Newton’s Method From The Ground Up

Peter Amerkhanian

2023-11-03

import numpy as np  
import os  
import matplotlib.pyplot as plt  
import matplotlib  
import sympy  
import imageio  
from typing import Callable  
import seaborn as sns

## 1 Newton’s Method for Finding Roots: What, How, and Why

In the context of a Differential Calculus course, [Newton’s Method](https://en.wikipedia.org/wiki/Newton%27s_method#Applications), also referred to as The Newton-Raphson Method, seems to typically come up near the end of the semester, offering a brief look into the world of “numerical methods” and how we might solve complex problems in the real world. I think that it’s a cool topic and I wanted to write an extended blog post about it. The main purpose of this is to ensure that I always have personal reference materials for Newton’s Method, but perhaps it can be helpful to other readers.

I draw on two key sources for thinking about Newton’s Method:

1. [“Newton’s Method.”](https://en.wikipedia.org/w/index.php?title=Newton%27s_method&oldid=1177135929.) 2023. In Wikipedia
2. Strang, Gilbert, and Edwin Herman. 2016. [Calculus Volume 1](https://openstax.org/books/calculus-volume-1/pages/4-9-newtons-method). OpenStax College.

### 1.1 What is it

In many areas of pure and applied mathematics, we are interested in finding solutions to an equation of the form

(Strang and Herman 2016)

Newton’s Method is a numerical method that helps us solve .

There are many cases where we need to solve equations like that, but the application area I work in involves statistical modeling, so I jump to the case where we want to “fit” a line as close as possible to some set of data points, thus creating a model of a data-generating-process. The fitting process rests on getting the line as close to the data points as possible, thus minimizing error. If we can formulate error as a function, then we can minimize it – we differentiate it and set the derivative to 0, and solve . Therein lies the opportunity to apply Newton’s Method to a real problem.

However, we’ll take a step back from statistics and return to the domain of an introductory calculus course. Newton’s Method is useful for finding the root – the x-intercept – of a function. We’ll explore the method by walking through an example

### 1.2 Using Newton’s Method to Solve a Simple Problem

Say we are given the function and we want to find the x-intercept of the function with the domain .

We’ll first set this function up in sympy, a python library for symbolic computation.

x, y = sympy.symbols('x y')  
y = 4\*sympy.sin(x)  
y

This is a problem with a known answer (you can google x-intercepts of for a table), so it’s not particularly useful to use Newton’s Method here, but for our purposes it will be helpful that we can check our answer with the “right” one.

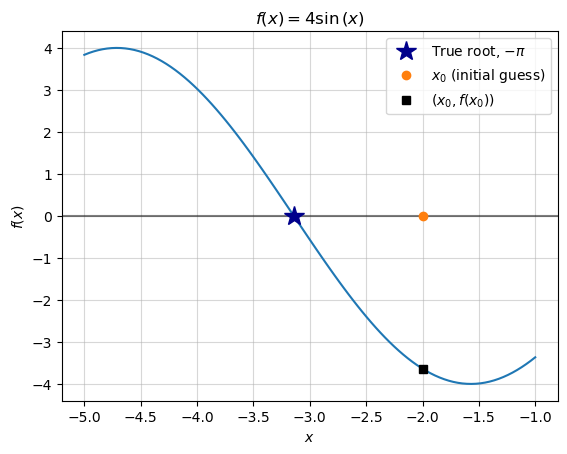
def plot\_f(  
 f: Callable[[float], float],  
 x\_low: int,  
 x\_high: int,  
 ax: matplotlib.axes.Axes) -> None:  
 """  
 Plots a given function f within a specified range on a provided axes.  
  
 Parameters:  
 f (Callable[[float], float]): The function to be plotted.  
 x\_low (int): The lower bound of the x-axis.  
 x\_high (int): The upper bound of the x-axis.  
 ax (matplotlib.axes.Axes): The matplotlib axes object on which the function will be plotted.  
  
