráfica generada por MATLAB para interpolación con spline

También Graficando todos los métodos en una sola figura puede servir para su comparación. ejecutando la siguiente serie de comandos, donde aparecen todos, para así notar la diferencia con cada método.

```
clear all
t = [1 2 3 4 5 6 7 8];
 p = [3 5 7 5 6 7 7 5];
x = 1:0.1:8;
y = interp1 (t, p, x, 'spline') ; plot (t, p,'o',x, y); hold on
y = interp1 (t, p, x, 'linear') ; plot (x, y,'r')
y = interp1 (t, p, x, 'nearest') ; plot (x, y,'g')
y = interp1 (t, p, x, 'pchip') ; plot (x, y,'b')
y = interp1 (t, p, x, 'cubic') ; plot (x, y,'c')
y = interp1 (t, p, x, 'v5cubic') ; plot (x, y,'m')
legend('Puntos de ref','splines','linear','nearest','pchip',
'cubic','v5cubic','Location','southeast')
 ylabel('eje y')
 xlabel('eje x')
```

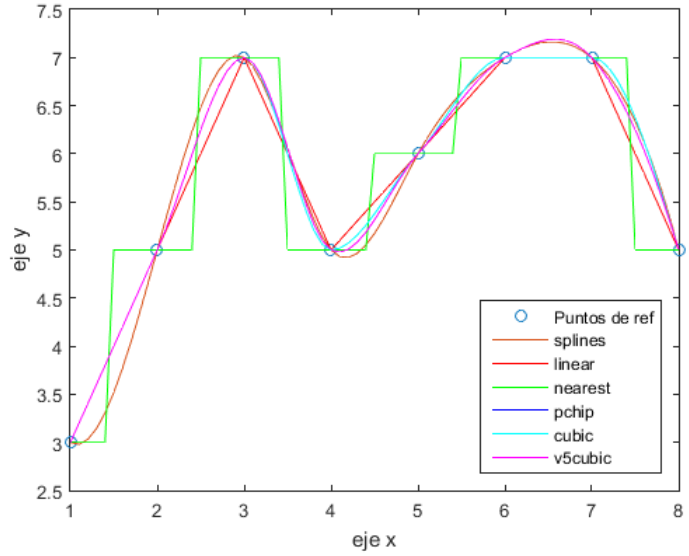

Figura 9: Gráfica generada por MATLAB utilizando todos los métodos de interpolación

5. Condicional if

5.1. Introducción

Una sentencia condicional es una instrucción MATLAB que permite tomar decisiones sobre si se ejecuta un grupo de comandos que cumplen una condición o, por el contrario, omitirlos.
En una sentencia condicional se evalúa una expresión condicional. Si la expresión es verdadera, el grupo o bloque de comandos se ejecutan. Si la expresión es falsa, MATLAB no ejecuta (salta) el grupo de comandos en cuestión. [4]

Para este trabajo usaremos el condicional if en 3 formas diferentes, y cuya estructura general es la siguiente:

```
if expresión(verdadera)
    acción 1
elseif expresión(verdadera)
    acción 2

    ...

else(falso)
    accion "n"
end
```

Además, la sentencia else es opcional. Esto significa que en el caso de que haya varios elseif y ningún else, si alguna condición de los elseif es verdadera, los comandos serán ejecutados, pero en otro caso (todas las condiciones de los elseif son falsas) no se ejecutan ni se realizará ninguna operación.

5.2. Primer forma de condicional if

Analizaremos la primera forma del condicional if (figura (10)) que tiene la estructura:

```
If expresión (verdadero)
acción
End.
```

El siguiente código es para expresar la función que se encuentra a continuación dentro del intervalo x ∈ [-5,5]:

$$f(x) = \begin{cases} 1 & si \quad x < 0 \\ \\ x^2 + 1 & si \quad x \geq 0 \end{cases} \tag{6}$$

5.2.1. Código

```
clear all
clf; hold on
for k=1:101;
    x=-5+(k-1)*0.1;
    y=1;
    if x>=0
        y=x^2+1;
    end
    plot(x,y,'.b')
    axis([-6 6 -3 30])
end
 ylabel('eje y')
 xlabel('eje x')
```

5.2.2. Gráfica

La gráfica producida por el código anterior, que corresponde al primer caso del condicional if, es la siguiente:

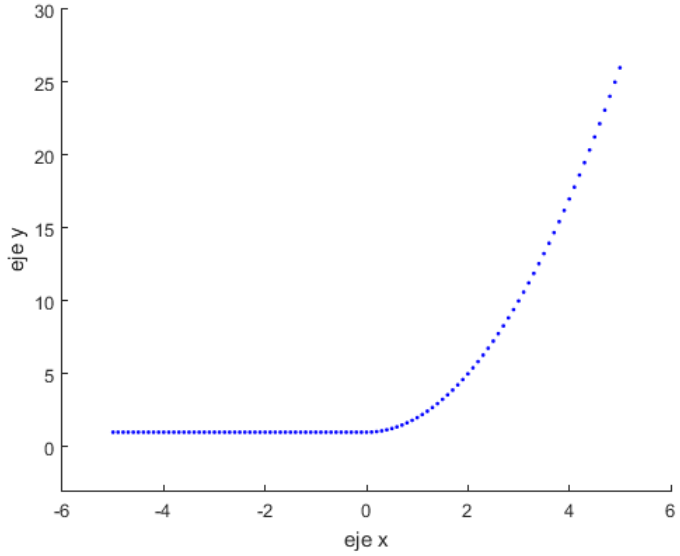

Figura 10: Gráfica generada por MATLAB para primer condicional

5.3. Segunda forma de condicional if

Analizaremos la segunda forma del condicional if (figura (11)) que tiene la estructura:

```
If expresión (verdadero)
acción 1
else (Falso)
acción 2
End.
```

El siguiente código es para expresar la función que se encuentra a continuación dentro del intervalo x ∈ [-5,5]:

$$f(x) = \begin{cases} x^3 & si \quad x < 2 \\ \\ -x^2 + 12 & si \quad x \geq 2 \end{cases} \tag{7}$$

5.3.1. Código

```
clear all
clf; hold on
for k=1:101;
    x=-5+(k-1)*0.1;
    if x>=2;
        y=-(x^2)+12;
    else
        y=x^3;
    end
     plot(x,y,'.b')
      axis([-6 6 -140 20])
       ylabel('eje y')
       xlabel('eje x')
end
```

5.3.2. Gráfica

La gráfica producida por el código anterior, que corresponde al segundo caso del condicional if, es la siguiente:

