CS164 ASSIGNMENT 2:

UNCONSTRAINED & EQUALITY CONSTRAINED OPTIMIZATION

Minerva University

CS164 - Optimization Methods

Prof. Levitt

October 22, 2022

Table of Contents

UNCONSTRAINED & EQUALITY CONSTRAINED OPTIMIZATION

Part 1

The cross-sectional area can be calculated via 2 methods: (i) the formula for the area of a right trapezoid and (ii) the sum of the right triangle's and rectangle's areas. The result is the same: $A = a\text{^}2 * (sin(\theta))\text{^}2 - 3 * a * sin(\theta) + a\text{^}2 * sin(\theta) + 0.5 * a\text{^}2 * cos(\theta) * sin(\theta)$, where $a \in (0, 3)$ in units $[m]$ and $\theta \in (0, \pi)$ in units $[rad]$ are physically-realistic values because a cannot be longer than metal's length and angle θ is strictly less than 180 because pair of angles along one of the legs (i.e., a) are supplementary angle (i.e., their sum must be $= 180°$).

Producing a surface and a contour plot of the function $A(a, \theta)$, we can notice unique maximum area over the domain of physically-realistic values that appears at $a \approx 1.1\ m$, $\theta \approx 1\ rad$, and the max value of A is $\approx 1.2\ m^2$ (Figure 1, Figure 2, Appendix A). We estimated these values visually based on 3d surface and 2d contour plots.

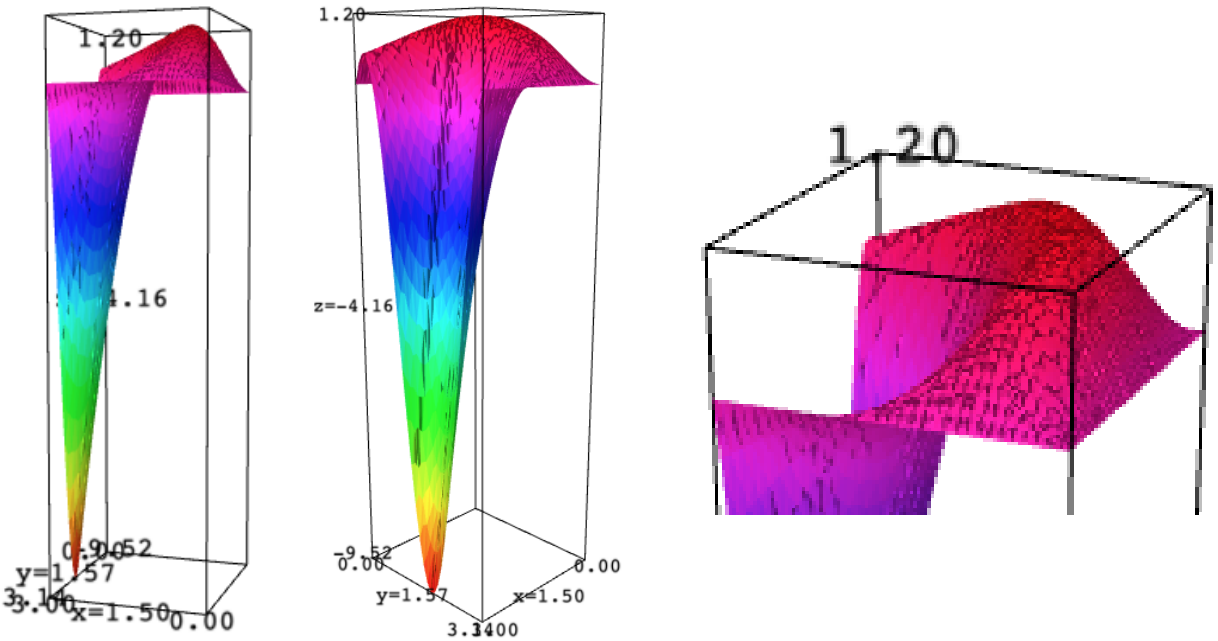


Figure 1. *3D surface plot for the cross-sectional area function, where the x and y-axis represent the length of the side a and angle θ values respectively. Colours change from red to ruby pink based on the area's value. The colour bar can be found below in Figure 2.*

