Implementación de Algoritmos para Soluciones Iterativas, búsqueda de Raíces de Funciones y obtención de Valores y Vectores Propios

Descripción

Se realizó la implementación de los siguientes algoritmos:

1. Algoritmo de Jacobi para solución de sistema lineales de ecuaciones.
2. Algoritmos de Gauss-Seidel para solución de sistema lineales de ecuaciones.
3. Algoritmo de Bisección para encontrar raíces de funciones.
4. Algoritmo de Newton-Raphson para encontrar raíces de funciones.
5. Algoritmo Método de Potencia para encontrar el valor y vector propio más grande de una matriz.

Algoritmo de Jacobi para solución de sistema lineales de ecuaciones.

El Método de Jacobi es un método iterativo para encontrar soluciones a sistemas de ecuaciones lineales de la forma $Ax = b$. Se define como sigue:

Sea una matriz cuadrada a A_{nxn}, A se puede escribir de la forma:

$$A = L + D + U \tag{1}$$

Dónde,

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{pmatrix} \tag{2}$$

$$L = \begin{pmatrix} 1 & 0 & \dots & 0 \\ a_{2,1} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \dots & 1 \end{pmatrix} \tag{3}$$

$$D = \begin{pmatrix} a_{1,1} & 0 & \dots & 0 \\ 0 & a_{2,2} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & a_{n,n} \end{pmatrix} \tag{4}$$

$$U = \begin{pmatrix} 0 & a_{1,2} & \dots & a_{1,n} \\ 0 & 0 & \dots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} \tag{5}$$

Donde D es una matriz diagonal, L es una matriz triangular inferior y U es una matriz triangular superior. Así sea

$$R = L + U \tag{6}$$

y del sistema $Ax = b$ se tiene:

$$Dx + Rx = b \tag{7}$$

$$x = D^{-1}(b - Rx) \tag{8}$$

Expresando la ecuación 3 de forma iterativa tenemos:

$$x^{(k)} = D^{-1}(b - Rx^{(k-1)}) \tag{9}$$

Desglosando la función:

$$x_i^{(k)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k-1)}), i = 1, 2, \ldots, n \tag{10}$$

Algoritmo

```
n -> Tamaño de la matrix
a[n][n] -> Matrix a
b[n] -> Vector b
xprev[n] -> Vector solucion previo
xnext[n] -> Vector solucion siguiente
MAX_ITER -> Número máximo de iteraciones
EPSILON -> Valor máximo de error esperado

xprev = b
for iter in [0, MAX_ITER]:
    error = 0.0
    for i in [0, n]:
        xnext[i] = b[i]
        for j in [0, n]:
            if i == j: continue
            xnext[i] -= a[i][j] * xprev[j]
        xnext[i] /= a[i][i]
        error += (xnext[i] - xprev[i]) ^ 2 / xnext[i] ^ 2
    error = sqrt(error)
    xprev = xnext
    if error <= EPSILON:
        break

xnext <- Contiene la solución
```

Ejemplo de prueba

Entrada

Matriz A:

```
3 3
4.000000 -1.000000 0.000000
-1.000000 4.000000 -1.000000
0.000000 -1.000000 4.000000
```

Matriz B:

```
2.0000000000
6.0000000000
2.0000000000
```

Salida

```
x_0: 1.0000004768
x_1: 2.0000002384
x_2: 1.0000004768
```

Observaciones y mejoras

- Debido a la división en la fórmula podemos tener problemas si existe un cero sobre la diagonal, así que antes de empezar a implementar el algoritmo podemos realizar un pivoteo sobre las filas para evitar la existencia de ceros sobre la diagonal.
- En las primera iteraciones usando el como entrada la matriz *M_BIG.txt* hay un overflow sobre el cálculo del error debido a que se divide sobre números muy pequeños, pero en las iteraciones sucesivas el error se corrige al aproximarse los valores de los vectores $xprev$ y $xnext$.

Algoritmos de Gauss-Seidel para solución de sistema lineales de ecuaciones.

El Método de Gauss-Seidel es un método iterativo para encontrar soluciones a sistemas de ecuaciones lineales de la forma $Ax = b$.

Se define como sigue:
Sea una matriz cuadrada a A_{nxn}, A se puede escribir de la forma:

$$A = L + D + U \tag{11}$$

Dónde,

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{pmatrix} \tag{12}$$

$$L = \begin{pmatrix} 1 & 0 & \dots & 0 \\ a_{2,1} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \dots & 1 \end{pmatrix} \tag{13}$$

$$D = \begin{pmatrix} a_{1,1} & 0 & \dots & 0 \\ 0 & a_{2,2} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & a_{n,n} \end{pmatrix} \tag{14}$$

$$U = \begin{pmatrix} 0 & a_{1,2} & \dots & a_{1,n} \\ 0 & 0 & \dots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} \tag{15}$$

