Paul Spelt

Paul.spelt95@gmail.com

Contents

[Introduction: 3](#_Toc70842915)

[Problem Description: 3](#_Toc70842916)

[Model Description and Assumptions: 4](#_Toc70842917)

[Population Model: 4](#_Toc70842918)

[Drone Service Time Model: 4](#_Toc70842919)

[Weather Model: 5](#_Toc70842920)

[Wind Gust Model: 6](#_Toc70842921)

[Order Arrivals: 7](#_Toc70842922)

[Other Considerations: 8](#_Toc70842923)

[Demonstration of System Dynamics: 8](#_Toc70842924)

[Results: 9](#_Toc70842925)

[Conclusions and Future Improvements: 13](#_Toc70842926)

[Bibliography 14](#_Toc70842927)

[Appendix – Wind Gust Derivation: 15](#_Toc70842928)

[Appendix - Code: 15](#_Toc70842929)

[PythonSim Updates: 15](#_Toc70842930)

[Imports: 16](#_Toc70842931)

[Population Data and Sampling Functions: 17](#_Toc70842932)

[Distance and Bearing Functions: 18](#_Toc70842933)

[Service Time Function: 19](#_Toc70842934)

[Weather Data and Functions: 20](#_Toc70842935)

[Classes For Updating Weather State and Logging Delivery Metrics: 21](#_Toc70842936)

[UAVSim Function: 23](#_Toc70842937)

[Confidence Intervals: 29](#_Toc70842938)

[Example of Running Simulation: 29](#_Toc70842939)

[System Dynamics Experiments: 30](#_Toc70842940)

[Policy Definitions: 31](#_Toc70842941)

[Subset Selection Example: 35](#_Toc70842942)

[Plot Delivery Coordinates: 37](#_Toc70842943)

# Introduction:

This project is based off the paper “Analysis of Future UAS-Based Delivery – Balaban, Mastaglio, Lynch” [1]. The paper assessed the commercial use of UAS (Unmanned Aerial System) by small businesses via distribution centers. A future operation model was envisioned based on factors such as drone velocity, flying altitude, number of drones, delivery demand and number of orders completed.

This project aims to refine the modeling approach using more sophisticated data and techniques to evaluate the impact of weather on a drone-based delivery system. Orders will be sampled from population data representing the Greater Toronto Area (GTA) and weather conditions sampled from historical data sourced from Environment Canada. Random adverse weather events will be modeled using historical wind gust data sourced from the German National Meteorological Service. Different operating policies will be compared to evaluate the impact of weather conditions on a drone delivery system.

# Problem Description:

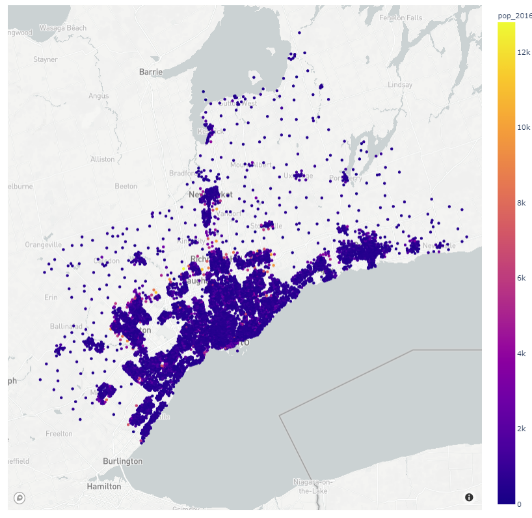
The evaluation metric considered is the number of completed deliveries. From a business point of view this metric makes the most sense as it maximizes the expected utility of the drone delivery system. The metric is also interesting in that it can accommodate for the systems ability to avert risk. This simulation assumes that drones are required to abort their delivery if they experience wind gusts in excess of their safety threshold. If the system does not effectively accommodate for risk, it will inevitably lose more drones therefore reducing its delivery capacity.

A secondary metric which will be considered is the number of lost drones. While some policies may be able to process a comparable number of deliveries, they may do so at the cost of more lost drones. This is an additional cost to consider when evaluating policies.

# Model Description and Assumptions:

This simulation model leverages a modified version of the “PythonSim” library. PythonSim is provided by Professor Barry Nelson of Northwestern University and provides simple python classes for discrete event simulation as well as random number generation [2]. This package was modified slightly to suit the specific needs of this simulation.

## Population Model:

Dissemination block (DB) population data is available for Toronto from 2016. This provides population at a very granular level and is suitable for sampling the coordinates of order demand.

These DB coordinates were filtered for the GTA and are plotted in Figure 1. It is sufficient to simply sample the coordinates weighted by population as the dissemination blocks provide data at very a granular level (for example one dissemination block may be equivalent to the area of a downtown street block). A theoretical distribution center was manually selected in North York near Highway 401.

Figure : Population Distribution GTA 2016

## Drone Service Time Model:

Once coordinates are sampled, the distance of the coordinate relative to the distribution center is determined. Flight paths are assumed to be “as the crow flies” and obstacles such as airports, office towers, etc. are not considered. For the purposes of this simulation, we will assume the drones are quadcopters with symmetrical drag profiles and that wind speed impacts the flight time irrespective of wind bearing. Imagine two flight paths, the first is a trip with a direct headwind/tailwind and the second is a trip with a perpendicular crosswind. For simplicity we will assume the net impact of wind is equivalent for both trips (i.e., the lost time from the drone “crabbing” or angling into the cross-wind for the trips entire duration is equivalent to the lost time flying into the headwind/tailwind). For example, let drone speed be 25km/h, wind speed 5 km/h and one-way distance 50km:

Therefore, we can see the effect wind has on total flight time in a direct headwind/tailwind flight. This model assumes similar lost time would occur if the drone experienced equivalent cross-winds for the entire duration of the flight.

The package delivery service time is modelled as a 2-Phase Erlang distribution with a mean of 5 minutes. This accounts for the time it takes the recipient to meet the drone and the time it takes for the recipient to sign for the package.

The drones are assumed to have a top speed of 80km/h based on research of commercial drones currently available for use in Canada [3]. As a rule of thumb, the max wind speeds a drone can safely fly in are equivalent to two-thirds of its top speed [4]. Therefore, we will assume that the drones can handle wind speeds up to 50km/h. Should the wind speed exceed this threshold the drone will be forced to emergency land.

## Weather Model:

Weather data is sourced from Environment Canada from March 2020 to 2021. Drones will operate on a “9-5” schedule and the hourly data for this period will be stepped through as discrete events. Modeling weather as a random variable would be a complex task due to the interaction effects between wind direction, wind speed and precipitation. It is not as straightforward as sampling a single value because an entire timeseries needs to be sampled to model the weather conditions over the course of a day.

The randomness of weather can be represented more easily by collecting a long timeframe of historical data which can sufficiently cover the range of conditions the system could expect to see. Furthermore, this approach allows the system to easily leverage Common Random Numbers (CRN) to reduce variance across simulations. Recall:

The variance of the difference between two estimators, θ(*x*), is decreased if there is covariance between them [5]. Therefore, to apply CRN, both simulations being compared will use the same source of randomness to introduce covariance. This will be done by simulating policies over the exact same days.

## Wind Gust Model:

The paper “Maximum Daily Wind Gusts Related to Mean Daily Wind Speed” by Richard Weggel 1999 is used for the derivation of the wind gust model [6]. This model can be used to estimate the risk of a wind gust exceeding a threshold in a day based on the gust factor and its standard deviation. Let the gust factor be:

Where *G* is the gust factor, *U* is the wind speed and *ug* is the wind gust speed. Using the CDF of a normal probability distribution one can find the probability that the wind gusts remain under the threshold using 1 – F(x) (See Appendix for the derivation). This model is applied to 10-minute wind gust and wind speed data from the Germany National Meteorological Service (Deutscher Wetterdienst) [7]. This allows the derivation of wind gust factors and their standard deviations at 10-minute intervals and the use of this model for estimating the probability of a wind gust exceeding a threshold in the next 10 minutes. For example, we can plot the risk of a gust exceeding 50km/h for the month of June:

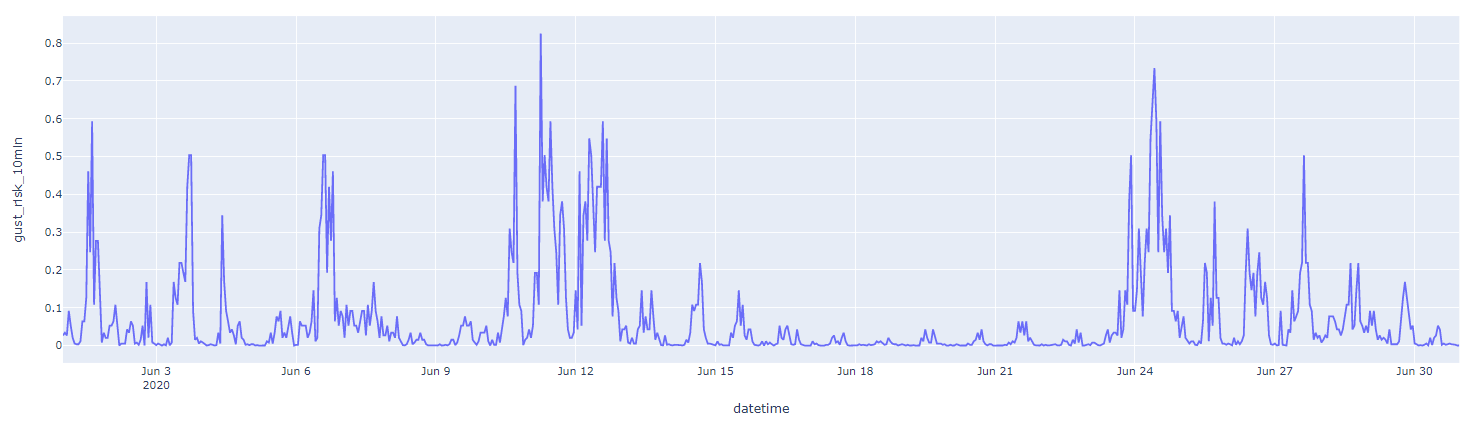


Figure : Probability of Gust Exceeding 50km/h in June 2020

When orders are simulated, the service time is divided into 10-minute intervals to determine how many times the gust risk distribution is sampled. For example, say a trip is 30 minutes, the gust risk is sampled 3 times. This forces the drones to account for the increased risk of longer duration journeys.