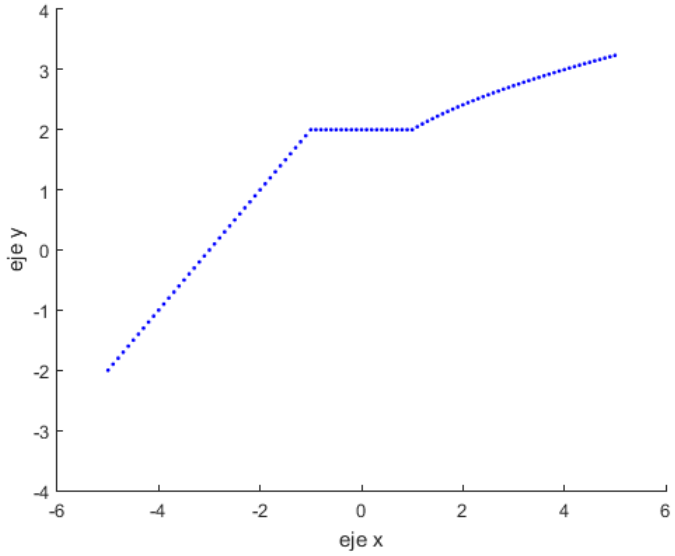

Figura 11: Gráfica generada por MATLAB para segundo condicional

5.4. Tercera forma de condicional if

Analizaremos la tercer forma del condicional if (figura (12))que tiene la estructura:

```
If expresión (verdadero)
acción 1
elseif expresión (verdadero)
acción 2
. . .
else (Falso)
acción "n"
End
```

El siguiente código es para expresar la función que se encuentra a continuación dentro del intervalo x ∈ [-5,5]:

$$f(x) = \begin{cases} x+3 & si \quad x \leq -1 \\ 2 & si \quad -1 < x < 1 \\ \sqrt{x}+1 & si \quad x \geq 1 \end{cases} \tag{8}$$

5.4.1. Código

```
clear all
clf; hold on
for k=1:101
    x=-5+(k-1)*0.1;
    if x>=1
        y=sqrt(x)+1;
    elseif x<=-1
        y=x+3;
    else
        y=2;
    end
    plot(x,y,'.b')
        axis([-6 6 -4 4])
       ylabel('eje y')
       xlabel('eje x')
end
```

5.4.2. Gráfica

La gráfica producida por el código anterior, que corresponde al tercer caso del condicional if, es la siguiente:

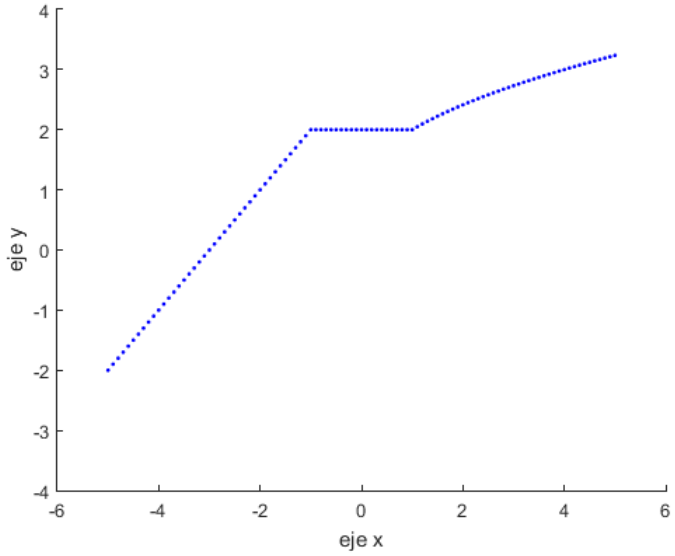

Figura 12: Gráfica generada por MATLAB para tercer condicional

5.5. Ejemplo de Trafico

Un ejemplo que podemos tomar es el siguiente código, en donde es posible apreciar que se utilizan diferentes condicionales con la intensión de simular el trafico de una carretera. Se cuenta con semáforos y se trata de evitar la acumulación de automóviles en puntos, es decir, evitar la congestión del trafico, optimizando los tiempos de los semáforos según las distancias. Además en la figura generada se tienen unos histogramas para distinguir de forma gráfica la acumulación de autos. Este un ejemplo de aplicación de los condicionales.

5.5.1. Código

```
clear all
ao=[1+rand(1,3)];
uo=[ zeros(1,3)];
xo=[ zeros(1,3)];

L1=2000; d1=100; L2=4000; d2=100;
y=[1 2 3];
dt=1.0; TS=80;
n1=1; rv1=(-1)^n1; n2=1; rv2=(-1)^n2;

for k=1:800
t=k*dt;
x=0.5*ao.*dt.^2 + uo.*dt + xo;
u=    ao.*dt    + uo ;
uo=u; xo=x;
ao(u>60*10/36)=0;

if mod(t,TS)==0; n1=n1+1; rv1=(-1)^n1; end
if mod(t,TS)==0; n2=n2+1; rv2=(-1)^n2; end

clf
subplot 211
plot(x,y,'sk','MarkerSize',16);axis([-10 6000 0 4])
text(500,3.7,[num2str(t) 'seg'],'fontsize',16)
hold on

if rv1>0; plot(L1,0.5,'.g','MarkerSize',20); end
if rv1<0; plot(L1,0.5,'.r','MarkerSize',20); end

if rv2>0; plot(L2,0.5,'.g','MarkerSize',20); end
if rv2<0; plot(L2,0.5,'.r','MarkerSize',20); end

if rv1<0
ao(xo>L1-d1 & xo<=L1)=0;
uo(xo>L1-d1 & xo<=L1)=0;
xo(xo>L1-d1 & xo<=L1)=L1;
else
ao(uo==0)=1+rand(size(ao(uo==0)));end;
if rv2<0
ao(xo>L2-d2 & xo<=L2)=0;
uo(xo>L2-d2 & xo<=L2)=0;
xo(xo>L2-d2 & xo<=L2)=L2;
else
ao(uo==0)=1+rand(size(ao(uo==0)));
```

```
subplot 413; hist(x(:),[250:500:5750]);
axis([0 6000 0 50]); title('pos')
subplot 414; hist(u(:),[  5: 10: 100]);
axis([0  100 0 50]); title('vel')

drawnow

if mod(t,40)==0;
ao=[ao; 1+rand(1,3)];
uo=[uo;  zeros(1,3)];
xo=[xo;  zeros(1,3)];
end

ao(x>6010)=NaN;
uo(x>6010)=NaN;
xo(x>6010)=NaN;
end
```