Donde D es una matriz diagonal, L es una matriz triangular inferior y U es una matriz triangular superior. Así sea

$$R = L + D \tag{16}$$

y del sistema $Ax = b$ se tiene:

$$Rx = b - Ux \tag{17}$$

$$x = R^{-1}(b - Ux) \tag{18}$$

Expresando la ecuación 3 de forma iterativa tenemos:

$$x^{(k)} = R^{-1}(b - Ux^{(k-1)}) \tag{19}$$

Desglosando la función:

$$x_i^{(k)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)}), i = 1, 2, \dots, n \tag{20}$$

Algoritmo

Observando la fórmula de arriba podemos notar que solo se necesitan los x_i previos a la iteración i actual, así que podemos usar solo vector para almacenarlos.

```
n -> Tamaño de la matrix
a[n][n] -> Matrix a
b[n] -> Vector b
xnext[n] -> Vector solucion siguiente
MAX_ITER -> Número máximo de iteraciones
EPSILON -> Valor máximo de error esperado

xnext = b
for iter in [0, MAX_ITER]:
    error = 0.0
    for i in [0, n]:
        err_prev = xnext[i]
        xnext[i] = b[i]
        for j in [0, n]:
            if i == j: continue
            xnext[i] -= a[i][j] * xnext[j]
        xnext[i] /= a[i][i]
        error += (xnext[i] - err_prev[i]) ^ 2 / xnext[i] ^ 2
    error = sqrt(error)
    if error <= EPSILON:
        break
```

```
22
23 xnext <- Contiene la solución
```

Ejemplo de prueba

Entrada

Matriz A:

```
1 3 3
2 4.000000 -1.000000 0.000000
3 -1.000000 4.000000 -1.000000
4 0.000000 -1.000000 4.000000
```

Matriz B:

```
1 2.0000000000
2 6.0000000000
3 2.0000000000
```

Salida

```
1 x_0: 1.0000000596
2 x_1: 2.0000000298
3 x_2: 1.0000000075
```

Observaciones y mejoras

- Debido a la división en la fórmula podemos tener problemas si existe un cero sobre la diagonal, así que antes de empezar a implementar el algoritmo podemos realizar un pivoteo sobre las filas para evitar la existencia de ceros sobre la diagonal.
- Con la entrada la matriz *M_BIG.txt* no hubo error un overflow sobre el cálculo del error y convergió más rápido.

Algoritmo de Bisección para encontrar raíces de funciones.

Es un método basado en el Teorema del Valor Intermedio, el cual nos indica que para cualquier función continua f en el intervalo $[a, b]$ f toma todos los valores que hay entre $f(a)$ y $f(b)$. Así bajo este teorema fácilmente podemos notar que si $f(a) > 0$ y $f(b) < 0$ existe un $f(c) = 0$ con $c \in [a, b]$.

Algoritmo

```
1 x_min -> Valor de rango de búsqueda inicial
2 x_max -> Valor de rango de búsqueda final
3 EPSILON -> Valor de error máximo
4 f() -> Función de evaluación
5
6 error = EPSILON + 1
7 while error > EPSILON:
8     x_middle = (x_min + x_max) / 2.0;
```

```
    if f(x_middle) == 0:
        break
    if sign(f(x_middle)) == sign(f(x_min)):
        x_min = x_middle
    else if sign(f(x_middle)) == sign(f(x_max)):
        x_max = x_middle
    error = abs(x_min - x_max)

x_middle <- Contiene el valor aproximado
```

Cabe destacar los siguientes puntos:

- *x_min* y *x_max* deber ser forzosamente tal que el signo de $f(x\_min)$ es diferente del signo de $f(x\_max)$.
- $x\_min < x\_max$

Ejemplo de prueba

Entrada

```
Choose a function:
        0) Quit
        1) f(x) = x^2
        2) f(x) = x^2 - 2
        3) f(x) = sin(x)
        4) f(x) = 1 / (x^2)
        5) f(x) = x^3 + 3x^2
        6) f(x) = (x-3)^2 - 2
Option: 6
Search range: -3 4.3
```