## Order Arrivals:

A non-stationary Poisson arrival process (NSPP) is leveraged. There is no order demand data for our proposed delivery system, so we can construct an arrival process for our needs. The drones are only permitted to fly within working hours from 9am to 5pm. The system starts with 10 drones and we will set the maximum order arrival rate to 20/hour from noon to 1pm with rates tapering to a minimum of 12/hour. This data will be used to demonstrate a thinning algorithm which can be fit to empirical order volume observations.

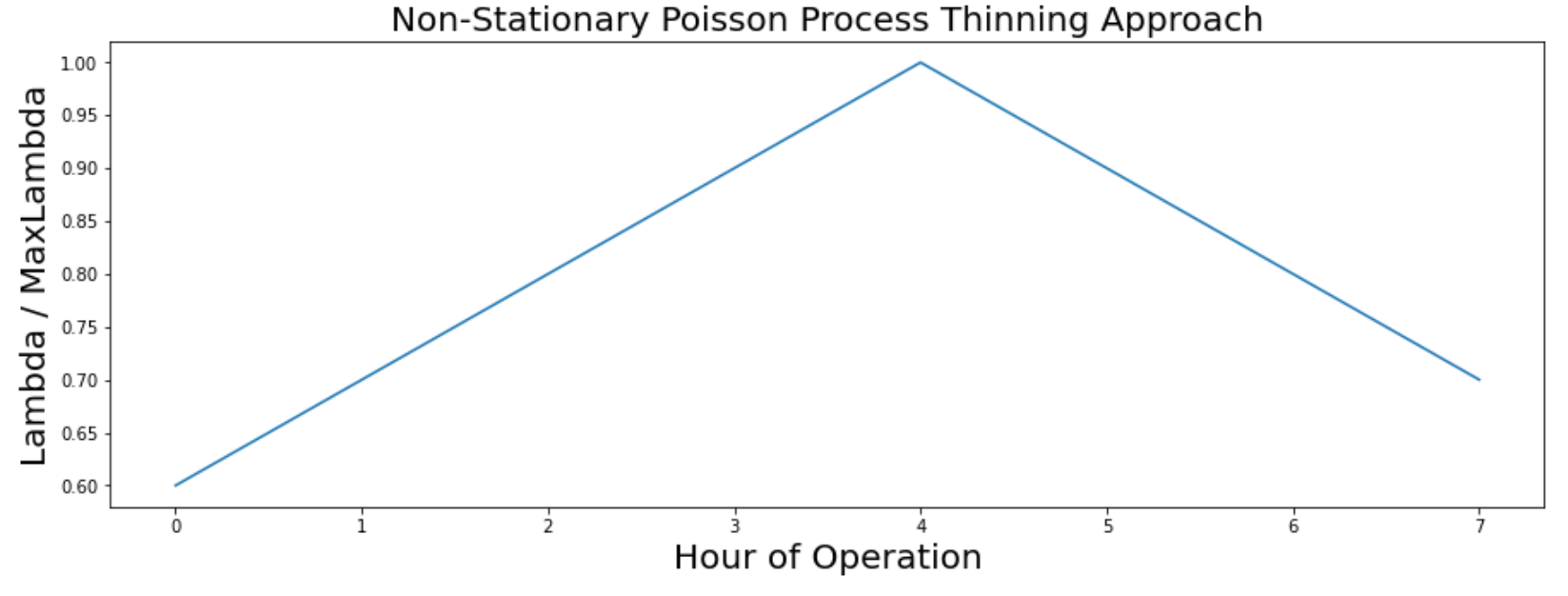


Figure : Ratio of Lambda/MaxLambda By Hour of Operation Used for Thinning Algorithm

## Other Considerations:

Unlike other queuing systems such as M/G/1, there is no warmup period or steady state in this model. This is because the simulation runs over a constrained timeframe and because the system state is constantly in flux. For example, the demand is non-stationary throughout the day, weather conditions change hourly, policies can introduce complex behaviour and the number of drones available for deliveries can change based on random weather events.

# Demonstration of System Dynamics:

The system was tested using a Naïve policy and where wind speed was modeled as a constant value throughout the day for speeds from 5km/h to 45km/h. As the wind speed increased the response of metrics such as delivery time, queue length and number of lost drones were recorded. The system remains robust until the wind speed approaches the maximum threshold, this demonstrates the need for careful policy design.

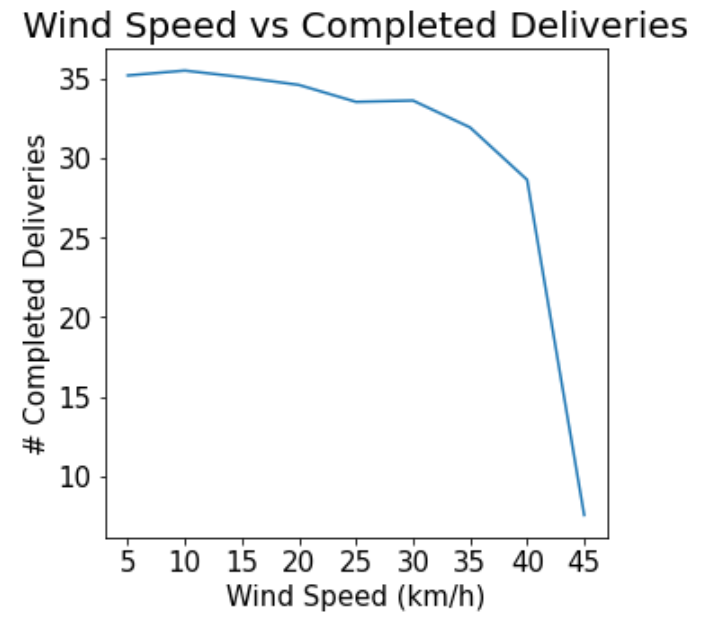
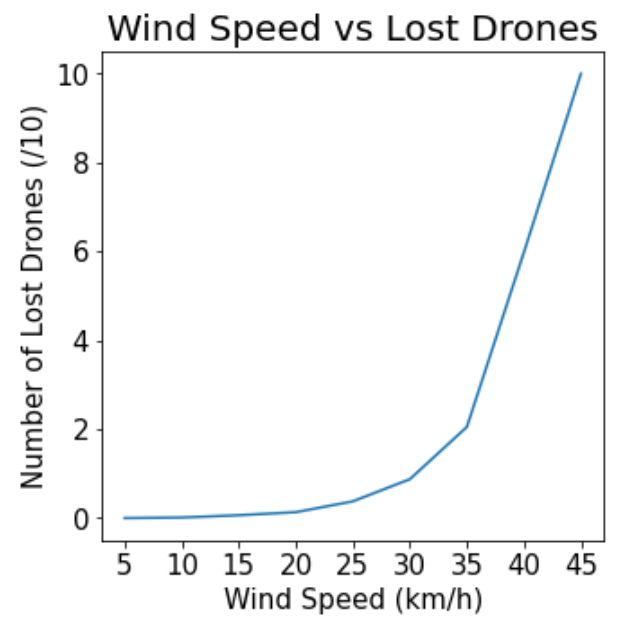
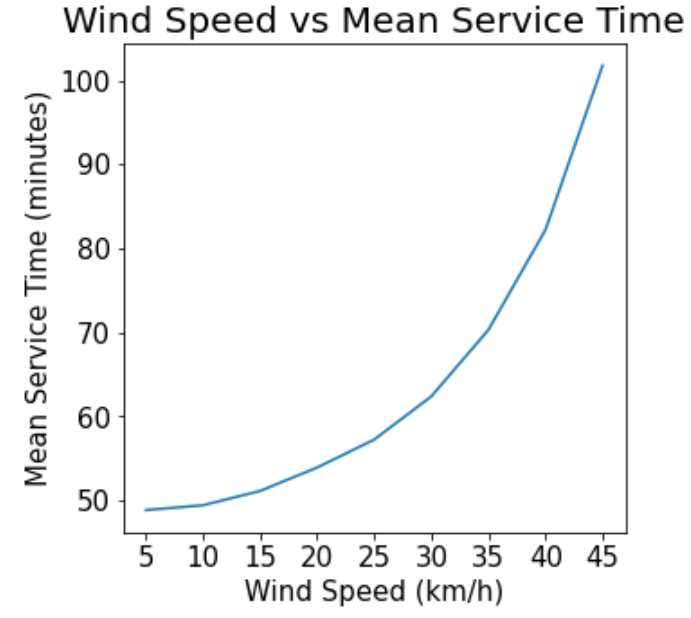


Figure : Wind Speed vs Service Time, Lost Drones and Number of Completed Deliveries

Policies can be implemented which evaluate delivery risk prior to acceptance. This will be done by estimating the service time of the trip and determining the binomial probability of a wind gust occurring. Recall the wind gust risk gives the probability of a wind gust exceeding the threshold in the next 10-minute interval. If a trip is expected to span *n* 10-minute intervals, the gust risk is sampled *n* times. The binomial distribution parameter *p* is equivalent to the gust risk and *k* is set to 0. This will give the probability of 0 wind gusts occurring in the next *n* trials.

# Results:

Subset Selection is used to differentiate between the policies with confidence [5]. Subset selection delivers a set of feasible scenarios *I* with a guarantee that:

Recall the t-quantiles are calculated with:

Where *K* is the number of policies in the set of feasible scenarios *I,* and *n* is the number of replications. As *K* increases, so does the t-quantile. Pairwise estimation error thresholds are calculated between policies *i* and *h*, based on the respective variances *S2*:

And the final subset selection is calculated as:

A policy joins the final subset selection if it is better than every other policy and their respective estimation error thresholds. The thresholds account for the variance of both policies and the level of

confidence specified (0.05 or 95% confidence in most cases). Seven policies were experimented with initially.