5.5.2. Simulación

A continuación se muestra la simulación de trafico con el código anterior

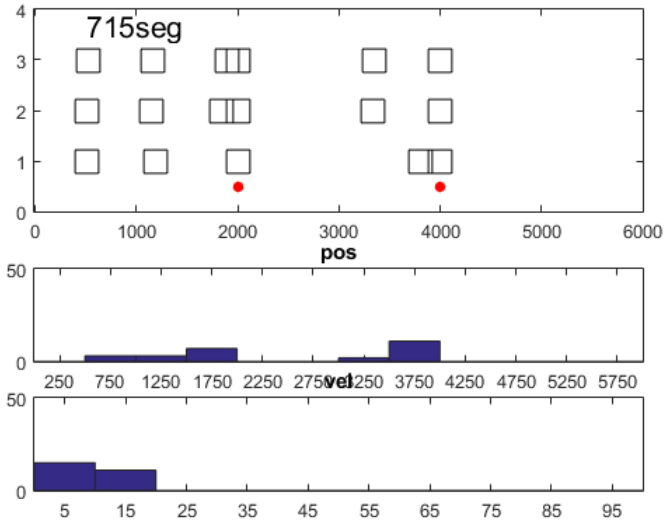

Figura 13: Simulación de tráfico

6. Polyfit y Polyeval

6.1. Introducción

Los polinomios son expresiones matemáticas muy utilizadas en la resolución de problemas matemáticos y modelados. En muchos casos, el polinomio representa una forma práctica de crear ecuaciones.
Los polinomios son funciones que tienen la forma:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0 \tag{9}$$

donde los coeficientes a_n, a_{n-1}, a_1, a_0 son números reales y n un número entero positivo que es el grado u orden del polinomio.
En MATLAB los polinomios se representan mediante un vector fila en el cual los elementos simbolizan los coeficientes del polinomio. El primer término es el coeficiente de la variable x de mayor grado. El vector debe contener todos los coeficientes, incluso los que son cero.

6.2. Valor de un polinomio (Polyval)

El valor de un polinomio en un punto x se puede calcular mediante la función polyval, que tiene la siguiente sintaxis:

$$polyval(p, x) \tag{10}$$

Donde p es un vector con los coeficientes del polinomio. x puede ser un número, una variable con un valor numérico asignado, x puede ser también una matriz o un vector. En este caso el valor se calculará para cada uno de los elementos, utilizando operaciones elemento a elemento; el resultado será un vector o una matriz, con los valores correspondientes al polinomio.[5]

6.2.1. Ejemplo

Como ejemplo, es posible evaluar el siguiente polinomio. Sea la función:

$$f(x) = x^3 - 2x^2 + 1 \tag{11}$$

Evaluarla en el punto $x = 5$.
El problema puede resolverse fácilmente mediante de la siguiente manera:

$$\begin{aligned} p &= [1\,2\,3\,0\,1]; \\ E &= polyval(p, 9) \end{aligned} \tag{12}$$

Como puede verse, se crea un vector p con los coeficientes del polinomio, y se evalúa directamente en el punto $x = 9$ con ayuda del comando polyval.

6.3. Curvas de ajuste mediante polinomios (polyfit)

El cálculo de curvas de ajuste, también llamado análisis de regresión, es un proceso que consiste en ajustar mediante una función un conjunto de datos representados por puntos. Se pueden emplear polinomios para realizar ajustes sobre datos. Cuando se tienen n puntos es posible definir un polinomio de grado menor de n-1 que no tiene por qué pasar necesariamente por todos los puntos dados, pero que puede darnos una buena aproximación de los datos. El método mas común para encontrar el mejor ajuste es el método de mínimos cuadrados. La forma de realizar ajustes con polinomios en MATLAB es mediante la función polyfit, cuya sintaxis mas básica se muestra a continuación:

$$p = polyfit(x, y, n) \tag{13}$$

Donde: p es el vector de los coeficientes de ajuste.
x es un vector con las coordenadas horizontales de los datos (variable independiente).
y es el vector con las coordenadas verticales (variable dependiente).
n es el grado de polinomio de ajuste deseado.
La función polyfit se utiliza para calcular polinomios de ajuste sobre m puntos para cualquier grado, hasta máximo m-1. Si $n = 1$, el polinomio resultante será una linea recta, si $n = 2$ será una parábola y así sucesivamente.[7]

6.4. Ejemplos

A continuación se mostrarán ejemplos de ajustes de primer hasta tercer orden. Como ya se mencionó, en general si se tiene una serie de datos x e y, o incluso y como una función cualquiera $y = f(x)$, es posible ajustar un polinomio de grado n a dicho par ordenado de datos. Un caso particular y muy sencillo para mostrar lo anterior es cuando se tienen datos cualesquiera que presentan una dependencia aproximadamente lineal entre ellos, por ejemplo los puntos:

$$(1, 1{,}1), (2, 4{,}6), (3, 5{,}1), (4, 8{,}8), (5, 10{,}1) \tag{14}$$

Si graficamos x contra y, resulta fácil observar que dichos datos presentan aproximadamente un comportamiento lineal, por ello es posible ajustarle un polinomio de grado $n = 1$. Esta tarea es muy sencilla al realizar un pequeño programa como el que se muestra en seguida, en el cual al principio es necesario definir nuestras variables análogamente a como se mostró en la tabla anterior, para posteriormente graficar estos puntos y verificar el comportamiento de los mismos. Una vez verificado el comportamiento, le ajustamos el polinomio de grado 1, para después una vez que tengamos el polinomio, definir una nueva variable (en este caso $x1$), la cual se utilizará para evaluar dicho polinomio obtenido en un cierto intervalo continúo generalmente en el cual están contenidos los datos y finalmente graficar dicha recta a la par de los puntos, mostrando la dependencia antes mencionada.

```
clear all
%Definimos x e y como una serie de puntos cualesquiera
x=[1,2,3,4,5];
y=[1.1,4.6,5.1,8.8,10.1];
plot(x,y,'or')
hold on
p=polyfit(x,y,1);
%Definimos una nueva variable para evaluar polinomio obtenido
x1=linspace(0,6);
y1=polyval(p,x1);
plot(x1,y1,'b--')
title('Ajuste grado 1','FontName','Times','Fontsize',16)
legend('Datos','P1')
xlabel('Eje x','FontName','Times','Fontsize',13)
ylabel('Eje y','FontName','Times','Fontsize',13)
saveas(gcf,'linearecta','png')
```

A continuación se muestra la gráfica obtenida al ejecutar el programa anterior, donde se obtiene que el polinomio de grado 1 ajustado a los datos es $P1 = 2{,}22000x - 0{,}70000$.

