Project 1 Computational Essay
Modeling the Impacts of Storms on Air Traffic and Finding Recovery Factors

Kate Mackowiak and Laurel Mitton
Prof: Jason Woodard
Class: Modeling and Simulation of the Physical World
Olin College of Engineering

20th September 2019

1 Question

Our question is an explanatory question asking how long a storm at a major airport hub creates delays, and what factors impact system recovery? This question is important to ask since travelers everywhere are impacted by delays in the airport system. We aim to model how quickly an air traffic system can recover from a delay, and vary parameters to find what may help a system recover faster.

2 Model/Methodology

To answer our question, we created a series of classes for airports, airplanes, and an air traffic controller in order to have objects of each class work through a mock schedule, so we could plot out where the airplanes are and if there are any delays. Then, to analyze how a storm impacts the schedule we had a function which adds delays of the duration of a storm to flights arriving at or departing from a specific airport. Then we plotted the number of delays per hour to show how the domino effect of a few delays impacts the system and how the system recovers.

We made a few assumptions in our model, one large one being that all of the planes are the same size and are interchangeable to take different flights if another plane is delayed. We are also assuming that each airport is the same size and that a scaled system with fewer airports will accurately show the impact of a storm.

To begin our model, we import a selection of libraries to use later for mathematical operations (NumPy), CSV reading (CSV), and plane labeling (String).

```
import numpy as np
import csv
import string
import matplotlib.pyplot as plt
```

Next, we create a function to read the CSV file of airports and flight times and process it into the lists we made earlier. For each line of the CSV, if it is the first line we place the values into the list of airport names. For the next rows, we decide if each value is one of the airport codes or a flight time, and places it in the appropriate array.

```
def read_csv(flight_time_array):
    '''reads airport names and flight times from csv'''
    airport_names = []
    with open('flight_times.csv', newline='') as csvfile:
        reader = csv.reader(csvfile, delimiter=' ', quotechar='|')
        for row in reader:
            if airport_names == []: # the first row is the names of the airports
                airport_names = (row[0].split(','))[1:]
                # exclude the first cell because it's a label for humans
            else: # the rest of the file is flight times to be separated out
                split = row[0].split(',')
                # split rows to give each airport flight times to the others
                split_ints = []
                for i in split:
                    if i not in airport_names:
                    # if it's not an airport name it's a flight time
                        split_ints.append(int(i))
                flight_time_array.append(split_ints)
    return airport_names
```

Our model has various probabilities that we can change as necessary, which are hard coded into the beginning of the file. These include a threshold for the probability of a plane flying at a given time, the length of the simulation, a threshold for a storm occurring, the number of planes each airport starts with, and the maximum number of hours a storm can go on for.

```
'''these global constants are parameters that can be varied as desired'''
flying_prob = .95
# probability of a flight being scheduled at a particular time
length_of_simulation = 25 # in hours
airport_size = 3
# number of airplanes distributed to each airport at the beginning
max_storm_length = 6 # max and min storm length
min_storm_length = 3
length_of_schedule = 10
# in hours, the simulation will continue to run after the schedule ends
plot_arrivals = False
# if True, will plot arrivals at the end; if False, will plot departures
```

Not included here are the definitions of three classes in our code. Each class (ATC, Airport, Plane) has its own set of variables that an instance of that class needs to know, and some basic functions related to the class. The ATC class is in charge of moving the time of the simulation and determining when, where, and for how long the next storm will be. In the ATC class' storm function, it decides whether or not to have a storm and then generates the location and length of it. This information is returned, and later passed on to instances of the Airport class.

```
def storm_schedule(self):
        '''generates information about storms'''
        global length_of_schedule
```

```
        self.storm_length = np.random.randint(min_storm_length, max_storm_length)
        # randomly determines storm length
        if(self.storm_length < round(length_of_schedule/4)):
        # keeps time at the beginning of the model length for a full prestorm
            self.storm_time = np.random.randint(self.storm_length,
                (round(length_of_schedule/4)))
        else:
            self.storm_time = self.storm_length
            # we want the storm at the beginning to see the effects later
        self.storm_location = airport_names[np.random.randint(0, len(airport_names))]
        print('Storm time {}, location {}, length {}'.format(self.storm_time,
            self.storm_location, self.storm_length))
        self.storm_info = [self.storm_time, self.storm_location, self.storm_length]
```

To initialize the various airports, we return to the global space to create a function. For each airport we listed in the CSV and added to the list earlier, we create an instance of the Airport class and add it to a dictionary, so we can refer to each airport by its real-world code to talk to the object.

```
'''loops through each airport name and creates an object based on that list'''
airport_dict = {}
for i in airport_names:
    name = i # the name is the airport code as a string
    schedule = [] # empty for initialization
    arrivals = [] # empty for initialization
    planes_ready = [] # empty for initialization
    i = Airport(name, schedule, planes_ready, arrivals)
    airport_dict[name] = i
    # dictionary to make it easier to refer to airport objects
return airport_dict
```

