Examen 2
Métodos Numéricos

Rafael Alejandro García Ramírez

4 de diciembre de 2023

Problema 1

Para la integral realizamos los siguientes métodos

n	Método	Resultado	Error Absoluto
2	Newton-Cotes cerrado	0.694444444444444	0.001297263884499
	Newton-Cotes abierto	0.692063492063492	0.001083688496453
	Cuadratura Gaussiana	0.692307692308649	0.000839488251296
3	Newton-Cotes cerrado	0.693750000000000	0.000602819440055
	Newton-Cotes abierto	0.692377645502646	0.000769535057300
	Cuadratura Gaussiana	0.693121693121693	0.000025487438252

El código utilizado para el problema fue el siguiente (que también viene anexo a este documento como `problema1.c`)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

double f(double x){
    return 1.0/x;
}

double newton_cotes_cerrado(double(*func)(double), double a, double b, int n){
    if (n < 1 || n > 4) {
       printf("\n---->Error: n debe ser un entero entre 1 y 5\n");
       return 0.0;
    }
    double h = (b-a)/n;
    double resultado;
    switch (n) {
        case 1:
            resultado = 0.5*h*(func(a) + func(b));
            break;
        case 2:
            resultado = (h/3.0)*(func(a) + 4.0*func(a + h) + func(b));
            break;
        case 3:
            resultado = (3.0*h/8.0)*(func(a) + 3.0*func(a + h) + 3.0*func(a + 2.0*h) + func(b));
            break;
        case 4:
            resultado =(2.0*h/45.0)*(7.0*func(a) + 32.0*func(a + h) + 12.0*func(a + 2.0*h) + 32.0*func(
    a + 3.0*h) + 7.0*func(b));
            break;
    }
    return resultado;
}

double newton_cotes_abierto(double(*func)(double), double a, double b, int n){
    if (n < 0 || n > 3) {
       printf("\n---->Error: n debe ser un entero entre 0 y 3\n");
       return 0.0;
    }
    double h = (b-a)/(n+2.0);
    double resultado;
    switch (n) {
    case 0:
        resultado = 2*h*func(a + h);
        break;
    case 1:
```

```
        resultado = (3.0*h/2.0)*(func(a + h) + func(a + 2.0*h));
        break;
    case 2:
        resultado = (4.0*h/3.0)*(2.0*func(a + h) - func(a + 2.0*h) + 2.0*func(a + 3.0*h));
        break;
    case 3:
        resultado = (5.0*h/24.0)*(11.0*func(a + h) + func(a + 2.0*h) + func(a + 3.0*h) + 11.0*func(a + 4.0*h));
        break;
    }
    return resultado;
}
double cuadratura_gaussiana(double a, double b, double(*func)(double), int n){
    if (n < 1 || n > 4) {
        printf("\n---->Error: n debe ser un entero entre 1 y 4\n");
        return 0.0;
    }
    if (n == 1){
        return 2 * func((a + b) / 2)* (b - a) / 2;
    }
    else if (n == 2){
        double coeficientes[] = {1.0, 1.0};
        double raices[] = {0.5773502692, -0.5773502692};
        double resultado = 0.0;
        double cambio_limites;
        for(int i = 0; i < 2; i++) {
            cambio_limites = ((b-a)*raices[i] + (b+a))/2;
            resultado += coeficientes[i]*func(cambio_limites);
        }
        return resultado*(b - a) / 2.0;
    }
    else if (n == 3){
        double coeficientes[] = {0.555555555555556, 0.888888888888889, 0.555555555555556};
        double raices[] = {-0.774596669241483, 0.0, 0.774596669241483};
        double resultado = 0.0;
        double cambio_limites;
        for(int i = 0; i < 3; i++){
            cambio_limites = ((b-a)*raices[i] + (b+a))/2;
            resultado += coeficientes[i]*func(cambio_limites);
        }
        return resultado*(b - a) / 2.0;
    }
    else if (n == 4){
        double coeficientes[] = {0.652145154862546, 0.652145154862546, 0.347854845137454, 0.347854845137454};
        double raices[] = {-0.339981043584856, 0.339981043584856, -0.861136311594053, 0.861136311594053};
        double resultado = 0.0;
        double cambio_limites;
        for(int i = 0; i < 4; i++){
            cambio_limites = ((b-a)*raices[i] + (b+a))/2;
            resultado += coeficientes[i]*func(cambio_limites);
        }
        return resultado*(b - a) / 2.0;
    }
    return 0.0;
}