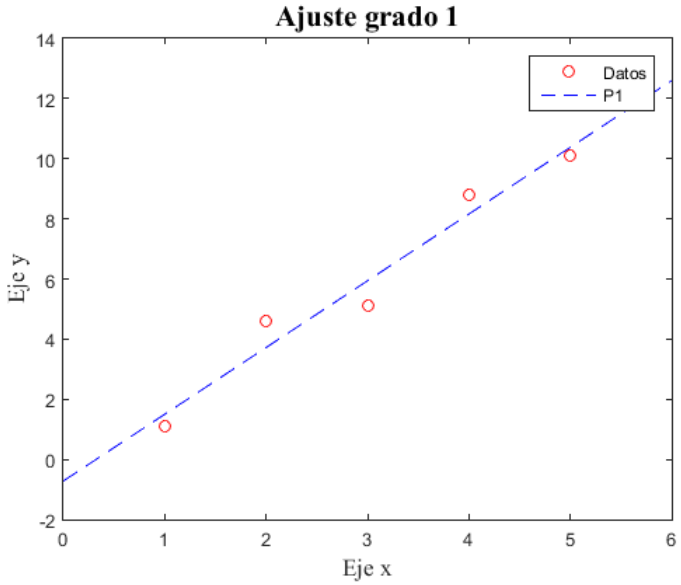

Figura 14: Ajuste lineal

Se realizó un ajuste con polinomio de grado dos de manera similar, se definieron los vectores x, y representando una toma de datos que asemejan una parábola:

$$(1,2{,}3), (2,3), (3,3{,}8), (4,5{,}4), (5,6), (6,6{,}7), (7,7{,}2), (8,2), (9,6{,}3), \\ (10,4{,}8), (11,4), (12,3{,}3), (13,2{,}9) \tag{15}$$

Se utilizó la función de matlab polyfit para calcular el vector p que contiene los coeficientes del polinomio que se utiliza para ajustar, después se definió xp que es un intervalo en x donde evaluaremos la función (desde 1.5 a 15.5 con un incremento de 0.5). Finalmente se evaluó con la función polyval el polinomio p en xp y se graficó como se muestra a continuación.

```
clear all
x=[1 2 3 4 5 6 7 8 9 10 11 12 13];
y=[2.3 3 3.8 5.4 6 6.7 7.2 7 6.3 4.8 4 3.3 2.9];
p=polyfit(x,y,2) %da los coeficientes del polinomio ajustado.
xp=(1.5:0.5:15.5);
yp=polyval(p,xp);
clf
plot(x,y,'*r',xp,yp,'--k','linewidt',2,'MarkerSize',5)
axis([0 14 1 8])
title('Ajuste de orden 2')
xlabel('EJE X')
ylabel('EJE Y')
```

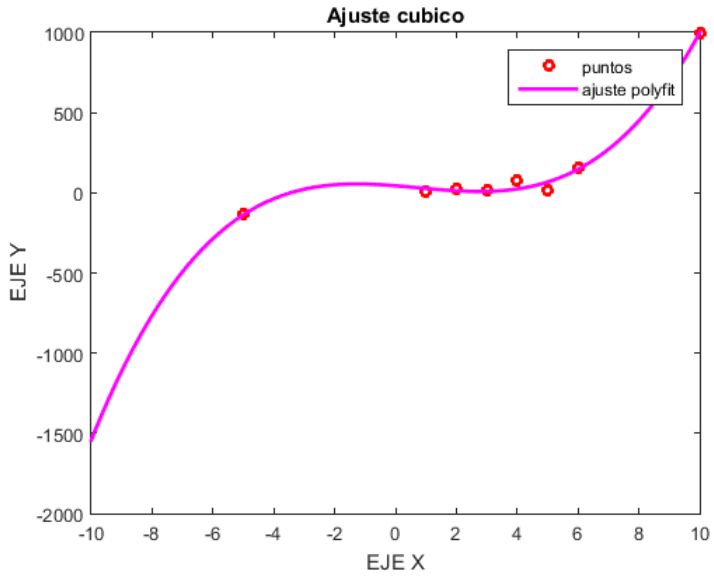

Figura 15: Ajuste cuadrático

Realizando el mismo procedimiento anterior, pero con los siguientes puntos:

$$(-5,-136),(1,7{,}3),(2,23),(3,12),(4,73),(5,12),(6,157),(10,1000) \quad (16)$$

Como muestran un comportamiento cúbico, el ajuste polinomial recomendado es el de orden 3, por lo tanto, primeramente se crean los vectores x e y con los valores de las coordenadas de los puntos a estudiar, enseguida se crea un vector p utilizando polyfit y un vector xp para representar gráficamente el polinomio. También se define un vector yp con los valores del polinomio en cada punto de xp utilizando polyval. Finalmente se grafican justos los puntos y el ajuste de orden 3 como se muestra a continuación.

```
clc
clear all
x=[-5 1 2 3 4 5 6 10 ];
y=[-136 7.3 23 12 73 12 157 1000 ];
p=polyfit(x,y,3);
xp=-10:0.1:10;
yp= polyval(p,xp);
clf
plot(x,y,'or', xp,yp,'m','linewidt',2,'MarkerSize',5)
title('Ajuste cúbico')
xlabel('EJE X')
ylabel('EJE Y')
legend('puntos','ajuste polyfit')
```

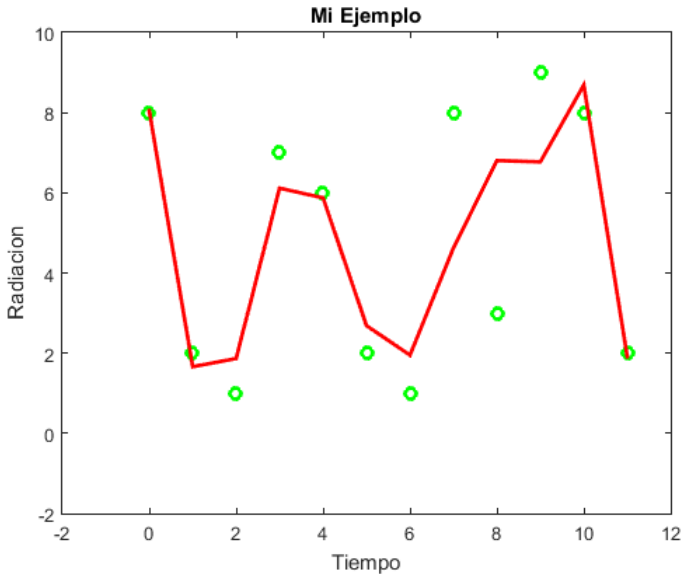

Figura 16: Ajuste cubico

Finalmente, aquí tenemos otro ejemplo haciendo uso de las funciones polyfit y polyval. Las variables x e y son un arreglo de vectores, representando el tiempo y algún otro valor, en este caso radiación. Polyfit encuentra los coeficientes de un polinomio $P(x)$ de grano N que mejor se ajusta a los datos usando mínimos cuadrados. polyval regresa los valores de un polinomio evaluado en X en forma de un vector de longitud $N + 1$. El siguiente programa muestra esta situación:

```
clear all
x=[0:11]; %Tiempo
y=[8 2 1 7 6 2 1 8 3 9 8 2]; %Radiacion
coefs = polyfit(x,y,8);
curva = polyval(coefs,x);
plot(x,y,'go',x,curva,'r','Linewidth',2);
xlabel('Tiempo')
ylabel('Radiacion')
title('Mi Ejemplo');
axis([-2 12 -2 10])
```

