

# Método de Newton-Raphson y sus Fractales

Proyecto Final - Laboratorio de Computación - Facet-UNT - Año 2021

Atuel Villegas

## Resumen

El método de Newton-Raphson es un conocido algoritmo numérico que sirve para obtener las raíces de una función. Se explicará el funcionamiento del algoritmo y se mostrarán ejemplos a partir de una subrutina creada por el alumno. Luego se explicará cómo se obtienen los fractales de Newton utilizando el plano complejo, y se mostrarán los resultados de forma gráfica.

## 1. Introducción

El algoritmo de Newton es un método simple que nos permite obtener las raíces de una función  $f(x)$  de la cual conocemos su derivada  $f'(x)$ .

Se puede explicar el funcionamiento del algoritmo con los siguientes pasos:

1. Proponer un valor de  $x$  al que se denomina  $x_0$  como un cero tentativo de la función.
2. Calcular  $f(x_0)$ .
3. Si  $|f(x_0)|$  es menor que un error de tolerancia definido por el usuario, entonces  $x_0$  es un cero de la función (según la tolerancia establecida) y el algoritmo termina.
4. Si  $|f(x_0)|$  es mayor que el error de tolerancia, entonces se calcula  $f'(x_0)$ .
5. Calcular la intersección entre el eje de abscisas y una recta que pase por el punto  $f(x_0)$  con pendiente  $f'(x_0)$  (ver fig. 1). A esta intersección se le da el nombre de  $x_1$ :

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

6. Se vuelve al paso 2 pero utilizando  $x_1$ .

De forma general se puede expresar una iteración del algoritmo de la siguiente manera:

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

Es decir, en cada iteración el algoritmo va a corregir el valor de una dada  $x_n$  y va a entregar una  $x_{n+1}$  que en principio debería estar más cerca de un cero real de la función que el valor anterior ( $x_n$ ). Se puede implementar este algoritmo fácilmente utilizando cualquier lenguaje de programación, y en este caso se utilizará el lenguaje **Fortran**.

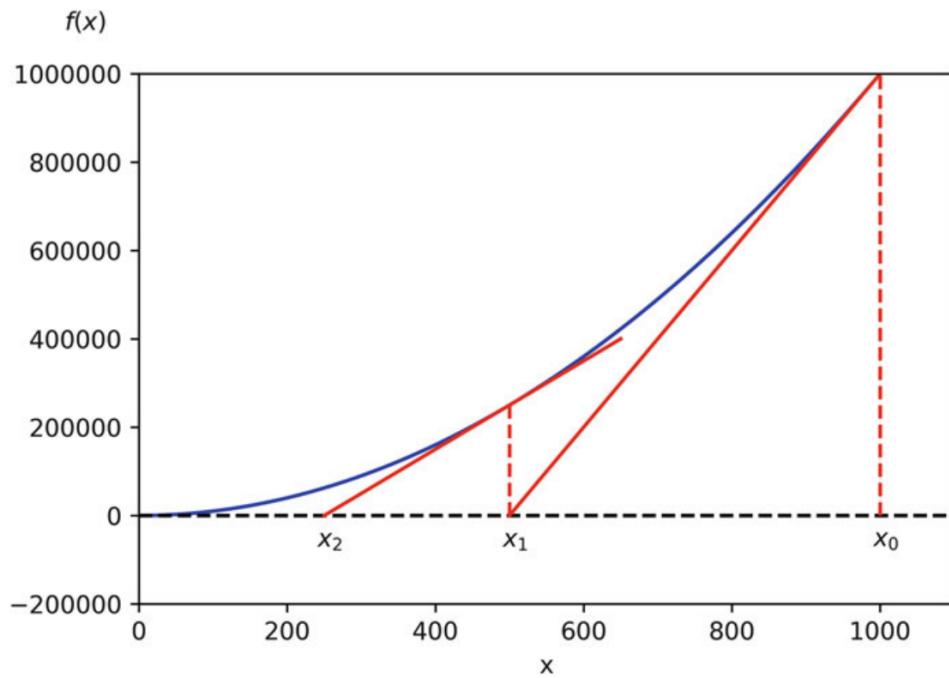


Figura 1: Método de Newton siendo aplicado en la función  $f(x) = x^2 - 9$

## 2. Primera Implementación

```

1 program main
2   ! entrada: valor de x_0 y valor n_max
3   ! salida: output.txt con las iteraciones del metodo de Newton
4   implicit none
5   real :: x
6   integer :: n
7
8   write(*,"(a)", advance="no") "x0 -> "
9   read(*,*) x
10  write(*,"(a)", advance="no") "n_max -> "
11  read(*,*) n
12
13  call newton(x, n)
14 end program main
15
16 subroutine newton(x, n)
17   ! el valor de x es la propuesta inicial, y n es el numero maximo de iteraciones
18   ! el valor de x cambia con cada iteracion y se imprime en la consola
19   ! a la vez se genera el archivo de salida "output.txt"
20   implicit none
21   real ,intent(inout) :: x
22   integer,intent(in) :: n
23   real :: f, df, step
24   integer :: i
25
26   open(15, file="output.txt")
27   write(15, *) "iteracion | x_n      "
28   write(15, *) 0, x
29   do i = 1, n
30     step = f(x)/df(x)
31     x = x - step
32     write(15, *) i, x
33     print*, i, x
34     if (abs(step) < 0.00001) then
35       exit

```

```

36      endif
37  enddo
38
39  close(15)
40 end subroutine newton
41
42 function f(x) result(r)
43   implicit none
44   real, intent(in):: x
45   real :: r
46
47   r = x**2 - 9
48 end function f
49
50 function df(x) result(r)
51   implicit none
52   real, intent(in):: x
53   real :: r
54
55   r = 2*x
56 end function df

```

Listing 1: primera implementación

En esta primera implementación, el código realizado cuenta con dos funciones (una para  $f(x)$  y otra para  $f'(x)$ ) y una subrutina llamada *newton* que se encarga de ejecutar el algoritmo.

