75.12 Analisis Numerico - Curso 7

Informe TP1

Nombre	Padron
Seminara, Roberto Dario	85604
Garcia, Omar	87500

Introduccion

Problema 1:

Se solicita calcular tres raices con exponentes **enteros:** $\sqrt{2}\ \sqrt{5}\ \sqrt[3]{3}$.

Para resolverlas se optó por plantear matemáticamente un polinomio cuya raíz sea el número buscado para cada caso y hallar la aproximacion aplicando metodos iterativos, se evaluara el uso de tres metodos posibles:

- Punto fijo
- Newton-Raphson
- Biseccion

Estas son las ecuaciones correspondientes a cada raiz, denominadas F(x):

- $x = \sqrt{2} \Rightarrow x^2 - 2 = 0$
- $x = \sqrt{5} \Rightarrow x^2 - 5 = 0$
- $x = \sqrt[3]{3} \Rightarrow x^3 - 3 = 0$

Punto fijo

Para utilizar el metodo de punto fijo se plantea estas ecuaciones

F(x)	f(x)	f'(x)
x^2 - 2 = 0	2/x	- 2/x^2
x^2 - 5 = 0	5/x	- 5/x^2
x^3 - 3 = 0	3/x^2	-6/x^3

En todos los casos se puede definir un intervalo a-b para el cual f(x) cae en ese mismo intervalo, pero sin embargo cualquier valor mayor que la raiz da una derivada cuyo modulo es mayor que 1, demostrando que **el metodo de punto fijo NO es aplicable,** al menos no directamente.

Newton Raphson

Newton-Raphson define una funcion cuyo punto fijo es la raiz de otra (la funcion de la cual queremos hallar la raiz)

f(x) se expresa según lo indica el metodo de Newton-Raphson:

$$f(x) = x - \frac{F(x)}{F'(x)}$$

Siendo F(x) el polinomio cuya raíz coincide con la raíz que se está buscando hallar:

F(x)	F'(x)	$f(x)$
$x^2 - 2 = 0$	**2x**	$x - \frac{x^2-2}{2x}$
$x^2 - 5 = 0$	**2x**	$x - \frac{x^2-5}{2x}$
$x^3 - 3 = 0$	**$3x^2$**	$x - \frac{x^3-3}{3x^2}$

Esas funciones y todas sus derivadas de todos los ordenes son continuas, las derivadas no se anulan en la raiz por lo que las condiciones para usar Newton-Raphson se dan y **el metodo de Newton-Raphson puede utilizarse**

Biseccion

Las funciones cuya raiz hay que calcular son crecientes en el intervalo del 0 al infinto, por lo que el metodo puede utilizarse.

La razon por la que no se optara por este metodo es porque requiere mas cantidad de iteraciones.

Hay que obtener un error relativo inferior al 0.0001, el error del metodo de la biseccion es del orden de 2^{-i}, por lo tanto se requeriran al menos 14 iteraciones, se sabe que el metodo de biseccion es menos restrictivo, mas simple y mas predible, pero requiere mas iteraciones para lograr la misma cota de error.

Finalmente se opto por Newton-Raphson

Para resolverlas se calcularán iteraciones del punto fijo de f(x) hasta obtener un valor cuya diferencia con el valor calculado por las funciones matemáticas del lenguaje sea menor que la cota de error buscado (0.01%).
Números de punto flotante con doble y simple precisión

Para la programación de los algoritmos se utilizará el lenguaje de programación Ruby por sus caracteristicas de flexibilidad y desarrollo rapido.
Sin embargo, la librería estándar de ese lenguaje no provee numeros de punto flotante con distintos niveles de precisión, solo provee un tipo de datos que es el Float, el cual internamente encapsula al **double** nativo de C (52 bits en la mantisa, 11 en el exponente)

Para cubrir esta falta y así poder probar los algoritmos usando números de punto flotante con distintos niveles de precision, se implementará un tipo numérico que decore al float, y simule la falta de precisión quitándole bits a la mantisa aplicando redondeo después de cada operación.

