TEMA 4

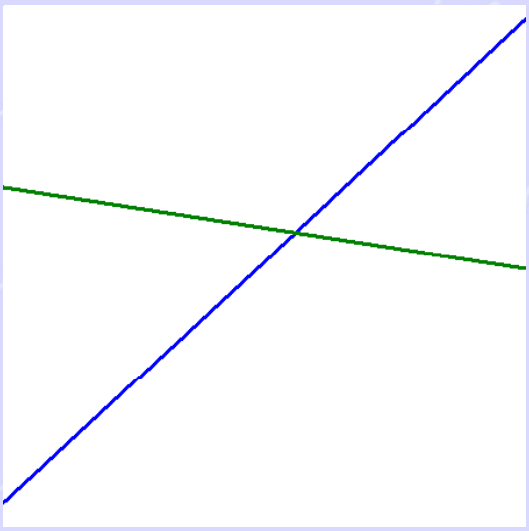
RESOLUCIÓN ECUACIONES NO LINEALES

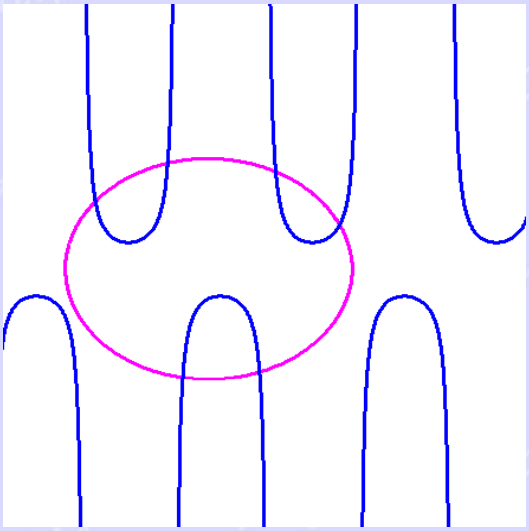
Tema 4: Resolución Ecuaciones Algebraicas.

	Lineales / No lineales	
Dimensionalidad	LINEALES	NO LINEALES
1 incógnita	$7x+3=0$	$x=\cos(x)$
Múltiples incógnitas	$3x+2y-z=1$ $2x+y+z=0$ $x-2y+3z=3$	$x=\cos(x+y)$ $x^2+2y^2=3$

~~TEMA 5: Sistemas de ECUACIONES LINEALES.~~

Ecuaciones Lineales frente a No Lineales





Ecuaciones no Lineales (1D)

Objetivo: hallar solución o soluciones de una ecuación no lineal.

Planteamiento: 1) Hallar x que verifique la ecuación x=cos(x)
2) Hallar x tal que f(x)=0:

En el 1er caso hablamos de hallar la solución de la ecuación, en el segundo de hallar el cero o la raíz de la función f(x).

TODOS los métodos que veremos se plantean de la 2ª forma: hallar s, que hace cero la función f(x), f(s) = 0.

El primer paso es siempre escribir la ecuación como f(x) = … = 0

Resolver ecuación x = cos(x) → hallar x tal que f(x) = x – cos(x) = 0

No existen fórmula generales para hallar la/s soluciones de una ecuación no lineal, ni tan siquiera para saber si existen.

Punto de Partida

¿Con qué contamos para hallar o aproximar s tal que f(s)=0?

- La función f(x) para evaluarla donde queramos:

Podemos disponer de una expresión para f(x) o considerarla una "caja negra" que recibe un número y devuelve otro (R→R)

- Una cierta idea de por dónde anda la solución en la forma de:
  - Un intervalo [a, b] que contenga la solución: $s \in [a, b]$
  - Un punto de partida $s_0 \sim s$ como solución aproximada.

¿De dónde sacamos este conocimiento previo?

Acotar la solución

En los problemas de clase (típicamente 1D) podemos acotar la solución si detectamos un cambio de signo:

- Si f(x) continua y $f(a)\cdot f(b) < 0$, → al menos 1 raíz en [a,b]

Tendríamos unicidad si podemos demostrar que la función es monótona (creciente o decreciente), esto es, la derivada f'(x) no cambia de signo → poco realista, exige hallar raíces de f'(x).