int main(){

    printf("Problema 1\n");

    double a = 1.0;
    double b = 2.0;

    double real = log(2.0);
    double nc, na, cg;

    for (int i = 2; i <= 3; i++){
        printf("\n");
        printf("n = %d\n", i);
        nc = newton_cotes_cerrado(f, a, b, i);
        na = newton_cotes_abierto(f, a, b, i);
        cg = cuadratura_gaussiana(a, b, f, i);
        printf("Newton-Cotes cerrado: %.15lf\n", nc);
        printf("El error absoluto es: %.15lf\n", fabs(real - nc));
        printf("Newton-Cotes abierto: %.15lf\n", na);
        printf("El error absoluto es: %.15lf\n", fabs(real - na));
        printf("Cuadratura Gaussiana: %.15lf\n", cg);
```

```
        printf("El error absoluto es: %.15lf\n", fabs(real - cg));

        printf("\n");

    }


    return 0;
}
```

Problema 2

Para este problema derivamos con respecto a x y y la función que se nos proporciona, al igualarlo a cero tenemos

$$\begin{cases} \frac{\partial f(x,y)}{\partial x} = \cos x - \sin x + y = 0 \\ \frac{\partial f(x,y)}{\partial y} = \cos y - \sin x + y = 0 \end{cases}$$

Utilizando estas ecuaciones como un sistema de ecuaciones no lineales con el Método de Gradiente Conjugado No Lineal que se presenta en el siguiente código (que también viene anexo como el archivo `problema2.c`) utilizando como condición incial $x = (4\pi/3, 4\pi/3)$ para una tolerancia de $\varepsilon = 1 \times 10^{-10}$ tenemos

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PI 3.14159265358979323846

// funcion para sin(x) + sin(y) + cos(x + y)

// derivada de la funci n x cos(x) - sen(x + y)
double dfx(double x, double y){
    return cos(x) - sin(x + y);
}
//  derivada de la funci n y cos(y) - sen(x + y)
double dfy(double x, double y){
    return cos(y) - sin(x + y);
}

void gradiente_conjugado_nolineal(double *solucion, double tolerancia, int iteraciones, double (*f1)(
    double, double), double (*f2)(double, double)) {
    int iteracion = 0;
    double F[2], direccion[2], gradiente[2];
    double gradiente_anterior[2], direccion_anterior[2];
    double alpha, beta;

    while (iteraciones--) {
        F[0] = f1(solucion[0], solucion[1]);
        F[1] = f2(solucion[0], solucion[1]);
        gradiente[0] = F[0], gradiente[1] = F[1];

        if (iteracion == 0) {
            direccion[0] = -gradiente[0];
            direccion[1] = -gradiente[1];
        } else {
            beta = (gradiente[0] * gradiente[0] + gradiente[1] * gradiente[1]) /
                   (gradiente_anterior[0] * gradiente_anterior[0] + gradiente_anterior[1] *
    gradiente_anterior[1]);
            direccion[0] = -gradiente[0] + beta * direccion_anterior[0];
            direccion[1] = -gradiente[1] + beta * direccion_anterior[1];
        }
        alpha = 0.3;

        solucion[0] += alpha * direccion[0];
        solucion[1] += alpha * direccion[1];

        iteracion++;
        if (fabs(gradiente[0]) < tolerancia && fabs(gradiente[1]) < tolerancia) {
            printf("Gradiente Conjugado NoLineal: Convergencia alcanzada en %d iteraciones\n",
    iteracion);
            return;
        }
        gradiente_anterior[0] = gradiente[0], gradiente_anterior[1] = gradiente[1];
        direccion_anterior[0] = direccion[0], direccion_anterior[1] = direccion[1];
    }
    printf("Error Gradiente Conjugado NoLineal: No convergi  despu s de %d iteraciones", iteracion);
}
```

```
int main(){

    double a = (4*PI) / 3;
    double x0[2] = {a, a};
    double tolerancia = 1e-10;
    int iteraciones = 1e5;

    gradiente_conjugado_nolineal(x0, tolerancia, iteraciones, dfx, dfy);

    printf("x0 = %.15lf\n", x0[0]);
    printf("y0 = %.15lf\n", x0[1]);

    return 0;
}
```