Si bien el método es útil en casos dónde la resolución analítica es imposible o demasiado complicada, es una buena idea ponerlo a prueba con funciones simples, de las cuales se puede obtener las raíces analíticamente, para comprobar que el método funciona correctamente.

El programa pide al usuario dos *inputs*:  $x_0$  y  $n_{max}$ (máximo número de iteraciones permitidas) y genera un archivo de texto *output1.txt* con las iteraciones y los valores de  $x_n$ .

Para la función  $f(x) = x^2 - 9$  con un  $x_0 = 1000$  se obtiene:

	iteracion   x_n
1	0 10000.0000
2	1 5000.00049
3	2 2500.00122
4	3 1250.00244
5	4 625.004822
6	5 312.509613
7	6 156.269211
8	7 78.1633987
9	8 39.1392708
10	9 19.6846085
11	10 10.0709095
12	11 5.48228645
13	12 3.56196856
14	13 3.04433060
15	14 3.00032282
16	15 3.00000000

Listing 2: output1.txt

Se observa fácilmente como el valor de  $x$  converge al valor real del cero  $x = 3$ . Si el valor de  $x_0$  fuese -10000 en lugar de 1000, el algoritmo convergería al valor  $x = -3$ .

### 3. Problemas con el Algoritmo

Es fácil notar que si el valor propuesto de  $x_0$  resulta en una derivada nula,  $f'(x_0) = 0$ , el siguiente valor estará indeterminado.

Pero sin llegar a este caso, es importante notar que hay casos dónde el valor de  $x$  va a diverger con cada iteración del método, es decir, que en vez de acercarnos al valor deseado, nos va a alejar.

Un ejemplo lo vemos con la función  $f(x) = \tanh(x)$ . Esta función tiene un cero en  $x = 0$ , y usando el método con valores de  $x$  pequeños (por ejemplo  $x_0 = 1$ ) se obtiene el resultado deseado. Sin embargo al aumentar un poco este valor ( $x_0 = 1,1$ ) el resultado va a diverger:

$x_0 = 1,0$			$x_0 = 1,1$		
1	iteracion	x_n	1	iteracion	x_n
2		0 1.00000000	2		0 1.10000002
3		1 -0.813430429	3		1 -1.12855256
4		2 0.409402847	4		2 1.23413098
5		3 -4.73051667E-02	5		3 -1.69516480
6		4 7.06091523E-05	6		4 5.71534348
7		5 0.00000000	7		5 -23020.5938
8		6 0.00000000	8		6 Infinity
9		7 0.00000000	9		7 NaN

Cuadro 1: función  $f(x) = \tanh(x)$

Otro posibilidad es que el algoritmo entre en un ciclo. Esto ocurre por ejemplo en la función  $f(x) = x^3 - 2x + 2$ . Si se introduce el valor de  $x_0 = 0$  el algoritmo entra en un ciclo pasando de  $x = 1$  a  $x = 0$  indefinidamente.

Todo esto se mostrará en las siguientes secciones de forma gráfica con la ayuda de los fractales de Newton.

## 4. Segunda Implementación: Plano Complejo

```

1 program main
2   ! entrada: ninguna. las variables y parametros utilizados estan dentro del
3   ! codigo
4   ! salida: newton.txt con la matriz de enteros que representan el fractal
5   implicit none
6
7   integer, parameter :: N = 100
8   real, parameter :: xmax = 3, imax = 3      ! rango en el plano complejo
9   integer :: i, j, k                         ! variables para los do loops
10  ! la recta real
11  real :: reals(N+1) = [(real(i)/N*2*xmax, i = 0, N)] - xmax
12  ! la recta imaginaria
13  real :: imag(N+1) = [(real(i)/N*2*xmax, i = N, 0, -1)] - imax
14  integer :: image(N+1,N+1) = 0             ! la matriz final
15  real :: error = 0.01                      ! error de tolerancia
16  integer, parameter :: rn = 3              ! numero de raices
17  real :: aux(rn)                          ! array auxiliar para comparar raices
18  complex :: roots(rn)                    ! raices verdaderas del polinomio
19  complex :: x                           ! la raiz compleja para iterar
20
21  roots = [(-1, 0), (0.5,0.866), (0.5,-0.866)] ! raices del polinomio
22
23  open(12,file="newton.txt")
24
25
26  do j = 1, N+1
27    do i = 1, N+1
28      x = complex(reals(i), imag(j)) ! la raiz propuesta x
29      call newton(x, 50)
30      aux = abs(roots - x)
31      do k = 1, rn
32        if (aux(k) < error) then
33          image(j,i) = k
34          exit
35        endif
36      enddo