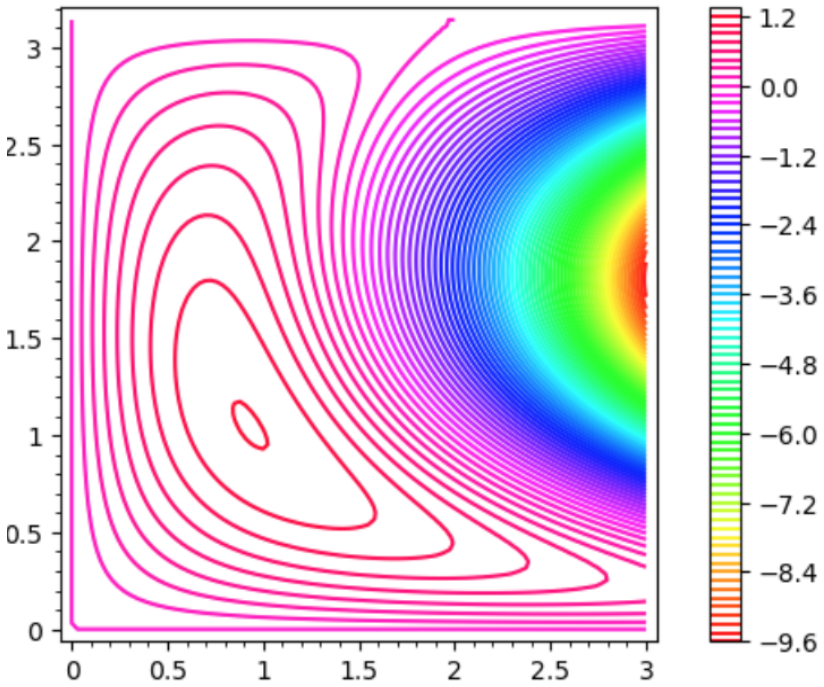


Figure 2. *Contour plot for the same function, where the x and y-axis represent the same values.*

Usually, the 3d surface plot is easier to interpret, but in our case, negative area values have wider variance and take a significant part of the plot while bringing zero value to estimation — the area cannot be negative. Still, we can see that having a wider angle and longer a does not lead to a bigger area. The ruby pink peak on the plot corresponds with the highest value of the area and occurs at approximated a and o mentioned above.

Looking at the contour plot, we can indicate a peak within the range of a and o at the centre of the concentric shape. At the area's highest value, contour lines are not spaced close to each other, so values change more slowly.

Part 2

Using Lagrange multipliers on the area function A(a, b, θ), we can find the exact values of a and θ that maximize the cross-sectional area of the channel. To do this, we rewrite the constraint and form the Lagrangian for $w = 3$:

$$f(a,\theta) = 0.5 * (2 * b + a * cos(\theta)) * a * sin(\theta)$$

$$Lagr(a, \theta, b, \lambda) = f(a,\theta) + \lambda * (b - w + a * sin(\theta) + a)$$

Now we study the function over the boundary of the domain D by using the Lagrange method with constraint equation (Appendix B). As a result, we solve the following system of equations in Sage: $\frac{dL}{da} = 0, \frac{dL}{d\theta} = 0, \frac{dL}{db} = 0, \frac{dL}{d\lambda} = 0$ (Figure 3).

```
([a == -2*(b + l)*cos(o)/(cos(o)^2 - sin(o)^2), a == 0],
 [a == -2*(b + l)*cos(o)/(cos(o)^2 - sin(o)^2), a == 0],
 [a == -l/sin(o)],
 [a == -(b - 3)/(sin(o) + 1)])
```

Figure 3. *The system has the following equations. Computations are conducted in Sage.*

Having the updated system of equations, we can find 2 real solutions:

1. $a = 4\sqrt{3} - 6$, $b = 3 - \sqrt{3}$, $\lambda = 3(\sqrt{3} - 2)$, $\Theta = 1/3\,\pi\,(6n + 1)$, $n \in Z$

