Actividades 8

Prácticas de Cálculo Numérico I (doble grado)

26 de abril de 2022

1. Iteraciones de punto fijo

Antes de introducir de los métodos de Jacobi y Gauss-Seidel, ya habíamos visto un ejemplo de iteraciones de punto fijo para resolver ecuaciones no lineales. Concretamente, la recurrencia $x_{n+1} = \frac{1}{2}(x_n + 2/x_n)$, uno de los algoritmos más antiguos de la historia, que, como muestra el siguiente programa, da una aproximación muy rápida a $\sqrt{2}$.

```
format long
N = 4;
x = 1;
for k = 1:N
        x = (x+2/x)/2;
end
disp(x)
err = abs(x-sqrt(2));
disp(['Error = ' num2str(err)])
```

Con `N = 5` ya tenemos un error comparable al épsilon máquina.

Según la teoría, la idea es que $x_{n+1} = g(x_n)$ genera una sucesión $\{x_n\}_{n=0}^{\infty}$ que tiende a un cero α de $f(x) = g(x) - x$ siempre que x_0 esté suficientemente cerca de α y g, supuesta regular, cumpla $\left|g'(\alpha)\right| < 1$.

Por ejemplo, la elección más tonta $f(x) = x^2 - 2$ no es adecuada para aproximar $\sqrt{2}$ porque $g(x) = x^2 - 2 + x$ cumple $g(\sqrt{2}) > 1$. El siguiente código ilustra esta situación. A pesar de que el punto de partida está cerca de $\sqrt{2}$, no se observa ninguna convergencia, ni incrementando el valor de `N`.

```
N = 10;
x = 1.4;
for k = 1:N
        x = x^2 - 2 + x
end
```

Imitando, lo hecho con los sistemas lineales, introducimos una constante $c \neq 0$ y consideramos $f(x) = c(x^2 - 2)$. De esta forma, no se alteran los ceros y $g(x)$ pasa a ser $x + c(x^2 - 2)$. Por ejemplo con $c = -1/2$ se tiene

$\left|g'(\sqrt{2})\right| = \left|1-\sqrt{2}\right| < 1$ y la convergencia está asegurada suficientemente cerca de $\sqrt{2}$.

```
1 N = 10;
2 c = -1/2
3 x = 1;
4 for k = 1:N
5         x = c*(x^2 - 2) + x
6 end
7 disp(['Error absoluto (con signo) = ' num2str(x-sqrt(2))])
```

La elección de la constante es importante. Si cambiamos la línea 2 por `c = -5/14` obtendremos un error más pequeño con solo tres iteraciones.

ACTIVIDAD 8.1.1. *Busca una fracción de denominador menor que 20 que reemplace a* `-5/14` *para obtener menor error todavía y compruébalo experimentalmente.*

Cuanto menor sea $|g'(\alpha)|$, más rápida será la convergencia. El éxito de `c = -5/14` se debe a que $|g'(\alpha)|$ es del orden de 10^{-2} y con `c = -0.3536` pasaría a ser 10^{-4}, resultando un método todavía más rápido. Esta idea nos explica de por qué el método inicial es tan bueno. Allí $g'(x) = \frac{1}{2}(1-2/x^2)$ y por tanto $g'(\sqrt{2}) = 0$. Dicho sea de paso, aparentemente es más simple ajustar c para que $g'(\sqrt{2}) = 0$, pero eso es un espejismo en general porque si no conocemos un cero α de $f(x) = g(x) - x$, es casi seguro que no conozcamos $g'(\alpha)$.

Aunque cero sea el menor valor absoluto posible, hay diferentes categorías de ceros porque los órdenes de anulación aceleran la convergencia. Por ejemplo, $g(x) = x + \operatorname{sen} x$ cumple no solo $g'(\pi) = 0$ sino también $g''(\pi) = 0$. Eso tiene que ver con la fantástica rapidez de convergencia a π del siguiente algoritmo que se puede replicar en una antigua calculadora de bolsillo sin más que partir de 3 y pulsar sucesivamente las teclas $\boxed{+}$, $\boxed{\text{sen}}$ y $\boxed{=}$.

```
1 N = 2;
2 x = 3;
3 for k = 1:N
4         x = x + sin(x);
5 end
6 disp(x)
7 err = abs(x-pi);
8 disp(['Error = ' num2str(err)])
```

Con dos iteraciones tenemos un error al aproximar π menor que `2e-11` y con tres iteraciones se llega a algo comparable al épsilon máquina.