 Returns:  
 None  
 """  
 x\_vec = np.linspace(x\_low, x\_high, 100)  
 ax.plot(x\_vec, f(x\_vec))  
  
  
def base\_plot(  
 y: sympy.core.mul.Mul,  
 x: sympy.core.mul.Mul,  
 x\_low: int = -5,  
 x\_high: int = 5) -> None:  
 """  
 Creates a base plot for a mathematical expression and its graph.  
  
 Parameters:  
 y (sympy.core.mul.Mul): The mathematical expression to be plotted.  
 x (sympy.core.mul.Mul): The symbol representing the independent variable in the expression.  
 x\_low (int): The lower bound of the x-axis (default is -5).  
 x\_high (int): The upper bound of the x-axis (default is 5).  
  
 Returns:  
 tuple: A tuple containing the matplotlib figure and axes used for plotting.  
  
 Note:  
 The mathematical expression is first converted to a Python function using sympy.lambdify.  
 The function is then plotted on the specified axes along with gridlines and labels.  
 """  
 f = sympy.lambdify(x, y)  
 fig, ax = plt.subplots()  
 ax.grid(alpha=.5)  
 ax.axhline(0, color="black", alpha=.5)  
 plot\_f(f, x\_low, x\_high, ax)  
 ax.set(title=f"$f(x)={sympy.latex(y)}$", xlabel="$x$", ylabel="$f(x)$")  
 return fig, ax  
  
  
def plot\_truth(ax: matplotlib.axes.Axes) -> None:  
 """  
 Plots the true root of a function as a marker on the graph.  
  
 Parameters:  
 ax (matplotlib.axes.Axes): The matplotlib axes on which the marker will be plotted.  
  
 Returns:  
 None  
 """  
 ax.plot(-np.pi,  
 0,  
 "\*",  
 markersize=15,  
 color="darkblue",  
 label="True root, $-\pi$")  
  
  
def plot\_guess(  
 ax: matplotlib.axes.Axes,  
 guess: int,  
 label: str) -> None:  
 """  
 Plots a guess or estimate as a marker on the graph.  
  
 Parameters:  
 ax (matplotlib.axes.Axes): The matplotlib axes on which the marker will be plotted.  
 guess (int): The estimated value to be marked on the graph.  
 label (str): Label for the marker.  
  
 Returns:  
 None  
 """  
 ax.plot(guess,  
 0,  
 "o",  
 label=label)  
  
  
def plot\_guess\_coords(  
 ax: matplotlib.axes.Axes,  
 guess: int,  
 label: str,  
 y: sympy.core.mul.Mul = y,  
 x: sympy.core.mul.Mul = x):  
 """  
 Plots a guess or estimate with specific coordinates as a marker on the graph.  
  
 Parameters:  
 ax (matplotlib.axes.Axes): The matplotlib axes on which the marker will be plotted.  
 guess (int): The estimated x-coordinate where the marker will be placed.  
 label (str): Label for the marker.  
 y (sympy.core.mul.Mul): The mathematical expression corresponding to the y-coordinate.  
 x (sympy.core.mul.Mul): The symbol representing the independent variable (x).  
  
 Returns:  
 None  
 """  
 ax.plot(guess,  
 y.subs(x, guess).evalf(),  
 "s",  
 label=label, color="black")  
  
  
def euclidean\_dist\_to\_truth(x): return np.sqrt((-np.pi - float(x))\*\*2)

Now we’ll begin the process of using Newton’s Method to arrive at that same answer of . #### Step 1: Make a first guess and evaluate its (x, y) coordinates For a first guess, it’s typical to start close to 0. In our case, we’ll try -2.

x\_0 = -2

Let’s get a sense of how good of a guess this is by plotting it

fig, ax = base\_plot(y, x, x\_low=-5, x\_high=-1)  
plot\_truth(ax)  
plot\_guess(ax, x\_0, "$x\_0$ (initial guess)")  
plot\_guess\_coords(ax, x\_0, "$(x\_0, f(x\_0))$")  
ax.legend();



