PROJECT 1: ROOTS OF EQUATIONS

DEVON REED, AYODEJI ODETOLA, JAMES MILLER

Date: February 2020.
Dr. Juan B. Gil ☺.

Given that an electric power cable is suspended from two towers that are 100 meters apart. We want to determine the length of the cable under the assumption that it does not touch the ground but it is allowed to dip 5 meters below the straight line connecting the points at which the cable is attached to the poles. For this, we must consider two possible cases:

(a) The ground is level and the poles are at the same height.
(b) The ground is uneven, in which case the cable is suspended at points of different heights.

Where case (b) is the most realistic case.

1. Simple Case: Level Ground

1.1. **Finding Lambda.** To find λ, the bisection method, Newtons Method and the secant method (using a stopping tolerance of 10^{-7}) were utilized in Python.[1] Below, the approximated roots and the iteration history is reported with a graph showing roughly how each converges.

(1) **Bisection**. $\lambda \approx 250.8289337158203$ in 17 iterations.
(2) **Secant**. $\lambda \approx 250.828920164603$ in 3 iterations.
(3) **Newtons**.[2] $\lambda \approx 250.82893172917275$ in 2 iterations.

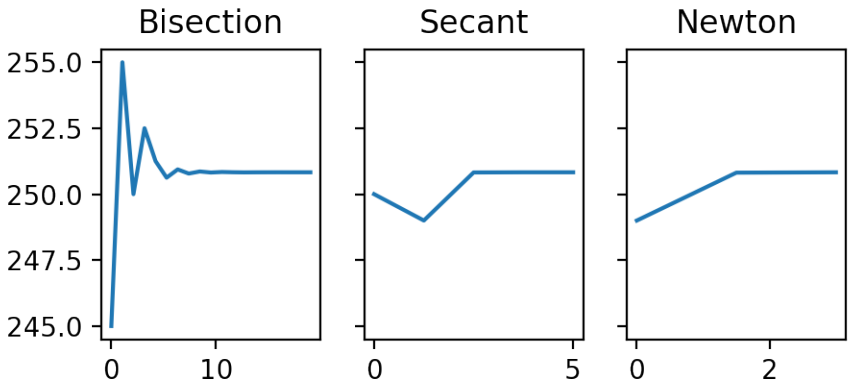

Figure 1. Iterations for each of our approximation algorithms.

[1]See Appendix 3.1

[2]Newtons Method converges the fastest so the value of λ generated by this algorithm is the value we used for the rest of the part 1

As expected, we observe that Newton's method converges the fastest with only 2 iterations since we know it converges quadraticly. Newton's is closely followed by the Secant method with 3 iterations which converges slightly slower with a super linear convergence. Then lastly, the Bisection method with 17 iterations converges even slower since its convergence is only linear.

1.2. **Finding the Cable's Length.** To determine the length of the cable, we look at the equation:

$$y(x) = h_0 + \lambda \cosh\left(\frac{x}{\lambda}\right)$$

We then calculated the derivative with respect to x to get $\frac{dy}{dx}$ given by the equation below.

$$\frac{dy}{dx} = \sinh\left(\frac{x}{\lambda}\right)$$

Using the derivative, we derive a formula for the length of the curve in terms of λ as follows:

$$\begin{aligned} L(\lambda) &= \int_{50}^{50} \sqrt{1 + \left(\frac{dy}{dx}\right)^2} \\ &= \int_{-50}^{50} \sqrt{1 + \sinh^2\left(\frac{x}{\lambda}\right)} dx \\ &= \int_{-50}^{50} \cosh\left(\frac{x}{\lambda}\right) dx \\ &= \lambda \sinh\left(\frac{x}{\lambda}\right) \Big|_{-50}^{50} \\ \implies L(\lambda) &= 2\lambda \sinh\left(\frac{50}{\lambda}\right) \end{aligned} \tag{1}$$

Thus, with $\lambda \approx 250.828$, we evaluate $L(\lambda)$ to get the approximated length as such.

$$L(250.82893172917275) = 100.664$$

Using the current information we have found a visualization of the simple case is provided as a sanity check and visual proof that our methods are working.

