Extended Introduction to Computer Science
CS1001.py , Lecture 9

Numeric Integral; Floating Point Arithmetic; Finding Zeroes of Real Functions

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Amir Gilad, Michal Kleinbort

Founding Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Spring Semester, 2017
http://tau-cs1001-py.wikidot.com

Lecture 8 Highlights

- The integral as a high order function; Numerical definite integrals.
- Floating point arithmetic and its wonders.
- Finding roots of real valued functions: Binary search, and Newton-Raphson iteration.

Finding Roots of a Real Valued Function, Take 1 *

You are given a black box that computes a real valued function, $f(x)$.

You are asked to find a root of $f(x)$ (namely a value a such that $f(a) == 0$ or at least $|f(a)| < \varepsilon$ for a small enough ε).

What can you do?

Not much, I'm afraid. Just go over points in some arbitrary/random order, and hope to hit a root.

*thanks to Prof. Sivan Toledo for helpful suggestions and discussions related to this part of the lecture

Finding Roots of Real Valued Function, Take 2

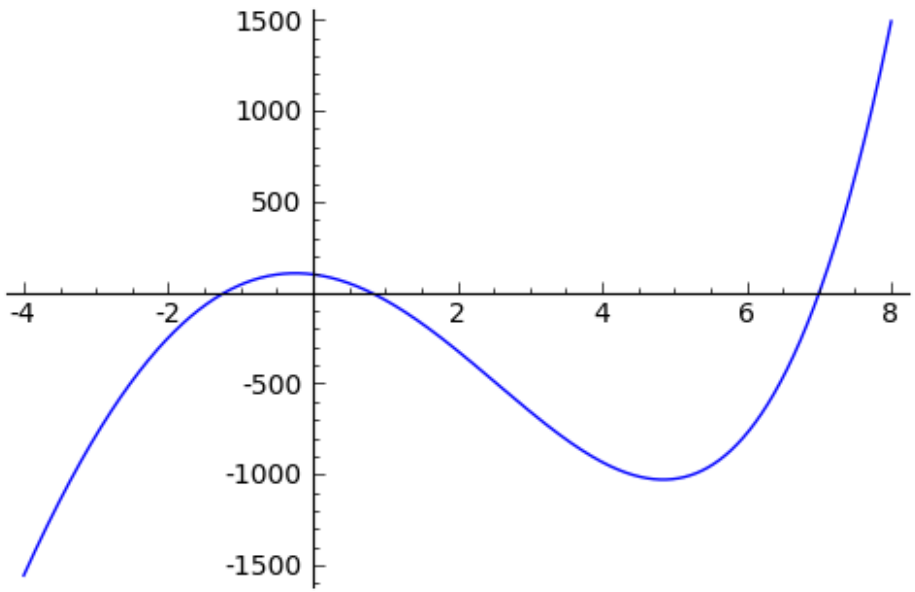
You are given a black box to compute the real valued function $f(x)$. On top of this, you are told that $f(x)$ is continuous, and you are given two values, L and U, such that $f(L) < 0 < f(U)$.

You are asked to find a root of $f(x)$ (namely a value a such that $f(a) == 0$ or at least $|f(a)| < \varepsilon$ for a small enough ε).

What can you do?

The Intermediate Value Theorem

Suppose that $f(x)$ is a continuos real valued function, and $f(L) < 0 < f(U)$ (where $L < U$, and both are reals). The intermediate value theorem (first year calculus) claims the existence of an intermediate value, C, $L < C < U$, such that $f(C) = 0$. There could be more than one such root, but the theorem guarantees that at least one exists.



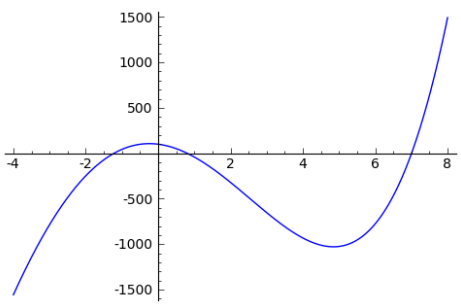
For example, in this figure, $f(-4) < 0 < f(8)$, so there is a C, $-4 < C < 8$, such that $f(C) = 0$ (in fact there are three such C .