We can assess the quality of that guess by calculating the distance from the guess to the right answer ():

print(f"error for guess x\_0:", euclidean\_dist\_to\_truth(x\_0))

error for guess x\_0: 1.1415926535897931

#### 1.2.1 Step 2: Find the equation of the tangent line at those coordinates

We will proceed to improve upon that initial guess by computing the linear approximation of the function at that point, then retrieve its root to make a next-guess. Our first guess wasn’t based on any relevant information other than the domain of our search (between -5 and 0). Our next guess is going to be better-informed, as it is an estimate based on an approximation of the function.

(Note that the linear approximation at this point is typically referred to as a tangent line, and I’ll use those two phrases interchangeably.)

We compute the tangent by first differentiating the function and plugging in our previous guess. This yields the slope of the tangent line:

I’ll note that sympy is capable of doing all of these routine calculations as well:

print(" f(x) = ", y)  
print(" f'(x) = ", y.diff())  
print("f'(x\_0) = ", y.diff().subs(x, -2))  
print("[note: sympy converted x\_0=-2 to x\_0=2 because cos(-x)=cos(x)]")  
print("f'(x\_0) = ", y.diff().subs(x, x\_0).evalf())

f(x) = 4\*sin(x)  
 f'(x) = 4\*cos(x)  
f'(x\_0) = 4\*cos(2)  
[note: sympy converted x\_0=-2 to x\_0=2 because cos(-x)=cos(x)]  
f'(x\_0) = -1.66458734618857

Now, given each of these terms:

| Term (math) | Value |
| --- | --- |
| (X, Last Guess) |  |
| (Y at ) |  |
| (Slope of tangent at ) |  |

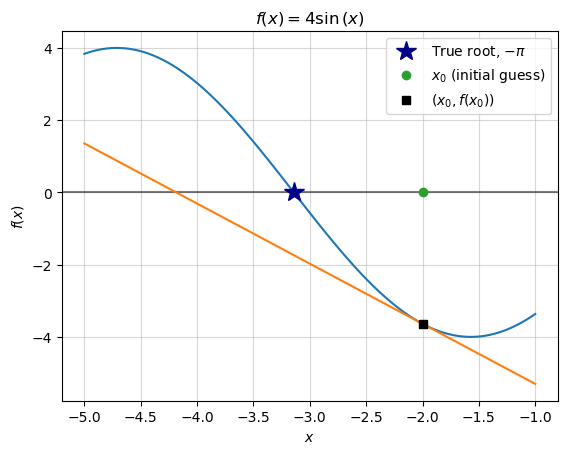
We can proceed to find the full equation of the tangent line by writing out the point-slope form of a linear equation with slope .

Plugging in our values, we get:

We’ll save that into a python function and plot it to make sure it does look like the tangent.

def f\_1(x): return -1.6645873\*x - 6.9663643

fig, ax = base\_plot(y, x, x\_low=-5, x\_high=-1)  
plot\_truth(ax)  
  
plot\_f(f\_1, x\_low=-5, x\_high=-1, ax=ax)  
  
plot\_guess(ax, x\_0, "$x\_0$ (initial guess)")  
plot\_guess\_coords(ax, x\_0, "$(x\_0, f(x\_0))$")  
  
ax.legend();