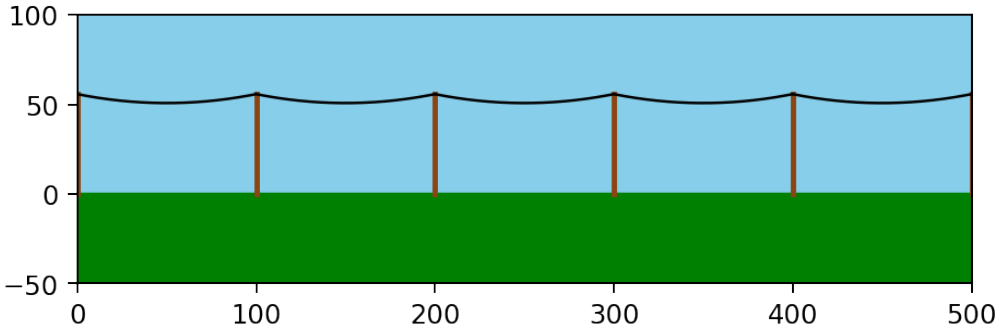

FIGURE 2. Dramatization of the approximated lambda applied to power lines to allow a sag of 5 meters over a distance of 100 meters. ☺

1.3. **Length of a Heavier Cable.** From Section 1.2, we know that the length of the cable is dependent upon lambda from equation (1). We want to show that if we replace the cable by a heavier one but of the same length, the sag remains the same. In order to show this, we must first prove that our equation for length is injective (one-to-one). In order to prove the injectivity of $L(\lambda)$ we will rely on the following Lemma.

Lemma. *If $f(a) = g(a)$ and $f'(x) < g'(x)$ for $x > a$, then $f(x) < g(x)$.*

To prove this we need to show that $\exists b \in \mathbb{R}$ such that $f(b) < g(b)$

Proof. Let $b \in \mathbb{R}$ and $h(x) = g(x) - f(x)$ such that $b > a$. Thus, $h(x) > 0$ and $h(a) = g(a) - f(a) = 0$. Using the mean value theorem we get the following,

$$h'(c) = \frac{h(b) - h(a)}{b - a} > 0$$
$$\implies (b - a)h'(c) + h(a) = h(b)$$

We know that $h(a) = 0$, so $h(b) = h'(c)(b - a) > 0$. Thus $h(b) > 0 \implies g(b) > f(a)$. □

In order to show that L is injective, we prove that L is monotonic on the interval $(0, \infty)$.

Proof. In order to show that L is injective, we must show $L'(\lambda) < 0$ for $\lambda \in (0, \infty)$
We know the equations of L and L'. Simplifying $L'(\lambda) < 0$, we get this inequality.

$$L'(\lambda) = 2\sinh\left(\frac{50}{\lambda}\right) - \frac{100}{\lambda}\cosh\left(\frac{50}{\lambda}\right) < 0$$
$$\frac{\lambda \sinh\left(\frac{50}{\lambda}\right)}{50\cosh\left(\frac{50}{\lambda}\right)} < 1$$
$$\tanh\left(\frac{50}{\lambda}\right) < \frac{50}{\lambda}$$

let $f(x) = \tanh x$ and $g(x) = x$, now we must show that $f'(x) < g'(x)$, to use our previous lemma.

$$f'(x) = \frac{1}{\cosh^2 x} < g'(x) = 1$$
$$\left(\frac{e^x + e^{-x}}{2}\right)^{-2} < 1$$
$$\frac{4}{\left(e^x + e^{-x}\right)^2} < 1$$
$$4 < e^{2x} + e^{-2x} + 2$$