2. $a = -6 - 4\sqrt{3}$, $b = 3 + \sqrt{3}$, $\lambda = -3(2 + \sqrt{3})$, $\Theta = 1/3\,\pi\,(6n - 1)$, $n \in Z$

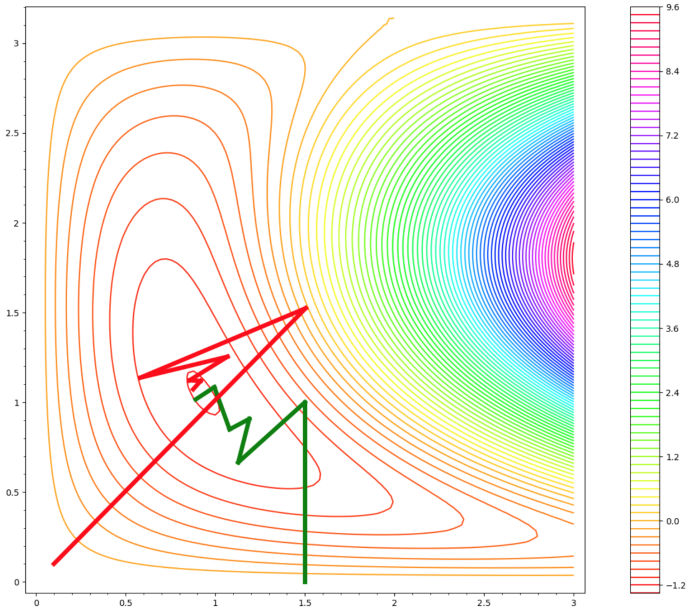
Evaluating the results, we tried to not list phisically impossible critical points. The 2nd solution here also is impossible (i.e., a < 0) The only feasible solution is the first one that gives us $A = 0.5 * (2 * (3 - \sqrt{3}) + (4 * \sqrt{3} - 6) * cos(pi/3)) * (4 * \sqrt{3} - 6) * sin(pi/3)$

$A = 1.2057\ m^2$. By 3D and contour plot, we know that this value is the only physically possible global maximum.

Part 3

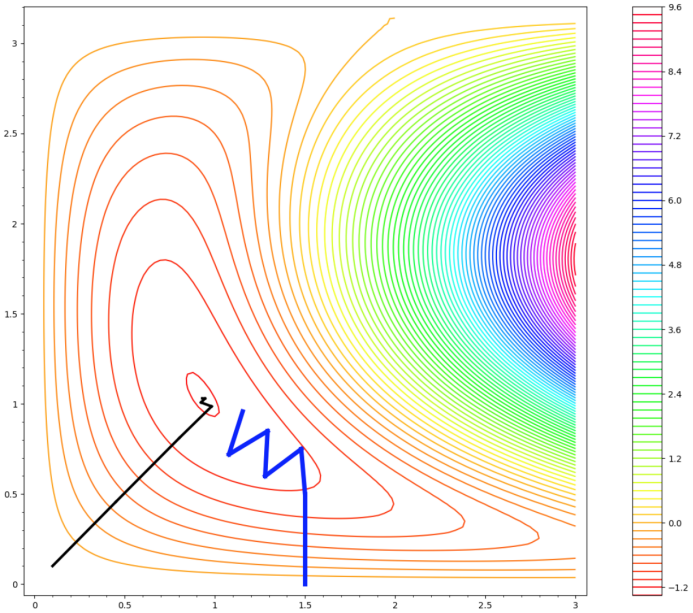
To verify that we reach the same optimal solution, we apply a gradient descent code with the backtracking line search we created in the break-out group with Saad and Artur during Session 10. We try the code on two initial states: (1.5, 0) and (0.1,0.1). The table of steps shows how the algorithms converge (Figure 4, Appendix C). We repeat this process using a conjugate gradient, momentum, and Newton's method codes (Figure 5, Appendix E; Figure 6,Appendix F;

Figure 7, Appendix G).