Una situación habitual, que ya habíamos implementado en entregas anteriores, es que nuestro objetivo no es tanto llevar a cabo un número definido de iteraciones como alcanzar un error, o bien absoluto o bien relativo, por

debajo de cierta tolerancia. En las iteraciones de punto fijo, el error está aproximado por $x_n - x_{n+1}$, ya que pensamos que x_{n+1} sirve como sustituto del valor buscado porque aproxima mejor que x_n. Con esta idea en mente, el siguiente código establece como criterio de parada que la estimación del error relativo sea menor que `tol`. Si quisiéramos usar el error absoluto como criterio de parada, bastaría eliminar `*abs(xn)` de la línea 6.

```
1  Nmax = 100;
2  tol = 1e-4;
3  x = 1;
4  for k = 1:Nmax
5          xn = x+1/2*(2-x^2);
6          if abs(xn-x)<tol*abs(xn)
7                  x = xn;
8                  break;
9          end
10         x = xn;
11 end
12 fprintf('%d iteraciones. Error = %.6e.\n',k,x-sqrt(2));
```

El uso de `fprintf` en vez de `disp`, que funciona como el `printf` de C, es simplemente para practicar con diferentes comandos.

2. Aceleración de la convergencia

Seguramente te sorprenda que ya L. Euler en el siglo XVIII inventó varios procedimientos para acelerar series de convergencia lenta. En principio eso suena imposible porque una serie es la que es y parece que solo hay un método para hacer una suma.

A modo de introducción, a menudo en cálculo numérico un algoritmo que requiere un esfuerzo computacional de N pasos u operaciones, consigue una aproximación $A(N)$ de una cantidad objetivo α de la forma $A(N) = \alpha + CN^{-p} + \dots$ con C una constante, p típicamente un entero positivo y los puntos suspensivos representando términos de orden inferior, en el sentido de que al multiplicar por N^p tienden a cero. Cuanto mayor sea p, mejor aproxima a la larga el algoritmo. La *extrapolación de Richardson* [Wik20], permite eliminar el término CN^{-p} por medio de

$$\frac{2^p A(2N) - A(N)}{2^p - 1} = \alpha + \dots$$

por tanto el primer miembro de esta expresión mejora a $A(N)$ a la larga, acelera el algoritmo.

El esquema anterior no se aplica tal cual a la iteración de punto fijo porque en ellas esperamos desarrollos exponenciales. En el caso de orden uno, sin entrar en la definición, la idea es que en cada iteración el error

se reduce multiplicándose aproximadamente por una constante C de valor absoluto menor que 1 y entonces

$$x_n = \alpha + AC^n + \dots$$

donde A es constante y los puntos suspensivos son términos de orden inferior. Definamos

$$x_n^* = x_{n-2} - \frac{(x_{n-1} - x_{n-2})^2}{x_n - 2x_{n-1} + x_{n-2}}.$$

Esto es lo que se llama *fórmula de extrapolación de Aitken* [Atk89, §2.6]. Operando un poco vemos que en $x_n^* - \alpha$ solo quedan términos de orden inferior. Además, el cálculo de x_n^* sigue requiriendo las mismas n iteraciones.

ACTIVIDAD 8.2.1. *Crea una función* `acel` *que, utilizando el comando* `circshift`*, transforme una sucesión* $\{x_n\}_{n=1}^N$ *en una sucesión acelerada* $\{x_n^*\}_{n=1}^N$ *por medio de la fórmula anterior. Define* $x_n^* = x_n$ *para* $n = 1, 2$.