In order to show this inequality holds for $0 < x < \infty$ we name the right half of the previous expression $h(x)$. $h(x)$ needs to be greater than four and monotonic increasing for $x > 0$. We know that $h(0) = 4$, so we just need to show $h(x)$ is increasing for $x > 0$ by looking at $h'(x)$. So we have

$$h(x) = e^{2x} + e^{-2x} + 2$$
$$h'(x) = 2e^{2x} - 2e^{-2x}$$
$$h'(x) = 2e^{2x}\left(1 - e^{-4x}\right) > 0$$

We know that $(1 - e^{-4x})$ is always positive and the proof for this is left as an exercise to the reader ☺. thus, $\frac{d}{dx}(\tanh x) < 1$ so, using our lemma this implies that $\tanh x < x$. Therefore, if we set $x = \frac{50}{\lambda}$ we get

that $\tanh \frac{50}{\lambda} < \frac{50}{\lambda}$. This concludes our proof that $L(\lambda)$ is injective for $\lambda \in (0, \infty)$ □

2. Realistic Case: Uneven Ground

2.1. **System of Equations for Lambda and c.** To find a system of equations for the unknown parameters, we simplified the equation given for the function y and obtained the following functions where H is the discrete function of the heights every 100 meters:

$$f(a+100) - f(a) = H(a+100)$$
$$f(a+50) = \frac{f(a+100) + f(a)}{2} - 3$$

After obtaining this equation, we were able to get the function in terms of just λ and c. The variables a and b are the x-positions where a is the x-position of the first pole, and b is the x-position of the next pole. Separating both sides we find the following equations:

$$\begin{cases} f_1(c,\ \lambda) = \lambda \cosh\left(\frac{a+c}{\lambda}\right) - \lambda \cosh\left(\frac{b+c}{\lambda}\right) = H[a] - H[b] \\ \\ f_2(c,\ \lambda) = \lambda \cosh\left(\frac{\left(\frac{a+b}{2}\right)+c}{\lambda}\right) + 3 = \left(\frac{\lambda \cosh\left(\frac{a+c}{\lambda}\right) + \lambda \cosh\left(\frac{b+c}{\lambda}\right)}{2}\right) \end{cases} \tag{2}$$

2.2. **2D Newtons and Convergence.** For our Python application of Newton's method in two dimension see the Appendix section 3.1.5. Next we discuss the convergence of Newton's method in two dimensions. To show that it is still quadratically convergent we let the following be true.

$$\hat{z_n} = \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$
$$\hat{r} = \begin{pmatrix} r_x \\ r_y \end{pmatrix}$$
$$C = max\{C_1, C_2\}$$

Given that $|x_{n+1} - r_x| < |x_n - r_x|$ and $|y_{n+1} - r_y| < |y_n - r_y|$. In order to find the rate of convergence we need to prove that $||\hat{z}_{n+1} - \hat{r}|\,| \leq ||\hat{z_n} - \hat{r}|\,|^2$, we know that

$$||\hat{z}_{n+1} - \hat{r}|\,| = \left\| \begin{pmatrix} x_{n+1} - r_x \\ y_{n+1} - r_y \end{pmatrix} \right\| = \sqrt{(x_{n+1} - r_n)^2 + (y_{n+1} - r_y)^2}$$

. Using the given inequalities we know that,

$$\sqrt{(x_{n+1}-r_x)^2+(y_{n+1}-r_y)^2} \leq \sqrt{C_1(x_n-r_x)^4+C_2(y_n-r_y)^4}$$

$$\sqrt{C_1(x_n-r_x)^4+C_2(y_n-r_y)^4} = C\sqrt{((x_n-r_x)^2)^2+((y_n-r_y)^2)^2}$$

Using the inequality that $a^2+b^2 \leq (a+b)^2$, we let a = $(x_n-r_x)^2$, and b = (y_n-r_x), we can make the claim that

$$C\sqrt{((x_n-r_x)^2)^2+((y_n-r_y)^2)^2} \leq C\sqrt{((x_n-r_x)^2+(y_n-r_y)^2)^2}$$

$$C\sqrt{((x_n-r_x)^2+(y_n-r_y)^2)^2} = \left\| \begin{pmatrix} x_n-r_x \\ y_n-r_x \end{pmatrix} \right\|^2$$

Therefore $||\hat{z}_{n+1}-\hat{r}|\,| \leq ||\hat{z}_n-\hat{r}|\,|^2$.

