A 2-D Computational Model for Coil Dynamics in Platinum Coil Embolization

Clayton Peacock

April 2020

1 Abstract

Over time, the force of blood flow in the brain can rupture aneurysms causing health complications including death. There are many medical techniques used to prevent aneurysm rupture: one of which is "Platinum Coil Embolization" in which a platinum coil is inserted and allowed to uncoil inside of an aneurysm, partway filling its volume. If enough of the aneurysm's volume is filled, the chance of rupture is drastically reduced. The ability to dynamically model the uncoiling of these coils inside aneurysms of various geometries has the potential to definitively increase the surgery's success rate. In this paper, a model is presented to dynamically model a 2-Dimensional coil inside of a 2D constraint which, when extended to model the system in 3-D, can be used to increase the success rate and tunability of the coil-inserting surgery.

2 Introduction

Arteries are the vessels which carry blood throughout our brains [1]. Due to various environmental and genetic factors, locations on some arteries can become

weak and thin. This, combined with constant pounding of blood through the artery, can cause an aneurysm [1]. Aneurysms often appear at junctions in arteries where the force of the blood is directed toward a junction, and the aneurysm is then formed at the vertex of that junction. Geometrically, an aneurysm is a ballooning bulge in the wall of an artery, and with enough time and blood flow this bulge will rupture causing "hemorrhagic stroke, brain damage, coma, and even death" [1].

Once an aneurysm is detected in the brain, it is imperative to try to prevent it from rupturing. This can be done in a variety of ways. In one method a metal mesh stent is inserting into the artery, covering the opening to the bulging aneurysm [1]. The stent doesn't prevent blood from flowing into the bulge, but it does slow it down enough to where the bulge has a much lower change of rupturing and the blood can pool and clot in the aneurysm. Another method is "Platinum Coil Embolization" which involves a tiny metal coil of platinum wire which itself is coiled very tightly like a spring. This wire is heat baked into a certain configuration and has the property of wanting to return to its original configuration when free to do so. The wire is straightened and carefully inserted through a catheter into the bulge of an patient's aneurysm where it is allowed to uncoil back to its original shape as it enters the bulge [1]. The wire is redirected by the aneurysm walls and eventually starts to fill the bulge (see Fig. 1). It has been found that filling 28% of an aneurysm is usually enough to greatly lower the chance of rupture. This process can be random and hard to predict. Thus it would be useful to have a way to model the dynamics of the coil upon entering the aneurysm. Therefore, in this paper a 2-Dimensional computational model is formulated to predict the dynamics of these coiled wires inside the aneurysm which can be extended to 3-Dimensions in the near future.

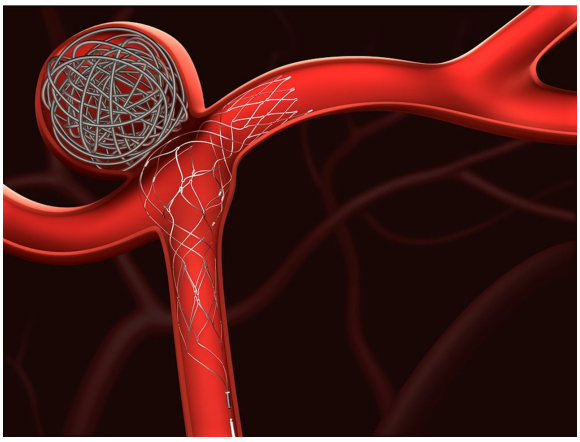

Figure 1: Artistic depiction of Platinum coil embolization [2]

3 Methods

3.1 The Coil

During the surgery the platinum coil first is twisted and baked into an initial configuration, then straightened, inserted into the aneurysm, and allowed to uncoil. Mathematically, we consider the wire to be made of many small discrete segments, and the wire is thought to be inserted one segment at a time. The energy is calculated at each position due to the angular difference between the segment's radial direction from its position of lowest energy (determined from the initial configuration), as well as a stiffness factor S_i determined by the material of the wire for each segment length $L_i,\ i \in \{i, 0, N\}$. The energy function is shown in Equations (1) and (2).