Por ejemplo, la iteración de punto fijo con $g(x) = x - (x^2 - 2)/2$

```
N = 10;
x = 1;
for k = 1:N-1
        x = x-(x^2 - 2)/2;
end
disp(['Error absoluto (con signo) = ' num2str(x-sqrt(2))])
```

muestra un error `7.916e-05`. Sabiendo que el método es de orden 1, en el sentido anterior, vamos a acelerarlo con las mismas 10 iteraciones:

```
N = 10;
x = ones(N,1);
x(1) = 1;
for k = 1:N-1
        x(k+1) = x(k)-(x(k)^2 - 2)/2;
end

for k = 3:N
        disp('----------')
        disp(['Iteración ' num2str(k)])
        xa = x(k-2)-
            ↪ (x(k-1)-x(k-2))^2/(x(k)-2*x(k-1)+x(k-2));
        disp(['Error sin acelerar ' num2str(x(k)-sqrt(2))])
        disp(['Error acelerado ' num2str(xa-sqrt(2))])
end
```

El error pasa a ser ahora `-3.1169e-08`, lo cual es una mejora más que considerable, más de mil veces menor. El programa ofrece la comparativa entre los valores anteriores sin acelerar y los acelerados. La diferencia es bastante espectacular.

La fórmula anterior que transforma una sucesión en otra acelerada tiene la desventaja de que solo sirve para orden uno. Además es un poco absurdo

utilizar en cada paso los valores no acelerados x_n en lugar de las mejoras x_n^* que podemos calcular. Por ello, una forma más conveniente de la *extrapolación de Aitken*, y además válida para todos los órdenes, es [QS07, §2.4]:

$$x_{n+1} = x_n - \frac{\big(g(x_n) - x_n\big)^2}{g(g(x_n)) - 2g(x_n) + x_n}.$$

ACTIVIDAD 8.2.2. *Incorpora esta forma de la extrapolación de Aitken en la comparativa anterior. Seguramente te sea conveniente definir una* función anónima. *Si las desconoces, mira la siguiente sección.*

ACTIVIDAD 8.2.3. *Comprueba que este método también acelera el algoritmo inicial* $x_{n+1} = (x_n + 2/x_n)/2$ *para aproximar* $\sqrt{2}$*, que es de orden 2. Observarás que el error es tan pequeño que enseguida se producen problemas de división por cero.*

3. El método de la bisección

Según el algoritmo visto en la teoría, ligeramente reordenado, el método de la bisección para $f(x) = x^2 - 2$ partiendo del intervalo $[1, 2]$ se implementa como:

```
1  f = @(x) x^2 -2; % función anónima
2
3  Nmax = 10;
4  a = 1;
5  b = 2;
6
7  for k = 1:Nmax
8          c = (a+b)/2;
9          if f(a)*f(c)<0
10                 b = c;
11         else
12                 a = c;
13         end
14 end
15 disp(['Error: ' num2str((a+b)/2-sqrt(2))])
```

Aquí se ha usado una *función anónima* `f`. En vez de definir `f` en un fichero externo, escribimos `@(x)` seguido de la fórmula en términos de `x`. Como siempre en `matlab/octave`, la variable `x` puede ser matricial, aunque no sea el caso aquí.

La siguiente modificación del código anterior, muestra la elección de los intervalos sucesivos en el método de la bisección. El error en el paso n está acotado por $(b-a)2^{-n}$.

```
1  f = @(x) x^2 -2; % función anónima
```

```
2
3  Nmax = 10;
4  a = 1;
5  b = 2;
6
7  for k = 1:Nmax
8          c = (a+b)/2;
9          if f(a)*f(c)<0
10                 b = c;
11                 disp(['Izquierdo ' num2str(a) ' , '
                       ↪ num2str(b)])
12         else
13                 a = c;
14                 disp(['Derecho ' num2str(a) ' , '
                       ↪ num2str(b)])
15         end
16 end
```

En `matlab/octave`, con una sintaxis que ya aparece en las funciones anónimas, es posible pasar funciones definidas en ficheros como argumentos de otras funciones. Simplemente se precede el nombre de la función a la que se apela con `@`.