```

```

37     enddo
38     write(12, *) image(j, :)
39   enddo
40   close(12)
41 end program main
42
43 subroutine newton(x, n)
44 ! x: raiz propuesta. Cambia con cada iteracion
45 ! n: numero maximo de iteraciones
46 implicit none
47 complex,intent(inout) :: x
48 integer,intent(in) :: n
49 complex :: f, df, step
50 integer :: i
51
52 do i = 1, n
53   step = f(x)/df(x)
54   x = x - step
55   if (abs(step) < 0.00001) then
56     exit
57   endif
58 enddo
59 end subroutine newton
60
61 function f(x) result(r)
62 implicit none
63 complex, intent(in):: x
64 complex :: r
65
66 r = x**3 + 1
67 end function f
68
69 function df(x) result(r)
70 implicit none
71 complex, intent(in):: x
72 complex :: r
73
74 r = 3*x**2
75 end function df

```

Listing 3: segunda implementación

Se sabe por el Teorema Fundamental del Algebra que un polinomio de grado  $N$  tiene  $N$  raíces en el plano complejo. Por ejemplo como la función  $z^3 - 1$  tiene una raíz real, significa que las otras dos están en el plano complejo.

Por lo tanto es de interés comprobar como se comporta el método cuando el dominio de  $x$  pasa a ser el conjunto de los números complejos (a partir de ahora se usará  $z$  y  $z_0$  en lugar de  $x$  y  $x_0$  debido al cambio de dominio).

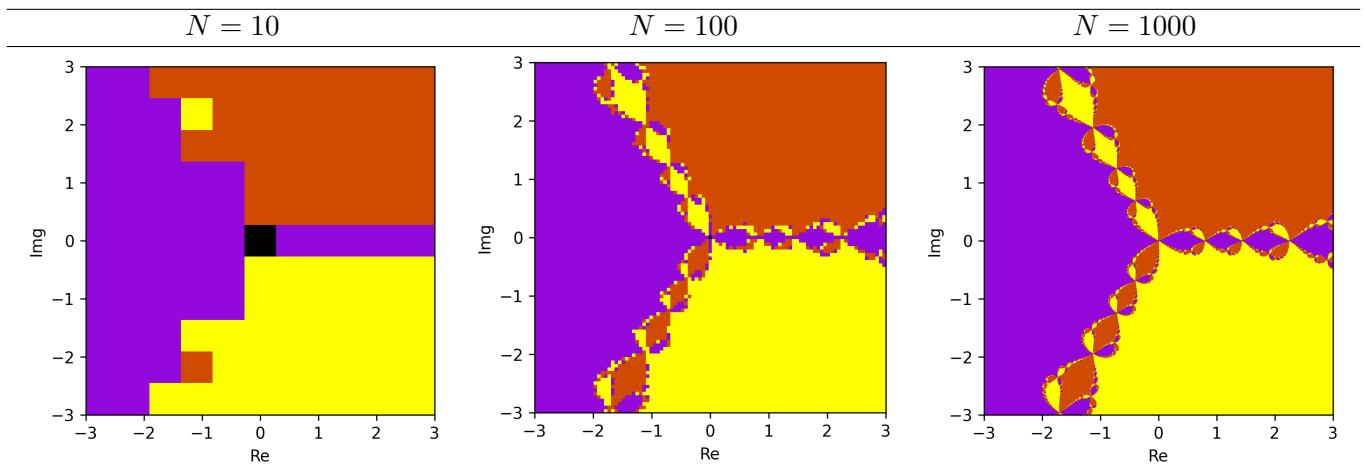
En la segunda implementación directamente se toma todo un plano acotado del espacio complejo cuyos límites están definidos en el código. Dentro del código se divide el plano en una matriz de tamaño  $N \times N$  que es una discretización del espacio en el plano. Luego se aplica el método de newton en cada punto y en función de a que cero diverge ese punto, se le coloca un color u otro (cada cero va a tener un color asociado) y en caso de diverger se deja en color negro.

El programa genera como salida un archivo de texto *newton.txt* el cual es solo una matriz de enteros. Luego se puede utilizar programas como **gnuplot** para ver los datos graficamente:

```
1 plot "newton.txt" matrix with image
```

Listing 4: comando gnuplot

En el cuadro 2 se observan los resultados graficados para 3 casos de  $N$  para  $z^3 - 1 = 0$ . En el caso de  $N = 10$  no se observa nada fuera de lo común; la función  $z^3 - 1$  tiene tres ceros, por lo tanto



Cuadro 2:  $z^3 - 1 = 0$  para tres diferentes *resoluciones*.

vemos tres colores, y en el centro un punto negro porque ahí la derivada es cero, lo que conduce a una indeterminación en el método.

Sin embargo con  $N = 100$  se empieza a notar que los bordes entre las regiones no están muy bien definidos. Con  $N = 1000$  ya se puede observar sin problemas lo que se conoce como **fractal de Newton**.