Table : Tested Policies

|  |  |
| --- | --- |
| Policy | Description |
| Policy 0 | Naïve dispatch |
| Policy 1 | Dispatch immediately to minimal risk |
| Policy 2 | Dispatch to minimal risk if Pr{0 Gusts} > 50% |
| Policy 3 | Dispatch to minimal risk if Pr{0 Gusts} > 25% |
| Policy 4 | Dispatch immediately to minimal distance |
| Policy 5 | Dispatch to min distance if < 10km |
| Policy 6 | Dispatch to min distance if < 40km |

The Naïve policy represents the baseline performance, the distance controlled policies represent a simplistic approach to minimalizing risk and the risk controlled policies represent an intelligent method for managing risk.

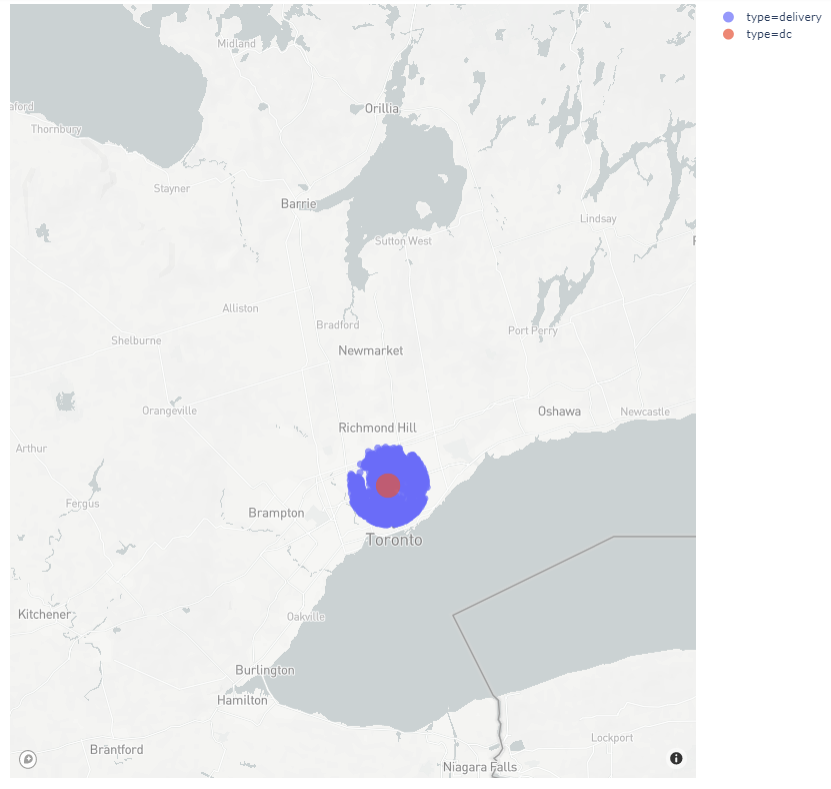
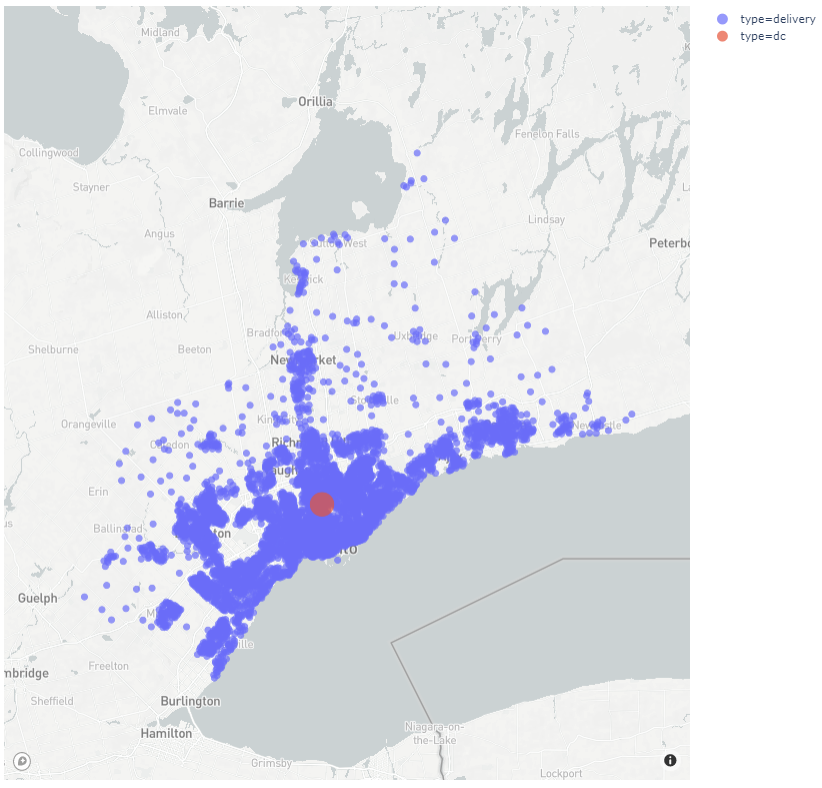
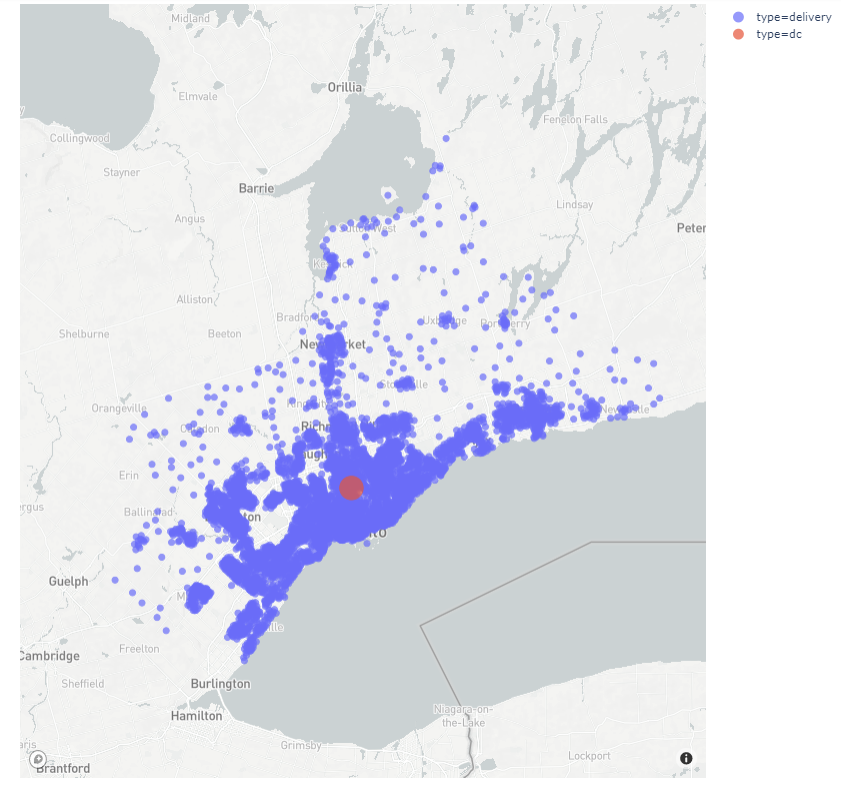
These policies were tested on sampled weather conditions from the entire year and run on 1000 replications each. The results show that the subset selection includes Policies 0, 1, 2, 3 and 4 as the best policies for maximizing the number of deliveries with 95% confidence. Subset selection applied to the number of drones selects Policy 5 as the best with 95% confidence.

Table : Initial Simulation Results Across Policies

|  |  |  |
| --- | --- | --- |
| Policy | Mean Deliveries / 95% CI | Lost Drones / 95% CI |
| Policy 0: Naïve | 14.99 +/- 0.53 | 2.88 +/- 0.19 |
| Policy 1: Minimize Risk | 14.84 +/- 0.54 | 2.92 +/- 0.19 |
| Policy 2: Risk Threshold 50% | 14.83 +/- 0.48 | 1.68 +/- 0.10 |
| Policy 3: Risk Threshold 25% | 15.18 +/- 0.52 | 2.29 +/- 0.14 |
| Policy 4: Minimize Distance | 14.96 +/- 0.54 | 2.92 +/- 0.19 |
| Policy 5: Distance Threshold 10km | 6.78 +/- 0.13 | 1.24 +/- 0.12 |
| Policy 6: Distance Threshold 40km | 10.26 +/- 0.32 | 2.09 +/- 0.17 |

While the policies appear very similar based solely on the number of completed deliveries, a cost-benefit analysis shows that Policy 2 is preferable when we factor in the number of lost drones. This policy is able to maintain equivalent delivery volume without losing as many drones.

To further understand these results, the weather conditions and delivery distances must be analyzed in greater detail. The coordinates of the completed deliveries for the Naïve policy, the 50% threshold policy and the 10km threshold policy are shown below in Figure 5.



***Policy 5***

***Policy 2***

***Policy 0***

Figure : Simulated Delivery Coordinates Across Different Policies

The distribution center is shown in red and the completed deliveries are shown in blue. For the Naïve policy, drones cover deliveries across the GTA while the 10km policy lets drones focus on shorter, faster and safer deliveries. The impact of this is that only 1.2 drones are lost compared to 2.9 drones in the Naïve system. However, the penalty of restricting the delivery distance is that it artificially reduces order demand for the system. The benefits of focusing on shorter and faster deliveries are outweighed by the costs of rejecting so many deliveries. However, when we look at the 50% risk policy, we can see that it offers the best of both worlds. It matches the throughput and range of the Naïve system while losing a comparable number of drones to the 10km restricted system.