Por ejemplo, supongamos que hemos definido en `pf4.m` una función que aplica cuatro veces la iteración de punto fijo partiendo de $x_0 = 1$:

```
1  function x = pf4( f )
2          x = 1;
3          for k =1:4
4                  x = f(x);
5          end
6  end
```

y, para replicar el ejemplo inicial, queremos que la aplique a la función `f1` definida en un fichero `f1.m`

```
1  function res = f1( x )
2          res = (x+2/x)/2;
3  end
```

Si intentamos `pf4( f1 )` obtendremos un error. Esto es lógico porque, cuando llame a `f1`, `matlab/octave` no sabrá qué es `x`. Lo correcto es escribir `pf4( @f1 )`. Las funciones anónimas se comportan como si ya incorporasen `@`. De este modo, `pf4( @f1 )` es equivalente a

```
1  g = @(x) (x+2/x)/2;
2  pf4( g )
```

Si tienes una versión (¿muy?) antigua de `matlab/octave`, pasar funciones anónimas como argumento hará saltar errores raros y las dos líneas anteriores no funcionarán. Para resolverlo tendrás que sustituir `f(x)` por `feval(f,x)` dentro de la definición de `pf4`.

ACTIVIDAD 8.3.1. *Escribe una función* `bisec(f,a,b,err)` *que aplique el método de la bisección donde los argumentos son la función* `f`*, el intervalo*

inicial `[a,b]` *y el* `err` *el tamaño del intervalo a partir del cual se detiene la iteración.*

4. El método de Newton

El esquema

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

es el conocido *método de Newton* o *método de Newton-Raphson* para hallar soluciones de $f(x) = 0$. Geométricamente corresponde a aproximar cerca de x_n la función por su tangente y calcular su corte con el eje X para obtener x_{n+1}.

Una implementación `mnewton( f, df, x0, N )` con `f` la función, `df` su derivada, `x0` el valor inicial y `N` el número de iteraciones es:

```
1  function x = mnewton( f, df, x0, N )
2          x = x0;
3          for k = 1:N
4                  x = x-f(x)/df(x);
5          end
6  end
```

En el caso $f(x) = x^2 - 2$, para replicar el ejemplo inicial, lo llamaríamos con `mnewton(@newf, @newdf, 1, 4)` definiendo

```
1  function res = newf( x )
2          res = x^2-2;
3  end
```

y

```
1  function res = newdf( x )
2          res = 2*x;
3  end
```

Por la teoría, sabemos que el método de Newton converge para hallar α con $f(\alpha) = 0$ siempre que $f''(\alpha) \neq 0$ y x_0 esté suficientemente cerca de α. Fuera de estas condiciones, podemos entrar en problemas. En [SB93, §5.4] se da como ejemplo $f(x) = \arctan x$ que da una sucesión divergente para $\arctan |x_0| > 2|x_0|/(1 + x_0^2)$, lo cual se cumple cuando x_0 es grande.

Por otro lado, no hay una respuesta fácil a la pregunta de para qué valores converge el método de Newton ni a qué converge. La complicación de la situación se manifiesta si lo aplicamos en el plano complejo $\mathbb{C}$. Típicamente la frontera de los puntos en los que converge a cierto cero de la función es un fractal (un *conjunto de Julia*). Para explorar esta situaciones, introduzcamos

primero el comando `imshow` que, en nuestro uso, transforma los valores de una matriz en tonos de gris. El segundo argumento es un intervalo que indica el valor mínimo que se asocia al negro y el valor máximo que se asocia al blanco. Por ejemplo:

```
un = ones(50);
A = [0*un,1*un;2*un,3*un];
imshow(A, [0,3])
```

muestra un cuadrado dividido en otros cuatro de 50×50 píxeles con tonos de gris que van del negro al blanco.