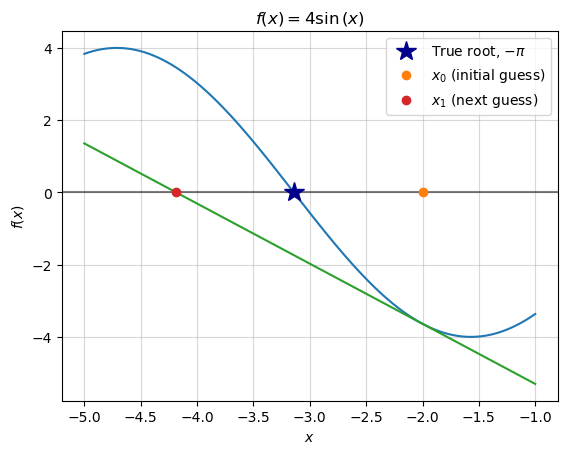


#### 1.2.2 Step 3: Find the x-intercept of the tangent line

Given that the tangent line is the **best linear approximation** of the original function, we can use its x-intercept as an approximation of the x-intercept of the original function. Thus, the root of the tangent line is the new “best-guess” of the original function’s root.

x\_1 = -4.1850

fig, ax = base\_plot(y, x, x\_low=-5, x\_high=-1)  
plot\_truth(ax)  
plot\_guess(ax, x\_0, "$x\_0$ (initial guess)")  
plot\_f(f\_1, x\_low=-5, x\_high=-1, ax=ax)  
plot\_guess(ax, x\_1, "$x\_1$ (next guess)")  
ax.legend();



While this guess still isn’t particularly great, we can see that we have actually reduced the “error” of our guess:

for i, x\_n in enumerate([x\_0, x\_1]):  
 print(f"error for guess x\_{i}:", euclidean\_dist\_to\_truth(x\_n))

error for guess x\_0: 1.1415926535897931  
error for guess x\_1: 1.0434073464102065

The big reveal of Newton-Raphson is that this error will continue to shrink as we repeat steps 1-3.

#### 1.2.3 Step 4: Repeat

We will again find the tangent line at this new point, . We could take the old tangent line equation, and simply update all of those to , but at this point it will benefit us to move towards a more general equation:

This allows us to generate the tangent line at any given guess, . The following code leverages sympy to write that equation as a python function.

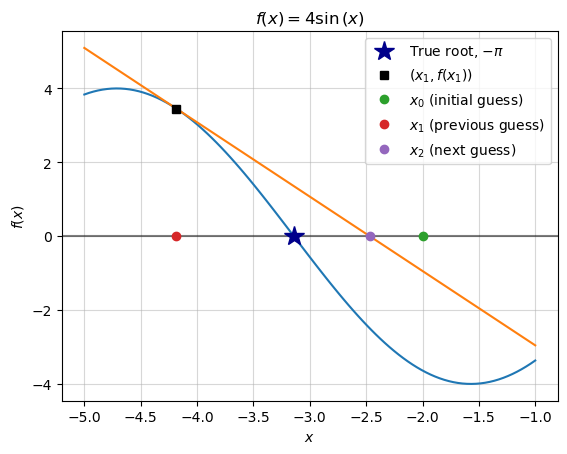
def y\_n(x\_n): return (  
 y.diff().subs(x, x\_n) \* (x - x\_n) + # f\_1(x\_1)(x-x\_1) +  
 y.subs(x, x\_n) # f(x\_1)  
)

y\_n(x\_1)

We’ll also use sympy to easily solve for the new tangent line’s x-intercept,

x\_2 = sympy.solve(y\_n(x\_1), x)[0]  
x\_2

fig, ax = base\_plot(y, x, x\_low=-5, x\_high=-1)  
plot\_truth(ax)  
plot\_f(sympy.lambdify(x, y\_n(x\_1)), x\_low=-5, x\_high=-1, ax=ax)  
  
plot\_guess\_coords(ax, x\_1, "$(x\_1, f(x\_1))$")  
  
plot\_guess(ax, x\_0, "$x\_0$ (initial guess)")  
plot\_guess(ax, x\_1, "$x\_1$ (previous guess)")  
plot\_guess(ax, x\_2, "$x\_2$ (next guess)")  
  
ax.legend();



We can verify that this new guess again reduces our “error,” which should encourage us to continue this process.

for i, x\_n in enumerate([x\_0, x\_1, x\_2]):  
 print(f"error for guess x\_{i}:", euclidean\_dist\_to\_truth(x\_n))

error for guess x\_0: 1.1415926535897931  
error for guess x\_1: 1.0434073464102065  
error for guess x\_2: 0.6735814493932737

#### 1.2.4 Step 5: Generalize the procedure

##### 1.2.4.1 A.) Make the equation more direct

Thus far we have used the general equation , where is our current guess, and we solve for to define our next guess. Given that we solve the equation for , we can rewrite it as follows:

This expresses *one step* of Newton’s Method – solving for the x-intercept of the tangent line at the point .