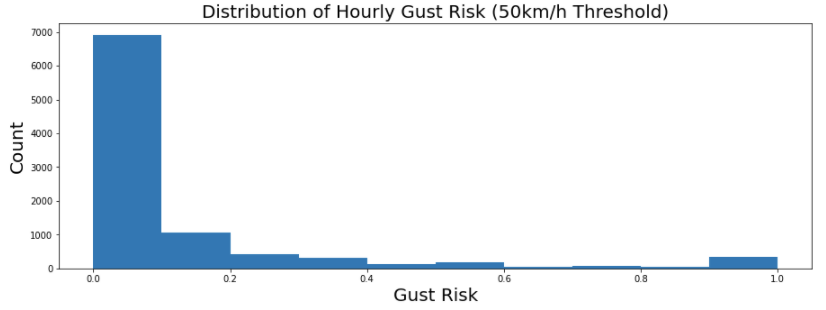
To understand why it is difficult to differentiate between these policies we also need to analyze the weather conditions from the past year. In Figure 6 we can see that the vast majority of days have a 10-minute gust risk within the 0-10% range. Using the binomial probability calculations, this suggests that drones can be guaranteed a 50% probability of not experiencing a 50km/h gust for trips up to 60 minutes in length. This is a very generous window and causes the risk averse systems to behave similarly to the Naïve systems. 

Figure : Distribution of Gust Risk Values (50km/h Threshold)

To better evaluate these policies, they will be simulated with 1000 replications of inclement weather conditions. Inclement weather is defined as days where the gust risk exceeds 50% within the operating hours of the system. Under these conditions the only trips which can be guaranteed probabilities of 0 gusts over 50% are trips which are less than 10 minutes long.

Table : Inclement Weather Simulation Results Across Policies

|  |  |  |
| --- | --- | --- |
| Policy | Mean Deliveries / 95% CI | Lost Drones / 95% CI |
| Policy 0: Naïve | 8.56 +/- 0.46 | 6.01 +/- 0.21 |
| Policy 1: Minimize Risk | 8.46 +/- 0.45 | 5.98 +/- 0.21 |
| Policy 2: Risk Threshold 50% | 9.51 +/- 0.41 | 2.56 +/- 0.11 |
| Policy 3: Risk Threshold 25% | 9.46 +/- 0.42 | 3.92 +/- 0.14 |
| Policy 4: Minimize Distance | 8.83 +/- 0.46 | 5.91 +/- 0.21 |
| Policy 5: Distance Threshold 10km | 6.50 +/- 0.18 | 3.53 +/- 0.18 |
| Policy 6: Distance Threshold 40km | 6.73 +/- 0.31 | 4.85 +/- 0.22 |

From the results above subset selection selected Policies 2, 3 and 4 as the best for maximizing delivery throughput, however it was unable to differentiate between them with 95% confidence. The subset selection chose Policy 2 as the best for the number of lost drones. So as before, a cost benefit analysis shows that Policy 2 is optimal, losing only 2.6 drones, the lowest out of all the policies. For comparison, the Naïve policies lose more than 6 drones on average to inclement weather and the 10km distance policy loses more than 3.5 drones.

Having selected Policy 2 as the best performing policy we can refine this analysis even further by testing a range of similar policies with thresholds including 30%, 40%, 50%, 60% and 70%. Choosing less policies for the subset analysis will increase the algorithms’ ability to differentiate between them (recall the impact of the *K* parameter on the t-quantiles). And running the policies over 10,000 replications will reduce the estimation errors.

Table : Refined Inclement Weather Simulation Results

|  |  |  |
| --- | --- | --- |
| Policy | Mean Deliveries / 95% CI | Mean Number of Lost Drones |
| Risk Threshold 30% | 9.40 +/- 0.13 | 3.69 +/- 0.04 |
| Risk Threshold 40% | 9.45 +/- 0.13 | 3.16 +/- 0.04 |
| Risk Threshold 50% | 9.42 +/- 0.12 | 2.55 +/- 0.03 |
| Risk Threshold 60% | 9.23 +/- 0.12 | 1.99 +/- 0.03 |
| Risk Threshold 70% | 8.56 +/- 0.12 | 1.30 +/- 0.03 |

Subset selection shows thresholds of 30-50% as the optimal thresholds for delivery throughput. It is unnecessary to test thresholds down to 20% or 10% as the cost benefit analysis shows that 50% is the best scenario. A 50% threshold matches the delivery throughput of less restrictive systems while minimizing the number of lost drones by nearly 60%. Policies which are less restrictive lose more drones and policies which are more restrictive have lower delivery throughput.

# Conclusions and Future Improvements:

This simulation was designed to stay faithful to real world conditions. To accomplish this the most accurate drone specifications, population data and historical weather data were leveraged. The results suggest that the adverse effects of weather do not have a significant impact on the delivery output of drones over the course of a year. Commercial drones with 80km/h top speeds are capable of handling wind gusts up to 50km/h and on any given day the risk of a wind gusts is so low that it is difficult to differentiate between a Naïve policy and a risk averse policy.

Weather effects are notable in that they can force drones into emergency landings, and policies are required to mitigate these additional operating costs. This study found that policies which leverage wind gust models and binomial probabilities can maintain delivery throughput while reducing lost drones by up to 40% under normal operating conditions. Furthermore, when focusing exclusively on inclement weather conditions, the risk averse policies can improve system throughput by more than 10% while minimizing the number of lost drones by up to 60%.

The most significant improvement to this model could be made by implementing NOAA weather radar data for more precise weather estimations. This could be combined with more sophisticated flight models factoring in wind bearing, correction angles, flying altitude, charging times and flight paths to avoid airports, buildings and weather radar zones.

# Bibliography

|  |  |
| --- | --- |
| [1] | M. Balaban, T. Mastaglio and C. Lynch, "Analysis of Future UAS-Based Delivery," *2016 Winter Simulation Conference,* 2016. |
| [2] | B. Nelson, "Foundations and Methods of Stochastic Simulation," Northwestern University, [Online]. Available: http://users.iems.northwestern.edu/~nelsonb/IEMS435/. |
| [3] | Drone Delivery Canada, "Drone Delivery Canada Technology," 2021. [Online]. Available: https://dronedeliverycanada.com/technology/. |
| [4] | M. Kuss, "Can a Drone Fly in Strong Winds," 2021. [Online]. Available: https://tipsfordrones.com/can-a-drone-fly-in-strong-winds/. |
| [5] | B. Nelson, Foundations and Methods of Stochastic Simulation, Springer, 2013. |
| [6] | R. Weggel, "Maximum Daily Wind Gusts Related to Mean Daily Wind Speed," *Journal of Structural Engineering,* 1999. |

# Appendix – Wind Gust Derivation:

Using logarithms to the base 10:

# Appendix - Code:

## PythonSim Updates:

Customer Entity Class updated to store delivery data:

1. **class** Entity():
2. *# This is the generic Entity that has a single attribute CreateTime*
3. **def** \_\_init\_\_(self,coord,distance,service\_time,risk):
4. *# Executes with the Entity object is created to initialize variables*
5. ***# Add additional problem-specific attributes here***
6. self.CreateTime = Clock
7. self.Classtype = 0
8. self.coord = coord
9. self.distance = distance
10. **self.service\_time = service\_time**
11. self.risk = risk

FIFOQueue Class updated with a “pop” function which removes a specified index from the list. Allows for more dynamic policies than simply “first out”:

1. **class** FIFOQueue():
2. *# This is a generic FIFO Queue object that also keeps track*
3. *# of statistics on the number in the queue (WIP)*
5. **def \_\_init\_\_(self):**
6. *# Executes when the FIFOQueue object is created to add queue statistics*
7. *# to TheCTStats list*
8. self.WIP = CTStat()
9. self.ThisQueue = []
11. **def** NumQueue(self):
12. *# Return current number in the queue*
13. **return** len(self.ThisQueue)
15. **def Add(self,X):**
16. *# Add an entity to the end of the queue*
17. self.ThisQueue.append(X)
18. numqueue = self.NumQueue()
19. self.WIP.Record(float(numqueue))
21. **def** Remove(self):
22. *# Remove the first entity from the queue and return the object*
23. *# after updating the queue statistics*
24. **if** len(self.ThisQueue) > 0:
25. **remove = self.ThisQueue.pop(0)**
26. self.WIP.Record(float(self.NumQueue()))
27. **return** remove
29. **def** Pop(self,idx):
30. ***# Remove the i'th entity from the queue and return the object***
31. **if** len(self.ThisQueue) > 0:
32. remove = self.ThisQueue.pop(idx)
33. self.WIP.Record(float(self.NumQueue()))
34. **return** remove
36. **def** Mean(self):
37. *# Return the average number in queue up to the current time*
38. **return** self.WIP.Mean()

## Imports:

1. *# Imports*
2. **from** math **import** radians, cos, sin, asin, sqrt
3. **import** numpy **as** np
4. **import** pandas **as** pd
5. **from sklearn.mixture import GaussianMixture**
6. **import** scipy.stats **as** stats
8. *# PySim*
9. **import** SimFunctions
10. **import SimRNG**
11. **import** SimClasses
13. *# Plotting Libraries*
14. **import** matplotlib.pyplot **as** plt
15. **import plotly.express as px**
16. mapbox\_access\_token = 'pk.eyJ1IjoicHNwZWx0IiwiYSI6ImNqdW4xemNtYzB3ZDI0ZXM4Z2VkYWwwMTgifQ.UodsHknkt8fNv3Viszo1Zg'
17. px.set\_mapbox\_access\_token(mapbox\_access\_token)
19. *# Weather Data Libraries*
20. **! pip install wetterdienst**
21. **from** wetterdienst **import** Wetterdienst