2.3. **Approximating Lambda, c, and, $\mathbf{h_0}$.** The values in the tables below represent the calculated[3] h_0, c, and λ for each interval of 100 meters in the horizontal direction.

Cable Number	λ	c	h_0
1	449.0997616	124.8206483	-441.5577399
2	466.1158912	73.9011686	-433.9339452
3	449.0997616	-424.8206482	-391.5577399
4	421.8000077	-412.8888102	-361.9969427
5	435.4121888	-578.4650695	-412.5013875
6	423.1091060	-621.4217413	-410.6515063
7	418.4890348	-683.3642477	-413.8197301
8	419.2316040	-708.2451312	-414.3126861
9	423.8233397	-774.2917477	-409.6032837
10	420.1374396	-899.8216974	-387.1374774

TABLE 1. All values of λ, C, and, h_0 calculated from 2D Newton's

[3]See Appendix section 3.2.3 for Python code

Cable Number	Length (m)
1	107.89525363549083
2	111.9750627832064
3	107.8952536354884
4	101.35051368573532
5	104.61373379999178
6	101.66433077532892
7	100.55681738477843
8	100.73482251646952
9	101.83554786523833
10	100.95196609364065

TABLE 2. Length of each cable for all intervals. See appendix

From the above data, a visualization was made to show that with the calculated values of h_0 the cables fit the layout of the terrain.

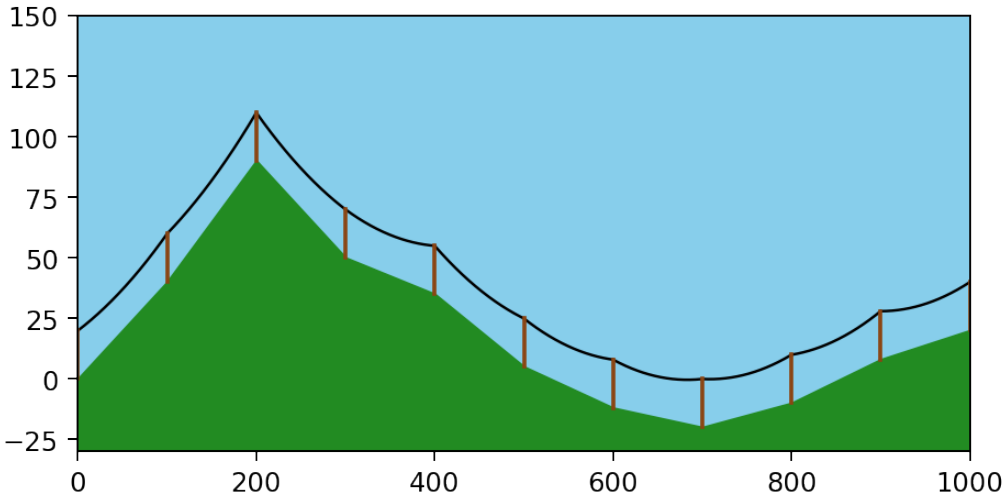

FIGURE 3. Visualization of the approximated λ, c, and, h_0

3. Appendix

3.1. Root Approximation Functions.

3.1.1. Bisection Method.

```
def superBiSecMethod(f, a, b, e = 1e-10):
 '''Finds roots of a function (f) on the interval
  [a, b] for f(r) < e using bisection method'''
 iterations = [a, b]
 if f(a) * f(b) < 0:
   while True:
     c = (a + b) / 2
     if f(a) * f(c) < 0:
       b = c
     else:
       a = c
     iterations.append(c)
     if abs(f(c)) < e:
       break
     elif f(c) == 0:
       print('Exact root located at:', c)
       break
 else:
   raise ValueError("function has no zero
   on given interval or special case")
 return c, iterations
```