La gráfica del ajuste y los puntos de estudio se muestra a continuación:

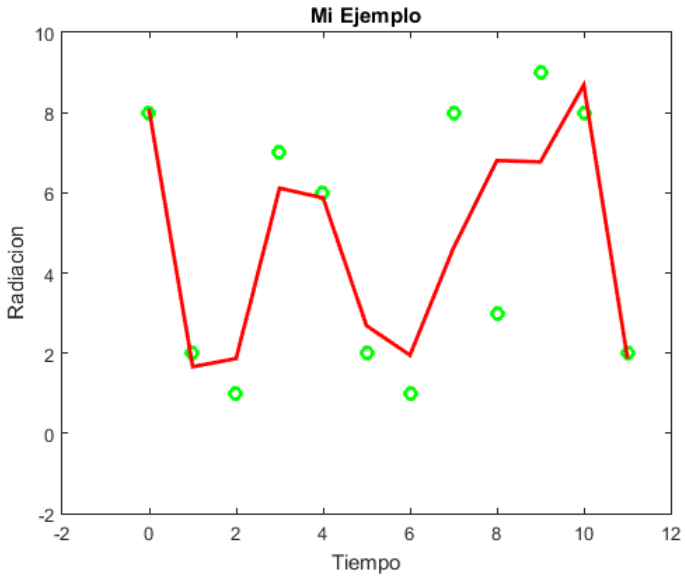

Figura 17: Ajuste cubico

7. Rutina roots

7.1. Introducción

Esta rutina nos devuelve las raíces o soluciones de un polinomio de la forma,

$$C_n x^n + C_{n-1} x^{n-1} + C_{n-2} x^{n-2} + ... + C1x + C_0 \tag{17}$$

Éste polinomio debe ser de una variable simple y sin exponentes negativos que expresamos en un vector P, especificando únicamente los coeficientes, en el que el primer termino será el coeficiente de exoponente 'n' (x^n) y los siguientes $n-1$, $n-2$...

$$P = [C_n \;\; C_{n-1} \;\; C_{n-2} \;\; ... \;\; C_1 \;\; C_0] \tag{18}$$

Con $roots(p)$ la rutina nos devuelve las raíces (r) del polinomio.[8]

```
>> roots (P)
      => raiz 1
```

```
=> raiz 2
   .
   .
   .
=> raiz n-1
=> raiz n
```

7.2. Ejemplo

En este ejemplo el polinomio que se utiliza es $3x^2 - 2x - 4 = 0$, para el cual las raíces son

$$r_1 = 1{,}5352$$

$$r_2 = -0{,}8685$$

Que se pueden observar en la figura (18)

7.2.1. Código

Aquí se muestra el codigo para encontrar las raices del polinomio $3x^2 - 2x - 4 = 0$

```
clear all
%Esta parte del codigo se declaran las variables "C1", "C2",
%"C3" y "C4" ,y los valores que tomara x.

c1 = 0;
c2 = 3;
c3 = -2;
c4 = -4;
x = -5:5;


%se define la función "f" con las variables de arriba.

f = c1*x.^3 + c2*x.^2 + c3*x +c4;


%Ahora hacemos con el polinomio un vector "c", y encontramos
%sus raíces "r" con el comando roots.

c = [c1 c2 c3 c4];
r = roots(c);

%graficamos el polinomio y se muestran las raíces.

plot(x,f, 'k')
hold on
plot(r,0, 'or')
grid on
ylabel('eje y')
xlabel('eje x')
```

7.2.2. Simulación

Raíces del polinomio $3x^2 - 2x - 4 = 0$

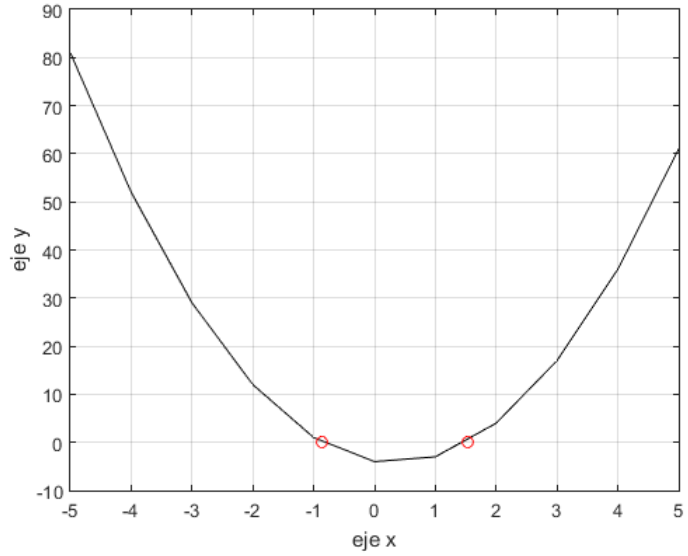

Figura 18: Gráfica del polinomio y sus raíces

8. Derivadas

8.1. Introducción

El uso de la derivada en las matemáticas no es para nada nuevo, y puede resultar en algunos casos un proceso muy tardado si se quiere obtener una aproximación numérica para un punto dado o varios de ellos. Pero gracias a la tecnología de hoy en día podemos hacer uso de un equipo de computo con el que podemos ahorrar mucho tiempo.
Existen distintos métodos en Matlab con el que podemos encontrar la aproximación para la primera derivada. Los que veremos son el método de diferencias progresivas, diferencias adelantadas, y el de diferencias centradas respectivamente.