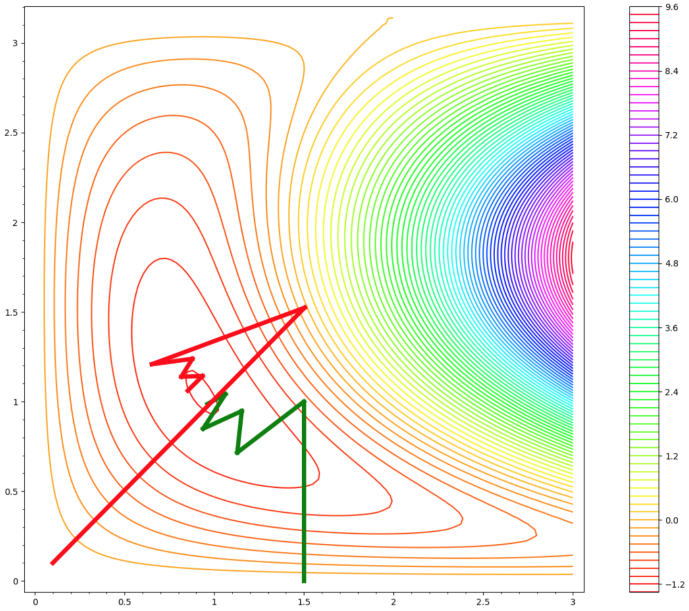
```
TABLE OF STEPS FOR INITITAL POINT (1.5,0)
#:  1 Convergence:  [(1.5, 1.0), (1.1, 0.67)]
#:  2 Convergence:  [(1.1, 0.67), (1.2, 0.91)]
#:  3 Convergence:  [(1.2, 0.91), (1.1, 0.85)]
#:  4 Convergence:  [(1.1, 0.85), (1.0, 1.1)]
#:  5 Convergence:  [(1.0, 1.1), (0.89, 1.0)]
TABLE OF STEPS FOR INITITAL POINT (0.1,0.1)
#:  1 Convergence:  [(1.5, 1.5), (0.58, 1.1)]
#:  2 Convergence:  [(0.58, 1.1), (1.1, 1.3)]
#:  3 Convergence:  [(1.1, 1.3), (0.86, 1.1)]
#:  4 Convergence:  [(0.86, 1.1), (0.92, 1.1)]
#:  5 Convergence:  [(0.92, 1.1), (0.88, 1.1)]
```

Figure 4. *Contour plot for the cross-sectional area function, where the colour bar and assigned values are reversed because we maximize rather than minimize area. x and y-axis represent the same values as in Figure 2. Green and red lines represent the gradient descent method with a backtracking line search from the initial point (1.5, 0) and (0.1, 0.1), respectively. The search terminates after five iterations. This amount of steps is insufficient to find the actual value.*



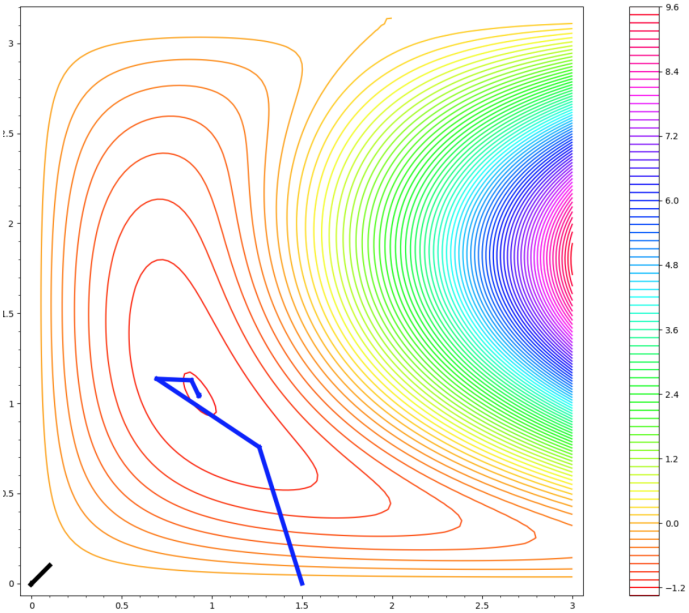
```
TABLE OF STEPS FOR INITITAL POINT (1.5,0)
#:  1 Convergence:  [(1.5, 0.50), (1.5, 0.75)]
#:  2 Convergence:  [(1.5, 0.75), (1.3, 0.60)]
#:  3 Convergence:  [(1.3, 0.60), (1.3, 0.85)]
#:  4 Convergence:  [(1.3, 0.85), (1.1, 0.72)]
#:  5 Convergence:  [(1.1, 0.72), (1.2, 0.96)]
TABLE OF STEPS FOR INITITAL POINT (0.1,0.1)
#:  1 Convergence:  [(0.45, 0.46), (0.80, 0.81)]
#:  2 Convergence:  [(0.80, 0.81), (0.98, 0.99)]
#:  3 Convergence:  [(0.98, 0.99), (0.92, 1.0)]
#:  4 Convergence:  [(0.92, 1.0), (0.95, 1.0)]
#:  5 Convergence:  [(0.95, 1.0), (0.93, 1.0)]
```