Root Finding Using Binary Search

Suppose that $f(x)$ is a continuos real valued function, and $f(L) < 0 < f(U)$ (where $L < U$). Compute $M = f((L+U)/2)$.

- If $M = 0$, then M is a root of $f(x)$.
- If $M < 0$, then by the intermediate value theorem, there is a root of $f(x)$ in the open interval $((L+U)/2, U)$.
- If $M > 0$, then by the intermediate value theorem, there is a root of $f(x)$ in the open interval $(L, (L+U)/2)$.

Looks familiar?



By performing binary search on the interval, we converge to a root of $f(x)$ (will stop when $|M| < \varepsilon$).

Finding Roots of Real Valued Function, Take 3

You are given a black box that on input x outputs the value $f(x)$. On top of this, you are told that $f(x)$ is differentiable (is smooth enough to have a derivative), and you also get access to a black box that on input x outputs the value $f'(x)$.

Your mission, should you choose to accept it, is to find a root of $f(x)$ (namely a value a such that $f(a) == 0$ or at least $|f(a)| < \varepsilon$ for a small enough ε).

What can you do?
(here, we'll start discussing the Newton-Raphson iteration.)

The Newton–Raphson (Isaac and Joseph) Method (1685–1690)

Let $f(x)$ be a real valued function of one variable, which it is differentiable everywhere.

We seek a root of the equation $f(x) = 0$.

Denote, as usual, the derivative of f at point x by $f'(x)$.

The Newton–Raphson method, or iteration, starts with some initial guess, denoted x_0, and iteratively computes

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

until some halting condition is met.

A Geometric Interpretation of Newton–Raphson Method

The Newton–Raphson method has a geometric interpretation:
Let x_n be the current approximation to the root of $f(x) = 0$ (including the initial guess, x_0). The next approximation, x_{n+1}, is the intersection point of the x-axis with the tangent to f at the point x_n (why?).

The Newton–Raphson Method, cont.

The Newton–Raphson method, or iteration, starts with some initial guess, denoted x_0, and iteratatively computes

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

until some halting condition is met.

It is easy to see that if the sequence converges to some limit, x_∞, then $f(x_\infty) = 0$. But it is far from obvious why convergence should occur at all.

Under mild conditions on the function (*e.g.* existence of a root) and the starting point, it can be shown that the process does converge to a root of $f(x)$.

Convergence Rate of the Newton–Raphson Method

It can be rigorously shown that if $f'(x_0) \neq 0$ and the root is of multiplicity one[†], then convergence in the neighborhood of a root holds, and furthermore has a quadratic rate. Roughly speaking, the number of correct digits doubles at each iteration.

The claim guarantees that under reasonable initial conditions, this sequence of tangents' intersection points converges to a root (and even converges fast).

The proof uses the Taylor series expansion of $f(x)$, close to a root. For obvious reasons (*e.g.* insufficient background), the proof will not be given or even attempted here.

Incidentally, the version we use is not due to Newton, but to the much less known Raphson. (Joseph Raphson, approx. 1648–1715).

[†] $f(x) = x \cdot (x-3)^2$ has two roots, 0 and 3. 0 has multiplicity one, while 3 has multiplicity two.