## Population Data and Sampling Functions:

1. *# Read Toronto Dissemination Tract Population (Population in 2016 by Lat-Long)*
2. pop = pd.read\_csv("db\_pop\_2016.csv",encoding='latin-1')*#("toronto\_db.csv")*
3. *# Filter on Toronto Area and Group on Unique Lat-Long*
4. pop = pop.loc[pop.name5=="Toronto"][['pop\_2016','lat','long']].copy()
5. **pop = pop.groupby(by=['lat','long'])['pop\_2016'].sum().reset\_index()**
6. *# Convert Pandas to Numpy Array (Sampling w Numpy is ~10x Faster than Dataframe Sampling w Pandas)*
7. pop = pop[['lat','long','pop\_2016']].values
8. *# Pull Lat-Longs from Array*
9. pop\_latlong = [tuple(x[:2]) **for** x **in** pop]
10. ***# Sum Total Pop From Array***
11. tot\_pop = np.sum(pop[:,2])
12. *# Calculate Probability of Lat-Long (Pop / Tot Pop)*
13. pop\_p = np.array([x[2]/tot\_pop **for** x **in** pop])
14. *# Create Index to Sample From*
15. **pop\_idx = np.arange(0,len(pop))**
16. *# Function uses numpy random choice to sample coordinates weighted by population*
17. **def** sample\_geo(pop=pop):
18. **return** pop\_latlong[np.random.choice(pop\_idx,p=pop\_p)]
19. *# Plot Population*
20. ***#fig = px.scatter\_mapbox(pd.DataFrame(pop,columns=['lat','long','pop\_2016']),lat='lat',lon='long',color='pop\_2016',zoom=12,width=1000,height=1000)***
21. *#fig.show()*

## Distance and Bearing Functions:

1. **def** haversine(latlong1, latlong2):
2. """
3. Calculate the great circle distance between two points
4. on the earth (specified in decimal degrees)
5. **"""**
6. *# convert decimal degrees to radians*
7. lon1 = latlong1[1]
8. lat1 = latlong1[0]
9. lon2 = latlong2[1]
10. **lat2 = latlong2[0]**
11. lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
13. *# haversine formula*
14. dlon = lon2 - lon1
15. **dlat = lat2 - lat1**
16. a = sin(dlat/2)\*\*2 + cos(lat1) \* cos(lat2) \* sin(dlon/2)\*\*2
17. c = 2 \* asin(sqrt(a))
18. r = 6371 *# Radius of earth in kilometers. Use 3956 for miles*
19. **return** c \* r
21. *# Test Distance Calc Between Distribution Center (dc) and sampled latlong*
22. test\_latlong = sample\_geo()
23. dc = np.array([43.74767,-79.40574])
24. haversine(test\_latlong,dc)
26. *# Mean Distance*
27. dist = []
28. **for** x **in** np.arange(0,int(10e4)):
29. dist.append(haversine(sample\_geo(),dc))
30. **np.mean(dist)**
32. **def** calc\_bearing\_off\_N(latlong1, latlong2):
33. """
34. Function calculates the bearing in degrees from North (where North is 0, East 90, South 180, West 270 etc.)
35. **"""**
36. dlong = latlong2[1] - latlong1[1]
37. X = np.cos(latlong2[0])\*np.sin(dlong)
38. Y = np.cos(latlong1[0])\*np.sin(latlong2[0]) - np.sin(latlong1[0])\*np.cos(latlong2[0])\*np.cos(dlong)
39. bearing = np.arctan2(X,Y)
40. ***# Return degrees from the North***
41. **return** np.degrees(bearing)
43. b = calc\_bearing\_off\_N(dc,test\_latlong)
45. **def vectorize(magnitude,bearing):**
46. """
47. Function converts a vector magnitude and bearing into (x,y) format
48. """
49. quadrant = np.ceil(bearing / 90)
50. **x = magnitude\*np.cos(math.radians((450-bearing)))**
51. y = magnitude\*np.sin(math.radians((450-bearing)))
52. **return** np.array([x,y])
54. **def** gb(x, y):
55. **"""**
56. Function converts (x,y) vector into a bearing
57. """
58. angle = degrees(atan2(y, x))
59. bearing1 = (angle + 360) % 360
60. **bearing2 = (90 - angle) % 360**
61. **return** bearing1

## Service Time Function:

1. **def** ServiceTime(distance,wind\_speed,wind\_dir,MeanST,Phases,drone\_speed=80):
2. """
3. Function calculates service time using drone speed, wind speed and delivery distance
4. Randomness comes from the 2 Phase Erlang distribution when the drone makes the actual delivery
5. **"""**
6. leg1\_speed = (drone\_speed - wind\_speed) / 60 *#km/h to km/min*
7. leg2\_speed = (drone\_speed + wind\_speed) / 60 *#km/h to km/min*
8. *# Randomness is in the drop-off time. Erlang phase 2*
9. service\_time = SimRNG.Erlang(Phases,MeanST,1)
10. **time = (distance / leg1\_speed) + (distance / leg2\_speed) + service\_time**
11. **return** time
13. distance = haversine(dc,shang)
14. bearing = calc\_bearing\_off\_N(dc,shang)
15. **ServiceTime(distance,bearing,wind\_speed=40,wind\_dir=0,MeanST=5,Phases=2,drone\_speed=80)**

## Weather Data and Functions:

1. *# Read Historical Weather (Toronto Intl Station merged with Toronto City Centre Station)*
2. cl = pd.read\_csv("toronto\_intl\_weather.csv")
3. cl = cl[['lat','lon','Year','Month','Day','Time (LST)','Temp (C)','Wind Chill','Wind Spd (km/h)','Wind Dir (10s deg)','Visibility (km)','Weather']].copy()
4. cl.columns = ['lat','long','year','month','day','time','temp','windchill','windspd','winddir','visib','weather']
5. **cl1 = pd.read\_csv("toronto\_citycentre\_weather.csv")[['Precip. Amount (mm)']]**
6. cl1.columns = ['precip']
7. cl = cl.join(cl1)*#.dropna()*
9. *# Read DWD Weather Station Data and Wind Speed Data and calculate G\_bar and G\_sigma*
10. **read\_dwd = False**
11. **if** read\_dwd == True:
12. *# Read DWD Weather Station Data and Wind Speed Data*
13. dwd\_stations = pd.read\_csv("dwd\_stations.csv",encoding='latin')
14. dwd\_ids = list(dwd\_stations.id.unique())
15. **wind = pd.read\_csv("dwd\_wind.csv")**
16. wind = wind.dropna().copy()
17. *# m/s to km/h*
18. wind['VALUE\_x'] = wind['VALUE\_x']\*1000/60
19. wind['VALUE\_x'] = wind['VALUE\_y']\*1000/60
20. **wind.loc[np.isinf(wind.G),'G'] = 0**
21. G\_bar = wind.G.mean()
22. G\_sigma = wind.G.std()
23. **else**:
24. G\_bar = 0.6327567537022766
25. **G\_sigma = 0.6769295605634029**
27. *# Gust Risk*
28. **def** probgust(mean\_windspd,threshold,G\_bar,G\_sigma):
29. """
30. **Function takes the G\_bar and G\_sigma from the DWD dataset, calculates the log\_G\_bar and sigma\_log\_G values**
31. Using a wind speed input returns the probability of a gust exceeding the defined threshold
32. """
33. U = mean\_windspd
34. v = threshold
35. **if U == 0:**
36. G=0
37. **else**:
38. G = (v / U) - 1
39. log\_G\_bar = np.log10(G\_bar) - 0.5\*np.log10((G\_sigma/G\_bar)\*\*2 + 1)
40. **sigma\_log\_G = np.sqrt((1/2.3026)\*np.log10((G\_sigma/G\_bar)\*\*2 + 1))**
41. x = (np.log10(G) - log\_G\_bar) / sigma\_log\_G
42. Fx = stats.norm.cdf(x)
43. Pr = 1 - Fx
44. **return** Pr
46. *# Define wind speed threshold for drones and apply gust risk function to the Toronto Weather data*
47. threshold = 50
48. col = "gust\_risk"
49. cl[col] = cl['windspd'].apply(**lambda** x: probgust(x,threshold,G\_bar,G\_sigma))
50. **cl[col].fillna(1,inplace=True) *# Nulls are already over threshold wind speed***
51. cl['hour'] = cl.time.apply(**lambda** x: int(x.split(":")[0]))
52. cl['datetime'] = pd.to\_datetime(cl[['year','month','day','hour']])
53. **def** sample\_gust(n\_samples):
54. """
55. Samples the gust risk n times depending on the duration of the trip
56. """
57. **rand = []**
58. **for** sample **in** range(n\_samples):
59. rand.append(SimRNG.Uniform(0,1,3))
60. **return** min(rand)
62. **def binomial\_prob(service\_time,gust\_risk):**
63. """
64. Determines the probability of 0 gusts occuring over the duration of the delivery
65. """
66. n = max(int(np.floor(service\_time / 10)),1)
67. **p = gust\_risk**
68. risk = (1-p)\*\*n
69. **return** risk, n