8.1.1. Expansión de serie de Taylor

Primero es necesario definir, la expansión en series de Taylor, para una función definida en (a,b) que tiene hasta la k-ésima derivada, entonces la expansión de f(x) usando series de Taylor alrededor del punto xi contenido

en el intervalo (a,b) será:[9]

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{f''(x_i)}{2!}(x - x_i)^2 + ... + \frac{f^n(x_i)}{n!}(x - x_i)^n \quad (19)$$

8.1.2. Diferencias progresivas

Este método también conocido como método de diferencias adelantadas, y se describe utilizando la ecuación (1) con k=2 y x=a+x, entonces queda:

$$f(x_i + \Delta x) = f(x_i) + \Delta x f'(x_i) + (\Delta x)^2 \frac{f''(\epsilon p)}{2!} \quad (20)$$

Donde para este caso el error de truncamiento correspondería a :

$$O_p(x) = -(\Delta x)^2 \frac{f''(\epsilon p)}{2!} \quad (21)$$

Sí de la la ecuación (2) despejamos el termino f'(x) que necesitamos, y los términos de la igualdad los expresamos en términos de f obtenemos:[10]

$$f'(a) = \frac{f_{i+1} - f_i}{\Delta x} \quad (22)$$

8.1.3. Diferencia regresiva

También conocida como diferencia atrasada, y muy parecido a la diferencia progresiva, se describe utilizando la ecuación (1) pero ahora con k=2 y $x = x_i \Delta x$,entonces queda:

$$f(x_i - \Delta x) = f(x_i) - \Delta x f'(x_i) + (\Delta x)^2 \frac{f''(\epsilon p)}{2!} \quad (23)$$

y el error que trae consigo esta aproximación de primer orden es:

$$O_p(x) = (\Delta x)^2 \frac{f''(\epsilon p)}{2!} \quad (24)$$

Ahora si despejamos de la ecuación(3) f' y expresamos los términos de la igualdad en términos de f, como lo hicimos para la diferencia progresiva, nos queda la siguiente expresión:[10]

$$f'(x_i) = \frac{f_i - f_{i-1}}{\Delta x} \quad (25)$$

8.1.4. Diferencias centradas

Tomando nuevamente la ecuación (1) pero ahora con k=3, $x = x_i + \Delta x$ y con $x = x_i \Delta x$, esto con la finalidad de obtener las siguientes dos ecuaciones:

$$f(x_i + \Delta x) = f(x_i) + \Delta x f'(x_i) + (\Delta x)^2 \frac{f''(x_i)}{2!} + (\Delta x)^3 \frac{f''(\epsilon p)(\epsilon p)}{3!} \quad (26)$$

$$f(x_i - \Delta x) = f(x_i) - \Delta x f'(x_i) + (\Delta x)^2 \frac{f''(x_i)}{2!} - (\Delta x)^3 \frac{f''(\epsilon p)(\epsilon p)}{3!} \quad (27)$$

Restamos de la ecuación (8) la ecuación (9)y obtenemos:[10]

$$f'(x) = \frac{f_{i+1} - f_{i-1}}{2\Delta x} \quad (28)$$

Que corresponde al método de diferencias centradas que estábamos buscando

8.2. Código

A continuación se realiza la descripción de todas las partes del código, con en el cual se determina y se grafica la figura (19), la primera y segunda derivada de la función sin(x) mediante los métodos antes expuestos: diferencia progresiva, adelantada y central

8.2.1. Código principal

```
clear all
dx=pi/10;
x=0:dx:4*pi;
y=sin(x);
clf; subplot(3,1,1); plot(x,y,'k')
grid on
legend('sin(x)')
ylabel('eje y')
xlabel('eje x')
yp=deriva(x,y);
ypp=deriva(x,yp);
subplot(3,1,2); plot(x,yp,'r')
grid on
legend('yp','cos(x)')
ylabel('eje y')
xlabel('eje x')
subplot(3,1,3); plot(x,ypp,'b')
grid on
legend('ypp','-sin(x)')
ylabel('eje y')
xlabel('eje x')
```

8.2.2. Rutina para derivada

Aquí se muestra el código de la rutina que es llamada en el programa principal, y es la encargada de realizar las operaciones numéricas de la derivada, utilizando el método de diferencia progresiva para el primer punto, el método de diferencia central para los puntos desde 2 hasta n-1. Y para el último punto se utiliza el método de diferencia regresiva.

```
function yp=deriva(x,y)
%
% x es el intervalo donde realizaremos la derivación;
% y es la función que se va a derivar
%
% fp=(x,f) dará como resultado la derivada de los puntos x,
% usando el método de diferencia progresiva para el primer
% punto, el método de diferencia central para los puntos
% desde 2 hasta n-1. Y para el último punto se utiliza el
% método de diferencia regresiva.

n=length(x);
yp(1)=(y(2)-y(1))/(x(2)-x(1));

for k=2:n-1
    yp(k)=(y(k+1)-y(k-1))/(2*(x(k)-x(k-1)));
end
yp(n)=(y(n)-y(n-1))/(x(n)-x(n-1));
```

8.3. Simulación

A continuación se muestran las gráficas generadas por el programa, y corresponden a la funcion $sin(x)$, a su primera derivada (que como se puede observar, es equivalente a la funcion cos(x)) y a su segunda derivada respectivamente(que corresponde a la funcion -sin(x)).

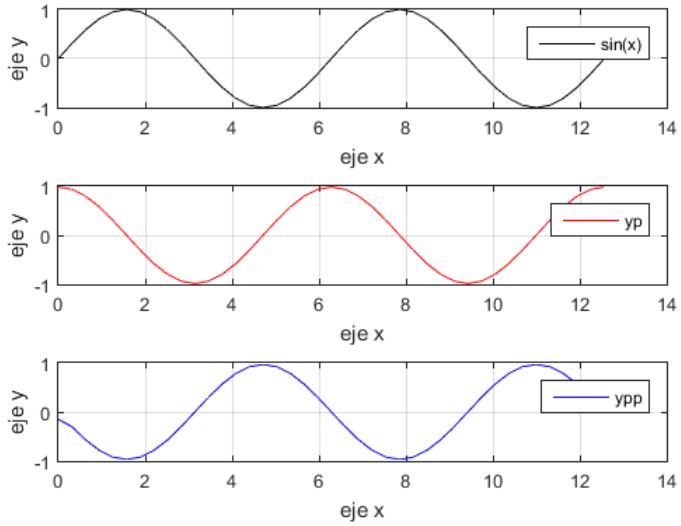

Figura 19: Gráfica generada por MATLAB para la primera y segunda derivada del sin(x)

9. Integral

9.1. Introducción

El calculo de primitivas no siempre es fácil, pude ser muy complicado o tardado, en especial cuando se trata de funciones muy largas. Pero otras veces puede resultar que simplemente nos encontramos con funciones que resultan no tener una primitiva, como en el caso por ejemplo de e^{-x^2}, $sen(x^2)$ entre otras.
Cuando se desea calcular una integral definida que contiene una función cuya primitiva no podemos hallar, entonces no se puede aplicar el teorema fundamental del cálculo y es aquí cuando se debe recurrir a una técnica de aproximación, y que mejor que utilizando una computadora que nos ahorre mucho tiempo y cálculos.