Con este código tenemos como resultado dentro de `11 iteraciones` de $\vec{x} = (4.712389, 4.712389)$ siento este el mínimo. Las iteraciones que se realizaron fueron gracias a la naturaleza del método de Gradiente Conjugado No Lineal.

Problema 3

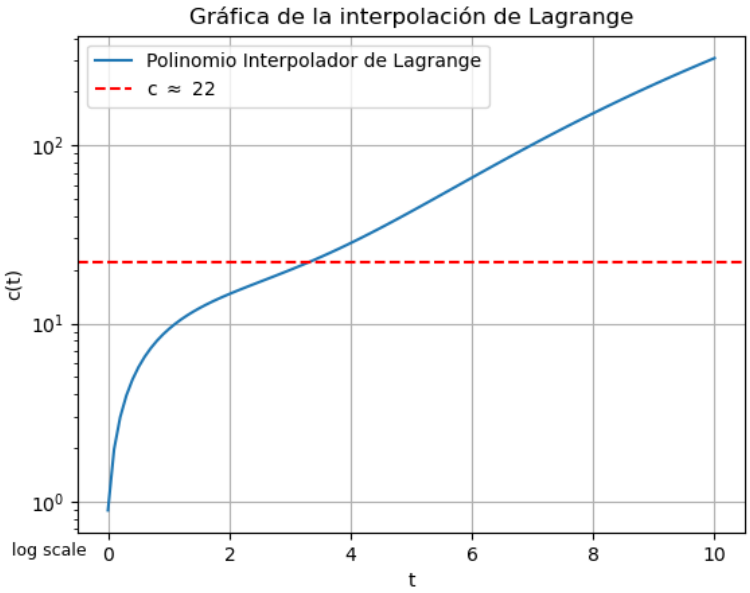
Para el siguiente problema se utilizó el siguiente código (que viene como anexo a este documento como `problema3.c`)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double interpolador_lagrange(int N, double *x, double *y, double punto){
    double L, P = 0.0;
    for (int i = 0; i < N; i++){
        L = 1;
        for (int j = 0; j < N; j++) if (i != j) L *= (punto - x[j])/(x[i] - x[j]);
        P += y[i] * L;
    }
    return P;
}

void interpolacion_cuadrados_exponencial(int N, double *x, double *y, double *alfa, double *beta,
    double regulador){
    double x1, suma_x = 0, suma_y = 0, suma_xy = 0, suma_x2 = 0;

    for (int i = 0; i < N; i++) {
        x1 = exp(-3 * x[i]);
        suma_x += x1;
        suma_y += y[i];
        suma_xy += x1 * y[i];
        suma_x2 += x1 * x1;
    }

    suma_x2 += regulador * N;
    double xx = suma_x2 - (suma_x * suma_x) / (double)N;
    double xy = suma_xy - (suma_x * suma_y) / (double)N;
    *beta = xy / xx;
    *alfa = (suma_y - (*beta) * suma_x) / (double) N;
}

int main(){
    int N = 4;
    double t[4] = {0.0, 3.0, 5.0, 7.0};
    double c[4] = {1.0, 20.0, 22.0, 23.0};
    //  calculamos con interpolaci n de lagrange t = 4
    double t0 = 4.0;
    double c0 = interpolador_lagrange(4, t, c, t0);
    printf("El valor de c(4) es: %lf\n", c0);
    // imprimimos el polinomio de interpolaci n
    printf("El polinomio de interpolaci n es:\n");
    printf("P(x) = %lf + %lf(x - %lf) + %lf(x - %lf)(x - %lf) + %lf(x - %lf)(x - %lf)(x - %lf)\n", c
    [0], (c[1] - c[0])/(t[1] - t[0]), t[0], (c[2] - c[1])/(t[2] - t[1]), t[1], t[0], (c[3] - c[2])/(t
    [3] - t[2]), t[2], t[1], t[0]);
    printf("\n\n");
    double w[] = {0, 0};
    double regulador = 0.0001;
    for (int i = 0; i < N; i++) c[i] = 1 / c[i];
    interpolacion_cuadrados_exponencial(N, t, c, &w[0], &w[1], regulador);
    printf("Parametros encontrados: alpha = %f, beta = %f\n", w[0], w[1]);