En un problema real (multidimensional) la cosa se complica (acotar la solución no es fácil en múltiples dimensiones).

En esos casos es fundamental tener información previa sobre dónde la localización de la solución o disponer de un método aproximado para obtener una primera estimación.

Métodos iterativos

Los métodos que veremos serán métodos iterativos.

Sucesión de estimaciones $\{x_n\}$ que verifiquen que $\lim_{n\to\infty} x_n = s$

PREOCUPACIONES:

- ¿Estamos seguros de la convergencia? Dependerá de x_0
- Incluso si hay convergencia, infinitos términos son muchos para calcular. Interesa una convergencia rápida para poder parar lo antes posible.
- Clasificaremos a los métodos según la evolución del error:

$$|e_n| = |x_n - s|$$

Convergencia lineal de métodos iterativos

Convergencia lineal: $\frac{|e_{n+1}|}{|e_n|} \approx K$, con K una constante <1

→ El error se reduce en un factor K (constante) en cada paso.

¿Qué efecto tiene esto en las cifras correctas de la solución?

Recordando que nº cifras decimales correctas ~ $-\log_{10}(|e|)$:

$$\log_{10}|e_{n+1}| \approx \log_{10}(K) + \log_{10}|e_n| \Rightarrow -\log_{10}|e_{n+1}| \approx -\log_{10}|e_n| - \log_{10}(K)$$

$$cifras(n+1) \approx cifras(n) + \log_{10}\left(\frac{1}{K}\right)$$

Ganamos aproximadamente $\log_{10}(1/K)$ cifras decimales en cada iteración (de forma constante).

Convergencia cuadrática

Convergencia cuadrática: $\frac{|e_{n+1}|}{|e_n|^2} \approx K$ → El error se reduce más rápidamente

$$\log_{10}|e_{n+1}| \approx \log_{10}(K) + 2\log_{10}|e_n|$$

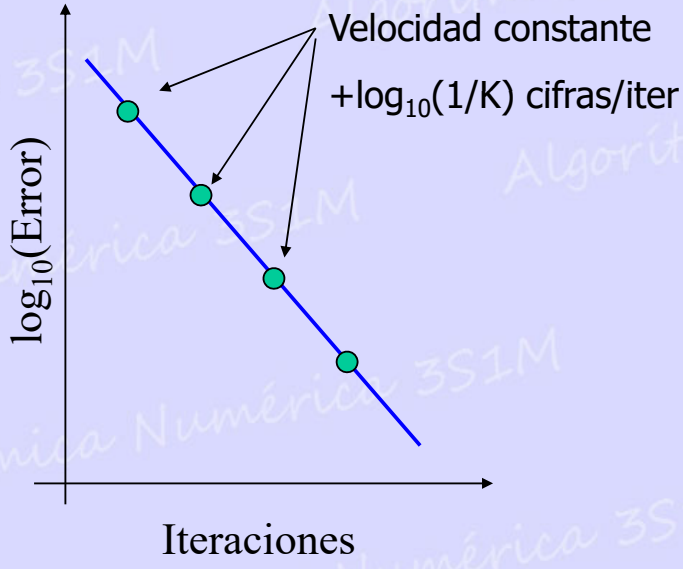
$$-\log_{10}|e_{n+1}| \approx -2 \cdot \log_{10}|e_n| - \log_{10}(K)$$

