

Devoir maison n°4 : Méthode de Newton

Jules Charlier, Thomas Diot, Pierre Gallois, Jim Garnier
TE1

Partie A - Description de la méthode de Newton

1) D'une part on sait que la fonction f est dérivable donc continue sur $[a, b]$ et qu'elle y est strictement monotone car f' strictement négative. D'autre part, on dispose de $f(a) > 0$ et de $f(b) < 0$.

Ainsi, d'après le corollaire du Théorème des Valeurs Intermédiaires, il existe un unique $\alpha \in [a, b]$ tel que $f(\alpha) = 0$.

2) a) Soit $u \in [a, b]$. On note τ_u la tangente à la courbe représentative de f au point d'abscisse u .

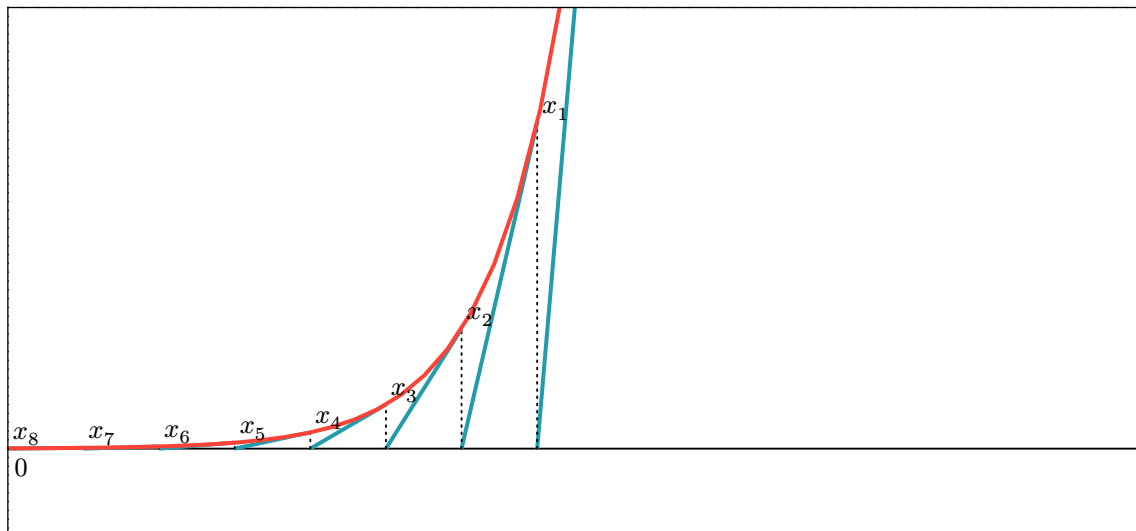
Ainsi, l'équation de τ_u est donnée par : $y = f'(u)(x - u) + f(u)$

Or $y = 0 \Leftrightarrow x = u - \frac{f(u)}{f'(u)}$.

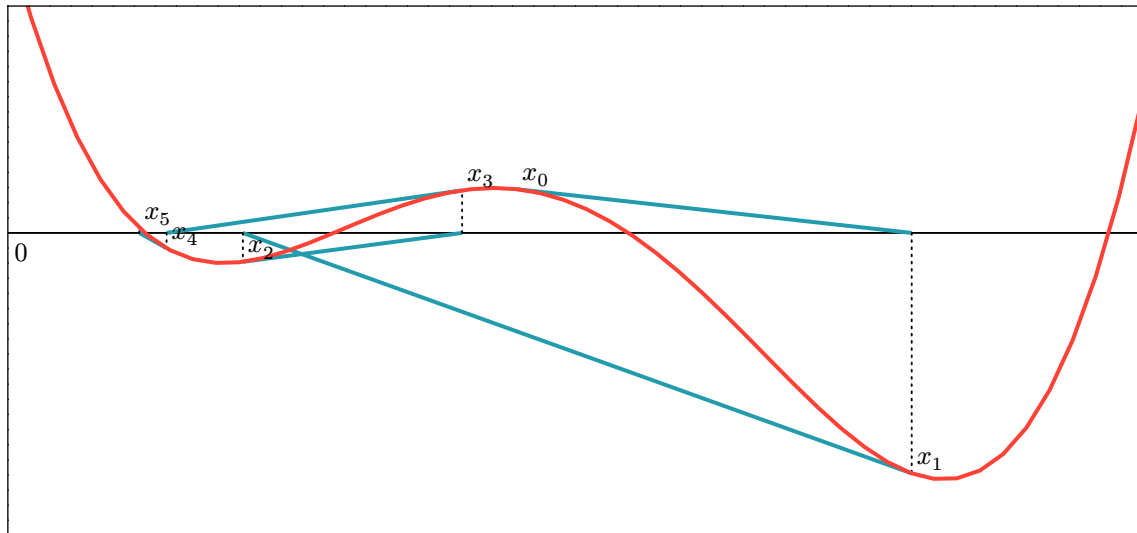
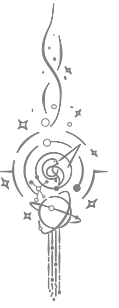
Par conséquent, τ_u coupe donc l'axe des abscisses au point d'abscisse $u - \frac{f(u)}{f'(u)}$.

b) Considérons maintenant la fonction g définie sur $[a, b]$ par $g : x \mapsto x - \frac{f(x)}{f'(x)}$ et la suite $(x_n)_{n \in \mathbb{N}}$ par $x_0 = a$ et $x_{n+1} = g(x_n)$.

Cette suite se construit donc de la manière suivante : on part du point d'abscisse x_n sur la courbe représentative de f , on trace la tangente à cette courbe en ce point, puis on reporte l'intersection de cette tangente avec l'axe des abscisses pour obtenir le point d'abscisse x_{n+1} .¹



¹Schémas générés automatiquement pour n'importe quelle fonction. (programme dans le code source du DM, cf Github).



3) a) f est deux fois dérivable sur $[a, b]$ donc f' est dérivable sur $[a, b]$. g est donc dérivable sur $[a, b]$ par opérations et $g'(x) = 1 - \frac{(f'(x))^2 - f''(x)f(x)}{(f'(x))^2} = -\frac{f''(x)f(x)}{(f'(x))^2}$. Comme f'' est strictement positive sur $[a, b]$, g' a le signe opposé de f .

b)

Partie B - Vitesse de convergence

Partie C - Algorithmes

1)

2)

3)

Tentons maintenant de simplifier et d'optimiser ce code :

```
f = lambda x: x**3 - 2

def newton(f, x, h=1e-4, epsilon=1e-6):
    while abs(y := f(x)) > epsilon:
        derivee = (f(x + h) - f(x - h)) / (2 * h)
        x -= (y / derivee)
    return x
```

Pour aller encore plus loin dans la simplification, changeons de langage pour Haskell :



```
f :: (Num r) => r -> r
f x = x^3 - 2

derivee f x h = (f (x + h) - f (x - h)) / (2*h)

newton f h e x =
  if (abs . f) x > e
  then newton f h e (x - (f(x) / (derivee f x h)))
  else x

main :: IO ()
main = do
  let initialGuess = 1.0 -- Initial guess for the root
      h = 1e-4           -- Small step for derivative approximation
      e = 1e-6           -- Tolerance level for convergence
      root = newton f h e initialGuess
  putStrLn $ "Root found: " ++ show root
```