Newton–Raphson in Python

```
from random import *
def NR(func, deriv=None, epsilon=10**(-8), n=100, x0=None):
    """ Given a real valued func and its real value derivative,
    deriv,NR attempts to find a root, using Newton-Raphson method.
    NR starts with a an initial x0 and performs n=100 iterations.
    If the absolute value function on some x_i is smaller
    than epsilon, None is the returned value """
    if deriv is None:
        deriv = diff_param(func)
    if x0 is None:
        x0 = uniform(-100.,100.)
    x=x0; y=func(x)
    for i in range(n):
        if abs(y)<epsilon:
            print ("x=",x,"f(x)=",y,"convergence in",i, "iterations
            return x
        elif abs(deriv(x))<epsilon:
            print ("zero derivative, x0=",x0," i=",i, " xi=", x)
            return None
        else:
            print("x=",x,"f(x)=",y)
            x = x - func(x)/deriv(x)
            y = func(x)
    print("no convergence, x0=",x0," i=",i, " xi=", x)
    return None
```

Newton–Raphson in Python: Some Comments

The function `NR` is a "high order function" in the sense that two of its arguments, `func` and `deriv`, are themselves functions (from reals to reals).

The function `NR` "assumes" that `deriv` is indeed the derivative of `func`. We supply a default argument, the good-ole `diff_param` of the function (numeric derivative). Do check the mechanism by which we do this (not the first one that comes to mind).

If you feed `NR` with functions that do not satisfy this relations, there is no reason why will return a root of `func`.

If an argument `x` for which the derivative is very close to `0` is encountered, `NR` prints a warning and returns `None`.

Newton–Raphson in Python: Some More Comments

The default argument `n=100` determines the number of iterations, `epsilon=10**(-8)` the accuracy, and `x0=None` the starting point.

Having `x0=None` as the starting point leads to executing the code with `x0=uniform(-100.,100.)` (a random floating point number, uniformly distributed in the interval [-100.0,+100.0]). Therefore, different computations may well converge to different roots (if such exist).

Syntactically, we could set the default directly as `x0=uniform(-100.,100.)`. This does not work in Python (trust, but check!), and you may be asked to explain why.

