Métodos iterativos para ecuaciones no lineales

Problem 1: Escribe un programa para calcular la constante matemática e, considerando la definición

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n ,$$

es decir, calcula $(1+1/n)^n$ para $n = 10^k$, $k = 1, 2, \ldots, 20$. Determina el error relativo y absoluto de las aproximaciones comparándolas con exp(1). **(1 punto)**

Solution.

Listing 1: Calcula el n-ésimo término del límite de e

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

/*
returns the nth term of the sequence (1+1/n)^n
 */
long double approx_exp(unsigned long int n) { return powl(1.0 + 1.0 / n,
    n); }

/*
prints the absolute and relative errors to the file inserted
 */
void print_relative_error(FILE *file, double *xn, double x, int dim) {
  fprintf(file, "Absolute Error, Relative Error\n");
  for (int i = 0; i < dim; i++) {
    fprintf(file, "%.15e, %.15e\n", fabs(*xn - x), fabs((*xn - x) / (*xn)
        ));
    xn++;
  }
}
int main() {
  unsigned long long int term = 10;
  short unsigned int pow = 1;       // we use powers of 10, that is, 10^k
      for 1\leq k \leq 20
  double e_n[20];     // storage of e_n = (1+1/n)^n
  double e = exp(1); // value of e^1 given by the math.h library

  while (pow <= 20) {
    e_n[pow - 1] = approx_exp(term);
    printf("e_{10^{%d}} = %.15lf\n", pow , e_n[pow - 1]); // prints the
        value of e_n

    term *= 10;
    pow++;
```

```
  }
  FILE *file;
  file = fopen("errors.csv",
               "w+"); // the file errors.csv will contain the absolute
                   and
                       // relative errors of the sequence and the value e
                           ^1 given
                       // by the math.h library
  print_relative_error(file, e_n, e, pow - 1);
}
```

Este nos genera la list

$$
\begin{aligned}
e_{10^1} &= 2.593742460100002 \\
e_{10^2} &= 2.704813829421528 \\
e_{10^3} &= 2.716923932235594 \\
e_{10^4} &= 2.718145926824926 \\
e_{10^5} &= 2.718268237192297 \\
e_{10^6} &= 2.718280469095753 \\
e_{10^7} &= 2.718281694132082 \\
e_{10^8} &= 2.718281798347358 \\
e_{10^9} &= 2.718282052011560 \\
e_{10^{10}} &= 2.718282053234788 \\
e_{10^{11}} &= 2.718282053357110 \\
e_{10^{12}} &= 2.718523496037238 \\
e_{10^{13}} &= 2.716110034086901 \\
e_{10^{14}} &= 2.716110034087023 \\
e_{10^{15}} &= 3.035035206549262 \\
e_{10^{16}} &= 1.000000000000000 \\
e_{10^{17}} &= 1.000000000000000 \\
e_{10^{18}} &= 1.000000000000000 \\
e_{10^{19}} &= 1.000000000000000 \\
e_{10^{20}} &= 1.000000000000000
\end{aligned}
$$

Notemos que en el término 15, la computadora deja de producir un valor prudente. Esto se debe a que un doble tiene precision de 15 digitos. Para este punto, $1/10^{15}$ tiene el decimal en el último punto de precisión. Entonces, al elevar a este mismo término, quedamos con algo que será impreciso. Para el término 16, $1/10^{16}$ ya es redondeado a 0 por el sistema. Entonces, solamente nos queda $1^{10^k} = 1$. ◇

Problem 2: La ecuación $x^3+x=6$ tiene una raíz en el intervalo $[1.55, 1.75]$, ¿cuántas iteracions se necesitan para obtener una aproximación de la raiz con error menor a 0.0001 con el método de bisección? Verifica con el método de bisección tu predicción de la raíz. **(2 puntos)**

Solution.

Listing 2: Programa que encuentra la raiz de un polinomio en un intervalo

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

double f(double x) {
  return x*x*x+x-6;
}

double bisection_method(double a, double b, double TOLERANCE, int
    MAX_ITER) {
  if (a > b) {
    return bisection_method(b, a, TOLERANCE, MAX_ITER);
  }
  // if ((fa < 0 && fb < 0) || (fa > 0 && fb > 0))  {
  //   fprintf(stderr, "Failed to apply method.\n");
  //   exit(1);
  // }
  int N = 1;
  while (N <= MAX_ITER) {
    double c = (a+b)/2;
    if (f(c) == 0 || (b-a)/2<TOLERANCE) {
      printf("Root found at iter = %d\n", N);

      return c;
    }
    N++;
    if ((f(c) < 0 && f(a) < 0) || (f(c) > 0 && f(a) > 0)) {
      a = c;
    } else{
      b = c;
    }
  }
  fprintf(stderr, "Method failed. MAX_ITER exceeded.\n");
  exit(1);
}

int main() {
  printf("root at x = %lf\n", bisection_method(1.55, 1.75, 0.0001, 100));
}
```

◇

Problem 3: Hallar una raíz de $f(x) = x^4 + 3x^2 - 2$ por medio de las siguientes 4 formulaciones de punto fijo utilizando $p_0 = 1$:

$$\text{a) } x = \sqrt{\frac{2 - x^4}{3}}, \qquad \text{b) } x = (2 - 3x^2)^{\frac{1}{4}}, \qquad \text{c) } x = \frac{2 - x^4}{3x}, \qquad \text{d) } x = \left(\frac{2 - 3x^2}{x}\right)^{\frac{1}{3}}$$

1. Las raíces de $f(x)$ deben de coincidir con las raíces de $x - g(x)$. Grafica $f(x)$ y $x - g(x)$. Comenta lo observado. **(1 punto)**

2. Crea una tabla comparativa para comparar el resultado de las raíces de $f(x)$ con la raíz alcanzada con cada una de las formulaciones. Usa máximo 20 iteraciones y tol = 0.0001. Explica lo sucedido. **(2 puntos)**

Solution.