## Classes For Updating Weather State and Logging Delivery Metrics:

1. **class** WeatherClass():
2. **def** \_\_init\_\_(self,weather\_data):
3. """
4. Class for loading weather data, forecasting the weather conditions for a day and updating the weather state for the system
5. **"""**
6. self.data = weather\_data
7. self.forecast = None
8. self.state = None
9. **def** Forecast(self,\*\*args):
10. **self.operating\_hours = args['operating\_hours']**
11. **if** (args['type'] == 'historical') | (args[‘type’] == ‘historical\_day’) | (args['type'] == 'inclement'):
12. *# Pull a historical weather day from the dataset*
13. self.forecast = list(self.data.loc[(self.data.year==args['year'])&(self.data.month==args['month'])&(self.data.day==args['day'])&(self.data.hour.isin(args['operating\_hours'])),['hour','windspd','winddir','precip','gust\_risk']].values)
14. **elif** args['type'] == 'manual':
15. ***# Pull an arbitrary weather day and overwrite the weather conditions with the specified conditions***
16. temp = self.data.loc[(self.data.year==2020)&(self.data.month==6)&(self.data.day==4)&(self.data.hour.isin(args['operating\_hours'])),['hour','windspd','winddir','precip','gust\_risk']].copy()
17. temp['windspd'] = args['windspd']
18. temp['winddir'] = args['winddir']
19. temp['precip'] = args['precip']
20. **temp['gust\_risk'] = temp['windspd'].apply(lambda x: probgust(x,threshold,G\_bar,G\_sigma))**
21. self.forecast = list(temp.values)
22. **def** Update(self,Queue):
23. hour, windspd, winddir, precip, gust\_risk = self.forecast.pop(0) *# Remove the latest row from the forecast*
24. self.state = {'windspd':windspd,'winddir':winddir,'precip':precip,'gust\_risk':gust\_risk} *# Update the system weather state*
25. **if Queue.NumQueue() > 0: *# If the Queue has remaining customers update their attributes (service time / gust risk can change based on new windspd conditions)***
26. **for** i,Cust **in** enumerate(Queue.ThisQueue):
27. Queue.ThisQueue[i].service\_time = ServiceTime(Cust.distance,wind\_speed=self.state['windspd'],wind\_dir=self.state['winddir'],MeanST=MeanST,Phases=Phases,drone\_speed=drone\_speed)
28. Queue.ThisQueue[i].risk,\_ = binomial\_prob(Cust.service\_time,self.state['gust\_risk'])
29. **def** Schedule(self,Calendar):
30. ***# Schedule hourly weather updates on the Calendar***
31. **for** i,hr **in** enumerate(self.operating\_hours):
32. update\_time = i\*60
33. SimFunctions.Schedule(Calendar,"UpdateWeather",update\_time)
35. **class DeliveriesClass(object):**
36. **def** \_\_init\_\_(self):
37. """
38. Class for logging delivery metrics (number completed, distance, coordinates, number of lost drones, number of delays, etc.)
39. """
40. **self.coord = []**
41. self.distance = []
42. self.service\_time = []
43. self.gust\_risk = []
44. self.wait = []
45. **self.status = []**
46. self.data = None
47. self.N = 0
48. self.delayed = 0
49. self.lost = 0
50. **def Update(self,coord,distance,service\_time,gust\_risk,wait,status):**
51. self.coord.append(coord)
52. self.distance.append(distance)
53. self.service\_time.append(service\_time)
54. self.gust\_risk.append(gust\_risk)
55. **self.wait.append(wait)**
56. self.status.append(status)
57. **if** status == 'Completed':
58. self.N +=1
59. **if** status == 'Failed':
60. **self.lost +=1**
61. **def** Dataframe(self):
62. *# Convert to a dataframe*
63. self.data = pd.DataFrame((self.coord,self.distance,self.gust\_risk,self.wait,self.status)).T
64. self.data.columns = ['coord','distance','gust\_risk','wait','status']

## UAVSim Function:

1. **def** UAVSim(n\_reps=1,n\_drones=10,dc=np.array([43.74767,-79.40574]),verbose=False,\*\*args):
2. *# Initialize Random Seeds*
3. ZSimRNG = SimRNG.InitializeRNSeed()
4. np.random.seed()
6. Queue = SimClasses.FIFOQueue()
7. Wait = SimClasses.DTStat()
8. Server = SimClasses.Resource()
9. Calendar = SimClasses.EventCalendar()
11. TheCTStats = []
12. TheDTStats = []
13. TheQueues = []
14. TheResources = []
15. **Deliveries = []**
17. TheDTStats.append(Wait)
18. TheQueues.append(Queue)
19. TheResources.append(Server)
21. drone\_speed = 80 *# kmh*
22. MaxLambda = args['MaxLambda'] *# For thinning function*
23. Lambda = args['Lambda'] *# For thinning function*
24. MeanST = 5 *# For actual delivery service (drone landing, customer receiving package, etc.)*
25. **Phases = 2**
27. AllWaitMean = []
28. AllQueueMean = []
29. AllQueueNum = []
30. **AllServerMean = []**
31. AllDeliveries = []
32. AllDelayedDrones = []
33. AllLostDrones = []
34. AllCompletedDeliveries = []
36. **def** NSPP():
37. """
38. Function applies thinning algorithm for Non-Stationary Poisson Process using MaxLambda and Lambda parameters
39. """
40. **PossibleArrival = SimClasses.Clock + SimRNG.Expon(1/(MaxLambda/60), 1)**
41. Hour = np.floor(SimClasses.Clock/60).astype(int)
42. **while** SimRNG.Uniform(0, 1, 3) >= (Lambda[Hour]/MaxLambda):
43. PossibleArrival = PossibleArrival + SimRNG.Expon(1/(MaxLambda/60), 1)
44. nspp = PossibleArrival - SimClasses.Clock
45. **return nspp**
47. **def** Arrival():
48. *# Schedule next arrival with exponential distribution*
49. SimFunctions.Schedule(Calendar,"Arrival",NSPP())
50. ***# Define New Customer - Sample from Geography, Calculate Distance and Service Time***
51. coord = sample\_geo()
52. distance = haversine(dc,coord)
53. service\_time = ServiceTime(distance,wind\_speed=Weather.state['windspd'],wind\_dir=Weather.state['winddir'],MeanST=MeanST,Phases=Phases,drone\_speed=drone\_speed)
54. *# Determine probability of 0 gusts occuring over duration of delivery (n\_samples is the number of 10 minute intervals elapsed by the delivery)*
55. **risk, n\_samples = binomial\_prob(service\_time,Weather.state['gust\_risk'])**
56. *# Define new Customer entity using coord, distance, service\_time, binomial risk*
57. Customer = SimClasses.Entity(coord,distance,service\_time,risk)
58. *# Separately create a list of the Customer data for the policy functions*
59. Queue\_Data.append([Customer.distance,Customer.service\_time,Customer.risk,Customer.CreateTime])
60. ***# Add Customer to the Queue***
61. Queue.Add(Customer)
62. *# Pass Customer / Queue / Server / Weather metrics into the Policy Function*
63. PolicyDecision = args['policy'](Queue\_Data,Server,Weather)
64. **if** PolicyDecision!='Delay':
65. ***# Remove Customer from Queue for Delivery***
66. CustomerEnteringService = Queue.Pop(PolicyDecision)
67. Queue\_Data.pop(PolicyDecision)
68. *# Record Time Spent in Queue*
69. WaitTime = SimClasses.Clock - CustomerEnteringService.CreateTime
70. ***# Sample Uniform for Gust Prob***
71. U = sample\_gust(n\_samples)
72. **if** U <= Weather.state['gust\_risk']:
73. *# Drone is hit with a wind gust and forced to land. Record failed delivery, subtract a drone from the total number available*
74. Deliveries.Update(CustomerEnteringService.coord,CustomerEnteringService.distance,CustomerEnteringService.service\_time,Weather.state['gust\_risk'],WaitTime,"Failed")
75. **Server.NumberOfUnits -=1**
76. **if** Server.NumberOfUnits == 0:
77. SimFunctions.Schedule(Calendar,"LostAllDrones",0)
78. **else**:
79. *# Drone is not hit with a wind gust and the delivery is successful. Seize a drone and schedule the EndOfService*
80. **Deliveries.Update(CustomerEnteringService.coord,CustomerEnteringService.distance,CustomerEnteringService.service\_time,Weather.state['gust\_risk'],WaitTime,"Completed")**
81. Server.Seize(1)
82. service\_time = CustomerEnteringService.service\_time
83. SimFunctions.Schedule(Calendar,"EndOfService",service\_time)
84. **else**:
85. ***# Cannot send a drone due to policy***
86. Deliveries.delayed +=1
87. SimFunctions.Schedule(Calendar,"EndOfService",60)
89. **def** EndOfService():
90. **if Queue.NumQueue() > 0: *# Queue is longer than 0. Check policy to see if we can send a new drone***
91. PolicyDecision = args['policy'](Queue\_Data,Server,Weather)
92. **if** PolicyDecision!='Delay':
93. *# Remove Customer from Queue for Delivery*
94. CustomerEnteringService = Queue.Pop(PolicyDecision)
95. **Queue\_Data.pop(PolicyDecision)**
96. *# Record Time Spent in Queue*
97. n\_samples = max(round(CustomerEnteringService.service\_time / 10),1)
98. *# Record Time Spent in Queue*
99. WaitTime = SimClasses.Clock - CustomerEnteringService.CreateTime
100. ***# Sample Uniform for Gust Prob***
101. U = sample\_gust(n\_samples)
102. **if** U <= Weather.state['gust\_risk']:
103. *# Drone is hit with a wind gust and forced to land. Record failed delivery, subtract a drone from the total number available*
104. Deliveries.Update(CustomerEnteringService.coord,CustomerEnteringService.distance,CustomerEnteringService.service\_time,Weather.state['gust\_risk'],WaitTime,"Failed")
105. **Server.NumberOfUnits -=1**
106. **if** Server.NumberOfUnits == 0:
107. SimFunctions.Schedule(Calendar,"LostAllDrones",0)
108. **else**:
109. *# Drone is not hit with a wind gust and the delivery is successful. Seize a drone and schedule the EndOfService*
110. **Deliveries.Update(CustomerEnteringService.coord,CustomerEnteringService.distance,CustomerEnteringService.service\_time,Weather.state['gust\_risk'],WaitTime,"Completed")**
111. Server.Seize(1)
112. service\_time = CustomerEnteringService.service\_time
113. SimFunctions.Schedule(Calendar,"EndOfService",service\_time)
114. **else**:
115. ***# Cannot send a drone due to policy***
116. Deliveries.delayed +=1
117. SimFunctions.Schedule(Calendar,"EndOfService",60)
118. **else**:
119. *# No customers to serve, can free up a drone*
120. **Server.Free(1)**
122. **def** UpdateWeather(Queue):
123. """
124. Updates the weather state of the system
125. **"""**
126. Weather.Update(Queue,MeanST,Phases,drone\_speed)
128. **def** print\_system\_state():
129. """
130. **For debugging: can print the system state and track the state transitions of the discrete event simulation**
131. """
132. event = NextEvent.EventType
133. clock = SimClasses.Clock
134. windspd = Weather.state['windspd']
135. **gust = Weather.state['gust\_risk']**
136. busy = Server.Busy
137. totdrone = Server.NumberOfUnits
138. qlen = Queue.NumQueue()
139. **print**((event,"Clock",clock,"Windspd",windspd,"Gust",gust,"BusyDrones",busy,"TotDrones",totdrone,"QueueLength",qlen))
141. **if** args['type'] == 'inclement':
142. *# Check passed arguments, if they call for inclement weather filter the weather data on days where gust risk exceeds 50% within the operating hours of the system*
143. ymd = cl.loc[(cl.gust\_risk>0.5)&(cl.hour.isin([9,10,11,12,13,14,15,16,17])),['year','month','day','windspd','winddir','precip']].dropna()[['year','month','day']].drop\_duplicates().values
144. weather\_args = {'type':'inclement','operating\_hours':[9,10,11,12,13,14,15,16,17]}
145. **else:**
146. *# Otherwise pull the entire weather dataset for simulation*
147. ymd = cl[['year','month','day','windspd','winddir','precip']].dropna()[['year','month','day']].drop\_duplicates().values
148. weather\_args = {'type':args['type'],'operating\_hours':[9,10,11,12,13,14,15,16,17]}
150. **for reps in range(0,n\_reps,1):**
151. weather\_idx = reps % len(ymd) *# Weather\_idx is an index for the weather data, when n\_reps exceeds the number of days, the mod math resets the index*
152. weather\_args['year'], weather\_args['month'], weather\_args['day'] = ymd[weather\_idx] *# Set the yyyymmdd parameters for our weather arguments*
153. **if** args['type']=='manual': *# If we are testing manual parameters override the winddir, windspd, precip values*
154. weather\_args['winddir'] = args['winddir']
155. **weather\_args['windspd'] = args['windspd']**
156. weather\_args['precip'] = args['precip']
158. Server.SetUnits(n\_drones) *# Reset number of drones each replication*
159. Queue\_Data = [] *# Reset queue data inbetween replications*
160. **SimFunctions.SimFunctionsInit(Calendar,TheQueues,TheCTStats,TheDTStats,TheResources)**
161. Weather = WeatherClass(cl) *# Initialize weather class*
162. Weather.Forecast(\*\*weather\_args) *# Forecast weather for this replication (using arguments specified above)*
163. Weather.Schedule(Calendar) *# Schedule weather updates each hour*
164. Deliveries = DeliveriesClass() *# Initialize deliveries class for tracking delivery stats*
165. **SimFunctions.Schedule(Calendar,"Arrival",NSPP()) *# Schedule first arrival using NSPP function***
167. NextEvent = Calendar.Remove()
168. SimClasses.Clock = NextEvent.EventTime
169. **if** NextEvent.EventType == "UpdateWeather":
170. **UpdateWeather(Queue)**
171. **elif** NextEvent.EventType == "Arrival":
172. Arrival()
173. **elif** NextEvent.EventType == "EndOfService":
174. EndOfService()
175. **elif NextEvent.EventType == "ClearIt":**
176. SimFunctions.ClearStats(TheCTStats,TheDTStats)
178. **while** len(Weather.forecast)>0: *# While we are within the operating hours of the system*
180. **NextEvent = Calendar.Remove()**
182. SimClasses.Clock = NextEvent.EventTime
183. **if** NextEvent.EventType == "UpdateWeather":
184. UpdateWeather(Queue)
185. **elif NextEvent.EventType == "Arrival":**
186. *#print\_system\_state()*
187. Arrival()
188. **elif** NextEvent.EventType == "EndOfService":
189. *#print\_system\_state()*
190. **EndOfService()**
191. **elif** NextEvent.EventType == "ClearIt":
192. SimFunctions.ClearStats(TheCTStats,TheDTStats)
193. **elif** NextEvent.EventType == "LostAllDrones":
194. **break**
196. AllWaitMean.append(Wait.Mean())
197. AllQueueMean.append(Queue.Mean())
198. AllQueueNum.append(Queue.NumQueue())
199. AllServerMean.append(Server.Mean())
200. **AllDeliveries.append(Deliveries)**
201. AllDelayedDrones.append(Deliveries.delayed)
202. AllLostDrones.append(Deliveries.lost)
203. AllCompletedDeliveries.append(Deliveries.N)
205. **if verbose:**
206. **print**("Estimated Expected Average wait: %.2f" % np.mean(AllWaitMean))
207. **print**("Estimated Expected Average queue-length: %.2f" % np.mean(AllQueueMean))
208. **print**("Estimated Expected Average utilization: %.2f" % np.mean(AllServerMean))
209. **print**("Estimated Expected Number of Delayed Drones: %.2f" % np.mean(AllDelayedDrones))
210. **print("Estimated Expected Number of Lost Drones: %.2f" % np.mean(AllLostDrones))**
212. **return** AllDeliveries,AllCompletedDeliveries,AllLostDrones,AllQueueMean