$$E_i = S_i \times (\Delta\theta_i)^2 \tag{1}$$

$$\Delta\theta_i = (\theta_i - \theta_0) \tag{2}$$

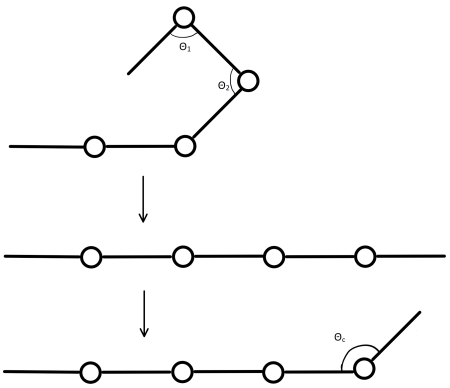

Figure 2: Initial Configuration $->$ Straightened $->$ Insertion

Computationally, the wire is divided into segments with N points and the energy is minimized in this manner for all points X_i. A simple generalized picture of this process is shown in Fig. (2) with initial angle θ_1 and final angle θ_c. For the 2-D case, segments will be allowed to cross to simulate the wires passing in front and behind themselves as they would in 3-D.

In general, one can obtain the position of each point $X_i \in X_0, X_1, \ldots X_N$ from a given array of angles θ_i:

$$X_{i+1} = X_i + L_{i,i+1} \begin{pmatrix} \cos\theta_i \\ sin\theta_i \end{pmatrix} \tag{3}$$

where $L_{i,i+1}$ represents the length of each segment. Thus we consider the array of segment angles to define the coil shape. There are many different ways to generate a coil from a given set of angles, not just the method shown in Equation (3). Throughout this project, the following equations were used to generate a Hypotrochoid coil [3] from a given array of angles:

$$\begin{aligned} X_x &= (R-r) * Cos(Angle) + d * Cos(((R-r) * Angle)/r) \\ X_y &= (R-r) * Sin(Angle) - d * Sin(((R-r) * Angle)/r) \end{aligned} \tag{4}$$

Here, R, r, and d are tunable variables and X_x, X_y represent the x and y components of each generated point X_i. Using these equations, one can generate a wide variety of coils of varying complexities and shapes.

We now have a method for generating a coil from a given array of angles. Given an initial configuration of angles, $\theta_0' s$, the energy equation (Eq. (1)) can be minimized using an optimization algorithm to find the final configuration, $\theta_i' s$, which will minimize the coil's energy. Without any constraints, the optimizer should find simply that $\theta_i = \theta_0$.

3.2 Constraints

The optimization problem becomes more difficult with the addition of constraints. The constraint in this minimization problem is that part of the coil cannot lie outside of the aneurysm. A ray-casting algorithm is used to implement this constraint. First, the shape of the aneurysm is modeled as a 2-D simple polygon. Then one considers a ray from each given point of the coil out to infinity and count the amount times it crosses the polygon's boundary. If the ray crosses an even amount of lines then the point lies outside of the curve, and if it crosses an odd number then the point is inside. To implement this in code, you must define the polygon by its segments and check systematically if a point's ray crosses an even or odd number of segments. A ray-casting algorithm created by Phillip Lemons [4] was used and adapted for this model.

There are varying degrees of implementation for the constraint in this minimization problem. One can first check if all points defining the coil lie inside the polygon. Then one can check if there are any line segments which lie partly outside the polygon even if their endpoints are contained inside. The former is implemented in this model however the latter can easily be added in the future. There are also various methods to implement constraints in an optimization

problem. For this model, an implementation of the Augmented Lagrange Constrained Minimizer was used (see the implementation in Appendix A) by first creating a function $c(x)$ which returns the integer number of coil points which lie outside a given polygon constraint. Then a new objective function, $g(x)$, is defined which contains the unconstrained objective function $f(x)$ (the energy of the coil, or Eq. (1) in our case) and a combination of the constraint function $c(x)$ with penalty multipliers r and v:

$$g(x) = f(x) + (1/r) * (c(x) - r/(2v))^2 \tag{5}$$

For each iteration of the optimizer, the penalty (1/r) is increased and v is redefined based on the current value for r and the constraint function evaluated at the current iteration's minimizing array, (xn):

$$v_i = 2 * c(xn)/r \tag{6}$$

In this method the algorithm first minimizes the unconstrained coil. Then the penalty for violating the constraint is increased gradually for each new iteration of the minimization and how much it is increased is based on how much the constraint was violated in the previous iteration.