Figure 5. *Blue and black lines represent conjugate gradient from the initial points (1.5, 0) and (0.1, 0.1). The search terminates after five iterations — insufficient to find the actual value for (1.5,0), but enough for (0.1, 0.1).*



```
TABLE OF STEPS FOR INITITAL POINT (1.5,0)
#:  1 Convergence:  [(1.5, 1.0), (1.1, 0.72)]
#:  2 Convergence:  [(1.1, 0.72), (1.2, 0.95)]
#:  3 Convergence:  [(1.2, 0.95), (0.94, 0.85)]
#:  4 Convergence:  [(0.94, 0.85), (1.1, 1.0)]
#:  5 Convergence:  [(1.1, 1.0), (0.96, 0.99)]
TABLE OF STEPS FOR INITITAL POINT (0.1,0.1)
#:  1 Convergence:  [(1.5, 1.5), (0.65, 1.2)]
#:  2 Convergence:  [(0.65, 1.2), (0.88, 1.2)]
#:  3 Convergence:  [(0.88, 1.2), (0.82, 1.1)]
#:  4 Convergence:  [(0.82, 1.1), (0.93, 1.1)]
#:  5 Convergence:  [(0.93, 1.1), (0.85, 1.1)]
```

Figure 6. *Green and red lines represent gradient decent with momentum from the initial points (1.5, 0) and (0.1, 0.1). The search terminates after five iterations — insufficient to find the max.*



```
TABLE OF STEPS FOR INITITAL POINT (1.5,0)
#:  1 Convergence:  [(1.3, 0.76), (0.69, 1.1)]
#:  2 Convergence:  [(0.69, 1.1), (0.89, 1.1)]
#:  3 Convergence:  [(0.89, 1.1), (0.93, 1.0)]
#:  4 Convergence:  [(0.93, 1.0), (0.93, 1.0)]
#:  5 Convergence:  [(0.93, 1.0), (0.93, 1.0)]
TABLE OF STEPS FOR INITITAL POINT (0.1,0.1)
#:  1 Convergence:  [(-0.0043, -0.0055), (-2.9e-6, -7.7e-6)]
#:  2 Convergence:  [(-2.9e-6, -7.7e-6), (-1.4e-12, -7.4e-12)]
#:  3 Convergence:  [(-1.4e-12, -7.4e-12), (3.4e-16, -3.4e-24)]
```

Figure 7. *Blue and black lines represent Newthon's method for the initial points (1.5, 0) and (0.1, 0.1). The search terminates after five iterations that is enough for (1.5, 0). We increased* $e = 10^{-9}$ *to achieve at least 3 steps for (0.1, 0.1) — Newthon's method does not work because the derivative is close to zero, so the tangent line overshoots because it is nearly horizontal.*

Comparing the performance of the methods on this problem, we consciously limited the number of steps to 5 to see which code would achieve the best parameters that maximize the area

and consequently maximizes the capacity of the roof drainage channel. For point (1.5, 0), the most efficient method (i.e., final result is (0.93,1.0)) was Newton's method, while for (0.1,0.1) — conjugate gradient. The real maximum of the function is (0.9282, 1.0472).