Newton–Raphson in Python: Execution Examples

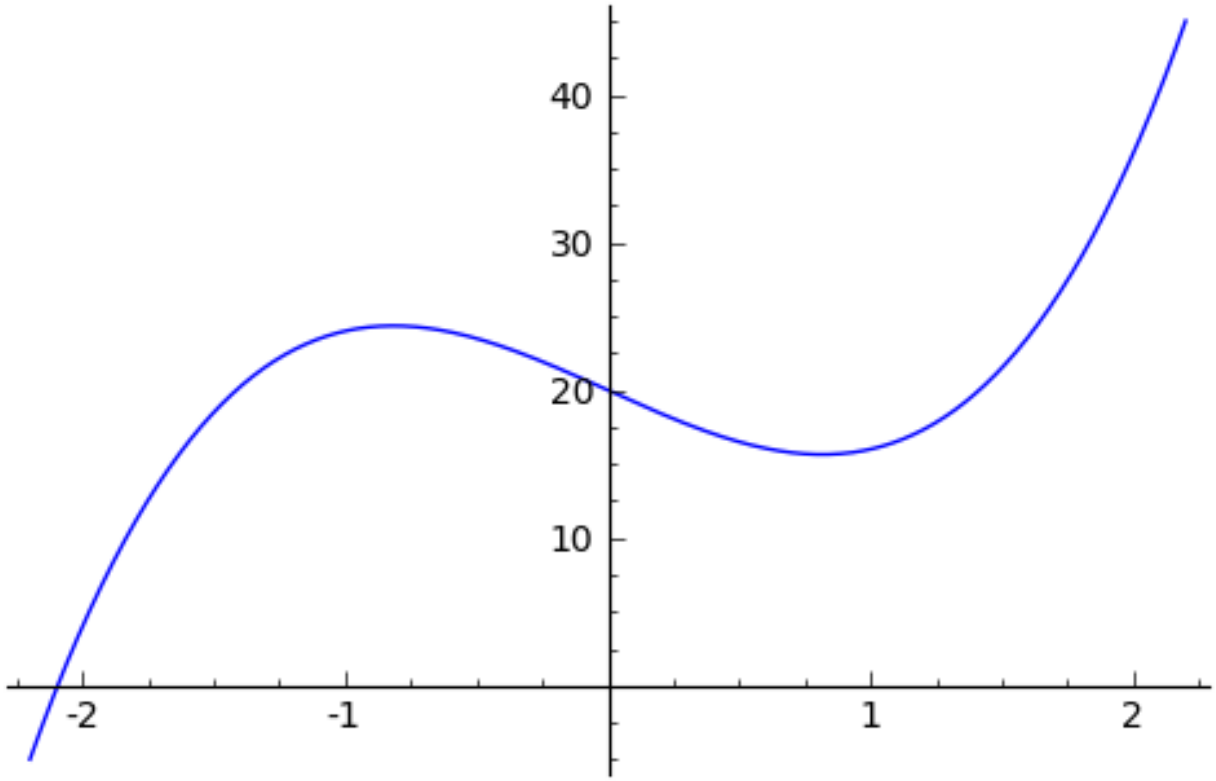
First, we run our function NR using the function $f(x) = x^2 + 2x - 7$, whose derivative is $f(x) = 2x + 2$. Observe that $f(x)$ has two real roots, $-1 - 2\sqrt{2} = -3.828427124, -1 + 2\sqrt{2} = 1.828427124$, and its derivative has a root at $x = -1$. Here, we decided to supply the two functions to NR as anonymous functions, using lambda expressions.

```
>>> NR(lambda x: x**2+2*x-7,lambda x:2*x+2)
x= 1.828427124746752 f(x)= 3.178790564106748 2e-12
convergence in 9 iterations
>>> NR(lambda x: x**2+2*x-7,lambda x:2*x+2)
x= -3.828427124759005 4 f(x)= 7.249489897276362e-11
convergence in 8 iterations
>>> NR(lambda x: x**2+2*x-7,lambda x:2*x+2)
1.828427124746 19
>>> NR(lambda x: x**2+2*x-7,lambda x: 2*x+2,x0=-1)
zero derivative, x0= -1  i= 0  xi= -1
>>> (lambda x: 2*x+2)(-1)
0
>>> NR(lambda x: x**2+2*x-7,lambda x:2*x+2, x0=-1.0000001)
x= -3.828427124746190 3 f(x)= 8.881784197001252e-16
convergence in 29 iterations
>>> NR(lambda x: x**2+2*x-7,lambda x: 2*x+2,x0=-2)
-3.828427125049864 3
```

Newton–Raphson in Python: A Second Example

Now, consider the function $4x^3 - 8x + 20$.

In the interval $[-2.2, 2.2]$ it looks as following:



The derivative of f is $f'(x) = 12x^2 - 8$. f has a local maximum at $x = -\sqrt{2/3} = -0.816$ and a local minimum at $x = \sqrt{2/3} = 0.816$.

We will try both $x_0 = 2$ and random starting points (`NR` default).

Newton–Raphson in Python: A Second Example

```
>>> NR(lambda x: 4*x**3-8*x+20, lambda x: 12*x**2-8, x0=2.)
x= -2.09455148175312 8 f(x)= -9.4113943305274e-09
convergence in 7 iterations

>>> NR(lambda x: 4*x**3-8*x+20,lambda x: 12*x**2-8)
                        # random starting point
x= -2.09455148154235 3 f(x)= -1.1830536550405668e-12
convergence in 32 iterations

>>> NR(lambda x: 4*x**3-8*x+20)  # default derivative
x= -2.0945514815701 3 f(x)= -1.2413075012318586e-09
convergence in 29 iterations
```