Por ejemplo, si se redondeara la mantisa a 23 bits tras cada operacion, se obtendría una limitación prácticamente igual a la que obtendríamos si usáramos el punto flotante de simple precision de C (el float de C), tambien existe la posibilidad de probar con otros límites más reducidos para la mantisa y ver qué ocurre en casos extremos

Explicación del algoritmo

Lo que el algoritmo hace es extraer el exponente, la mantisa y el signo del número al cual se le quiere reducir la precisión, para volver a construir el número usando la ecuación:

$$x = signo \, . \, 2^{exponente} \, . \, mantisa$$

Pero en lugar de usar exactamente la misma mantisa, se utiliza una versión redondeada a la cantidad de decimales que se estaría simulando caben en la mantisa (por ejemplo, 23 decimales si se esta simulando un numero de punto flotante de precision simple).

Respecto al exponente, teniendo en cuenta que los números que se manejaran no son enormes (nunca se saldrán del rango), se puede dejar de lado la emulación del exponente limitado.

Para extraer el exponente y la mantisa se utilizan métodos numericos, NO se accede a los datos binarios que componen el float.

Problema 2:

Se pide calcular $Sin(\frac{\pi}{3})$ y $Cos(\frac{\pi}{3})$ usando sus expansiones en serie:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x0).(x - x0)^n}{n!}$$

Sumar infinitos terminos de la serie, matematicamente hablando, da el valor exacto de la funcion evaluada en un punto x. En el ambito numérico, sólo se puede sumar un número finito n de terminos para aproximar el resultado asegurando una determinada cota del **error de truncamiento.**

La razón por la que la expansion en serie de Taylor puede ser útil para este caso, y muchos otros, es porque se puede elegir un x0 para el cual las derivadas de todos los órdenes son conocidos.

Si $f(x) = sin(x)$

Entonces $f^{(n)}(x)$es sin(x) para n = 0, cos(x) para n = 1, -sin(x) para n = 2, -cos(x) para n = 3, sin(x) para n=4 y asi...

Las opciones para x0 son 0, $\frac{\pi}{2}$, $\frac{-\pi}{2}$ y π ya que los valores de coseno y seno son conocidos para esos puntos

x_0	$f^{(n)}(x_0)$
0	0 para n = 0, 1 para n = 1, 0 para n = 2 y -1 para n = 3
$\frac{\pi}{2}$	1 para n = 0, 0 para n = 1, -1 para n = 2 y 0 para n = 3

Usar cero para x_0 nos ahorraría una operación de resta pero si se usa $\frac{\pi}{2}$ se convergerá mas rapido ya que este valor está más cerca del $\frac{\pi}{3}$ que cero, finalmente se optará por usar $x_0 = \frac{\pi}{2}$ ya que el error de la resta es despreciable, y de hecho la resta puede hacerse matematicamente antes

Las mismas resoluciones aplican tambien para el caso del calculo de $Cos(\pi/3)$

Niveles de Precision

Para el problema 1 y 2, se utilizaran 4 niveles distintos de precision

Tipo de dato	Caracteristicas	Justificacion
Precision doble	52 bits de mantisa	Lo pide el enunciado
Precision simple	23 bits de mantisa	Lo pide el enunciado
Precision a medida de 14 bits	14 bits de mantisa	Es la menor precision que permite alcanzar la cota de error buscado
Precision media	10 bits de mantisa	Esta en el estandar IEEE y su numero de maquina caracteristico esta por debajo del error de truncamiento de 0.01% siendo un ejemplo de precision que no sirve para el problema

Para el caso de la precision media, jamas se alcanzara el criterio de parada de los algoritmos porque la precision de 10 bits en la mantisa no alcanza para representar la diferencia del 0.01%

Problema 3

El tercer problema no presenta complejidades específicas del análisis numérico. Se utilizará un array de 1000 posiciones con booleanos para representar la criba de Erastóstenes.