9.1.1. Regla de los rectángulos

El método más simple de este tipo es hacer a la función interpoladora ser una función constante (un polinomio de orden cero) que pasa a través del

punto (a,f(a)). Este método se llama la regla del rectángulo:[11]

$$\int_{x=a}^{x=b} f(x)dx \sim f(a)(b-a) \tag{29}$$

9.1.2. Regla de los trapecios

Este método es mas exacto que el anterior, y se puede explicar fácilmente con la siguiente figura

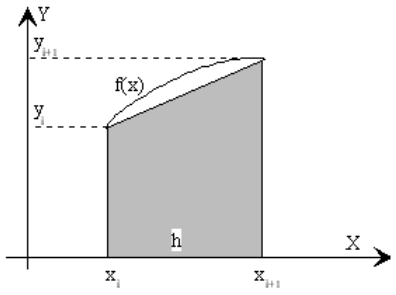

Figura 20: Regla del trapecio [13]

En la figura se puede ver que el área del trapecio corresponde a

$$\frac{h}{2}(y_i + y_{i+1}) \tag{30}$$

h corresponde a la base del trapecio, es decir la distancia $x_{i+1} - x_i$ o $h = b - a$ que son los limites de donde integraremos.
Pero para realizar una mejor aproximación no solo se toma un trapecio sino muchos trapecios pequeños, por lo que h ahora será

$$h = \frac{b-a}{n} \tag{31}$$

Donde ahora n es el numero de trapecios que se tendrá. Cabe señalar que entre mayor sea este numero mayor será la aproximación.
Por lo tanto el área total, correspondería a la suma de las áreas de los n pequeños trapecios de anchura h

$$\int_{x=a}^{x=b} f(x)dx \approx \frac{h}{2}(y_0 + y_1) + \frac{h}{2}(y_1 + y_2)) + ... + \frac{h}{2}(y_{n-2} + y_{n-1}) + \frac{h}{2}(y_{n-1} + y_n) \tag{32}$$

Agrupando términos se obtiene que

$$\int_{x=a}^{x=b} f(x)dx \approx h(\frac{y_0}{2} + y_2 + ... + y_{n-1} + \frac{n}{2} \tag{33}$$

Cuanto mayor sea el número de divisiones del intervalo [a, b] que hagamos, menor será h, y más nos aproximaremos al valor exacto de la integral. Sin embargo, no podremos disminuir h tanto como queramos, ya que el ordenador maneja números de precisión limitada.[11]

9.1.3. Regla de Simpson

Como se vio en la regla anterior, una forma de aproximar una integral definida en un intervalo [a,b] es mediante la regla del trapecio, es decir, que sobre cada subintervalo en el que se divide [a,b] se aproxima f por un polinomio de primer grado, para luego calcular la integral como suma de las áreas de los trapecios formados en esos subintervalos . El método utilizado para la regla de Simpson sigue la misma filosofía, pero aproximando los subintervalos de f mediante polinomios de segundo grado

En este procedimiento, se toma el intervalo de anchura 2h, comprendido entre x_i y x_{i+2} y se sustituye la función f(x) por la parábola que pasa por tres puntos $(x_i, y_i), (x_{i+1}, y_{i+1}) y (x_{i+2}, y_{i+2})$

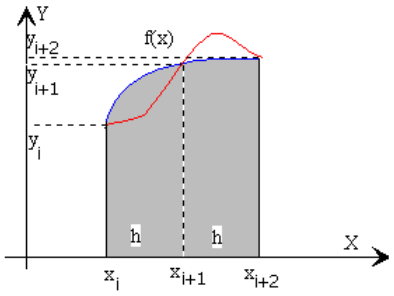

Figura 21: Regla de Simpson [12]

Calculamos la contribución a la integral del primer intervalo $(x_0, x_0 + 2h)$ y generalizaremos al resto de los intervalos. La ecuación de la parábola $y = ax^2 + bx + c$ que pasa por los puntos $(x_0, y_0), (x_0 + h, y_1), (x_0 + 2h, y_2)$ es

$$y_0 = ax_0^2 + bx_0 + c \tag{34}$$

$$y_1 = a(x_0 + h)^2 + b(x_0 + h) + c \tag{35}$$

$$y_2 = a(x_0 + 2h)^2 + b(x_0 + 2h) + c \tag{36}$$

Este sistema de tres ecuaciones con tres incógnitas, se reduce a

$$y_1 = y_0 + 2ax_0h + ah^2 + bh \tag{37}$$

$$y_2 = y_0 + 4ax_0h + 4ah^2 + 2bh \tag{38}$$

$$2ax_0 + b = \frac{4y_1 - 3y_0 - y_2}{2h} \tag{39}$$

Sustituimos la curva por la porción de parábola en el intervalo $(x_0, x_0 + 2h)$. La integral aproximada vale

$$\int_{x=x_0}^{x=x_0+2h} (ax^2+bx+c)dx = \frac{a}{3}(6x_0^2+12x_0h^2+8h^3)+\frac{b}{2}(4x_0h+4h^2)+c(2h) = \tag{40}$$

$$= 2h(ax_0^2 + bx_0 + c) + 2(2ax_0 + b)h^2 + \frac{8}{3}ah^3 = \frac{h}{3}(y_0 + 4y_1 + y_2) \tag{41}$$

En general, el valor del área aproximada, en el intervalo (xi, xi+2h) sombreada en la figura, es

$$\frac{h}{3}(y_i + 4y_{i+1} + y_{i+2}) \tag{42}$$

El área aproximada en el intervalo [a, b] es

$$\int_{x=a}^{x=b} f(x)dx \approx \frac{h}{3}(y_0+4y_1+y_2)+\frac{h}{3}(y_2+4y_3+y_4)+...+\frac{h}{3}(y_{n-2}+4y_{n-1}+y_n) \tag{43}$$

o bien, agrupando términos

$$\int_{x=a}^{x=b} f(x)dx \approx \frac{h}{3}((y_0+y_n)+4(y_1+y_3+...+y_{n-1})+2(y_2+y_4+...+yn-2)) \tag{44}$$

El primer paréntesis, contiene la suma de los extremos, el segundo, la suma de los términos de índice impar, y el tercero la suma de los términos de índice par. En el método de Simpson, el número de divisiones n debe de ser par.[12]