##### 1.2.4.2 B.) Move from equation to algorithm

We now build on this single step and express the general *process* of Newton’s method. To start, it’s more accurate to label the left hand side given that it represents the next guess:

Given this equation, we can think of Newton’s Method as essentially searching for good guesses – defining new – until it’s right. But how do we define “right”? Put in other words, **when do we stop?**

In our case, we are working with a simple function and we know the correct answer – we can just stop once we get close to – but in any real application that won’t be the case. In those cases, it is common practice to define “right” as when the guesses stop changing much with each iteration. Stated semi-formally, we wait until gets small.

For example: if the last guess was -3.14159 and the new guess is -3.141592, the guess only changed by .0000002, and we might conclude that we’ve gotten as close to the answer as is necessary. In this case, we set a stopping condition – when the next guess is less than or equal to .0000002 away from the previous one, we stop. We can write out the stopping condition as follows:

We can try writing out the recursive algorithm as a piece-wise equation:

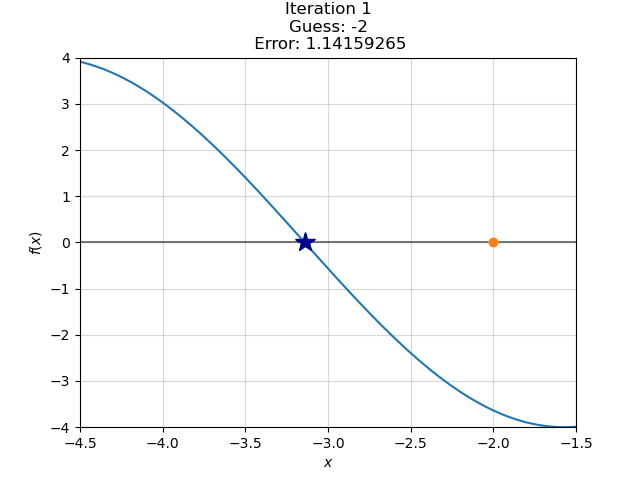
However, now that we are moving into the realm of algorithms, I think it’s clearer to write this as code:

x\_n = -2  
  
# We'll use this to count which guess we are on  
counter = 1  
while True:  
 # Disregard the following utility code  
 print(f"Guess {counter}:",  
 str(round(float(x\_n), 5)).ljust(8, '0'),  
 " --- Error:",  
 euclidean\_dist\_to\_truth(x\_n))  
 ################################################  
 # The following is the code for newton's method  
 # 1.) Check for the stopping condition,  
 # |f(x\_n)/f'(x\_n)| < 2 \* 10^-7  
 stop\_condition = (  
 np.abs(sympy.lambdify(x, y)(x\_n) /  
 sympy.lambdify(x, y.diff())(x\_n))  
 < 2e-7  
 )  
 if stop\_condition:  
 print(f"Converged in {counter} steps.")  
 break  
 # 2.) If stopping condition not met, make a new guess  
 x\_n = x\_n - (  
 # f(x\_n) /  
 sympy.lambdify(x, y)(x\_n) /  
 # f'(x\_n)  
 sympy.lambdify(x, y.diff())(x\_n)  
 )  
 ################################################  
 # Update the counter  
 counter += 1

Guess 1: -2.00000 --- Error: 1.1415926535897931  
Guess 2: -4.18504 --- Error: 1.0434472096717258  
Guess 3: -2.46789 --- Error: 0.6736989790751275  
Guess 4: -3.26619 --- Error: 0.1245936239793135  
Guess 5: -3.14094 --- Error: 0.00064874127215786  
Guess 6: -3.14159 --- Error: 9.101119857746198e-11  
Converged in 6 steps.