Salida

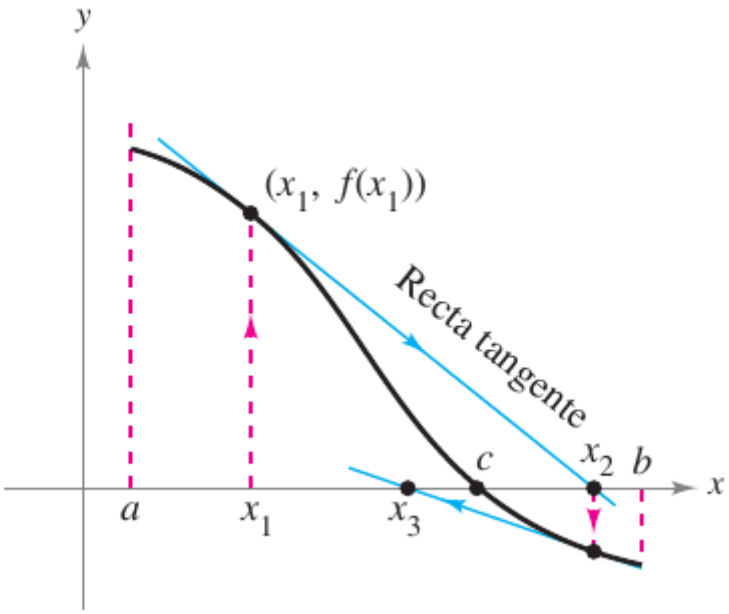
```
Estimated zero at: 1.585786
```

Observaciones y mejoras

- Se implementaron las validaciones correspondientes al algoritmo para cubrir los puntos mencionados arriba.
- Por tanto estamos atados a conocer muy bien la función y el intervalo a evaluar.
- Si la función contiene más de un cero en el rango dado solo se obtendrá uno de ellos.

Algoritmo de Newton-Raphson para encontrar raíces de funciones.

Es un método basado en el uso de la pendiente para acercarse a un cero de la función. Aproxima el siguiente punto a evaluar usando la recta punto pendiente del punto actual.



El siguiente punto está dado por

$$x_{i+i} = x_0 - \frac{f(x_o)}{f'_x} \tag{21}$$

Algoritmo

```
x -> Valor de rango de búsqueda inicial
EPSILON -> Valor de error máximo
f() -> Función de evaluación
fp() -> Derivada de f
MAX_ITER -> Número máximo de iteraciones

while MAX_ITER--:
    x_next = x - f(x) / fp(f, x);
    if f(x_middle) == 0:
        break

    if x - x_next == 0:
        break;
    x = x_next;

x_next <- Contiene el valor aproximado
```

Cabe destacar los siguientes puntos:

- La convergencia no está garantizada para todas las funciones.
- La implementación de la derivada de la función f se puede realizar mediante una aproximación usando la definición de la derivada.

Ejemplo de prueba

Entrada

```
Choose a function:
        0) Quit
        1) f(x) = x^2
        2) f(x) = x^2 - 2
        3) f(x) = sin(x)
        4) f(x) = 1 / (x^2)
        5) f(x) = x^3 + 3x^2
        6) f(x) = (x-3)^2 - 2
Option: 6
Search range: -3 4.3
```

Salida

```
Estimated zero at: 1.585786
```

Observaciones y mejoras

- Dado que la convergencia no está garantizada para todas las funciones hay que tener cuidado en su uso.

Algoritmo Método de Potencia para encontrar el valor y vector propio más grande de una matriz.

El Método de la Potencia sirve para calcular el valor propio más grande de una matriz . Sólo es válido (converge) cuando dicho valor propio de módulo más grande es real y es simple, o, en caso de ser múltiple, tiene asociados tantos vectores propios independientes como indique su multiplicidad.

Sea A una matriz de tamaño nxn, luego se toma cualquier vector x_0 y en cada paso k se calcula:

$$x_{k+1} = \frac{Ax_k}{||Ax_k||} \tag{22}$$

Dónde x_{k+1} es la convergencia al vector propio, cuyo valor propio está dado por el cociente del producto punto de los vectores.

$$\lambda = \frac{x_{k+1} . x_{k+1}}{x_{k+1} . x_k} \tag{23}$$

Algoritmo

```
n -> Tamaño de la matriz
a[n][n] -> Matriz a
xprev -> Vector de tamaño n inicializado con valores aleatoreos
xnext -> Vector de tamaño n
MAX_ITER -> Número máximo de iteraciones
lambda -> Valor propio

normalizar(xprev)

while MAX_ITER --:
    xnext = a * nprev
```

```
    lambda = xnext * xnext / xnext * xprev
    normalizar(xnext)
    xprev = xnext

xnext <- Contiene el vector propio
lambda <- Contiene valor propio
```

Ejemplo de prueba

Entrada

```
4 4
6.0000000000 -1.0000000000 -1.0000000000 4.0000000000
1.0000000000 -10.0000000000 2.0000000000 -1.0000000000
3.0000000000 -2.0000000000 8.0000000000 -1.0000000000
1.0000000000 1.0000000000 1.0000000000 -5.0000000000
```

Salida

```
Eigen Value: -9.435770
Eigen Vector:
0.13406326835450824153 0.95326676624072981259 0.07129840114863986167
-0.26120116855155278701
```

Observaciones y mejoras

Durante las pruebas se obtuvieron los resultados esperados excepto para la matriz contenida en el archivo *M_BIG.txt*. A pesar de obtener el valor propio correcto no se obtuvo la convergencia del vector propio. La convergencia hacia el valor propio fue rápida. hecho que le atribuyo a los pequeños valores de la matriz de entrada.