Errores

Ambos problemas 1 y 2 presentan errores de truncamiento **en su resultado final** (las raices en el problema 1 y el seno y coseno del problema 2), siendo precisamente el criterio de corte alcanzar un valor cuyo error de truncamiento sea menor al 0.01%. El error producido al interrumpir un algoritmo numerico iterativo como lo es el metodo iterativo de newton-raphson o la sumatoria de la serie de taylor se denominan errores de truncamiento.

Ambos problemas 1 y 2 presentan errores de redondeo por tener una mantisa de tamaño limitado, en este caso el tamaño de la mantisa de los tipos de datos utilizados alcanza ampliamente para representar las diferencias del error de truncamiento buscado, por lo que **el error de redondeo no influye en el resultado final significativamente**, aunque puede ocasionar que el algoritmo use mas iteraciones para converger

Solamente el problema 2 presenta errores inherentes que cambian el resultado final, se pide calcular $Sin(\frac{\pi}{3})$ y $Cos(\frac{\pi}{3})$ y para ello se utiliza la expansion en serie, pero el valor de -$\frac{\pi}{6}$ utilizado como x - x_0 en el programa es solamente una aproximacion, ya que el valor de $\overline{\pi}$ es una aproximacion de π y por lo tanto la serie NO converge a $Sin(\frac{\pi}{3})$ sino a $Sin(\frac{\overline{\pi}}{3})$. Como el valor de referencia se calcula tambien usando la aproximacion de π,

Sin embargo el error puede despreciarse, ya que la constante M_PI de c (utilizada en ruby) es tan precisa como permite la representacion de double, su error relativo es tan pequeño como el μ (el numero de maquina), es menor que 2^{-52} , totalmente despreciable en comparacion con los errores de truncamiento con los que se estaba trabajando.