3.1.2. **Secant Method**.

```
def superSecantMethod(f, x0, x1, e=1e-10):
  '''Finds roots of function f given two points
  x1 and x2 close enough to the root for
  f(r) < e using the secant method'''
  iterations = [x0, x1]
  while True:
    x2=(x0 * f(x1) - x1 * f(x0)) / (f(x1) - f(x0))
    x0 = x1
    x1 = x2
    iterations.append(x2)
    if abs(f(x2)) < e:
      break
  return x0, iterations
```

3.1.3. **Newton's Method**.

```
def superNewton(f, x0, e=1e-10):
  iterations = [x0]
  x = symbols('x')
  f_prime = lambdify(x, diff(f(x), x))
  while True:
    if f_prime(x0) == 0:
      raise ValueError('You messed up.')
    x1 = float(x0 - (f(x0) / f_prime(x0)))
    x0 = x1
    iterations.append(x0)
    if abs(f(x0)) < e:
      break
  return x0, iterations
```

3.1.4. **Evaluate for Lambda**.

```
g = lambda x: x*cosh(50 / x) - x - 5
error = 1e-7
rbs, ibs=superBiSecMethod(g, 245, 255, error)
rse, ise=superSecantMethod(g, 250, 249.9,error)
rnm, inm=superNewton(g, 250,error)
print('BISECTION. f(r) <', error,
    'r:', rbs, 'in', len(ibs)-2, 'iterations')
print('SECANT. f(r) <', error,
    'r:', rse, 'in', len(ise)-2, 'iterations')
print('NEWTONS. f(r) <', error,
    'r:', rnm, 'in', len(inm) -1, 'iterations')
```

3.1.5. Newton's Method in 2 Dimensions.

```
def twoDNewtons(f1, f2, x0, y0, e=1e-10):
  '''Aproximates the root vector [r1, r2]
  given the vector
  value function f(x, y) = [f1, f2]
  with an error of e'''
  x, y = symbols('x y')
  f1dx = lambdify((x, y), diff(f1(x, y), x))
  f1dy = lambdify((x, y), diff(f1(x, y), y))
  f2dx = lambdify((x, y), diff(f2(x, y), x))
  f2dy = lambdify((x, y), diff(f2(x, y), y))
  while True:
    detj=float(f1dx(x0, y0)*
       f2dy(x0, y0)-f1dy(x0, y0)*f2dx(x0, y0))
    assert detj != 0
    h1 = float((f2dy(x0, y0)*
     f1(x0, y0)-f1dy(x0, y0)*f2(x0, y0)) / detj)
    h2 = float((f1dx(x0, y0)*
     f2(x0, y0)-f2dx(x0, y0)*f1(x0, y0)) / detj)
    x1, y1 = x0 - h1, y0 - h2
    x0, y0 = x1, y1
    if float((f1(x0, y0)**2+
    f2(x0, y0)**2)**.5) < e:
      break
  return x0, y0
```

3.2. Graphs and Figures.

3.2.1. Figure 1.

```
g = lambda x: x*cosh(50 / x) - x - 5
error = 1e-7
rbs, ibs = superBiSecMethod(g, 245, 255, error)
rse, ise = superSecantMethod(g, 250, 249, error)
rnm, inm = superNewton(g, 249, error)
print('BISECTION. root aproximated | f(r) <', error,
    'r:', rbs, 'in', len(ibs)-2, 'iterations.')
print('SECANT. root aproximated | f(r) <', error,
    'r:', rse, 'in', len(ise)-2, 'iterations.')
print('NEWTONS. root aproximated | f(r) <', error,
    'r:', rnm, 'in', len(inm) -1, 'iterations.')
bisecit, secit, nit = np.asarray(ibs),
np.asarray(ise), np.asarray(inm)
xbs, xs, xn = np.linspace(0, len(ibs),
len(ibs)), np.linspace(0, len(ise),
len(ise)),np.linspace(0, len(inm), len(inm))
f, (bs, sc, nt) = plt.subplots(1, 3, sharey=True)
f.set_figheight(3)
f.set_figwidth(6)
f.set_dpi(175)
bs.set_title("Bisection Iterations")
bs.plot(xbs, bisecit)
#axins = zoomed_inset_axes(nt, 3, loc=1)
#x1, x2, y1, y2 = 10, 15, 250.5, 251.2
#plt.yticks(visible=False)
#plt.xticks(visible=False)
#axins.set_xlim(x1, x2)
#axins.set_ylim(y1, y2)
#mark_inset(nt, axins, loc1=3, loc2=4, ec='0.5')
sc.set_title("Secant Iterations")
sc.plot(xs, secit)
nt.set_title("Newton Iterations")
nt.plot(xn, nit)
plt.draw()
```