$$cifras(n+1) \approx 2 \cdot cifras(n) + \log_{10}\left(\frac{1}{K}\right)$$

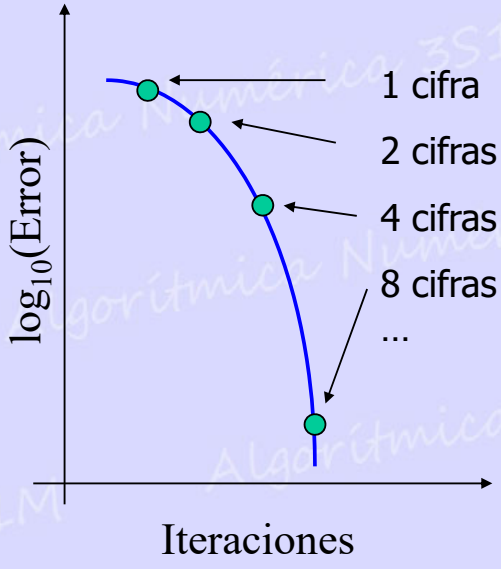
Ignorando el término en K, en cada paso se **duplican** las cifras.

- K no tiene mucha influencia: K=1 → n'=2n K=0.1 → n'=2n+1
- K afecta al error inicial permitido para tener convergencia.

Convergencia lineal



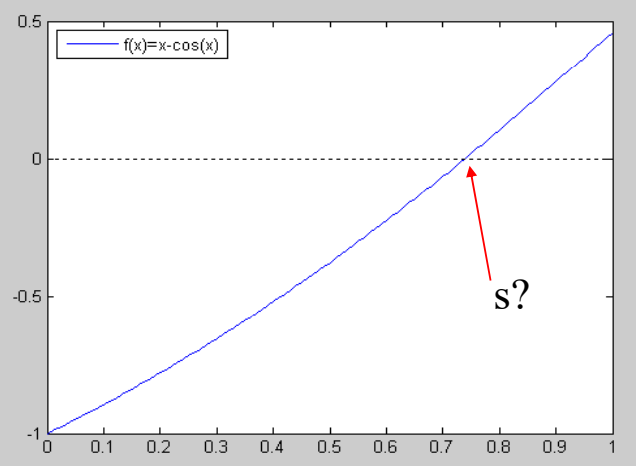
Convergencia cuadrática



Posible enfoque para f(x): R→R

Si se que la solución está en [a,b] y puedo evaluar la función f(x) en cualquier punto, ¿no podemos simplemente pintar la gráfica de f(x) y buscar gráficamente la solución?

```
% En ventana comandos o script
>> fun(0),   ans = -1
>> fun(1),   ans = 0.4597
>> x=(0:0.01:1);
>> plot(x,fun(x),[0 1],[0 0],'k');
% Funcion a resolver
function f = fun(x)
 f = x-cos(x);
return
```



¿Coste computacional?
¿Precisión alcanzada?

Coste computacional / Precisión

Coste computacional: en este tipo de métodos el número de operaciones será muy dependiente de la complejidad de f(x)

Coste computacional se mide en el nº de evaluaciones de f(x).

En este caso hemos realizado 100 evaluaciones de f(x) (cada 0.01)

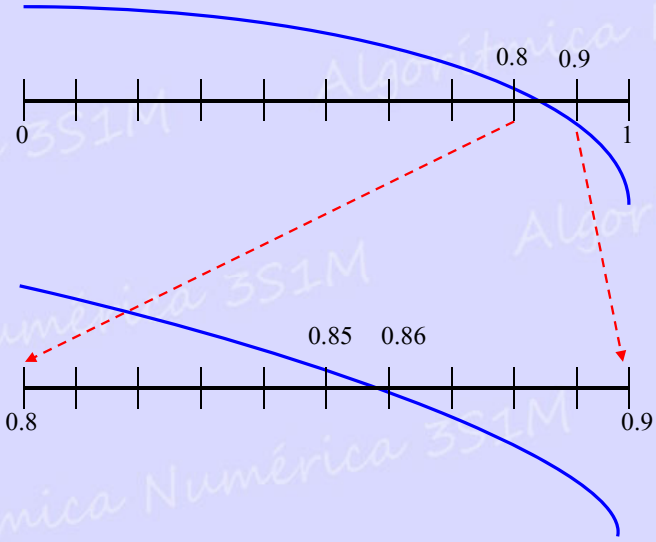
Precisión alcanzada: es del orden de $|\tilde{s} - s| \leq \frac{1}{2} \cdot 0.01$

Esto es equivalente a un par de decimales correctos.