## 5. Fractales de Newton y Tercera Implementación

```

1 program main
2     ! entrada: ninguna. las variables y parametros utilizados estan dentro del
3     ! codigo
4     ! salida: newton.txt y alpha.txt
5     implicit none
6     integer, parameter :: N = 5000
7     real, parameter :: xmax = 3, imax = 3      ! rango en el plano complejo
8     integer :: i, j, k                         ! variables para los do loops
9     ! la recta real
10    real :: reals(N+1) = [(real(i)/N*2*xmax, i = 0, N)] - xmax
11    ! la recta imaginaria
12    real :: imag(N+1) = [(real(i)/N*2*xmax, i = N, 0, -1)] - imax
13    integer :: image(N+1, N+1) = 0             ! matriz final
14    real :: alpha(N+1, N+1) = 0                ! matriz alpha
15    real :: error = 0.01                       ! error de tolerancia para las raices
16    integer, parameter :: rn = 3               ! numero de raices
17    real :: aux(rn)                          ! array auxiliar para comparar raices
18    complex :: roots(rn)                     ! Raices verdaderas del polinomio
19    complex :: x                            ! raiz compleja para iterar
20    real :: a                                ! valor individual para alpha
21
22    roots = [(-1.76929, 0), (0.88465, 0.58974), (0.88465, -0.58974)]    ! raices
23
24    open(12, file="newton.txt")
25    open(13, file="alpha.txt")
26
27    do j = 1, N+1
28        do i = 1, N+1
29            x = complex(reals(i), imag(j)) ! la raiz propuesta x
30            call newton(x, 50, a)
31            aux = abs(roots - x)
32            do k = 1, rn
33                if (aux(k) < error) then
34                    image(j,i) = k
35                    exit
36            end if
37        end do
38    end do
39
40    write(*, *) "Guardando resultados"
41    close(12)
42    close(13)
43
44    print *, "Proceso finalizado"
45
```

```

35         endif
36     enddo
37     alpha(j, i) = a
38   enddo
39   write(12, *) image(j, :)
40   write(13, *) alpha(j,:)
41 enddo
42 close(12)
43 close(13)
44
45 end program main
46
47 subroutine newton(x, n, a)
48   implicit none
49   complex, intent(inout) :: x ! x: raiz compleja para iterar
50   integer, intent(in) :: n      ! n: num. maximo de iteraciones
51   real, intent(inout) :: a      ! a: medida de convergencia
52   complex :: f, df, step
53   integer :: i
54
55   do i = 1, n
56     step = f(x)/df(x)
57     x = x - step
58     if (abs(step) < 0.00001) then
59       exit
60     endif
61   enddo
62
63   if (i < n/10) then
64     a = 0.6
65   else if (i < n/8) then
66     a = 0.7
67   else if (i < n/6) then
68     a = 0.75
69   else if (i < n/5) then
70     a = 0.8
71   else if (i < n/2) then
72     a = 0.9
73   else
74     a = 1
75   endif
76
77 end subroutine newton
78
79 function f(x) result(r)
80   implicit none
81   complex, intent(in):: x
82   complex :: r
83
84   r = x**3 -2*x +2
85 end function f
86
87 function df(x) result(r)
88   implicit none
89   complex, intent(in):: x
90   complex :: r
91
92   r = 3*x**2 -2
93 end function df