3.3 Optimization

A Python code was written to perform the optimization of the coil inside of a simple polygon constraint (see Appendix A for the full code). For the constrained optimization, the gradient-free algorithm "Powell" is used as implemented in the Python package "Scipy" [5]. The code models the coil's dynamics as follows:

1. The coil starts with each point pinned just outside of the walls of the

constraint, simulating how the neurosurgeon holds the coil straightened in a catheter just outside the aneurysm in the patient's blood vessel.

2. The first point of the coil, X_1, is unpinned. Then consider the vector from the origin (where X_1 was pinned) to X_1. Call this vector $\bar{d}_1$ and the angle it makes with the horizontal θ_1. The optimizer is run to find the value for θ_1 that minimizes the energy of the coil (which right now is just the energy of this first segment). By defining θ_1, the optimizer defines the direction of $\bar{d}_1$, which further defines X_1 as the endpoint of $\bar{d}_1$. The minimum energy configuration, left unconstrained, would simply be the initial configuration. So in the optimization routine, after X_1 is first defined, the ray casting algorithm is run to determine if X_1 lies within the given constraint. If it does, then the optimizer continues, and if it doesn't, the optimizer runs again until it finds a position for X_1 which both minimizes the energy of the coil and satisfies the constraint.

3. The second point, X_2, is now unpinned. This means the coil is shifted into the constraint by one segment and the optimizer is run again, along with the ray casting algorithm, to determine the lowest energy configuration of the two points such that they both lie inside the constraint.

4. This process then continues for N points, unpinning the points of the coil one at a time and running the optimizer each time. Through this method, the dynamics of the coil are simulated effectively, modeling the coil uncoiling in a given constraint.

3.4 Example

Consider a Heptagon shaped constraint and a Hypotrochoid coil which is generated using 25 segments with $R, r, d = 2, 13, 15$. The result of optimizing such

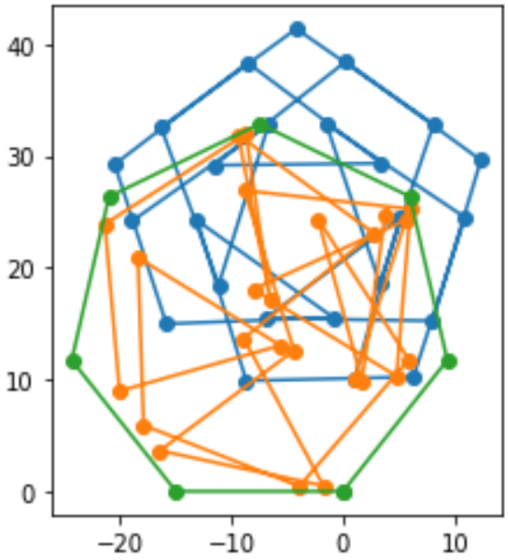

Figure 3: Example Hypotrochoid Coil (25 intervals, $R, r, d = 10, 1, 4$) and a Heptagon constraint

a coil inside the Heptagon is shown in Fig. (3) where the constraint is shown in green, the original shape of the coil is shown in blue, and the final shape of the coil is shown in orange. Note that three points of the coil lie outside the constraint initially but are effectively constrained to lie inside the Heptagon in the final configuration.

4 Conclusion

In conclusion, a Python code has been developed which models the dynamics of the Platinum Coil Embolization neurosurgery in 2-Dimensions through an optimization routine. This code can be built upon and improved in the future so that it models coils and aneurysms in 3-Dimensions (in which the coils will not be allowed to cross themselves). When the 3D code is formulated, it will

allow a neurosurgeon to predict how a given coil will unfurl inside of a given aneurysm before performing the surgery. As shown previously, the outcome of the optimization depends on the initial configuration of the coil and the geometry of the constraint. From a neurosurgery perspective, the geometry of the constraint (aneurysm) cannot be controlled. However, due to the properties of the metal used to fabricate it, the coil's initial configuration can be controlled and thus its dynamic uncoiling inside of a patient's aneurysm can be predicted. The model can also be used to predict the behavior of a coil if it is inserted part-way, taken out, and reinserted. Thus, when created, the 3-D model will allow neurosurgeons to carefully choose the initial configuration of coils for each patient's uniquely shaped aneurysm. This will give neurosurgeons the ability to better predict the dynamics of their surgeries and increase the success rate of the Platinum Coil Embolization neurosurgery, saving lives.

References

[1] NINDS. Cerebral aneurysms fact sheet, 2018.