MUY POCO EFICIENTE: para 3 decimales tendríamos que evaluar con un intervalo de 0.001 = 10x en el número de evaluaciones.

Nuestro primer método

Supongamos una raíz en [0,1]: s = 0.xxxx (conozco 1 cifra)



Evaluamos f(x) en 10 puntos:

- Solución en (0.8,0.9): s = 0.8xxx
- 10 evaluaciones → +1 cifra decimal

Repetimos para el intervalo [0.8, 0.9].

- Solución en (0.85,0.86): s = 0.85xx
- 1 cifra decimal/10 evaluaciones

Podemos optimizar este método con una estrategia de búsqueda binaria, dividiendo el intervalo por la mitad → MÉTODO BISECCIÓN

Método de la bisección

Datos de partida: intervalo [a,b] con f(a)·f(b)<0 (cambio signo)

Regla: 1) estimación de la solución = punto medio entre a y b
2) Evaluar f(x) en dicho punto medio y quedarse con la mitad del intervalo que mantenga un cambio signo.

Repetir 1) y 2).

```
Inicio:  [a, b],  con fa=fun(a), fb=fun(b) verificando que fa · fb < 0

while (condicion de parada)
 {
   s = (a+b)/2;   fs= fun(s);   % Punto  medio y evaluación

   if fs==0  break;   % TERMINAR:  hemos tenido (mucha) suerte

   if (fa * fs <0)   {  b=s;  fb=fs; }  else  { a=s; fa=fs; }
 }
```

Consideraciones sobre método de bisección

Coste computacional: 1 evaluación de f(x) por iteración

Precisión: en cada paso el intervalo se reduce a la mitad.

La cota del error en el paso n es:

$$e_n \leq \frac{1}{2^n}(b-a)$$

Intervalo inicial

Factor ½ a cada paso

Ventajas: sencillo y robusto (si hay una raíz la va a encontrar).

Desventajas: lentitud: ~ 0.3 cifras/iteración ($-\log_{10}(0.5)$) .

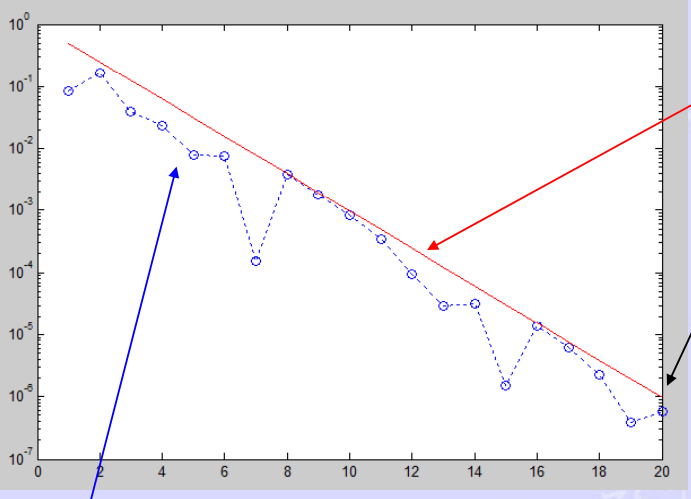
~ 50 iteraciones para precisión máquina (53 bits)

No aplicable a raíces dobles (sin cambio de signo).

Verificación cota de error

Aplicamos bisección a la función $f(x)=x^2-2$ empezando en [1,2].

Comparamos las sucesivas hipótesis con la solución $s=\sqrt{2}$



Errores reales $|x_n-s|$ en las 20 primeras iteraciones.

Cota error: $e_n \leq \frac{1}{2^n}(b-a)$

En la iteración 20, $E_{abs}\sim 10^{-6}$

~ unos 6 decimales correctos

El comportamiento general de la bisección es como si fuese un método lineal con K=1/2.