Newton–Raphson in Python: More Execution Examples

We now run our function NR using the function $f(x) = x^2 + 2x + 7$, whose derivative is $f(x) = 2x + 2$ as well. Observe that $f(x)$ has no real roots. Again, we supply the two functions to NR as anonymous functions, using lambda expressions.

```
>>> NR(lambda x: x**2+2*x+7,lambda x: 2*x+2)
no convergence, x0= 0  i= 99  xi= -0.29801393414

>>> NR(lambda x: x**2+2*x+7,lambda x: 2*x+2,n=1000)
no convergence, x0= 0  i= 999  xi= 10.9234003098

>>> NR(lambda x: x**2+2*x+7,lambda x:2*x+2,n=100000)
no convergence, x0= 29.3289256937  i= 99999  xi= 3.61509706324
# not much point in going on

>>> NR(lambda x: x**2+2*x+7,lambda x: 2*x+2,x0=-1)
zero derivative, x0= -1  i= 0  xi= -1
```

Newton–Raphson in Python: sin_by_million Example

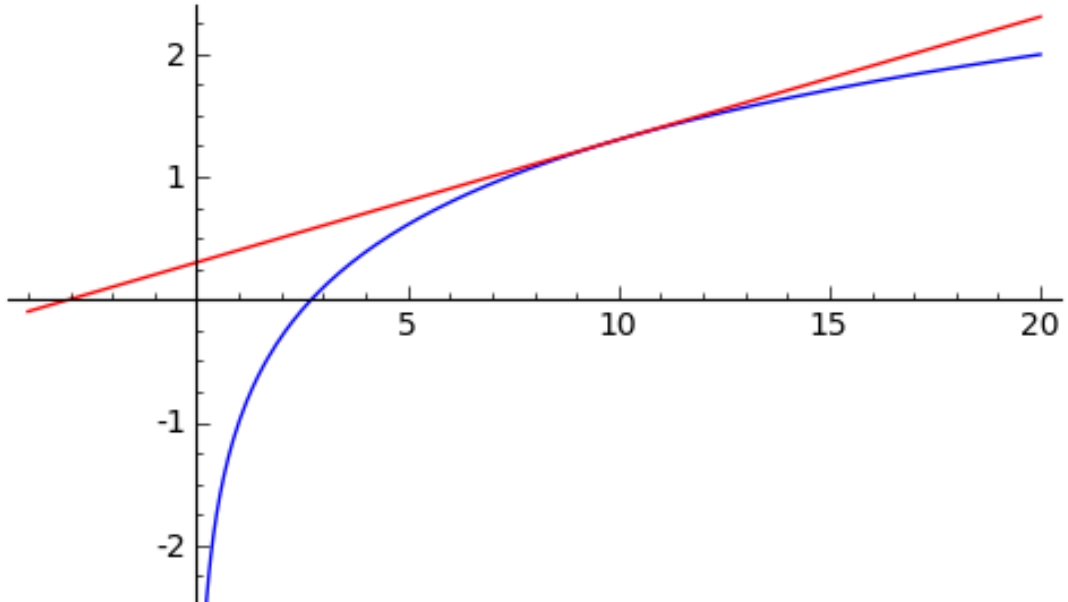
We now run our function NR using the beloved function $f(x) = \sin(10^6 x)$, whose derivative is $f'(x) = 10^6 \cos(10^6 x)$.

```
def sin_by_million(x):
    return math.sin(10**6*x)
def sin_by_million_deriv(x):
    return 10**6*math.cos(10**6*x)
#We start with the numeric derivative
>>> NR(sin_by_million, diff_param(sin_by_million))
89.2687780627 0.964613148463
...
x= 53.526379440327254 f(x)= -0.7126804756085253
no convergence, x0= 53.42557187766076  i= 99

#Let's try to increase the accuracy in the numeric derivation
>>> NR(sin_by_million, diff_param(sin_by_million,h=0.000001))
x= -0.0941671530286782 9 f(x)= -0.8698163554715612
...
x= -0.094166098198704 3 f(x)= -3.8425752320545916e-09
convergence in 10 iterations

#now let's use the symbolic derivative
>>> NR(sin_by_million, sin_by_million_deriv)
x= 32.24980202842823 5 f(x)= 0.5539528640085607
...
x= 32.2498026155329 3 f(x)= -2.4213945229881345e-09
convergence in 3 iterations
```