Desarrollo

Codigo del ejercicio 1

```
class Float
  def binary_round(decimals = 0)
    factor = 2**decimals
    (self * factor).round.to_f / factor
  end

  # extrae la mantisa del numero
  def mantissa
    compute_mantissa_and_exponent unless @mantissa
    @mantissa
  end

  def exponent
    compute_mantissa_and_exponent unless @exponente
    @exponent
  end

  def sign
    if self < 0
      -1
    elsif self > 0
      1
    else
      0
    end
  end

  def rounded_mantissa(decimals = 0)
    sign * 2 ** exponent * mantissa.binary_round(decimals)
  end

private

  def compute_mantissa_and_exponent
    if self == 0
      @mantissa = 0.0
      @exponent = 0.0
      return
    end

    x = self.abs
    exponente = 0

    while (x < 1.0 or x >= 2.0)
      if x < 1.0
        exponente = exponente - 1
        x = x * 2
      elsif x >= 2.0
        exponente = exponente + 1
        x = x / 2
      end
    end

    @exponent = exponente
```

```ruby
    @mantissa = x
  end
end

class Numeric
  def to_single_precision(decimals)
    FloatPrecisionDecorator.new(self.rounded_mantissa(decimals), decimals)
  end

  def single(decimals = 23)
    decimals > 52 ? self : to_single_precision(decimals)
  end
end

class FloatPrecisionDecorator
  def initialize(inner, decimals)
    @decimals = decimals
    @inner = inner.to_f
  end

  def to_f
    @inner
  end

  def ==(other)
    self.to_f == other.to_f
  end

  def method_missing(m,*x)
    # todas las operaciones sobre el numero se ejecutan sobre el float verdadero
    # y se obtiene el verdadero resultado con precision completa del Float original

    if x.size > 0
      if FloatPrecisionDecorator === x.first
        verdadero_resultado = @inner.send(m,x.first.instance_variable_get(:@inner))
      else
        verdadero_resultado = @inner.send(m,*x)
      end
    else
      verdadero_resultado = @inner.send(m,*x)
    end
    # si es numeric, truncar y wrappear
    if Numeric === verdadero_resultado
      # se reduce la precision, multiplicando por el factor, redondeando y volviendo a
dividir
      reduced = verdadero_resultado.to_f.rounded_mantissa(@decimals)
      FloatPrecisionDecorator.new(reduced, @decimals)
    else
      # si no, devolve el resultado como es
      verdadero_resultado
    end
  end

  def coerce(other)
    return self, other
  end

  def to_s
    @inner.to_s
  end

  def inspect
    # para que llame al inspect del float decorado
```

```
    @inner.inspect
  end
end

# encuentra la raiz de x = f[x] usando la tecnica del punto fijo
# hasta obtener un error menor que el especificado comparado con
# un valor de referencia
def punto_fijo(f, stop, inicial = 0)
  # asignamos a x el valor inicial
  x = inicial
  xprev = x

  # iteramos infinitamente mientras la diferencia en valor absoluto
  # entre el valor de referencia y x sean superiores al error maximo
  while not stop[x,xprev]
    # en cada iteracion, asignamos a x el resultado de evaluar f[x] con el x anterior
    # como podemos prescindir del x anterior pisamos la misma variable
    # si evaluaramos los errores usando las diferencias entre distintos x de la
secuencia
    # utilizariamos otra variable mas el x del ciclo anterior
    xprev = x
    x = f[x]
  end

  # devolvemos el ultimo x calculado
  return x
end

# encuentra la raiz de f[x] = 0 usando la tecnica de newton rapson
# la cual define una funcion cuyo punto fijo es tambien la raiz d f
# es necesario pasar como parametro la derivada
# de la funcion y el criterio de parada que usara el metodo del punto fijo

def newton_rapson(f, fd, stop, inicial = 0)
  # se plantea una funcion cuyo punto fijo es tambien
  # raiz de f[x], segun como lo estipula el metodo de new rapson
  g = lambda{|x| x - f[x]/fd[x] }
  return punto_fijo(g, stop, inicial)
end

# Calcula la raiz exponente de un numero usando el algoritmo iterativo
# de newton-rapson
#
# Ej:
#
#   print "raiz cuadrada de dos: ", raiz(2,2), "\n"
#   print "raiz cuadrada de tres: ", raiz(3,2), "\n"

def raiz(valor,exponente,referencia, contador = nil)
  # planteamos una funcion cuya raiz es tambien la raiz que intentamos aproximar
  funcion = lambda{|x| contador.call;
       x**(exponente) - valor} # NOTA: ** significa elevar x al exponente

  # definimos la derivada de la funcion de la cual queremos obtener el punto fijo
  # ya que el metodo de newton rapson requiere ese parametro tambien
  if exponente == 2
    derivada = lambda{|x| exponente*x }
  else
    derivada = lambda{|x| exponente*x**(exponente-1) }
  end

  # calculamos el valor inicial para usar en el metodo de punto fijo
  # como el promedio entre el rango en el que se supone estara el resultado
```

```
  # (ejemplo: la raiz cuadrada de dos esta entra 1 y 2)
  maximo_valor = valor
  minimo_valor = 1
  inicial = (maximo_valor + minimo_valor) / 2

  # invocar el metodo de newton rapson
  return newton_rapson(funcion, derivada, lambda{|x,xprev| ((x-referencia)/x).abs <
0.0001 }, inicial)
end

require "timeout"
iteraciones = 0

{53 => "doble", 23 => "simple simulada", 14 => "custom 14 bits", 10 => "media
simulada"}.each do |k,v|
timeout(10) do

  print "con precision #{v} (#{k} bits de mantisa)\n"

  contador = lambda{ iteraciones = iteraciones + 1}

  r23 = raiz(3.0.single(k),2.0.single(k), 3**0.5, contador)
  print "raiz cuadrada de tres: #{r23} resuelto con #{iteraciones} iteraciones\n"

  iteraciones = 0
  r25 = raiz(5.0.single(k),2.0.single(k), 5**0.5, contador)
  print "raiz cuadrada de cinco: #{r25}, resuelto con #{iteraciones} iteraciones\n"

  iteraciones = 0
  r33 = raiz(3.0.single(k),3.0.single(k), 3**(1/3.0), contador)
  print "raiz cubica de tres: #{r33}, resuelto con #{iteraciones} iteraciones\n"

  print "\n"
end
end
```