```

Listing 5: tercera implementación

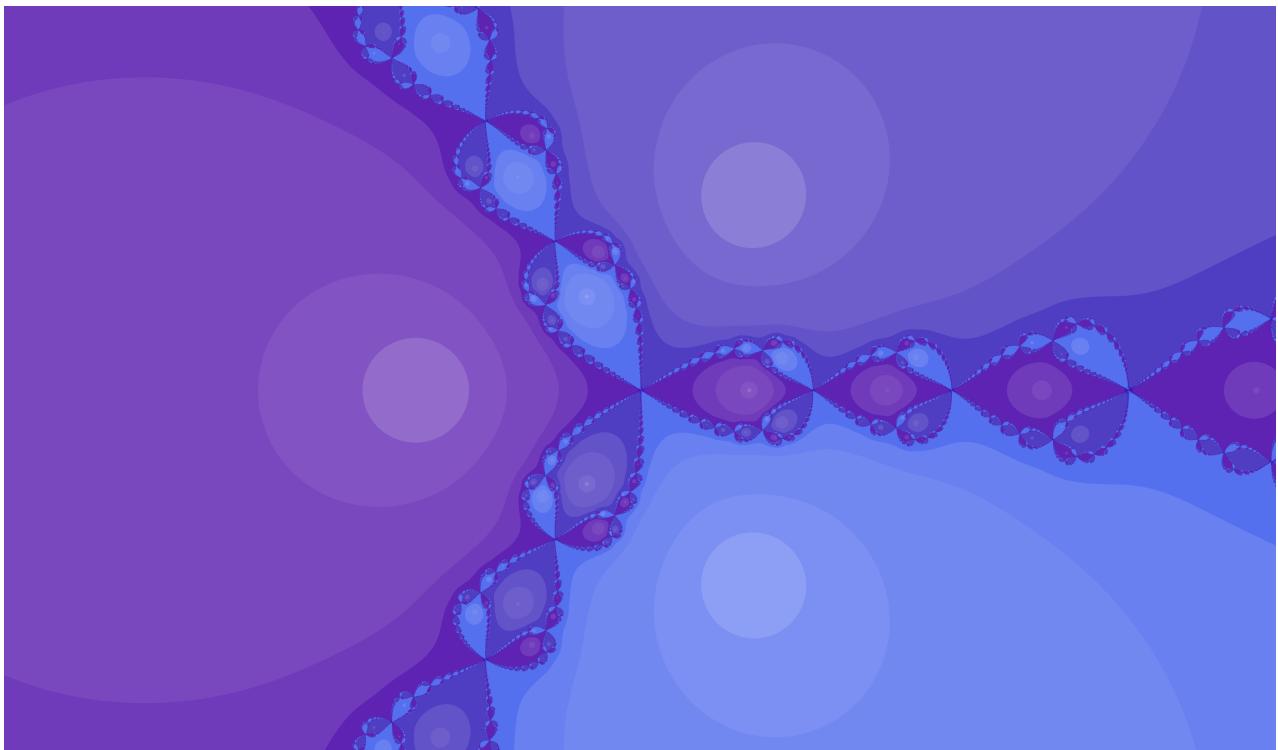


Figura 2: Función  $f(z) = z^3 - 1$

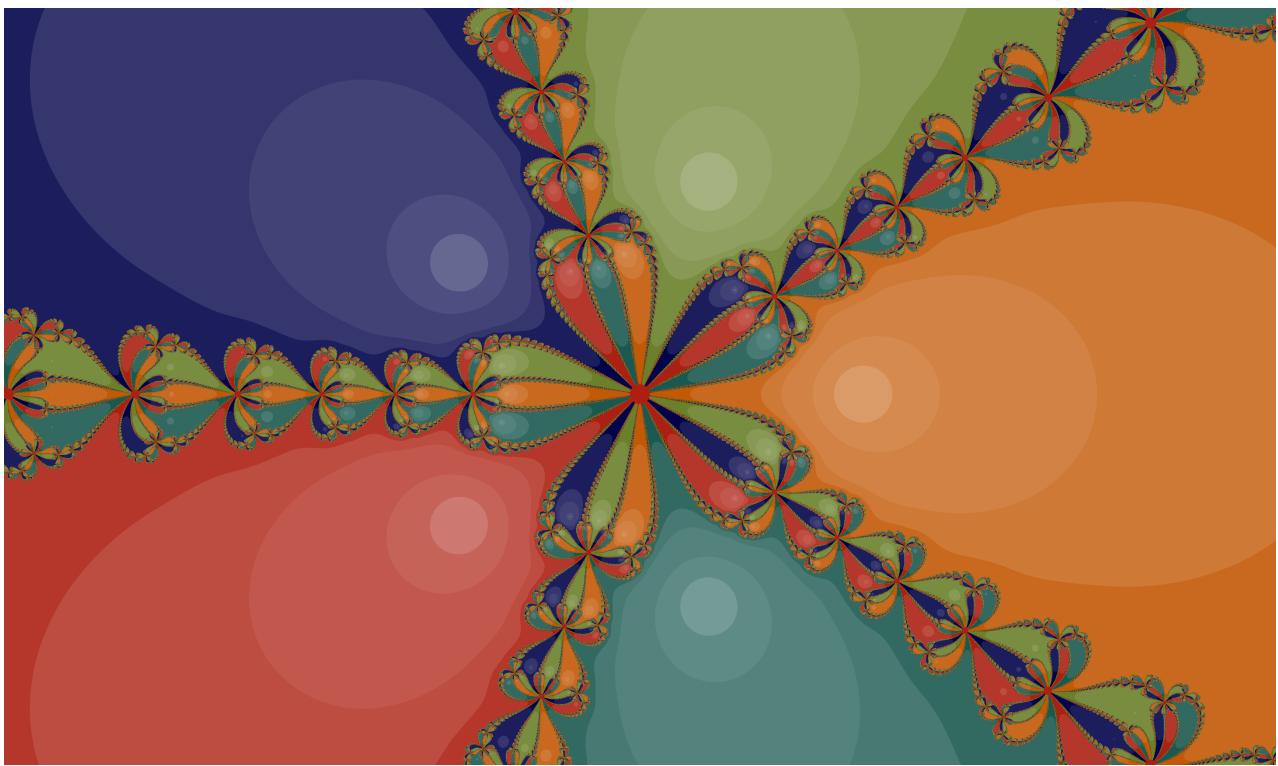


Figura 3: Función  $f(z) = z^5 - 1$

Lo que vemos gráficamente en los fractales de newton es que el método se comporta de forma muy caótica cuando el valor de  $z_0$  se encuentra en puntos medios entre los distintos ceros verdaderos de la función. Es decir, un cambio increíblemente pequeño en el valor inicial  $z_0$  puede cambiar el resultado al que el método va a convergir.

En la tercera implementación se agrega a la salida del programa una segunda matriz *alpha.txt*