3.2.2. **Figure 2**.

```
h = lambda x: rnm * np.cosh(x / rnm) - 200
x = np.linspace(-50, 50, 200)
fill = np.linspace(0, 500, 10)
fig = plt.figure(figsize=(6, 3), dpi=175)
ax = plt.axes()
ax.axis([0, 500, -50, 100])
ax.set_facecolor('skyblue')
ax.set_adjustable('box')
plt.gca().set_aspect('equal')
ax.fill_between(fill, -50, color='g')
for i in range(-50, 550, 100):
  ax.plot([i+50, i+50], [0, h(-50)], c='saddlebrown', lw=2)
for i in range(0, 500, 100):
  ax.plot(x+i+50, h(x), c='black', lw=1)
```

3.2.3. **Figure 3**.

```
# f1 and f2 given x = lambda and y = c
f = lambda x, l, c, h0: h0 + l*np.cosh((x+c)/l)
ints = np.array([0,100,200,300,400,500,600,700,800,900,1000])
relHeights =
np.array([0, 40, 90, 50, 35, 5, -12, -20, -10, 8, 20])
# heights between poles
file1 =
open(r"/content/drive/My Drive/rootsForLatek.txt",'a')
lambdas = []
cs = []

fig = plt.figure(1, (6,3), 175) # Case 2: Uneven Ground Figure
ax = plt.axes()
ax.axis([-3, 1003, -30, 150])
ax.set_facecolor('skyblue')

for i in range(10):
  a, b = i*100, (i+1)*100 # current interval [a, b]
  f1 = lambda x, y: x*cosh((a+y)/x)-x*cosh((b+y)/x)+
  relHeights[i+1]-relHeights[i] # Function 1 for interval
  f2 = lambda x, y: x*cosh((((a+b)/2)+y)/x)-x*cosh((b+y)/x)+
  ((relHeights[i+1]-relHeights[i])/2)+3 # Function 2 for interval
  r1, r2 = twoDNewtons(f1, f2,  100, 100, e=1e-10)
  # find lambda and c
  x = np.linspace(int(a), int(b), 200)
  # x array for the lines
  x_new = np.linspace(0, 1000, 500)
  # x array for the ground
  spl = make_interp_spline(ints, relHeights, k=3) # smooth ground
  power_smooth = spl(x_new)
  v_shift = 25 + relHeights[i] - f(a, r1, r2, 0) # find h0
  ax.plot(x, f(x, r1, r2, v_shift), c='black', lw=1)
  ax.plot([a, a], [f(a, r1, r2, v_shift)-25,
  f(a, r1, r2, v_shift)], c='saddlebrown', lw=1.5)
  lambdas.append(r1)
  cs.append(r2)
  file1.write('Interval ' + str(i+1) + r': \lambda \approx '
  + str(r1) + r' c \approx ' + str(r2) + r' h_0 \approx '
  + str(v_shift) + '\n')
file1.close()
ax.plot([1000, 1000], [f(1000, r1, r2, v_shift)-25,
f(1000, r1, r2, v_shift)] , c='saddlebrown', lw=1.5)
ax.fill_between(x_new, power_smooth, -30, color='forestgreen')
plt.show()
```