Código básico bisección

- Función MATLAB codificando la f(x) a resolver.
- Código del método propiamente dicho.

```
a=1;  b=2;              % Extremos del intervalo dado
fa=fun(a); fb=fun(b);   % Evaluación de fx en extremos intervalo

if (fa*fb>0),fprintf('ERROR: NO HAY CAMBIO SIGNO\n'); return; end

for k=1:55,  % Suficientes iteraciones
  s  = (a+b)/2; fs = fun(s); % Punto central s y evaluación f(s)
  if fs==0, break; end  % Hemos encontrado raiz
  if (fs*fa <0),  % raiz en [a,s]
      b=s; fb=fs;
  else            % raiz en [s b]
      a=s; fa=fs;
  end
end

fprintf('Mejor estimacion de s = %12.16f\n',s);
fprintf('Error es menor que: %.3e\n',(b-a)/2);
```

```
function f=fun(x)
  f = x*x-2;
end
```


Mejoras del código

- Especificar una tolerancia o precisión requerida para terminar.
- Combinar criterios de parada: precisión + máximo iteraciones
- Escribirlo como una función (bisecc.m), cuyos parámetros sean:
  a) intervalo inicial
  b) precisión requerida o número máximo de iteraciones
- ¿Y para resolver otra f(x) que no sea la codificada en fun.m?

  Opción A) si quieres rersolver otra f(x) cambia el código de fun.m
  Opción B) Más elegante: pasar a la función bisecc la función que queremos resolver como un parámetro adicional, de la misma forma que le pasamos el intervalo o la precisión

¿Cómo pasar una función como parámetro de otra función?

Paso de funciones como parámetros

Queremos pasar la función a resolver como parámetro a bisecc.m:

```
function s=bisecc(f,I)
% Entrada: f parámetro indicando la funcion a resolver
% I intervalo [a b] conteniendo raíz
% Salida: s estimación raíz
```

Debe existir por ahí la función en cuestión. Puede ser:

- Una función de MATLAB como sin(x), cos(x), etc.
- Una función guardada previamente en un fichero, p.e. fun.m.

Lo que haremos es pasar a bisecc() como argumento un puntero a la función en la que estamos interesados usando el comando @:

```
>> s = bisecc(@fun,[0 1]); % Halla raíz de fun(x) en [0 1]

>> s = bisecc(@sin,[-1 1]); % Halla raíz de sin(x) en [-1 1]
```


¿Cómo usar el parámetro f dentro de la función?

Sabemos como llamar a bisecc() pasandole un puntero a función:

```
s = bisecc(@fun1,[0 1]); % Halla raíz de fun1.m en [0 1]
```

¿Cómo usamos ese puntero dentro de bisecc()?

MUY SENCILLO: usar el nombre que habéis usado como argumento de entrada en bisecc() directamente como si fuese una función:

```
function s=bisecc(f,I)
% Entrada: f = puntero a la función a usar (p.e. @fun1)
%          I = intervalo donde buscar solucion (p.e [0,1])

a=I(1); b=I(2); % Extremo del intervalo
fa=f(a); fb=f(b); % Evaluación de f en los extremos.
...
```

Código bisección implementada como función

```
function s=bisecc(f,I,tol)
% Entrada: f puntero a función, I intervalo de raíz, tol = precisión
% Salida : s, estimación de la raíz
N=50; % Máximo número de iteraciones
if nargin==2, tol=1e-8; end
a=I(1); b=I(2); fa=f(a); fb=f(b); % Extremos de intervalo y evaluación f
if (fa*fb>0), fprintf('ERROR: sin cambio signo en I\n'); return; end

n=1; % contador de iteraciones
while ( ((b-a)/2 >tol) && (n<=N) )  % Condiciones salida
  m = (a+b)/2; % punto medio
  fm=f(m);     % Evaluación de f en punto medio m
  fprintf('%2d -> %17.16f  f(m)= %+4.2e  Cota %4.2e\n',n,m,fm,(b-a)/2);
  if (fm*fa<0), b=m;fb=fm; else a=m;fa=fm; end % Selecciono subintervalo
  n = n+1;
end
s = (a+b)/2;  % Mejor hipotesis dentro del intervalo final [a,b]
end
```

```
% Script o línea de comandos
s1=bisecc(@fun,[1,2],1e-6);
s2=bisecc(@fun2,[0,1]);
```

```
function f=fun2(x)
  f = x-cos(x);
end
```

```
function f=fun(x)
  f = x*x-2;
end
```


Iteración funcional

La bisección trabaja con un intervalo [a,b] más una regla para dividirlo.