We’ve converged at our best-guess after six steps, which we can see animated below.

%%capture  
x\_n = -2  
max\_iter = 10  
tolerance = 1e-6  
  
fig, ax = base\_plot(y, x, x\_low=-5, x\_high=-1)  
plot\_truth(ax)  
ax.set\_ylim(-4, 4)  
ax.set\_xlim(-4.5, -1.5)  
  
images = []  
breaking\_condition = False  
for j in range(max\_iter):  
 error = euclidean\_dist\_to\_truth(x\_n)  
 if error < tolerance:  
 breaking\_condition = True  
  
 ax.set\_title(  
 f"Iteration {j+1}\nGuess: {round(x\_n, 6)}\n Error: {round(error, 8)}")  
  
 # Plot the current guess  
 plot\_guess(ax, x\_n, "$x\_n$")  
 filename = f'newton\_iteration\_{j}\_0.png'  
 fig.savefig(filename)  
 images.append(imageio.imread(filename))  
  
 if breaking\_condition:  
 ax.text(x\_n, 2, s=f"CONVERGED IN {j} STEPS", size=15)  
 filename = f'newton\_iteration\_{i}\_3.png'  
 fig.savefig(filename)  
 images.append(imageio.imread(filename))  
 ax.text(x\_n, 2, s=f"CONVERGED IN {j} STEPS", size=15)  
 filename = f'newton\_iteration\_{i}\_4.png'  
 fig.savefig(filename)  
 images.append(imageio.imread(filename))  
 break  
 # Plot the coordinates of the current guess  
 plot\_guess\_coords(ax, x\_n, "$(x\_n, f(x\_n))$")  
 filename = f'newton\_iteration\_{j}\_1.png'  
 fig.savefig(filename)  
 images.append(imageio.imread(filename))  
 # Plot the tangent line of that coordinate to inform next guess  
 plot\_f(sympy.lambdify(x, y\_n(x\_n)), x\_low=-5, x\_high=-1, ax=ax)  
 filename = f'newton\_iteration\_{j}\_2.png'  
 fig.savefig(filename)  
 images.append(imageio.imread(filename))  
  
 # Reset plot  
 ax.clear()  
 fig, ax = base\_plot(y, x, x\_low=-5, x\_high=-1)  
 plot\_truth(ax)  
 ax.set\_ylim(-4, 4)  
 ax.set\_xlim(-4.5, -1.5)  
  
 x\_n -= (  
 sympy.lambdify(x, y)(x\_n) /  
 sympy.lambdify(x, y.diff())(x\_n)  
 )  
imageio.mimsave('newton\_iterations.gif', images, duration=1)  
  
# Clean out pngs  
wd = os.getcwd()  
files = os.listdir(wd)  
for item in files:  
 if item.endswith(".png"):  
 os.remove(os.path.join(wd, item))



### 1.3 Using Newton’s Method to Solve a Real Problem

In the previous example, we dealt with a function, with a well known analytical solution for its x-intercept. Other simple functions can typically be solved with known formulas – e.g. a second degree polynomial’s roots can be found using the quadratic formula. In cases of known analytical solutions or readily available root-finding formulas, there is no reason to use Newton’s Method beyond as a learning exercise.