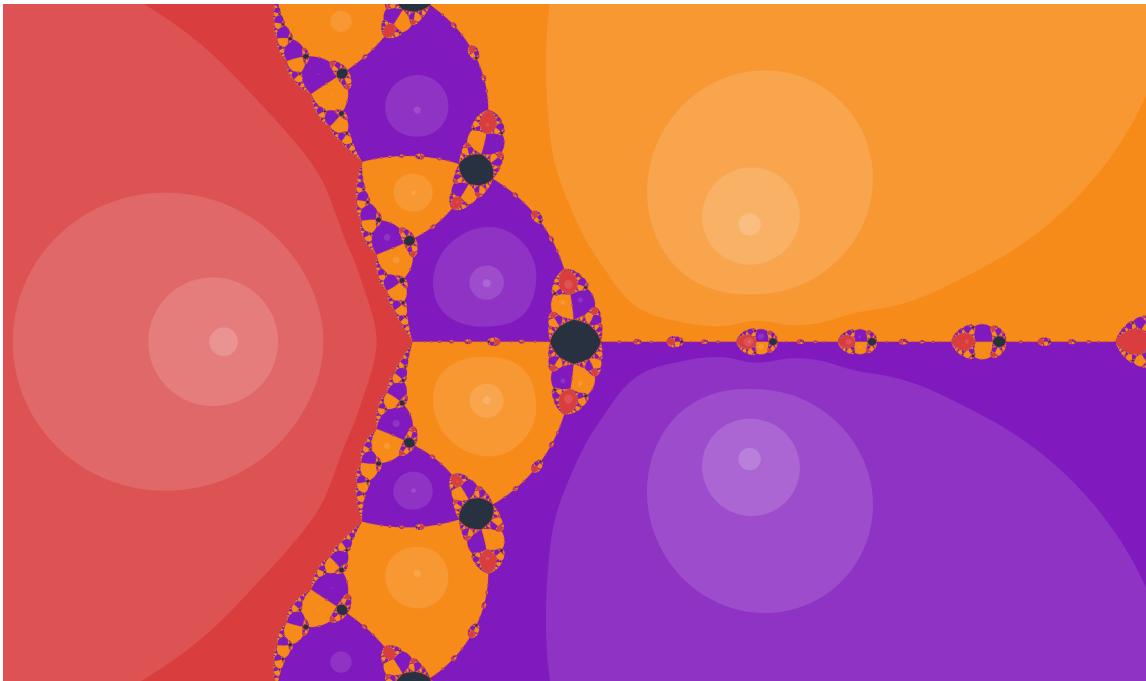


Figura 4: Función  $f(z) = z^3 - 2x + 2$

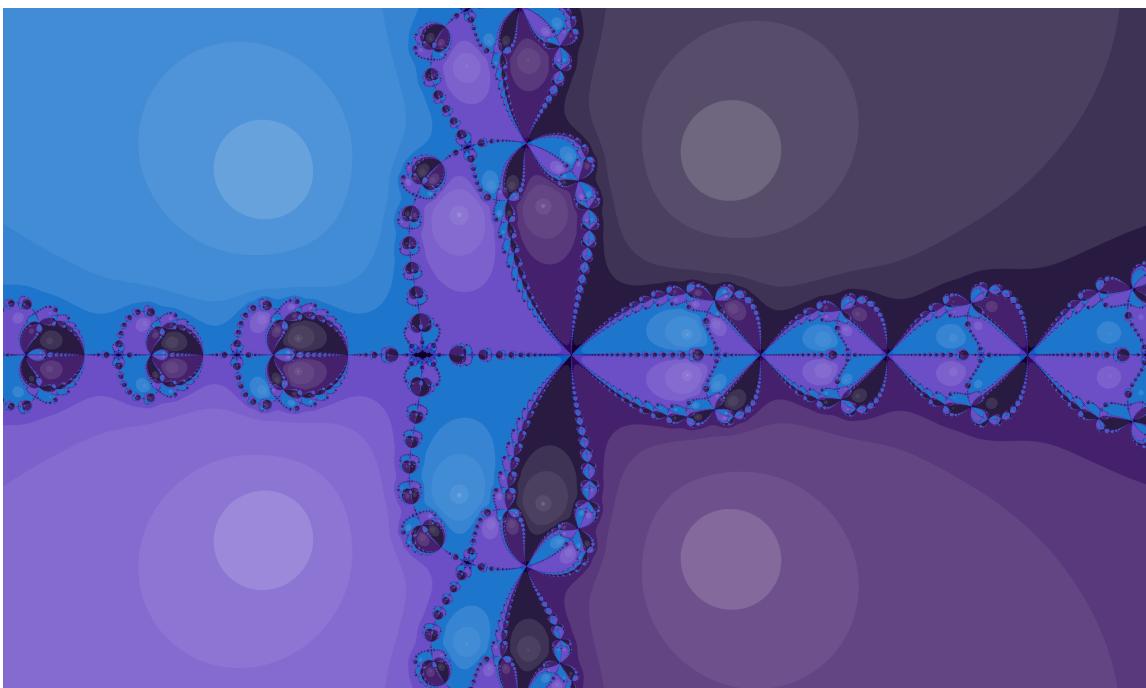


Figura 5: Función  $f(z) = z^4 - z^3 + 1$

con valores reales entre 0.5 y 1. Un valor cercano a 0.5 indica una rápida convergencia del método, y un valor cercano 1 indica una convergencia más lenta (o no convergencia en absoluto).

Luego, utilizando el lenguaje **python** con los módulos **numpy** para leer los datos y **matplotlib** para la implementación gráfica se realizan las gráficas restantes de este informe (el código utilizado se encuentra en los apéndices ya que no es tan relevante).

La matriz *alpha* se ve reflejada en las imágenes como el nivel de transparencia. *alpha* = 1,0 es transparencia nula, y *alpha* = 0,0 indica una transparencia total.

Las figuras 2 y 3 tienen una distribución de ceros bastante simétricas (se obtienen fácilmente haciendo la raíz).

Por otro lado, en la figura 4 vemos el caso mencionado anteriormente, para la función  $f(z) = z^3 - 2x + 2$ , donde las zonas totalmente negras son zonas de no convergencia (el algoritmo forma un ciclo cerrado). Y en la figura 5 usamos una función más compleja para obtener un fractal menos simétrico que los anteriores.

Estas últimas figuras se graficaron

## 6. Conclusiones

Se observó como un algoritmo tan simple en principio como el algoritmo de Newton, con un objetivo igual de simple, puede dar lugar a una complejidad inesperada.

Por una parte esto muestra la importancia de conocer a fondo el algoritmo que se va a utilizar, pues se corre el riesgo de encontrar resultados inesperados que solo se entienden a partir del funcionamiento del algoritmo en cuestión.