## Confidence Intervals:

1. **def** t\_mean\_confidence\_interval(data,alpha):
2. a = 1.0\*np.array(data)
3. n = len(a)
4. m, se = np.mean(a), stats.sem(a)
5. **h = stats.t.ppf(1-alpha/2, n-1)\*se**
6. **return** m, h

## Example of Running Simulation:

We must define the arguments to pass the simulation including MaxLambda, Lambda (for the NSPP order demand thinning algorithm). The operating hours, weather conditions, the policy to run, the number of drones, the number of replications (or days) to simulate and the location of the distribution center.

All of these arguments can be specified and passed to the simulation as shown below:

1. *# Example of Simulation Call*
2. MaxLambda = 20 *# per hour*
3. Lambda = {0:12,1:14,2:16,3:18,4:20,5:18,6:16,7:14}
4. arguments = {'type':'inclement','operating\_hours':[9,10,11,12,13,14,15,16,17],'MaxLambda':MaxLambda,'Lambda':Lambda}
5. **def policy0(Queue\_Data,Server,Weather):**
6. **if** Server.Busy < Server.NumberOfUnits: *# Naive Policy: Always Send Drone*
7. **return** 0
8. **else**:
9. **return** 'Delay'
10. **arguments['policy'] = policy0**
11. d,res,lost,queue = UAVSim(n\_reps=1,n\_drones=10,dc=np.array([43.74767,-79.40574]),verbose=False,\*\*arguments)

## System Dynamics Experiments:

The code used to generate plots of the system response as wind speed increases. We input the “manual” argument to manually specify the windspeed the system runs on.

1. resST = {}
2. resND = {}
3. resLD = {}
4. MaxLambda = 10
5. ***#Lambda = {0:6,1:7,2:8,3:9,4:10,5:9,6:8,7:7}***
6. Lambda = {0:1/(6/60),1:1/(7/60),2:1/(8/60),3:1/(9/60),4:1/(10/60),5:1/(9/60),6:1/(8/60),7:1/(7/60)}
7. **for** windspd **in** np.arange(5,50,5):
8. args = {'type':'manual','windspd':windspd,'winddir':0,'precip':0,'operating\_hours':[9,10,11,12,13,14,15,16,17],'MaxLambda':MaxLambda,'Lambda':Lambda,'policy':policy0}
9. deliveries,results,lost,queue = UAVSim(n\_reps=500,n\_drones=10,dc=np.array([43.74767,-79.40574]),verbose=False,\*\*args)
10. **resST[windspd] = np.mean([np.mean(x.service\_time) for x in deliveries])**
11. resND[windspd] = np.mean([np.mean(x.N) **for** x **in** deliveries])
12. resLD[windspd] = np.mean([np.mean(x.lost) **for** x **in** deliveries])
13. dfST = pd.DataFrame.from\_dict(resST,orient='index').reset\_index()
14. dfND = pd.DataFrame.from\_dict(resND,orient='index').reset\_index()
15. **dfLD = pd.DataFrame.from\_dict(resLD,orient='index').reset\_index()**
17. fig = plt.figure(figsize=(5,5))
18. plt.plot(dfST['index'],dfST[0])
19. plt.title("Wind Speed vs Mean Service Time",size=20)
20. **plt.xlabel("Wind Speed (km/h)",size=15)**
21. plt.ylabel("Mean Service Time (minutes)",size=15)
22. plt.xticks(size=15)
23. plt.yticks(size=15)
24. plt.show()
26. fig = plt.figure(figsize=(5,5))
27. plt.plot(dfND['index'],dfND[0])
28. plt.title("Wind Speed vs Completed Deliveries",size=20)
29. plt.xlabel("Wind Speed (km/h)",size=15)
30. **plt.ylabel("# Completed Deliveries",size=15)**
31. plt.xticks(size=15)
32. plt.yticks(size=15)
33. plt.show()
35. **fig = plt.figure(figsize=(5,5))**
36. plt.plot(dfLD['index'],dfLD[0])
37. plt.title("Wind Speed vs Lost Drones",size=20)
38. plt.xlabel("Wind Speed (km/h)",size=15)
39. plt.ylabel("Number of Lost Drones (/10)",size=15)
40. **plt.xticks(size=15)**
41. plt.yticks(size=15)
42. plt.show()

## Policy Definitions:

The policy functions used for this simulation:

1. **def** policy0(Queue\_Data,Server,Weather):
2. **if** Server.Busy < Server.NumberOfUnits: *# Naive Policy: Always Send Drone*
3. **return** 0
4. **else**:
5. **return 'Delay'**
7. **def** policy1(Queue\_Data,Server,Weather):
8. **if** Server.Busy < Server.NumberOfUnits:
9. **if** len(Queue\_Data) > 0:
10. **qd = np.array(Queue\_Data)**
11. **return** qd[:,2].argsort()[-1] *# Minimum Binomial Risk Policy*
12. **else**:
13. **return** 0
14. **else**:
15. **return 'Delay'**
17. **def** policy2(Queue\_Data,Server,Weather):
18. **if** Server.Busy < Server.NumberOfUnits:
19. **if** len(Queue\_Data) > 0:
20. **qd = np.array(Queue\_Data)**
21. idx = qd[:,2].argsort()[-1]
22. **if** qd[idx,2] > 0.5: *# Minimum Binomial Risk if >50% Probability of 0 Gusts*
23. **return** idx
24. **else**:
25. **return 'Delay'**
26. **else**:
27. **return** 0
28. **else**:
29. **return** 'Delay'
31. **def** policy3(Queue\_Data,Server,Weather):
32. **if** Server.Busy < Server.NumberOfUnits:
33. **if** len(Queue\_Data) > 0:
34. qd = np.array(Queue\_Data)
35. **idx = qd[:,2].argsort()[-1]**
36. **if** qd[idx,2] > 0.25: *# Minimum Binomial Risk if >25% Probability of 0 Gusts*
37. **return** idx
38. **else**:
39. **return** 'Delay'
40. **else:**
41. **return** 0
42. **else**:
43. **return** 'Delay'
45. **def policy4(Queue\_Data,Server,Weather):**
46. **if** Server.Busy < Server.NumberOfUnits:
47. **if** len(Queue\_Data) > 0:
48. qd = np.array(Queue\_Data)
49. **return** qd[:,0].argsort()[0] *# Shortest Distance Available Policy*
50. **else:**
51. **return** 0
52. **else**:
53. **return** 'Delay'
55. **def policy5(Queue\_Data,Server,Weather):**
56. **if** Server.Busy < Server.NumberOfUnits:
57. **if** len(Queue\_Data) > 0:
58. qd = np.array(Queue\_Data)
59. idx = qd[:,0].argsort()[0]
60. **if qd[idx,0] <= 10: *# Shortest Distance Available if < 10km Distance***
61. **return** idx
62. **else**:
63. **return** 'Delay'
64. **else**:
65. **return 'Delay'**
66. **else**:
67. **return** 'Delay'
69. **def** policy6(Queue\_Data,Server,Weather):
70. **if Server.Busy < Server.NumberOfUnits:**
71. **if** len(Queue\_Data) > 0:
72. qd = np.array(Queue\_Data)
73. idx = qd[:,0].argsort()[0]
74. **if** qd[idx,0] <= 40: *# Shortest Distance Available if < 40km Distance*
75. **return idx**
76. **else**:
77. **return** 'Delay'
78. **else**:
79. **return** 'Delay'
80. **else:**
81. **return** 'Delay'
83. **def** policy7(Queue\_Data,Server,Weather):
84. **if** Server.Busy < Server.NumberOfUnits:
85. **if len(Queue\_Data) > 0:**
86. qd = np.array(Queue\_Data)
87. idx = qd[:,2].argsort()[-1]
88. **if** qd[idx,2] > 0.40: *# Minimum Binomial Risk if >40% Probability of 0 Gusts*
89. **return** idx
90. **else:**
91. **return** 'Delay'
92. **else**:
93. **return** 0
94. **else**:
95. **return 'Delay'**
97. **def** policy8(Queue\_Data,Server,Weather):
98. **if** Server.Busy < Server.NumberOfUnits:
99. **if** len(Queue\_Data) > 0:
100. **qd = np.array(Queue\_Data)**
101. idx = qd[:,2].argsort()[-1]
102. **if** qd[idx,2] > 0.30: *# Minimum Binomial Risk if >30% Probability of 0 Gusts*
103. **return** idx
104. **else**:
105. **return 'Delay'**
106. **else**:
107. **return** 0
108. **else**:
109. **return** 'Delay'
111. **def** policy9(Queue\_Data,Server,Weather):
112. **if** Server.Busy < Server.NumberOfUnits:
113. **if** len(Queue\_Data) > 0:
114. qd = np.array(Queue\_Data)
115. **idx = qd[:,2].argsort()[-1]**
116. **if** qd[idx,2] > 0.60: *# Minimum Binomial Risk if >60% Probability of 0 Gusts*
117. **return** idx
118. **else**:
119. **return** 'Delay'
120. **else:**
121. **return** 0
122. **else**:
123. **return** 'Delay'
125. **def policy10(Queue\_Data,Server,Weather):**
126. **if** Server.Busy < Server.NumberOfUnits:
127. **if** len(Queue\_Data) > 0:
128. qd = np.array(Queue\_Data)
129. idx = qd[:,2].argsort()[-1]
130. **if qd[idx,2] > 0.70: *# Minimum Binomial Risk if >70% Probability of 0 Gusts***
131. **return** idx
132. **else**:
133. **return** 'Delay'
134. **else**:
135. **return 0**
136. **else**:
137. **return** 'Delay'
139. **def** policy11(Queue\_Data,Server,Weather):
140. **if Server.Busy < Server.NumberOfUnits:**
141. **if** len(Queue\_Data) > 0:
142. qd = np.array(Queue\_Data)
143. idx = qd[:,2].argsort()[-1]
144. **if** qd[idx,2] > 0.80: *# Minimum Binomial Risk if >80% Probability of 0 Gusts*
145. **return idx**
146. **else**:
147. **return** 'Delay'
148. **else**:
149. **return** 0
150. **else:**
151. **return** 'Delay'
153. policies = [policy0,policy1,policy2,policy3,policy4,policy5,policy6]
154. policies = [policy7,policy8,policy9,policy10]

## Subset Selection Example:

Example of how a subset selection procedure is run. First calculate means and standard deviations for number of deliveries and number of lost drones:

1. *# Parameters*
2. MaxLambda = 20 *# per hour*
3. Lambda = {0:12,1:14,2:16,3:18,4:20,5:18,6:16,7:14}
4. arguments = {'type':'historical','operating\_hours':[9,10,11,12,13,14,15,16,17],'MaxLambda':MaxLambda,'Lambda':Lambda}
5. **policies = [policy0,policy1,policy2,policy3,policy4,policy5,policy6]**
6. *# Subset Selection Variables*
7. K = len(policies)
8. results = {}
9. deliveries = {}
10. **lost = {}**
11. queue = {}
12. t\_quantiles = {}
13. sample\_means\_deliveries = {}
14. sample\_means\_lost = {}
15. **ci\_deliveries = {}**
16. ci\_lost = {}
17. sample\_vars\_deliveries = {}
18. sample\_vars\_lost = {}
19. n\_reps = 1000
20. ***# Loop Through Policies***
21. **for** p **in** policies:
22. arguments['policy'] = p
23. deliveries[str(p)],results[str(p)],lost[str(p)],queue[str(p)] = UAVSim(n\_reps=n\_reps,n\_drones=10,dc=np.array([43.74767,-79.40574]),verbose=False,\*\*arguments)
24. *# Calculate t quantiles @ 95% confidence*
25. **t\_quantiles[str(p)] = stats.t.ppf((1-0.05)\*\*(1/(K-1)), n\_reps-1)**
26. *# Calculate sample means and variances for the number of completed deliveries*
27. sample\_means\_deliveries[str(p)] = np.mean(results[str(p)])
28. ci\_deliveries[str(p)] = t\_mean\_confidence\_interval(results[str(p)],0.05)
29. sample\_vars\_deliveries[str(p)] = np.var(results[str(p)],ddof=1)
30. ***# Calculate sample means and variances for the number of lost drones***
31. sample\_means\_lost[str(p)] = np.mean(lost[str(p)])
32. ci\_lost[str(p)] = t\_mean\_confidence\_interval(lost[str(p)],0.05)
33. sample\_vars\_lost[str(p)] = np.var(lost[str(p)],ddof=1)

Now calculate the estimation error threshold values (W) and compare the pairwise policies:

1. *# Calculate Subset W Threshold Values and Compare Policies to Determine the Optimum Subset (For Number of Deliveries)*
2. W = {}
3. subset = []
4. **for** i,policy\_i **in** enumerate(policies):
5. **i\_ = str(policy\_i)**
6. W[i\_] = {}
7. means = [sample\_means\_deliveries[i\_]]
8. policies\_ = [i\_]
9. **for** h,policy\_h **in** enumerate(policies):
10. **h\_ = str(policy\_h)**
11. **if** h\_ != i\_:
12. W[i\_][h\_] = np.sqrt((t\_quantiles[i\_]\*\*2)\*(sample\_vars\_deliveries[i\_]/n\_reps) + (t\_quantiles[h\_]\*\*2)\*(sample\_vars\_deliveries[h\_]/n\_reps))
13. means.append(sample\_means\_deliveries[h\_] - W[i\_][h\_])
14. policies\_.append(h\_)
16. max\_idx = np.array(means).argsort()[-1]
17. **if** i\_ == policies\_[max\_idx]:
18. subset.append(i\_)
20. **print(subset)**

Run again for the number of lost drones:

1. *# Calculate Subset W Threshold Values and Compare Policies to Determine the Optimum Subset (For Number of Lost Drones)*
2. W = {}
3. subset = []
4. **for** i,policy\_i **in** enumerate(policies):
5. **i\_ = str(policy\_i)**
6. W[i\_] = {}
7. means = [sample\_means\_lost[i\_]]
8. policies\_ = [i\_]
9. **for** h,policy\_h **in** enumerate(policies):
10. **h\_ = str(policy\_h)**
11. **if** h\_ != i\_:
12. W[i\_][h\_] = np.sqrt((t\_quantiles[i\_]\*\*2)\*(sample\_vars\_lost[i\_]/n\_reps) + (t\_quantiles[h\_]\*\*2)\*(sample\_vars\_lost[h\_]/n\_reps))
13. means.append(sample\_means\_lost[h\_] + W[i\_][h\_])
14. policies\_.append(h\_)
16. max\_idx = np.array(means).argsort()[0]
17. **if** i\_ == policies\_[max\_idx]:
18. subset.append(i\_)
20. **print(subset)**

## Plot Delivery Coordinates:

1. **def** plot\_deliveries(deliveries\_instance,dc):
2. Coord = []
3. **for** d **in** deliveries\_instance:
4. Coord+=d.coord
5. **Coord = pd.DataFrame(Coord,columns=['lat','long'])**
6. Coord['type'] = 'delivery'
7. Coord['size'] = 0.5
8. Coord = Coord.append(pd.DataFrame([list(dc)+['dc',5]],columns=['lat','long','type','size']))
9. Coord['Count'] = 1
10. **Coord = Coord.groupby(by=['lat','long','type','size'])['Count'].sum().reset\_index()**
11. fig = px.scatter\_mapbox(Coord,lat='lat',lon='long',color='type',size='size',hover\_data=['Count'],zoom=12,width=1000,height=1000)
12. fig.show()