Codigo del ejercicio 2

```
class Float
  def binary_round(decimals = 0)
    factor = 2**decimals
    (self * factor).round.to_f / factor
  end

  # extrae la mantisa del numero
  def mantissa
    compute_mantissa_and_exponent unless @mantissa
    @mantissa
  end

  def exponent
    compute_mantissa_and_exponent unless @exponente
    @exponent
  end

  def sign
    if self < 0
      -1
    elsif self > 0
      1
    else
      0
    end
  end
```

```ruby
  def rounded_mantissa(decimals = 0)
    sign * 2 ** exponent * mantissa.binary_round(decimals)
  end

private

  def compute_mantissa_and_exponent
    if self == 0
      @mantissa = 0.0
      @exponent = 0.0
      return
    end

    x = self.abs
    exponente = 0

    while (x < 1.0 or x >= 2.0)
      if x < 1.0
        exponente = exponente - 1
        x = x * 2
      elsif x >= 2.0
        exponente = exponente + 1
        x = x / 2
      end
    end

    @exponent = exponente
    @mantissa = x
  end
end

class Numeric
  def to_single_precision(decimals)
    FloatPrecisionDecorator.new(self.rounded_mantissa(decimals), decimals)
  end

  def single(decimals = 23)
    decimals > 52 ? self : to_single_precision(decimals)
  end
end

class FloatPrecisionDecorator
  def initialize(inner, decimals)
    @decimals = decimals
    @inner = inner.to_f
  end

  def to_f
    @inner
  end

  def ==(other)
    self.to_f == other.to_f
  end

  def method_missing(m,*x)
    # todas las operaciones sobre el numero se ejecutan sobre el float verdadero
    # y se obtiene el verdadero resultado con precision completa del Float original

    if x.size > 0
      if FloatPrecisionDecorator === x.first
        verdadero_resultado = @inner.send(m,x.first.instance_variable_get(:@inner))
```

```
      else
        verdadero_resultado = @inner.send(m,*x)
      end
    else
      verdadero_resultado = @inner.send(m,*x)
    end
    # si es numeric, truncar y wrappear
    if Numeric === verdadero_resultado
      # se reduce la precision, multiplicando por el factor, redondeando y volviendo a
dividir
      reduced = verdadero_resultado.to_f.rounded_mantissa(@decimals)
      FloatPrecisionDecorator.new(reduced, @decimals)
    else
      # si no, devolve el resultado como es
      verdadero_resultado
    end
  end

  def coerce(other)
    return self, other
  end

  def to_s
    @inner.to_s
  end

  def inspect
    # para que llame al inspect del float decorado
    @inner.inspect
  end
end

# efectua la sumatoria de los terminos de una sucesion hasta
# que se acerque suficiente a un referencia
# el tercer parametro, es el criterio de parada que debera determinar
# en funcion de x y de n si se debe parar o no
def serie(sucesion, stop)
  x = 0.0
  n = 0
  while not stop[x,n]
    x = x + sucesion[n]
    n = n + 1
  end

  x
end


def factorial(n)
  n > 1 ? (2..n).inject(&:*) : 1
end

# efectua la suma de taylor dada la derivada enesima en el punto x0
# y la diferencia entre x-x0 hasta que se acerque suficiente a una referencia
# el tercer parametro, es el criterio de parada
def taylor(derivada_n, diferenciax0, stop)
  # crear los terminos de la sumatoria de taylor en funcion de n
  sucesion = lambda{|n|
      derivada_n[n] * diferenciax0 ** n / factorial(n) }
  # efectuar la sumatoria de la serie
  serie(sucesion, stop)
end
```

```ruby
require "timeout"
{53 => "doble", 23 => "simple simulada", 14 => "custom 14 bits", 10 => "media
simulada"}.each do |k,v|

timeout(10) do

  print "Con precision #{v} (#{k} bits de mantisa):\n"

  ops = 0
  iteraciones = 0

  # la derivada enesima de sin(x) evaluada en pi/2
  derivada_n_sin = lambda{|n|
    iteraciones = iteraciones + 1
    if n%2 == 0 # si el numero es par
      if n%4 == 0
        1.0.single(k)
      else
        -1.0.single(k)
      end
    else # si el numero es impar, vale cero
      0.0.single(k)
    end
  }

  # x0 = pi/2
  print "sin(pi/3): "
  referencia = Math::sin(Math::PI/3.0)
  print taylor(derivada_n_sin, - Math::PI/6.0,  lambda{|x,n| ((x-referencia) / (x ==
0.0 ? 0.01 : x) ).abs < 0.0001}), " resuelto con #{iteraciones} iteraciones\n"

  iteraciones = 0
  # la derivada enesima de cos(x) evaluada en pi/2
  derivada_n_cos = lambda{|n|
    iteraciones = iteraciones + 1
    if n%2 == 0 # si el numero es par, vale cero
      0.0.single(k)
    else # si el numero es impar
      if n%4 == 1
        -1.0.single(k)
      else
        1.0.single(k)
      end
    end
  }
  # x0 = pi/2
  print "cos(pi/3): "
  referencia = Math::cos(Math::PI/3.0)
  print taylor(derivada_n_cos, - Math::PI/6.0,  lambda{|x,n| ((x-referencia) / (x ==
0.0 ? 0.01 : x) ).abs < 0.0001}), " resuelto con #{iteraciones} iteraciones\n"
  print "\n"
end

end
```