Listing 3: Programa que encuentra la raiz de un polinomio mediante la iteracion de punto fijo

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

double g1(float x) { return sqrt((2 - x * x * x * x) / 3); }

double g2(float x) { return pow(2 - 3 * x * x, 0.25); }

double g3(float x) { return (2 - x * x * x * x * x) / (3 * x); }

double g4(float x) {
  double d = (2 - 3*x*x);
  double t = d / x;
  if (t < 0) {
    return -pow(-t, 1.0/3);
  }
  return pow(t, 1.0/3);
}

double fixed_point_iteration(double p0, double TOLERANCE, unsigned int
    MAX_ITER,
                             double (*f)(float)) {
  unsigned int N = 1;
  while (N <= MAX_ITER) {
    double p = f(p0);
    printf("p_%d = %lf\n", N, p);
    if (fabs(p - p0) < TOLERANCE) {
      return p;
    }
    N++;
    p0 = p;
  }
```

```
  fprintf(stderr, "Method failed after %d iterations.\n", N);
  return 0;
}

int main() {
  printf("g1: root = %lf\n", fixed_point_iteration(1.0, 0.0001, 20, g1));
  printf("g2: root = %lf\n", fixed_point_iteration(1.0, 0.0001, 20, g2));
  printf("g3: root = %lf\n", fixed_point_iteration(1.0, 0.0001, 20, g3));
  printf("g4: root = %lf\n", fixed_point_iteration(1.0, 0.0001, 20, g4));
  return 0;
}
```

◇

Problem 4: Utiliza el método de bisección, método de Newton, método de la secante y método de la falsa posición para comparar los resultados de los siguientes problemas: Encontrar λ con una presición de 10^{-4} y $N_{iter,max} = 100$, para a ecuación de la población en términos de la tasa de natalidad λ,

$$P(\lambda) = 1,000,000e^{\lambda} + \frac{435,000}{\lambda}(e^{\lambda} - 1)$$

para $P(\lambda) = 1,564,000$ individuos por años. Usa $\lambda_0 = 0.01$. (Sugerencia: graficar $P(\lambda) - N$) **(4 puntos)**

Solution.

Listing 4: Programa que encuentra la raiz de $P(\lambda)$

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

#define TOLERANCE 0.0001
#define N_MAX_ITER 100

double newton(double p0, double (*f)(double), double (*df)(double),
    double TOL,
                unsigned int MAX_ITER) {
  unsigned int i = 1;
  while (i < MAX_ITER) {
    double p = p0 - f(p0) / df(p0);
    if (fabs(p - p0) < TOL) {
      return p;
    }
    i++;
    p0 = p;
  }
  fprintf(stderr, "Newton's method failed after N = %d iterations.\n",
          MAX_ITER);
  return 0;
```

```
}

double secant(double p0, double p1, double (*f)(double), double TOL,
              unsigned int MAX_ITER) {
  int i = 2;
  double q0 = f(p0), q1 = f(p1);
  double p = p1 - q1 * (p1 - p0) / (q1 - q0);
  while (i < MAX_ITER) {
    if (fabs(p - p0) < TOL) {
      return p;
    }
    i++;
    p0 = p1;
    q0 = q1;
    p1 = p;
    q1 = f(p);
  }
  fprintf(stderr, "Secant's method failed after N = %d iterations.\n",
          MAX_ITER);
  return 0;
}

double false_position(double p0, double p1, double (*f)(double), double
    TOL,
                      unsigned int MAX_ITER) {
  int i = 2;

  double q0 = f(p0), q1 = f(p1);
  double p = p1 - q1 * (p1 - p0) / (q1 - q0);
  while (i < MAX_ITER) {
    if (fabs(p - p0) < TOL) {
      return p;
    }
    i++;
    double q = f(p);
    if (q * q0 < 0) {
      p0 = p1;
      q0 = q1;
    }
    p1 = p;
    q1 = q;
  }

  fprintf(stderr, "False position's method failed after N = %d iterations
      .\n",
          MAX_ITER);
  return 0;
}
```

```c
double dP(double lambda) {
  double e_lambda = exp(lambda);
  return 1e6 * e_lambda +
         (435e3 / lambda * lambda) * (e_lambda * (lambda - 1) + 1);
}

double P(double lambda) {
  double e_lambda = exp(lambda);
  return -1564000 + 1e6 * e_lambda + (435e3 / lambda) * (e_lambda - 1);
}

int main() {
  double r = newton(0.01, P, dP, TOLERANCE, N_MAX_ITER);
  printf("newton found root at %lf\n", r);
  printf("P(%lf) = %lf\n", r, P(r));
  double r_sec = secant(0.01, 0.2, P, TOLERANCE, N_MAX_ITER);
  printf("secant found root at %lf\n", r_sec);
  printf("P(%lf) = %lf\n", r_sec, P(r_sec));
  double r_fp = false_position(0.01, 0.2, P, TOLERANCE, N_MAX_ITER);
  printf("root at %lf\n", r_fp);
  printf("P(%lf) = %lf\n", r_fp, P(r_fp));
  return 0;
}
```

◇