Since the accuracy of each method depends on the input, it is better to apply several ways to verify the solution. We also need to remember that each optimization algorithm differs in the amount of data needed, so sometimes, we need to make a trade-off between the correctness of the final parameters and the run-time it takes to perform. To do this, we calculate the error bar and time consumed.

HC Application

#dataviz: Throughout the paper, I paid great attention to data visualization and its interpretation. In Part 1, I immediately mention units and then build surface and contour plots. I provide detailed explanations for both figures mentioning the most important features of each type of plot. I also conducted additional research to explore SageMath's extensive plotting functionality. This helped me to add colours to contour plots, colour bars to figures, thickness to lines, etc. I also made sure that I included only the needed visualization and achieved it by careful design thinking (e.g., I avoided labels on contour lines because it made the figure staffy; instead, I implemented a colour bar). If something in the data visualization could be confusing, I did my best to address it (e.g., I mentioned that we add a minus to the function, so values are reversed). Most importantly, I used data visualization to solve optimization problems.

#scienceoflearning: This is funny, but it took me so long to conduct calculations for the area. I did three iterations via different formulas and still made multiple mistakes. The reason was my tiredness. I started the assignment very late in the evening, so we were not capable of paying enough attention to details. After reflection on it, I finally started the assignment the next day and it took me much less time. I realize that it is important to analyze when your brain is in the most active stage and utilise it. I also asked my peer about the formula she used, and she told me about the medium line - I completely forgot about this method! So, it was useful too. Also, I didn't give up and precisely analyzed why I couldn't complete the formula or code. The approach of iteration paid of, and I feel that I focus on active rather than passive learning by deriving formulas by hand.

Appendix A

```
#Let θ be symbol "o", where the cross-sectional area of the channel A is f(a,o):
w = 3
b = w - a*sin(o) - a
f(a,o) = 0.5*(2*b+a*cos(o))*a*sin(o)
contour = contour_plot(f,(0,3),(0,pi), contours = 100, fill = False, cmap='hsv', colorbar=true)
surface = plot3d(f,(0,3),(0,pi), viewer = 'threejs', adaptive=True, color='automatic', online = True, num_colors=256, show_curve=True, opacity=0.5)
show(surface)
```

Appendix B

```
Lagr(a, o, b, l) = f(a,o) + l*(b-w+a*sin(o)+a)
der_a = derivative(Lagr(a, o, b, l), a)
der_a = derivative(Lagr(a, o, b, l), o)
der_b = derivative(Lagr(a, o, b, l), b)
der_l = derivative(Lagr(a, o, b, l), l)
solve([der_a == 0],a), solve([der_o == 0],a), solve([der_b == 0],a), solve([der_l == 0],a)
```