However, many functions do not have have readily available methods for finding the root. For example, if is a polynomial of degree 5 or greater, it is known that no formula for finding its roots exist (Strang and Herman 2016). Consider the following polynomial of degree 5:

What if we are asked to solve the following:

There is no formula that solves this. Even plugging the polynomial into sympy and running its solver, sympy.solveset, doesn’t give a clear answer.

x, y = sympy.symbols('x y')  
y = x\*\*5 + 8\*x\*\*4 + 4\*x\*\*3 - 2\*x-7  
y

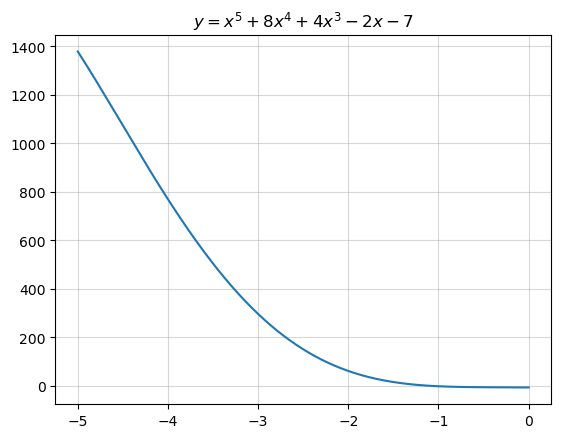
print(sympy.solveset(y, x, sympy.Interval(-5, 0)))

{CRootOf(x\*\*5 + 8\*x\*\*4 + 4\*x\*\*3 - 2\*x - 7, 1)}

(where CRootOf is an indexed complex root of the polynomial – not an analytical solution)

We might proceed to visually inspect the function on this domain – but it’s even pretty hard to visually spot a root here!

x\_vec = np.linspace(-5, 0, 1000000)  
fig, ax = plt.subplots()  
ax.grid(alpha=.5)  
ax.plot(x\_vec, sympy.lambdify(x, y)(x\_vec))  
ax.set\_title(f"$y={sympy.latex(y)}$");



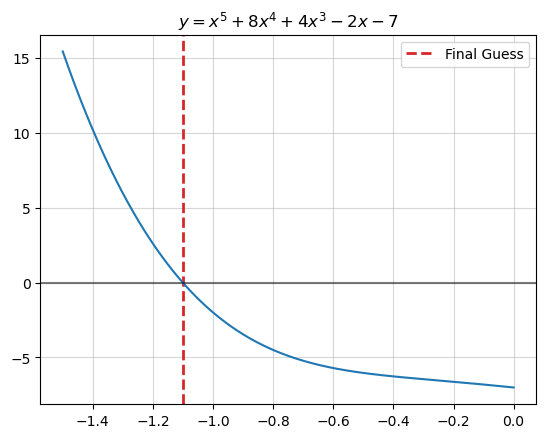
Here’s a good use-case for Newton’s Method. I set up the algorithm with and a stopping condition that .  
I quickly converge at an answer:

x\_n = 0  
tolerance = 1e-12  
counter = 1  
  
last = np.inf  
while True:  
 # Disregard the following utility code  
 print("Guess {0:3}:".format(counter),  
 str(round(float(x\_n), 5)).ljust(8, '0'),  
 " --- Change:",  
 round(np.abs(float(x\_n) - last), 8))  
 last = float(x\_n)  
 ################################################  
 # The following is the code for newton's method  
 # 1.) Check for the stopping condition,  
 # |f(x\_n)/f'(x\_n)| < 2 \* 10^-7  
 stop\_condition = (  
 np.abs(sympy.lambdify(x, y)(x\_n) /  
 sympy.lambdify(x, y.diff())(x\_n))  
 < tolerance  
 )  
 if stop\_condition:  
 print(f"Converged in {counter} steps.")  
 break  
 # 2.) If stopping condition not met, make a new guess  
 x\_n = x\_n - (  
 # f(x\_n) /  
 sympy.lambdify(x, y)(x\_n) /  
 # f'(x\_n)  
 sympy.lambdify(x, y.diff())(x\_n)  
 )  
 ################################################  
 counter += 1

Guess 1: 0.000000 --- Change: inf  
Guess 2: -3.50000 --- Change: 3.5  
Guess 3: -2.44316 --- Change: 1.05683755  
Guess 4: -1.81481 --- Change: 0.6283498  
Guess 5: -1.41471 --- Change: 0.40010602  
Guess 6: -1.19062 --- Change: 0.22409096  
Guess 7: -1.11070 --- Change: 0.07991323  
Guess 8: -1.10108 --- Change: 0.0096197  
Guess 9: -1.10095 --- Change: 0.00012977  
Guess 10: -1.10095 --- Change: 2e-08  
Converged in 10 steps.