[2] Benjamin Gory, Raphaël Blanc, Francis Turjman, Jérôme Berge, and Michel Piotin. The barrel vascular reconstruction device for endovascular coiling of wide-necked intracranial aneurysms: a multicenter, prospective, post-marketing study. *Journal of NeuroInterventional Surgery*, 10(10):969–974, 2018.

[3] Eric W. Weisstein. Hypotrochoid. from mathworld–a wolfram web resource.

[4] Phillip Lemons. Ray casting algorithm, 2017.

[5] The SciPy community. minimize(method='powell'), 2019.

5 Appendix A

```
import numpy as np
from matplotlib import pyplot
from scipy.optimize import minimize
#From this webpage - credit to Phillip Lemons:
#http://philliplemons.com/posts/ray-casting-algorithm
class Point:
    def __init__(self, x, y):
        """
        A point specified by (x,y) coordinates in the cartesian plane
        """
        self.x = x
        self.y = y
class Polygon:
    def __init__(self, points):
        """
        points: a list of Points in clockwise order.
        """
        self.points = points
    @property
    def edges(self):
        ''' Returns a list of tuples that each contain 2 points of an edge '''
        edge_list = []
        for i,p in enumerate(self.points):
            p1 = p
            p2 = self.points[(i+1) % len(self.points)]
            edge_list.append((p1,p2))
```

```
    return edge_list
def contains(self, point):
    import sys
    # _huge is used to act as infinity if we divide by 0
    _huge = sys.float_info.max
    # _eps is used to make sure points are not on the same line as vertexes
    _eps = 0.00001
    # We start on the outside of the polygon
    inside = False
    for edge in self.edges:
        # Make sure A is the lower point of the edge
        A, B = edge[0], edge[1]
        if A.y > B.y:
            A, B = B, A
        # Make sure point is not at same height as vertex
        if point.y == A.y or point.y == B.y:
            point.y += _eps

        if (point.y > B.y or point.y < A.y or point.x > max(A.x, B.x)):
            # The horizontal ray does not intersect with the edge
            continue
        if point.x < min(A.x, B.x): # The ray intersects with the edge
            inside = not inside
            continue
        try:
            m_edge = (B.y - A.y) / (B.x - A.x)
```

```
            except ZeroDivisionError:
                m_edge = _huge
            try:
                m_point = (point.y - A.y) / (point.x - A.x)
            except ZeroDivisionError:
                m_point = _huge
            if m_point >= m_edge:
                inside = not inside
                continue
        return inside
#MyPolygon: Pass True for plot, False for points
def MyPolygon(n, L, plot=False):
    Angles = np.ones(n)*(360/n)
    Angles = np.radians(np.array(Angles))
    current_x = Point(0,0)
    current_theta = 0
    xs = [0]
    ys = [0]
    points = [Point(xs,ys)]
    #Want xs and ys to be points in clockwise order
    for theta in Angles:
        start = current_x
        current_theta += theta
        end = Point(start.x + L*np.cos(current_theta),start.y + L*np.sin(current_theta))
        xs.append(end.x)
        ys.append(end.y)
        points.append(end)
```

```
        current_x = end
    if plot:
        pyplot.plot(xs,ys,"-o")
        #pyplot.scatter(p1.x,p1.y)
        #pyplot.scatter(p2.x,p2.y)
        pyplot.show()
    else:
        xs = xs[::-1]
        #xs = xs[:len(xs)-1]
        ys = ys[::-1]
        #ys = ys[:len(ys)-1]
        Points = []
        polypoints = []
        for i in range(len(xs)):
            p = Point(xs[i],ys[i])
            pp = [xs[i],ys[i]]
            polypoints.append(p)
            Points.append(pp)
        return polypoints

# ------------------------------------------------------------
# Optimizing
#Functions needed to to create coil (Hypotrochoid in this case)

#Function to create hypotrochoid from polar angle and return points
def HypoPoints(R,r,d,Intervals):
    dAngle = (4*np.pi-0)/Intervals
```

```
    Angles = np.arange(0,4*np.pi,dAngle)
    points = []