Appendix C

```
#function for area
f(a,o) = -0.5*(2*(3-a*sin(o)-a)+a*cos(o))*a*sin(o)

def backtracking_line_search(f, fg, a, d, alph, p=0.5, B=10^(-4)):
    o, g = f(*a), fg(*a)
    while (f(*(a + alph*d)) > o + B*alph*(g*d)):
        alph*= p
    return alph

def gradient_descent(f, a, step_size, termination, rounding=5):
    solutions, scores = list(), list()
    g = f.gradient()(*a)
    d = -g/(g.norm())
    old = new = f(*a)
    new_point = a
    all_points = []
    for i in range(1,termination+1):
        old = new
        a = new_point
        g = f.gradient()(*a)
        d = -g/(g.norm())
        d = vector([round(i,rounding) for i in d])
        step = backtracking_line_search(f,f.gradient(),a,d,step_size)
        new_point = a + step*d
        new = f(*new_point)
        all_points.append(new_point)
    return n(new_point), n(new), all_points

final_point, value, path1 = gradient_descent(f,vector([1.5,0]),2,6)
final_point, value, path2 = gradient_descent(f,vector([0.1,0.1]),2,6)

print("TABLE OF STEPS FOR INITITAL POINT (1.5,0)")
count1 = 0
location1 = 0
b1 = line([(1.5,0),path1[0]],color='green',thickness=5)
for i,j in zip(path1[:len(path1)-1],(path1[1:])):
    b1 += line([(i),(j)],color='green',thickness=5)
    count1 += 1
    location1 = [(i.n(digits = 2)),(j.n(digits = 2))]
    print("#: ", count1, "Convergence: ",  location1)

count2 = 0
location2 = 0
print("TABLE OF STEPS FOR INITITAL POINT (0.1,0.1)")
b2 = line([(0.1,0.1),path2[0]],color='red',thickness=5)
for i,j in zip(path2[:len(path2)-1],(path2[1:])):
    b2 += line([(i),(j)],color='red',thickness=5)
    count2 += 1
    location2 = [(i.n(digits = 2)),(j.n(digits = 2))]
    print("#: ", count2, "Convergence: ",  location2)

contour = contour_plot(f,(0,3),(0,3.14), contours = 100, fill = False, cmap='hsv', colorbar=true, figsize = 15) + line([(1.5,0),path1[0]]) + line([(0.1,0.1),path2[0]])

contour+b1+b2
```

Appendix D

```
def conjugate_gradient(f,a,steps,rounding=5,step_size = 5):
    fg = f.gradient()
    g = fg(*a)
    d = -g/(g.norm())
    d = vector([round(i,rounding) for i in d])
    g_1 = vector([I for i in range(len(a))])
    path = []
    for i in range(steps):
        step = backtracking_line_search(f,f.gradient(),a,d,step_size)
        a1 = a + step*d
        d1 = -fg(*a1)/(fg(*a1).norm())
        d1 = vector([round(i,rounding) for i in d1])
        B = max(0,((g*(g - g_1))/(g_1*g_1)))
        g1 = d1 + B*d
        path.append(a1)
        a = a1
        d = d1
        g_1,g = g, g1
    return a1,path

fin, path3 = conjugate_gradient(f,vector([1.5,0]),6,step_size = 0.5)
fin, path4 = conjugate_gradient(f,vector([0.1,0.1]),6,step_size = 0.5)

count1 = 0
location1 = 0
print("TABLE OF STEPS FOR INITITAL POINT (1.5,0)")
m1 = line([(1.5,0),path3[0]],color='blue',thickness=5)
for i,j in zip(path3[:len(path3)-1],(path3[1:])):
    m1 += line([(i),(j)],color='blue',thickness=5)
    count1 += 1
    location1 = [(i.n(digits = 2)),(j.n(digits = 2))]
    print("#: ", count1, "Convergence: ",  location1)

count2 = 0
location2 = 0
print("TABLE OF STEPS FOR INITITAL POINT (0.1,0.1)")
m2 = line([(0.1,0.1),path4[0]],color='black',thickness=3)
for i,j in zip(path4[:len(path4)-1],(path4[1:])):
    m2 += line([(i),(j)],color='black',thickness=3)
    count2 += 1
    location2 = [(i.n(digits = 2)),(j.n(digits = 2))]
    print("#: ", count2, "Convergence: ",  location2)

contour = contour_plot(f,(0,3),(0,3.14), contours = 100, fill = False, cmap='hsv', colorbar=true, figsize = 15) +
line([(1.5,0),path3[0]]) + line([(0.1,0.1),path4[0]])

contour+m1+m2
```