Codigo del ejercicio 3

```ruby
print "numeros primos entre el 1 y el 1000:\n"

criba = Array.new
2.upto(999) do |numero|
  # es primo hasta que se demuestre lo contrario
```

```ruby
  criba[numero] = true
end

2.upto(32) do |numero|
  # marcar todos los multiplos menores que 100
  # si el numero es primo
  if criba[numero]
    i = 2*numero
    while (i < 1000)
      criba[i] = false # no es primo
      i = i + numero
    end
  end
end

primos = (2..999).select(&criba.method(:[]))
2.upto(999) do |numero|
  if criba[numero]
    print numero, '-'
  end
end
print "\n"
```

Resultados

Resultado del ejercicio 1

```
con precision doble (53 bits de mantisa)
raiz cuadrada de tres: 1.7321428571428572 resuelto con 2 iteraciones
raiz cuadrada de cinco: 2.2360688956433634, resuelto con 3 iteraciones
raiz cubica de tres: 1.442351584357819, resuelto con 3 iteraciones

con precision simple simulada (23 bits de mantisa)
raiz cuadrada de tres: 1.7321428060531616 resuelto con 5 iteraciones
raiz cuadrada de cinco: 2.2360689640045166, resuelto con 3 iteraciones
raiz cubica de tres: 1.4423515796661377, resuelto con 3 iteraciones

con precision custom 14 bits (14 bits de mantisa)
raiz cuadrada de tres: 1.73211669921875 resuelto con 5 iteraciones
raiz cuadrada de cinco: 2.236083984375, resuelto con 3 iteraciones
raiz cubica de tres: 1.44232177734375, resuelto con 3 iteraciones

con precision media simulada (10 bits de mantisa)
ejercicio1.rb:4:in `binary_round': execution expired (Timeout::Error)
```