Cases Where the Newton–Raphson Method Fails

There are cases where the method fails to find a root, despite the fact that a real root does exist. A specific example is

- $f(x) = \ln x - 1$. This example is somewhat pathological since it is defined (over the reals) only for $x > 0$. Its root is $x = e = 2.718281\ldots$. If the starting point, x_0, satisfies $0 < x_0 < e$, the iteration will converge to the root. However, if $x_0 > e$, the intersection of the tangent with the x-axis is negative, we get into complex values, and never converge to a solution.



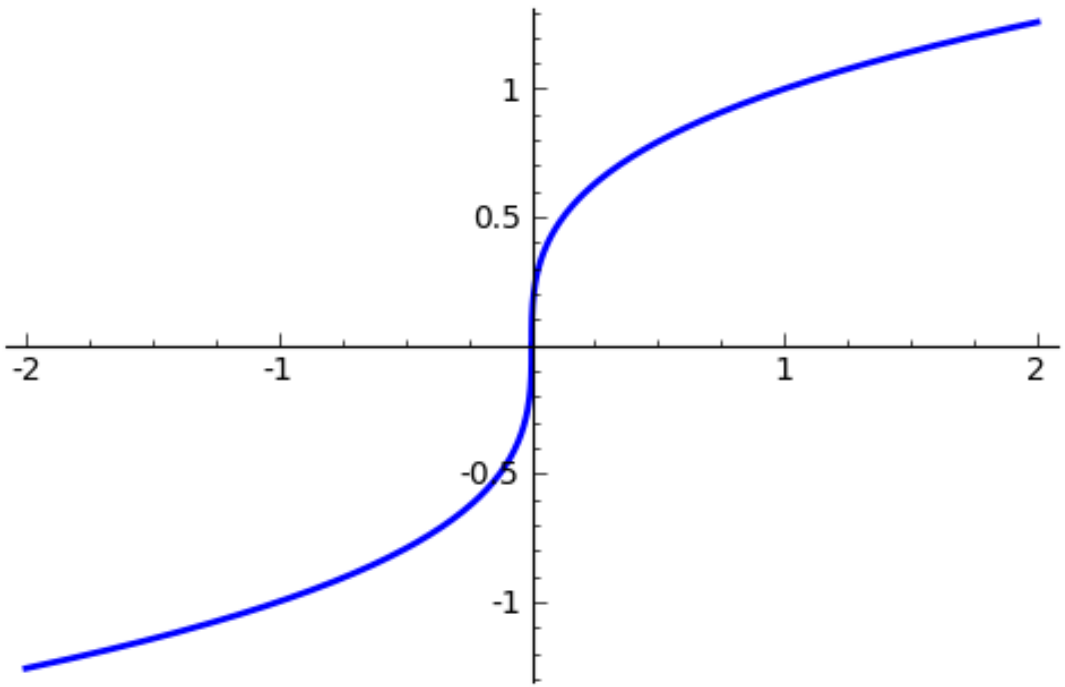
Cases Where the Newton–Raphson Method Fails: The $f(x) = \ln x - 1$ function. Execution Examples

```
>>> NR(lambda x: math.log(x) - 1, x0=1.5)
        # initial point smaller than e

x= 2.7182818408322724 f(x)= 4.551856003232046 5e-09
     convergence in 4 iterations

>>> NR(lambda x: math.log(x) - 1, x0=10)
        # initial point larger than e

x= 10 f(x)= 1.302585092994046
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    NR(lambda x: math.log(x) - 1, x0=10)
  File "/Users/benny/Dropbox/Intro_CS_Course/IntroCS_2017/Code2017/
    y = func(x)
  File "<pyshell#10>", line 1, in <lambda>
    NR(lambda x: math.log(x) - 1, x0=10)
ValueError: math domain error
```

Cases Where the Newton–Raphson Method Fails (2)

There are cases where the method fails to find a root, despite the fact that a real root does exist. A specific example is

- $f(x) = \sqrt[3]{x}$.