Appendix F

```
#function for area
f(a,o) = -0.5*(2*(3-a*sin(o)-a)+a*cos(o))*a*sin(o)

def backtracking_line_search(f, fg, a, d, alph, p=0.5, B=10^(-4)):
    o, g = f(*a), fg(*a)
    while (f(*(a + alph*d)) > o + B*alph*(g*d)):
        alph*= p
    return alph

def momentum(f, x, step_size, b, termination, rounding=5):
    g = f.gradient()(*x)
    d = -g/(g.norm())
    old = new = f(*x)
    new_point = x
    all_points = []
    d = vector([0,0])
    for i in range(1,termination+1):
        old = new
        x = new_point
        g = f.gradient()(*x)
        d = -g/(g.norm()) + b*d
        d = vector([round(i,rounding) for i in d])
        step = backtracking_line_search(f,f.gradient(),x,d,step_size)
        new_point = x + step*d
        new = f(*new_point)
        all_points.append(new_point)
    return n(new_point), n(new), all_points

final_point, value, path1 = momentum(f,vector([1.5,0]),2,0.1,6)
final_point, value, path2 = momentum(f,vector([0.1,0.1]),2,0.1,6)

print("TABLE OF STEPS FOR INITITAL POINT (1.5,0)")
count1 = 0
location1 = 0
b1 = line([(1.5,0),path1[0]],color='green',thickness=5)
for i,j in zip(path1[:len(path1)-1],(path1[1:])):
    b1 += line([(i),(j)],color='green',thickness=5)
    count1 += 1
    location1 = [(i.n(digits = 2)),(j.n(digits = 2))]
    print("#: ", count1, "Convergence: ",  location1)

count2 = 0
location2 = 0
print("TABLE OF STEPS FOR INITITAL POINT (0.1,0.1)")
b2 = line([(0.1,0.1),path2[0]],color='red',thickness=5)
for i,j in zip(path2[:len(path2)-1],(path2[1:])):
    b2 += line([(i),(j)],color='red',thickness=5)
    count2 += 1
    location2 = [(i.n(digits = 2)),(j.n(digits = 2))]
    print("#: ", count2, "Convergence: ",  location2)

contour = contour_plot(f,(0,3),(0,3.14), contours = 100, fill = False, cmap='hsv', colorbar=true, figsize = 15) +
line([(1.5,0),path3[0]]) + line([(0.1,0.1),path4[0]])

contour+b1+b2
```

Appendix G

```
#function for area
f(a,o) = -0.5*(2*(3-a*sin(o)-a)+a*cos(o))*a*sin(o)

def backtracking_line_search(f, fg, a, d, alph, p=0.5, B=10^(-4)):
    o, g = f(*a), fg(*a)
    while (f(*(a + alph*d)) > o + B*alph*(g*d)):
        alph*= p
    return alph

def newtons_method(f,a,steps,e,rounding=5,step_size = 5):
    d = vector([float('inf') for i in range(len(a))])
    k = 1
    H, fg = f.hessian(), f.gradient()
    all_points = []

    while d.norm() > e and k <= steps:
        d = H(*a)^(-1) * fg(*a)
        a -= d
        k += 1
        all_points.append(a)
    return a, all_points

final_point, path1 = newtons_method(f,vector([1.5,1]),6,0.000000001)
final_point, path2 = newtons_method(f,vector([0.1,0.1]),6,0.000000001)

print("TABLE OF STEPS FOR INITITAL POINT (1.5,0)")
count1 = 0
location1 = 0
b1 = line([(1.5,0),path1[0]],color='blue',thickness=5)
for i,j in zip(path1[:len(path1)-1],(path1[1:])):
    b1 += line([(i),(j)],color='blue',thickness=5)
    count1 += 1
    location1 = [(i.n(digits = 2)),(j.n(digits = 2))]
    print("#: ", count1, "Convergence: ",  location1)

count2 = 0
location2 = 0
print("TABLE OF STEPS FOR INITITAL POINT (0.1,0.1)")
b2 = line([(0.1,0.1),path2[0]],color='black',thickness=5)
for i,j in zip(path2[:len(path2)-1],(path2[1:])):
    b2 += line([(i),(j)],color='black',thickness=5)
    count2 += 1
    location2 = [(i.n(digits = 2)),(j.n(digits = 2))]
    print("#: ", count2, "Convergence: ",  location2)

contour = contour_plot(f,(0,3),(0,3.14), contours = 100, fill = False, cmap='hsv', colorbar=true, figsize = 15) +
line([(1.5,0),path1[0]]) + line([(0.1,0.1),path2[0]])

contour+b1+b2
```