Lo mas importante y llamativo de la salida del ejercicio 1 son la cantidad de iteraciones utilizadas para calcular la raiz cuadrada de tres que varia segun la precision utilizada. No fue posible calcular ninguna raiz cuando se uso 10 bits de mantisa

Resultado del ejercicio 2

```
Con precision doble (53 bits de mantisa):
sin(pi/3): 0.8660538834157472 resuelto con 5 iteraciones
cos(pi/3): 0.5000021325887924 resuelto con 6 iteraciones

Con precision simple simulada (23 bits de mantisa):
sin(pi/3): 0.8660539388656616 resuelto con 5 iteraciones
cos(pi/3): 0.5000021457672119 resuelto con 6 iteraciones

Con precision custom 14 bits (14 bits de mantisa):
sin(pi/3): 0.866058349609375 resuelto con 5 iteraciones
cos(pi/3): 0.4999847412109375 resuelto con 6 iteraciones

Con precision media simulada (10 bits de mantisa):
sin(pi/3): ejercicio2.rb:138:in `*': execution expired (Timeout::Error)
```

Esta vez lo unico llamativo es el hecho de que tampoco se pudo aproximar con la cota de error buscada cuando se usa una precision con solo 10 bits de mantisa

Resultado del ejercicio 3

```
numeros primos entre el 1 y el 1000:
2-3-5-7-11-13-17-19-23-29-31-37-41-43-47-53-59-61-67-71-73-79-83-89-97-101-
103-107-109-113-127-131-137-139-149-151-157-163-167-173-179-181-191-193-197-
199-211-223-227-229-233-239-241-251-257-263-269-271-277-281-283-293-307-311-
313-317-331-337-347-349-353-359-367-373-379-383-389-397-401-409-419-421-431-
433-439-443-449-457-461-463-467-479-487-491-499-503-509-521-523-541-547-557-
563-569-571-577-587-593-599-601-607-613-617-619-631-641-643-647-653-659-661-
673-677-683-691-701-709-719-727-733-739-743-751-757-761-769-773-787-797-809-
811-821-823-827-829-839-853-857-859-863-877-881-883-887-907-911-919-929-937-
941-947-953-967-971-977-983-991-997-
```

Esta salida no aporta datos interesantes

Conclusion

Menor precision (bits en la mantisa) en el tipo de dato de punto flotante utilizado aumenta los errores de redondeo tras cada operacion, en algoritmos numericos iterativos estos errores se propagaran a travez de las iteraciones haciendo que se requieran mas iteraciones para llegar al valor buscado. El mejor ejemplo es al calcular la raiz cuadrada de tres, al usar doble precision se requirieron 2 iteraciones mientras que al usar simple precision se necesitaron 5, las demas raices cuadradas no necesitaron mas iteraciones cuando se bajo la precision, esto demuestra que no siempre tiene porque ser asi, aunque la minima cantidad de iteracioens necesarias sera mayor si la precision es menor.

En casos extremos, cuando la precision es infinita, las iteraciones necesarias para llegar al resultado son las teorizadas simbolicamente, cuando la precision es tan baja (por ejemplo, al usar una mantisa de 10 bits) como para que no se pueda representar la diferencia en fraccion que determina el error de truncamiento buscado, el criterio de corte puede no darse nunca.

En el ejercicio 2 tambien puede observarse que con una precision media de 10 bits en la mantisa el criterio de corte tampoco se da, sin embargo en los demas niveles de precision en los cuales el metodo da resultados la cantidad de iteraciones no varian entre uno y otro (a diferencia de lo que ocurre en el ejercicio 1), la razon posible de esto es que la ecuacion de newton-rapson usada en el ejercicio 1 que implica una resta de valores similares propaga mas los errores de redondeo en comparacion a los terminos de taylor.