Y por otro lado, este informe también apunta simplemente al hecho de apreciar la complejidad visual de los fractales de newton, que surgen de extender el dominio inicial del algoritmo. En el apéndice A se menciona muy brevemente su relación con el conjunto de Julia, y algo del conjunto de Mandelbrot, temas de análisis complejo.

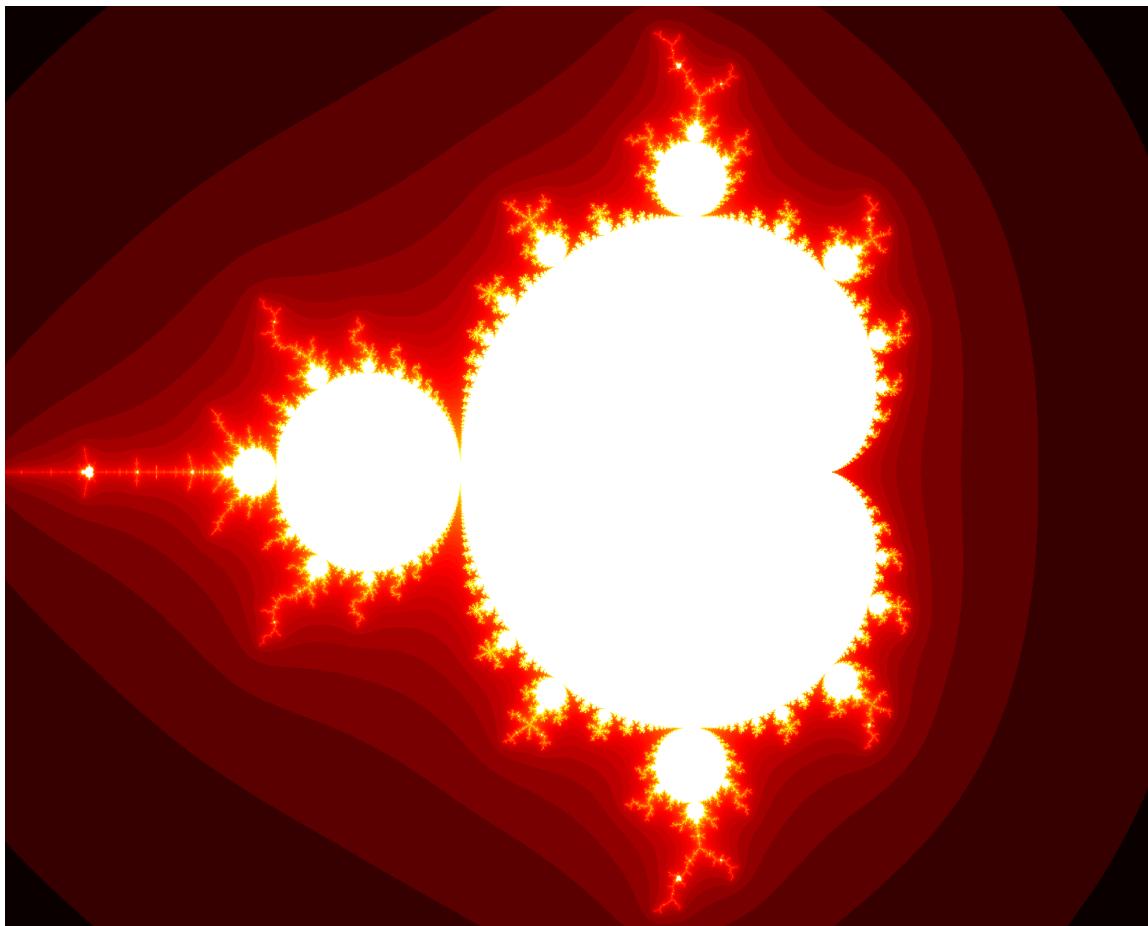


Figura 6: Conjunto de Mandelbrot (zona blanca). Un rojo claro indica una divergencia más lenta que la de un rojo oscuro.

## A. Apéndices

### A.1. Conjunto de Julia y Conjunto de Mandelbrot

Los fractales de Newton son un caso particular del conjunto de Julia. Son el conjunto de la función meromorfa  $z \rightarrow z - \frac{p(z)}{p'(z)}$  que está dada por el método de Newton.

Un conjunto de Julia se puede graficar a partir de un procedimiento muy similar al realizado para los fractales de Newton, pero no se lo realizará en este informe, ni tampoco se dará una definición formal.

Por otro lado, otro conjunto bastante conocido es el de Mandelbrot, el cuál tiene relación tanto con los fractales de Newton como con el conjunto de Julia, pero esta no se intentará explicar aquí.

El conjunto de Mandelbrot es el conjunto de valores de  $c$  en el plano complejo para los cuales la órbita del punto crítico  $z = 0$  permanece acotada bajo la iteración

$$z_{n+1} = z_n^2 + c$$

Por ejemplo, para  $c = 1$  la secuencia es  $0, 1, 5, 26, 677, \dots$ , por lo tanto no pertenece al conjunto, mientras que con  $c = -1$  la secuencia es  $0, 1, 0, 1, \dots$ , por lo tanto sí pertenece al conjunto.

El código utilizado para graficar el set de Mandelbrot que se observa en la figura 6 se encuentra en los siguientes apéndices.

### A.2. Nota sobre a los códigos

Cada implementación va a tener una función distinta, con sus distintas raíces. Para obtener el fractal de otra función habrá que cambiar las funciones  $f(x)$  y  $df(x)$ , además colocar las raíces de la función dentro del array correspondiente, e indicar el número total de estas (excepto en la primera implementación).