Otros métodos parten de x_0 y aplican una función iterativa: $x_{n+1}=g(x_n)$

Estos métodos pueden construirse despejando x en la ecuación f(x)=0.

Dada $f(x) = x^3 + x^2 - 4x - 4 = 0$ si despejamos p.e la x del término en x^2 se obtiene el método:

$$x_{n+1} = g(x_n) = \sqrt{4x_n + 4 - x_n^3}$$

Si pruebo con x=2 en esta fórmula anterior obtengo $x = \sqrt{8+4-8} = 2$
La ecuación se verifica (lo que me indica que 2 es una solución).

En general para un x_0 cualquiera $x_1 = g(x_0) \neq x_0$ al no ser una solución.
Lo que estos métodos esperan es que x_1 esté más cerca de la solución y si repetimos el proceso, $x_2=g(x_1)$, … se verifique que $x_n \rightarrow s$.

Si hay convergencia se verá porque los sucesivos x_n , x_{n+1} , … cada vez cambiarán menos y se irán pareciendo cada vez más a la solución s.

Iteración funcional

Hay muchas formas de despejar una x de una función dada.

Por ejemplo, a partir de $f(x) = x^3 - x^2 - x - 1 = 0$ podemos obtener:

$$x_{n+1} = x_n^3 - x_n^2 - 1, \quad x_{n+1} = \sqrt[3]{x_n^2 + x_n + 1}, \quad x_{n+1} = \sqrt[2]{x_n^3 - x_n - 1}$$

Estos métodos construidos de forma "ingenua" no tienen garantizada su convergencia (de los tres anteriores sólo el 2º es convergente).

Cuando funcionan lo habitual es que sean lentos (convergencia lineal).

En los problemas cuando aparece un método de éstos, nos suelen pedir aplicarle varias iteraciones y a partir de su velocidad de convergencia, determinar si es lineal o cuadrático, estimar cifras por iteración, etc.

Vamos a ver como podemos diseñar un método del tipo x=g(x) con el objetivo de conseguir que sea muy rápido (convergencia cuadrática).

Método de Newton-Raphson

Problema: hallar el cero de una función f(x) no lineal ¿f(x)=0?

➔ No sabemos hacerlo

Solución: cambiar f(x) por una función que sea más sencilla r(x)
Hallar el cero de la nueva función r(x)=0

Obviamente la solución hallada no será la correcta, pero esperamos que sea una "mejor" aproximación que la hipótesis inicial. Si ese es el caso, repetimos.

¿Qué funciones sencillas podemos usar para aproximar f(x)?