Coloreemos con tres tonos de gris diferentes cada x_0 en el cuadrado máx $\left(|\Re x_0|, |\Im x_0|\right) < 3/2$, dependiendo de si el método de Newton aplicado a $f(x) = x^3 - 2$ converge a cada una de sus tres ceros complejos.

```
N = 400;
Nmax = 30;
z0 = 2^(1/3);
z1 = z0*exp(2*pi*i/3);
z2 = z0*exp(-2*pi*i/3);
tol = 0.01;

A = meshgrid(linspace(-3/2,3/2,N),1:N);
A = A + i*A' + eps;
for l = 1:Nmax
        A = 2*(A.^3+1)./A.^2/3;
end

A = (abs(A-z0)<tol) + 2*(abs(A-z1)<tol) + 3*(abs(A-z2)<tol);

imshow(A, [0,3])
```

El dibujo evidencia la complicación de la convergencia. Hay ternas de valores iniciales arbitrariamente próximos que dan sucesiones que convergen a ceros distintos.

ACTIVIDAD 8.4.1. *Busca información sobre el comando* `subplot` *y completa el programa anterior para que muestre adosadas las imágenes correspondientes a* $f(x) = x^3 - 2$, $f(x) = x^4 - 2$ *y* $f(x) = x^5 - 2$. *Tendrás que incrementar* `Nmax` *en las dos últimas para no ver puntos negros asociados a falta de convergencia en la tolerancia permitida.*

Hay también un método de Newton válido para el caso de varias variables reemplazando la derivada por la matriz jacobiana [QS07] [SB93].

5. El método de la secante

No siempre tenemos la suerte de contar con una expresión analítica para la función de la que queremos aproximar las raíces ya que no es inusual

que f en realidad represente la salida de un algoritmo más o menos complicado. Incluso si la tenemos, la expresión de la derivada puede ser demasiado compleja, afectando al rendimiento. La solución es aproximar $f'(x_n)$ en el método de Newton por la derivada numérica $\big(f(x_n)-f(x_{n-1})\big)/(x_n-x_{n-1})$, lo que da lugar al método de la secante:

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}.$$

Hay también una justificación geométrica, que da nombre al método y consiste reemplazar en el método de Newton tangentes por secantes que unen los puntos $\big(x_n, f(x_n)\big)$ y $\big(x_{n+1}, f(x_{n+1})\big)$.

Una posible implementación para $f(x) = x^2 - 2$ es:

```
Nmax = 4;
x1 = 1;
x2 = 2;
for k = 1:Nmax
        x3 = x2-(x2^2-2)*(x2-x1)/(x2^2-x1^2);
        x1 = x2;
        x2 = x3;
end
disp(['Error secante = ',num2str(x3-sqrt(2))]);
```

El rendimiento es algo peor que el del método de Newton:

```
Nmax = 4;
x1 = 1;
x2 = 2;
x = x1;
for k = 1:Nmax
        x3 = x2-(x2^2-2)*(x2-x1)/(x2^2-x1^2);
        x1 = x2;
        x2 = x3;
        x = x/2+1/x;
end
fprintf('Error secante = %.6e.\n',x3-sqrt(2));
fprintf('Error Newton = %.6e.\n',x-sqrt(2));
```

Este código arroja como salida

```
Error secante = -2.123898e-06.
Error Newton = 1.594724e-12.
```

ACTIVIDAD 8.5.1. *Realiza un programa que aplique el método de la secante para $f(x) = x^2 - 2$ con $x_0 = 1$, $x_1 = 2$ y dibuje gráfica de f en $[1, 2]$ y, en diferente color, las secantes que unen $\big(x_n, f(x_n)\big)$ y $\big(x_{n+1}, f(x_{n+1})\big)$ con $n = 0, 1, 2$.*

Referencias

[Atk89] K. E. Atkinson. *An introduction to numerical analysis*. John Wiley & Sons, Inc., New York, second edition, 1989.

[QS07] A. Quarteroni and F. Saleri. *Cálculo Científico con MATLAB y Octave*. Springer, Milan, 2007.

[SB93] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*, volume 12 of *Texts in Applied Mathematics*. Springer-Verlag, New York, second edition, 1993. Translated from the German by R. Bartels, W. Gautschi and C. Witzgall.

[Wik20] Wikipedia contributors. Richardson extrapolation — Wikipedia, the free encyclopedia, 2020. [Online; accessed 1-May-2021].