    return 0;
}
```

1. Utilizando la interpolación de Lagrange tenemos que

$$c(t=4) = \mathbf{21.528571}$$

2. La naturaleza del polinomio interpolador de Lagrange es un polinomio que carece de una estabilización, es decir, que a diferencia de $c(t)$ este jamás se saturará y tendremos que $\lim_{x\to\infty} P(x)$ diverge, esto se puede ver en la siguiente gráfica



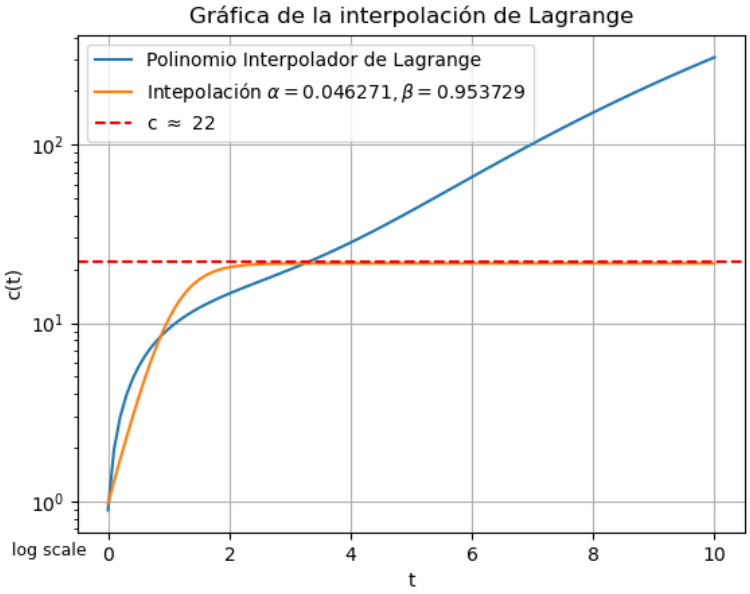
Por lo que para valores después de esta saturación (conocida o desconocida) todos los valores tenderán a un error cada vez mayor.

3. Realizando lo indicado dentro del problema, podemos encontrar los valores de las constantes que son

$$\alpha = \mathbf{0.046399}, \quad \beta = \mathbf{0.953221}$$

En este caso realizamos este código previo dado que tenemos la forma $\frac{1}{c(t)} = \alpha + \beta \exp -3t$ por lo que podemos considerar esto de forma lineal.

4. Graficando esta ecuación junto con el polinomio interpolador de Lagrange tenemos la siguiente gráfica escalada logarítmicamente.



Problema 4

1. Para el problema 4 proponemos construir un siguiente sistema de ecuaciones

$$\begin{cases} y' = \gamma \\ \gamma' = 2\gamma - y + \sin x \end{cases}$$

Con las condiciones iniciales

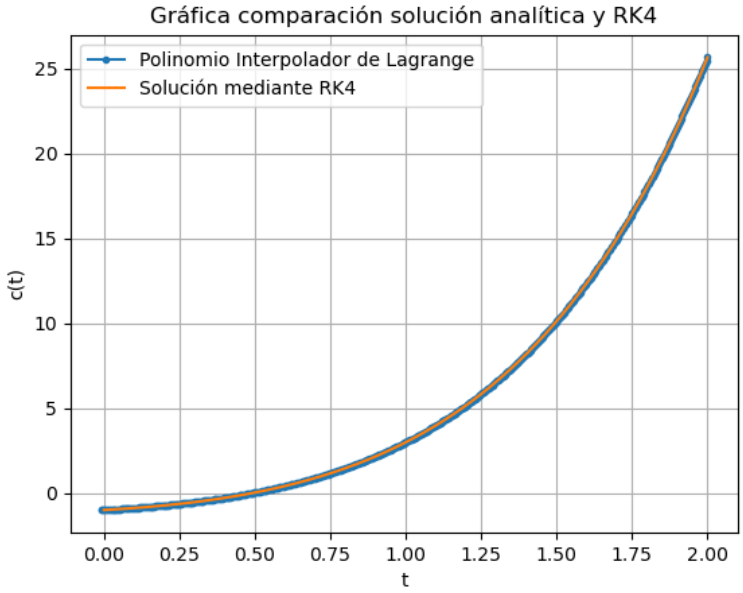
$$y(0) = -1, \quad \gamma(0) = 1$$

Para resolverlo utilizaremos el método de Runge - Kutta 4 que reescribimos e implementamos en el siguiente código (anexado a este documento como `problema4.c`)

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void f(double x, double *y, double *salida) {
    salida[0] = y[1];
    salida[1] = 2 * y[1] - y[0] + sin(x);
}
double *RK4_sistema(int N, double a, double b, double *y0, int dim, void (*f)(double, double *, double *)) {
    double h = (b - a) / N;
    double *y = (double *)malloc((N + 1) * dim * sizeof(double));
    double k1[dim], k2[dim], k3[dim], k4[dim], yTmp[dim];
    for (int j = 0; j < dim; j++) y[j] = y0[j];
    for (int i = 1; i <= N; i++) {
        f(a, &y[(i - 1) * dim], k1);