9.2. Código

Se muestra el código en donde se incluyen las tres formas de integración numérica, con el fin de apreciar las diferencias entre ellas de la figura (22) a la figura (29) cambiando algunos valores.

```
clear all
a=0;           %limite a
b=2*pi;        %limite b
c=62;          %numero de subintervalos
dx=(b-a)/(c);  %tamaño de base de rectángulo (dx necesario
               %para que n sea impar y tener mejor
               %aproximacion)
x=a:dx:b;
y=sin(x);          %funcion a integrar
fun='sin(x)';  %misma funció que arriba,entre comillas
n=length(x);   %calcula el numero de rectangulos(es impar)
%------------%cálculo de integral por medio de rectángulos
I1=sum(y*dx);
%--------------%cálculo de integral por medio de trapecios
I2=0.5*y(1)*dx+sum(y(2:n-1)*dx)+0.5*y(n)*dx;

%------------------%metodo de simpson
f=inline(fun);
h=dx;

sumai=0;   %se establece un punto inicial para las sumatoria
sumap=0;

for i=1:2:n-1                   %suma los impares
    sumai=sumai+feval(f,h*i+a);
end
for i=2:2:n-2                   %suma los pares
    sumap=sumap+feval(f,h*i+a);
end
                                %forma del metodo de Simpson
I3=(h/3)*(feval(f,a)+4*sumai+2*sumap+feval(f,b));

%-----------------------------------------------
%%%--- salída de los datos que observa el usuario

fprintf(['El valor numérico de la integral de la funcion
desde ',num2str(a),' hasta ',num2str(b),' es: ','con n=',
num2str(n),'\n','Método de triangulos: ',num2str(I1),'\n',
'Método de trapecios: ',num2str(I1),'\n', 'Método de Simpson:
',num2str(I3),'\n'])
```

9.3. Simulación

9.3.1. sin(x)

Para la primera simulación se utilizó c=62 para la función sin(x) desde a=0 hasta 2π

```
>> integracion
El valor numérico de la integral de la funcion desde 0 hasta 6.2832 es: con n=63
Método de triangulos: -4.585e-16
Método de trapecios: -4.585e-16
Método de Simpson: -5.9521e-16
```

Figura 22: Datos generados por MATLAB para primera simulación de sin(x)

Para la segunda, tercera y cuarta simulación se utilizó función sin(x) desde a=0 hasta π pero solo se cambio el valor de c, que corresponde al numero de subintervalos. Para la segunda se utilizó c=2, para la tercera c=50 y para c=200

```
>> integracion
El valor numérico de la integral de la funcion desde 0 hasta 3.1416 es: con n=3
Método de triangulos: 1.5708
Método de trapecios: 1.5708
Método de Simpson: 2.0944
```

Figura 23: Datos generados por MATLAB para segunda simulación de sin(x)

```
>> integracion
El valor numérico de la integral de la funcion desde 0 hasta 3.1416 es: con n=51
Método de triangulos: 1.9993
Método de trapecios: 1.9993
Método de Simpson: 2
```

Figura 24: Datos generados por MATLAB para tercera simulación de sin(x)

```
El valor numérico de la integral de la funcion desde 0 hasta 3.1416 es: con n=201
Método de triangulos: 2
Método de trapecios: 2
Método de Simpson: 2
```

Figura 25: Datos generados por MATLAB para cuarta simulación de sin(x)

9.3.2. exp(x)

Para la primera simulación se utilizó c=20 para la función exp(x) desde a=0 hasta 2

```
>> integracion
El valor numérico de la integral de la funcion desde 0 hasta 2 es: con n=21
Método de triangulos: 6.8138
Método de trapecios: 6.8138
Método de Simpson: 6.3891
```

Figura 26: Datos generados por MATLAB para primera simulación de exp(x)

Para la segunda simulación se utilizó c=200 para la función sin(x) desde a=0 hasta 2

```
>> integracion
El valor numérico de la integral de la funcion desde 0 hasta 2 es: con n=201
Método de triangulos: 6.4311
Método de trapecios: 6.4311
Método de Simpson: 6.3891
```

Figura 27: Datos generados por MATLAB para segunda simulación de exp(x)

9.3.3. Polinomios

Para la primera simulación de los polinomios se utilizó la función $x^4 + 3x^3 + 1$ con n=200, a=1 y b=4

```
>> integracion
El valor numérico de la integral de la funcion desde 1 hasta 4 es: con n=201
Método de triangulos: 402.2623
Método de trapecios: 402.2623
Método de Simpson: 398.85
```

Figura 28: Datos generados por MATLAB para primera simulación de polinomios

Para la segunda simulación de los polinomios se utilizó la función $x^3 + 3x^2 + 2$ con n=200, a=1 y b=4

```
>> integracion
El valor numérico de la integral de la funcion desde 1 hasta 4 es: con n=201
Método de triangulos: 133.6512
Método de trapecios: 133.6512
Método de Simpson: 132.75
```

Figura 29: Datos generada por MATLAB para segunda simulación de polinomios

Cómo se pudo observar, se utilizó la función sin(x), exp(x) y dos polinomios, y en todos los casos el código funciono perfectamente. También se pudo observar que en algunos casos solo se modificó el valor de c(numero de subintervalos) para algunas simulaciones y se encontró que entre mayor es el valor de c, se obtiene una mayor aproximación a la integral definida entre dos puntos.
Claramente en todas las simulaciones, el método de Simpson se acerca mas al valor que teóricamente se debe obtener. Luego le sigue el método utilizando trapecios, y por último el menos exacto es el de los rectángulos.Pero al utilizar muchos subintervalos(aumentar el numero de c) Todos los resultados son muy parecidos.[3]

Referencias

[1] https://www.fisicalab.com/apartado/movimiento-parabolicocontenidos

[2] http://metodosnumericos4s3.es.tl/SERIE-DE-TAYLOR.htm.

[3] http://www.aulavirtual-exactas.dyndns.org/ANUM/document/3$_{I}nterpolaci$

[4] http://www.utm.mx/ vero0304/HCPM/20.estructuras-control.pdf

[5] https://es.mathworks.com/help/matlab/ref/polyval.html

[6] https://es.mathworks.com

[7] https://es.mathworks.com/help/matlab/ref/polyfit.html

[8] https://es.mathworks.com/help/matlab/ref/roots.html

[9] https://es.wikipedia.org/wiki/Serie$_{d}e_{T}aylor$

[10] http://disi.unal.edu.co/ lctorress/MetNum/MeNuCl05.pdf

[11] https://es.wikipedia.org/wiki/Integraci

[12] http://www.sc.ehu.es/sbweb/fisica/*numerico*/*integral*/*simpson.html*

[13] http://www.sc.ehu.es/sbweb/fisica/cursoJava/numerico/integracion/trapecio/trapecio.htm