→ **Su recta tangente en el punto x_n**

Método de Newton-Raphson

1. Hipótesis inicial x_0 → calculo recta tangente a f(x) en x_0:

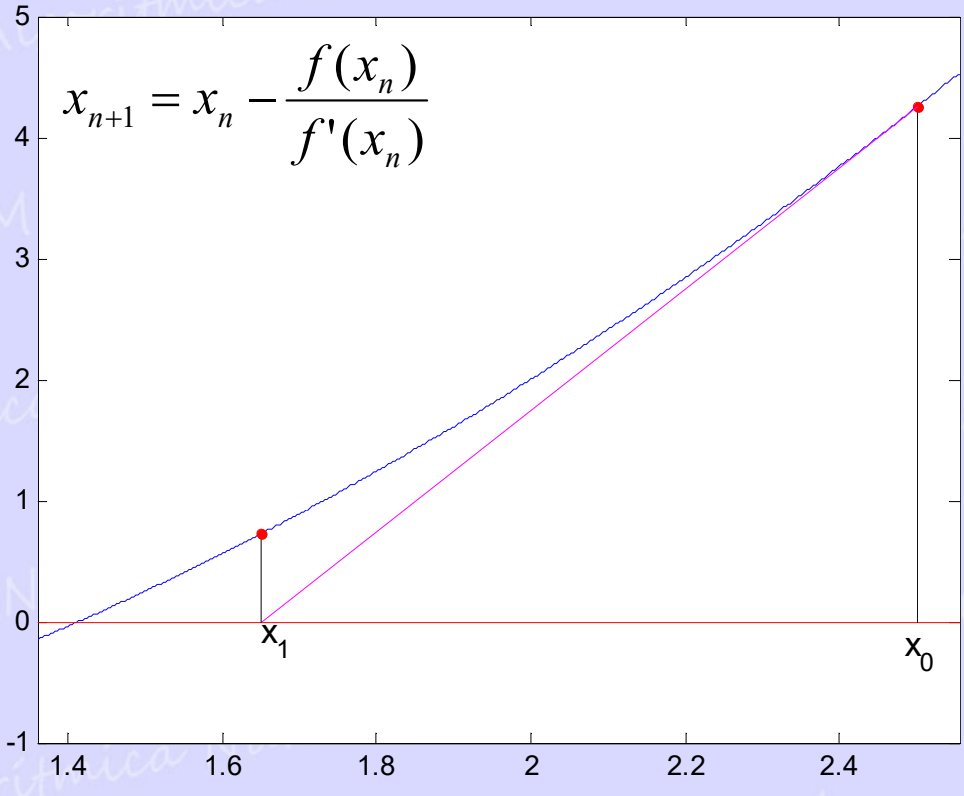
$$r(x) = f(x_0) + f'(x_0)(x - x_0)$$

2. Hallamos el cero de r(x): $r(x) = 0 \Rightarrow x = x_0 - \frac{f(x_0)}{f'(x_0)}$

3. Obviamente x ≠ s ya que x es la solución de la recta r(x) y no la de la función f(x). Nuestra esperanza es que sea una mejor aproximación a la verdadera solución que x_0

4. Repetimos el proceso desde el nuevo punto: x_0=x

Gráficamente



Ventajas y Desventajas de Newton-Raphson

Desventajas:

1. Doble evaluación de funciones en cada paso: f(x) + f'(x)
2. Implica poder calcular la derivada (no siempre posible)
3. Si f'(x)=0 en alguno de los pasos tenemos un problema.
4. No está claro que haya convergencia para un x_0 cualquiera.

Ventajas: cuando converge es MUY rápido (método cuadrático)

$$f(x) = x^2 - 2 \quad \Rightarrow \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - 2}{2x_n} = \frac{x_n}{2} + \frac{1}{x_n}$$

```
x0 = 1.5
x1 = 1.4166666666666665    e ~ 10^-3
x2 = 1.4142156862745097    e ~ 10^-6
x3 = 1.4142135623746899    e ~ 10^-12
x4 = 1.4142135623730949    e ~ 10^-16
```


Código básico de Newton en una función

```
function s=newton(f,x0,N)
% Entrada: puntero a función, hipótesis inicial, nº iteraciones
% Salida : estimación de la raíz tras N iteraciones
  s=x0; % arrancamos en x0
  for k=1:N  % N iteraciones
    [f fp]=f(s);  % Evaluación función y derivada
    if (f==0), return; end % Ya he llegado a la raíz
    s = s - f/fp; % Iteración de Newton
    fprintf('%2d -> s=%17.16f\n',k,s);
  end  % Al terminar la ultima iteracion el mejor valor está en s
end
```

```
function [f,fp]=fun(x) %Funcion a resolver: devuelve f y f'
  f = x^2 -2;    % Valor de la función en x
  fp= 2*x;       % Valor de su derivada en x
end
```

```
% En un script o en la ventana de comandos
x0=1.5; N=5; s=newton(@fun,x0,N);  fprintf('Solución = %f\n',s);
```