This example is also somewhat pathological, since the derivative at the root, $x = 0$, is $+\infty$.

Cases Where the Newton–Raphson Method Fails: The $f(x) = \sqrt[3]{x}$ function. Execution Examples

```
>>> NR(lambda x: x**(1/3))
zero derivative, x0= 17.583962736808118
i= 34  xi= (307104697948.0005-4569312853.5441675j)
     #x is complex. go figure

>>> NR(lambda x: x**(1/3))
zero derivative, x0= -95.35484532133836
i= 32  xi= (-384401949315.69366-3473926036.0956345j)
     #x is complex. go figure
```

The derivative is $f'(x) = \frac{1}{3} \cdot x^{-2/3}$, and it converges to 0 for values of x with large absolute values (both negative and positive). On top of this, Python tends to choose the cubic roots that are complex numbers (and not the real valued ones), which leads to this weird behavior.

Cases Where the Newton–Raphson Method Fails (3)

There are other cases where the method fails to find a root, despite the fact that a real root does exist.

- If we run in the course of the iteration into an x_i where the derivative is zero, $f'(x_i) = 0$.

- There are even polynomials with bad starting points. Usually one has to work hard to find them, and we won't even try.

A Distorted Newton–Raphson Method

Suppose we cruelly modify the Newton–Raphson method, as following: Let h(x) be any real valued function.

Start with some initial guess, denoted x_0, and iteratatively computes

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

until some halting condition is met.

It is still true that if the sequence converges to some limit, x_∞, then $f(x_\infty) = 0$. But it is even further from obvious if convergence holds.

A Distorted Newton–Raphson Method: Examples

```
>>> def f(x):
    return x**7+30*x**4+5*x-100
>>> def g(x):     # the proper derivative of f
    return 7*x**6+120*x**3+5
>>> def h(x):     # a cousin of the derivative of f
    return 7*x**6+10*x**3+55
>>> def k(x):    # a remote relative of the derivative of f
    return x**6+x**2+7

>>> NR(f,g)
x= -3.061222690393527 f(x)= -5.115907697472721e-13
convergence in 27 iterations
>>> NR(f,h)
x= -3.061222690397287 f(x)= -8.735909773349704e-09
convergence in 51 iterations
>>> NR(f,h)
x= -3.061222690397729 f(x)= -9.762217700881592e-09
convergence in 67 iterations
>>> NR(f,k)
no convergence, x0= 65.37916743242721  i= 99
```

Distorted Newton–Raphson Method: More Examples

```
>>> def f(x):
    return x**7+30*x**4+5*x-100
>>> def g(x):     # the proper derivative of f
    return 7*x**6+120*x**3+5
>>> def l(x):    # not even a relative of g
    return 5*x**4+x**2+77

>>> NR(f,l)
no convergence, x0= 0.561285440908  i= 99
>>> NR(f,l)
Traceback (most recent call last):
  File "<pyshell#98>", line 1, in <module>
    NR(f,l)
  File "/Users/benny/Documents/InttroCS2011/Code/Intro12/lecture12_
    y=func(x)
  File "<pyshell#14>", line 2, in f
    return x**7+30*x**4+5*x-100
OverflowError: (34, 'Result too large')
```

So apparently, to have convergence, the "fake derivative" should be reasonably close to the true one. Enough is enough!

Generalization of the Newton–Raphson Method

Newton–Raphson uses the first order derivative of a differentiable function, $f(x)$.

If $f(x)$ has derivatives of higher order (*e.g.* 2nd order, 3rd order, etc.), there are improved root finding methods that employ them, and typically achieve faster convergence rates.

These methods are generally known as the class of Householder's methods. We will not discuss them here.