When we plot our best guess, , we see that it is indeed the root of the function.

fig, ax = plt.subplots()  
  
ax.axvline(x\_n, linestyle="--", color="tab:red",  
 linewidth=2, label="Final Guess")  
ax.legend()  
  
x\_vec = np.linspace(-1.5, 0, 1000000)  
  
ax.grid(alpha=.5)  
ax.plot(x\_vec, sympy.lambdify(x, y)(x\_vec))  
ax.set\_title(f"$y={sympy.latex(y)}$")  
ax.axhline(0, color="black", alpha=.5);



We can also compare our answer to what sympy gets from numerically solving for the function’s root:

print("Newton's Method:", x\_n)

Newton's Method: -1.1009529619409872

print("Sympy's answer:", sympy.solveset(y, x, sympy.Interval(-5, 0)).evalf())

Sympy's answer: {-1.10095296194099}

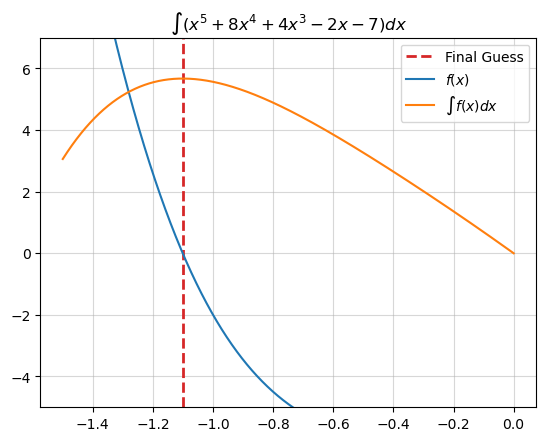
We see that we have basically the same answer as sympy.

## 2 Newton’s Method as Optimization

Recall that the derivative of a function is 0 at a critical point – its maximum or minimum. We have been using Newton’s Method to find the root of a function, but in the process **we’ve also implicitly been finding the critical point of the anti-derivative, or, integral of that function**.

For visual intuition of this fact, consider the following plot, in which we visualize the integral of our function, , and plot the root of that we just found using Newton’s Method:

fig, ax = plt.subplots()  
  
ax.axvline(x\_n, linestyle="--", color="tab:red",  
 linewidth=2, label="Final Guess")  
  
ax.grid(alpha=.5)  
ax.plot(x\_vec, sympy.lambdify(x, y)(x\_vec), label="$f(x)$")  
ax.plot(x\_vec, sympy.lambdify(x, y.integrate())(x\_vec), label="$\int f(x) dx$")  
ax.legend()  
ax.set\_ylim(-5, 7)  
ax.set\_title(f"$\int ({sympy.latex(y)})dx$");



The root of seems to also be the maximum of .

**Why is this important?** Based on this idea, we can extend Newton’s Method for general use in finding the critical value of a function, which means that we can use it for solving optimization problems. We can setup the optimization approach as follows:  
If the following equation will converge to the root of and the critical point of :

Then the following equation will converge to the root of and thus the critical point of (equivalent to ):

(“Newton’s Method” 2023, “Minimization and maximization problems”)

“Newton’s Method.” 2023. *Wikipedia*. <https://en.wikipedia.org/w/index.php?title=Newton%27s_method&oldid=1177135929>.

Strang, Gilbert, and Edwin Herman. 2016. *Calculus Volume 1*. OpenStax College.