    for Angle in Angles:
        point = [(R-r)*np.cos(Angle)+d*np.cos(((R-r)*Angle)/r),(R-r)*np.sin(Angle)-
        d*np.sin(((R-r)*Angle)/r)]
        points.append(point)
    return np.array(points)

#Function to turn hypotrochoid points into segment vectors
def HypoVectors(points):
    d = []
    current_point = points[0]
    for point in points[1::]:
        d_ = point - current_point
        current_point = point
        d.append(d_)
    return np.array(d)

#Function to turn segment vectors into local angles
def HypoAngles(Vectors):
    Angles = []
    current_vec = Vectors[0]
    for vector in Vectors[1::]:
        angle = np.arccos(np.dot(current_vec,vector)/(np.linalg.norm(current_vec)
        *np.linalg.norm(vector)))
        Angles.append(angle)
        current_vec = vector
```

```
    return np.array(Angles)


#Define hypotrochoid parameters
R,r,d,l = 10,1,4,15
Intervals = 20
Angle = HypoAngles(HypoVectors(HypoPoints(R,r,d,Intervals))) # Initial Coil Configuration
Stiffness = np.ones(Angle.size)
Constraint = Polygon(MyPolygon(7,15,False))


#Function to create coil (Hypotrochoid in this case)
def Hypotrochoid(Thetas,R,r,d,l,plot=False):
    current_x = Point(1,10)
    current_theta = 0
    xs = [1]
    ys = [10]
    points = [Point(xs,ys)]
    for theta in Thetas:
        start = current_x
        current_theta += theta
        end = Point(start.x + l*np.cos(current_theta),start.y
        + l*np.sin(current_theta))
        xs.append(end.x)
        ys.append(end.y)
        points.append(end)
        current_x = end
    if plot:
        pyplot.plot(xs,ys,"-o")
```

```
        pyplot.gca().set_aspect('equal')
#       pyplot.show()
    else:
        Points = []
        polypoints = []
        xs = np.array(xs)
        ys = np.array(ys)
        for i in range(xs.size):
            p = Point(xs[i],ys[i])
            pp = [xs[i],ys[i]]
            polypoints.append(p)
            Points.append(pp)
        return polypoints

#Objective function
def EnergyMin(x):
    thetas_f = x[0:Angle.size]
    E = Stiffness*(Angle-thetas_f)**2
    Energy = np.sum(E) #Energy
    return Energy

def AugmentedLagrangeMethod(f, c, x0, v0, r0, beta, tol=1e-6,
maxIter=100, maxTotalIter=5000):
    v = v0
    r = r0
    if beta > 1:
        beta = 1/beta
```

```
    print("Penalty scaling (beta) is greater than one so taking reciprocal")
totalIter = 0
nIter = 0
done = False
header_format = "\n{:^7} {:^7} {:^7} {:^10}
{:^10} {:^10} {:^10} {:^10} {:^10}"
row_format = "{:^7} {:>7.1e} {!s:^7} {:^10} {:^10} {:>10.3e}
{:>10.3e} {:>10.3e} {:>10.3}"
print(header_format.format('Solve #', 'r', 'Success', 'Iterations', 'Total Iter',
'f(x,y)', 'g(x,y)', 'L(x,y)','lambda'))
print("-"*91)
while not done:
    objective = lambda x: f(x) + (1/r)*(c(x) - r/2*v)**2
    sol = minimize(objective, x0, method="Powell")
    totalIter += sol.nit
    xn = sol.x
    if sol.success == False:
        done = True
        message = 'Local Minimizer Failed with Message: "{}"'.format(sol.message)
    if c(xn) < tol:
        done = True
        message = "Constraint Satisfied with Tolerance"
    if nIter == maxIter:
        done = True
        message = "Maximium Iterations Reached"
    v -= 2*c(xn)/r
    x0 = xn
```

```
        r *= beta
        nIter += 1
        print(row_format.format(nIter, r, sol.success,
        sol.nit, totalIter, f(sol.x), c(sol.x), sol.fun, v))
    print(message)
    return xn, c(xn), message
n,L = 7,15
def Constraints(x):
    ps = Hypotrochoid(x,R,r,d,l,False)
    count = 0
    for p in ps:
        if not Constraint.contains(p):
            count += 1
    return count
print('Original shape of coil:')
Hypotrochoid(Angle,R,r,d,L,True) #Plot initial coil
x0 = np.random.randn(Angle.size) #Intial vector
res = minimize(EnergyMin, x0, method='Powell',options={'disp':
True}) #Optimizing
print('Shape after optimization:')
v0 = 1
r0 = 1
beta = 1/2
x, C, message = AugmentedLagrangeMethod(EnergyMin,Constraints, x0, v0, r0,
beta, maxIter=1000)
Hypotrochoid(x,R,r,d,l,True) #Plot final coil
MyPolygon(n,L,True)
```