        for (int j = 0; j < dim; j++) yTmp[j] = y[(i - 1) * dim + j] + 0.5 * h * k1[j];
        f(a + 0.5 * h, yTmp, k2);
        for (int j = 0; j < dim; j++) yTmp[j] = y[(i - 1) * dim + j] + 0.5 * h * k2[j];
        f(a + 0.5 * h, yTmp, k3);
        for (int j = 0; j < dim; j++) yTmp[j] = y[(i - 1) * dim + j] + h * k3[j];
        f(a + h, yTmp, k4);
        for (int j = 0; j < dim; j++) y[i * dim + j] = y[(i - 1) * dim + j] + (h / 6.0) * (k1[j] + 2 * k2[j] + 2 * k3[j] + k4[j]);
        a += h;
    }
    return y;
}

int main(){
    double a = 0.0, b = 2.0;
    int N = 500;
    double *y0 = (double *)malloc(2 * sizeof(double));
    y0[0] = -1.0;
    y0[1] = 1.0;
    double *y = RK4_sistema(N, a, b, y0, 2, f);

    //  guardamos el resultado de x y en un archivo
    FILE *fp = fopen("problema4.txt", "w");
    for (int i = 0; i <= N; i++)
        fprintf(fp, "%lf %lf\n", a + i * (b - a) / N, y[i * 2]);
    fclose(fp);
    free(y);
    free(y0);

    return 0;
}
```

Esta elección la realizamos debido a la exactitud que podemos obtener de Runge - Kutta 4, esto también suponiendo que los datos no sean inestables o generen oscilaciones dentro de la solución.

2. Graficando la solución anterior contra la solución obtenida por esta forma tenemos



Podemos ver que la solución que obtenemos se empalma bastante bien con la solución analítica.

1. Referencias

1. Kong, Q., Siauw, T., & Bayen, A. (2020). Python programming and numerical methods: A guide for engineers and scientists. Academic Press.
2. Richard. L. Burden y J. Douglas Faires, Análisis Numérico, 7a Edición, Edito-rial Thomson Learning, 2002.
3. Samuel S M Wong, Computational Methods in Physics and Enginering, Ed. World Scientific, 3rd Edition, 1997.
4. Stewart, G. W. (2001). Matrix Algorithms: Volume II: Eigensystems. Society for Industrial and Applied Mathematics.
5. Teukolsky, S. A., Flannery, B. P., Press, W. H., & Vetterling, W. T. (1992). Numerical recipes in C. SMR, 693(1), 59-70.
6. William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd Edition, Cambridge University Press, 2007