### A.3. Código Python: Graficar datos implementación 3

```
1 # Este código de python esta pensado para usarse con la tercera implementacion.
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import matplotlib.colors as mcolors
5
6 filepath = "newton.txt"
7
8 image = np.loadtxt(filepath, int)
9 alpha = np.loadtxt("alpha.txt", float)
10
11
12 # -----
13 # Las siguientes funciones sirven solo para crear un mapa de color a partir de una
14 # lista con valores HEX
15
16 def hex_to_rgb(value):
17     """
18     convierte hex a rgb
19     value: string de 6 caracteres representando un color hex.
20     return: lista con los valores RGB ''
21     value = value.strip("#") # borra el hash si esta presente
22     lv = len(value)
23     return tuple(int(value[i:i + lv // 3], 16) for i in range(0, lv, lv // 3))
24
25
26 def rgb_to_dec(value):
27     """
28     Convierte RGB a valores decimales (divide cada valor en 256)
29     value: lista con los valores RGB
30     Returns: lista con los valores decimales ''
```

```

31     return [v/256 for v in value]
32
33 def get_continuous_cmap(hex_list, float_list=None):
34     ''' crea y regresa un mapa de color para ser utilizado en las figuras.
35         Si float_list no esta presente, el mapa de color gradua linealmente entre
36         cada color de la hex_list.
37         Si float_list esta presente, cada color es mapeado a la respectiva
38         localizacion.
39     '''
40     rgb_list = [rgb_to_dec(hex_to_rgb(i)) for i in hex_list]
41     if float_list:
42         pass
43     else:
44         float_list = list(np.linspace(0,1,len(rgb_list)))
45
46     cdict = dict()
47     for num, col in enumerate(['red', 'green', 'blue']):
48         col_list = [[float_list[i], rgb_list[i][num], rgb_list[i][num]] for i in
49                     range(len(float_list))]
50         cdict[col] = col_list
51     cmp = mcollections.LinearSegmentedColormap('my_cmp', segmentdata=cdict, N=256)
52
53     # -----
54
55     hexs = ["ae2012", "ca6702", "55a630", "03045e", "000000"]
56
57     plt.imshow(image, alpha=alpha, cmap=get_continuous_cmap(hexs))
58     plt.savefig("newton.png", dpi=300)
59     plt.show()

```

#### A.4. Código Fortran: Conjunto de Mandelbrot

```

1 program main
2     implicit none
3     integer, parameter :: N = 1000
4     real, parameter :: xmax = 2, imax = 2           ! rango en el plano complejo
5     integer :: i, j, k                           ! variables para los do loops
6     real :: reals(N+1) = [(real(i)/N*2*xmax, i = 0, N)] - xmax    ! la recta real
7     real :: imag(N+1) = [(real(i)/N*2*xmax, i = N, 0, -1)] - imax ! la recta
8     imaginaria
9     integer :: image(N+1,N+1) = 0                  ! Matriz con datos finales para
10    exportar
11    complex :: x
12
13    open(12,file="mandelbrot.txt")
14
15    do j = 1, N+1
16        do i = 1, N+1
17            x = complex(reals(i), imag(j))
18            call iteration(x, k)
19            image(j, i) = k
20        enddo
21        write(12, *) image(j, :)
22    enddo
23    close(12)
24
25 end program main
26
27 subroutine iteration(c, n)
28
29     implicit none
30     complex, intent(inout) :: c
31     integer, intent(inout) :: n

```

```

30 complex :: z
31 integer :: i
32
33 z = (0, 0)
34 do i = 1, 80
35   z = z**2 + c
36   if (abs(z) > 300) then
37     n = i
38     exit
39   endif
40 enddo
41 if (abs(z) < 5) then
42   n = 0
43 endif
44 end subroutine iteration

```

El archivo de texto de este último código se puede graficar tanto con gnuplot como con python.

## B. Bibliografía

- University of Cambridge, Department of Physics - Computational Physics - Self-study guide 2 Programming in Fortran 95
- Springer - Programming for Computations - Python (2nd. ed) Svein Linge · Hans Petter Langtangen
- <https://fortran-lang.org/learn/>
- [https://en.wikipedia.org/wiki/Newton%27s\\_method](https://en.wikipedia.org/wiki/Newton%27s_method)
- [https://en.wikipedia.org/wiki/Newton\\_fractal](https://en.wikipedia.org/wiki/Newton_fractal)
- <https://matplotlib.org/3.4.3/contents.html>

# Índice

<b>1. Introducción</b>	<b>1</b>
<b>2. Primera Implementación</b>	<b>2</b>
<b>3. Problemas con el Algoritmo</b>	<b>3</b>
<b>4. Segunda Implementación: Plano Complejo</b>	<b>4</b>
<b>5. Fractales de Newton y Tercera Implementación</b>	<b>6</b>
<b>6. Conclusiones</b>	<b>10</b>
<b>A. Apéndices</b>	<b>11</b>
A.1. Conjunto de Julia y Conjunto de Mandelbrot . . . . .	11
A.2. Nota sobre a los códigos . . . . .	11
A.3. Código Python: Graficar datos implementación 3 . . . . .	11
A.4. Código Fortran: Conjunto de Mandelbrot . . . . .	12
<b>B. Bibliografía</b>	<b>13</b>