We use a similar method to create instances of the Plane class, but for this we name each plane using the letter 'a' followed by numerals. From that, we then create a list of plane names and use the same procedure as above to create the plane objects and a dictionary to refer to them by their names.

```
def create_airplanes():
    '''creates airplanes and returns a dictionary with names and location'''
    plane_dict = {}
    names = []
    for i in range(len(airport_names) * airport_size):
    # creates a series of names, a0, a1, a2, etc, as many as necessary
        names.append('a'+ str(i))
    plane_num = 0
    for i in range(len(airport_names)):
        for j in range(airport_size): # evenly distributes planes to airports
            name = names[plane_num]
            plane_names.append(name)
            names[plane_num] = Plane(name, airport_names[i], airport_names[i],-1)
```

```
            # the -1 gives all the airports their planes at the same time
            plane_dict[name] = names[plane_num]
            plane_num += 1
    return plane_dict
```

To create the schedule, we need to know the flight times between two airports, many times over with different airports each time. This function simply takes two airport names and returns the time between them, referring to the flight time array by finding the corresponding numbers that the port names relate to.

```
def find_flight_time(airport_dict, airport_names, port_one, port_two):
    '''returns the time from port one to port two'''
    for i in range(len(airport_names)):
        if airport_names[i] == port_one: # find the first airport
            port_one = i
        if airport_names[i] == port_two: # find the second airport
            port_two = i
    return flight_time_array[port_one][port_two]
    # consult the flight time array for flight time
```

Next we create the schedule using a variety of inputs. First, we create empty lists for the schedule, arrivals, and the planes in each airport. The list of planes in each airport is populated based on the dictionary from earlier. Next, for each time step, for each airport, we create a list of destinations, determine the number of planes in the current airport, and remove the current airport from the list of destinations so a plane can't fly to its origin. If there are no available destinations, we skip the rest of the function.

```
 def create_schedule(airport_names, airport_size, total_time, airport_dict):
    '''
    takes in list of airports, number of planes each start with, and simulation length
    returns a schedule listing the time of departure, the departure airport, and the
    arrival airport
    '''
    sched = [] # sched: time leaving, departure airport, arrival airport
    arrivals = [] # arrivals: time, destination
    planes_in_ports = []
    for i in airport_names:
        planes_in_ports.append([i,airport_size])
    flight_number = 0
    for i in range(total_time):  #goes through each time to determine arrivals and
    departures for that time
        for port_number in range(len(planes_in_ports)):    #figure out departures
            destinations = airport_names.copy()
            port = planes_in_ports[port_number]
            destinations.remove(airport_names[port_number])
            if type(destinations) == None:
                return
```

Continuing from within the for loops above, we go through each airplane at the current airport and decide whether or not that plane will fly. We set the time to the destination to 0 to create a blank slate for later, then determine where the plane will fly to. If the destination is in the list of possible destinations, we set the flight time and arrival time and add the flight to the schedule. The list of destinations is then printed for debugging purposes. Finally, the destination just used is removed from the list of possible destinations.

```
 for individual_airplanes in range(planes_in_ports[port_number][1]):
 #this should go through and decide if each airplane available is flying
                if (planes_in_ports[port_number][1] > 0) and
                    (np.random.random() > 1- flying_prob) and len(destinations) > 0:
                    #PROBABLILITY HERE
                    planes_in_ports[port_number][1] -= 1
                    dest = np.random.randint(0,len(airport_names))
                    time_to_dest = flight_time_array[port_number][dest]
                    arrival_time = int(i) + int(time_to_dest) + 1
                    if airport_names[dest] in destinations:
                        arrivals.append([arrival_time,dest])
                        sched.append([i,port[0],airport_names[dest], flight_number])
                        flight_number+=1
                        if len(destinations) == 1:
                            destinations = []
                        else:
                            destinations.remove(airport_names[dest])
```

Still within the port number for loop above, we also check for arrivals. This function checks if there are planes that should be arriving at the current time. At the end of the function, we return the schedule.

```
    for x in arrivals:  #check for arrivals
        if x[0] == i:   #if the time the arrival specifies is now
            for plane_num in range(len(planes_in_ports)):
                if planes_in_ports[plane_num][0] == airport_names[x[1]]:
                    planes_in_ports[plane_num][1] += 1
return sched
```

Then we have a function called run simulation, which runs through moving planes around the airports using the functions from before and a schedule. This function checks if we are running a simulation with or without a storm, evaluates any storm delays, and checks for any departures. Then it updates the airplanes and, as they arrive at the airports, updates the airports so they know what planes are available.

```
def run_simulation(storm):
    planes_in_air_at_time = []
    delayed_per_hour = []
    total_delays = 0
    delays = 0
    #then, step through each time in the run_simulation
    for step in range(length_of_simulation):
```

```
        delays = 0 #reset the number of delays for time step
        atc.update() #update the time each step
        for airport in airport_names:
            #run through each airport and check if there are any storms
            #which affect the current flights and update accordingly,
            #also add any delays caused by this
            if storms:
                delays += airport_dict[airport].storm_check()
            #send any flights, and if there arent enough airplanes
            #then add that into our calculated delays too
            delays += airport_dict[airport].departure_update()
        for plane in plane_names:
            #have plane check if it should be arriving somewhere
            plane_dict[plane].update(airport_dict)
        for airport in airport_names:
            #then process the arriving planes in the airports
            airport_dict[airport].arrival_update()
        #determine how many planes are currently flying
        planes_currently_flying = 0
        for plane in plane_names:
            if plane_dict[plane].flying == True:
                planes_currently_flying += 1
        planes_in_air_at_time.append(planes_currently_flying)
        delayed_per_hour.append(delays)
        total_delays += delays
```

Two important functions that run_simulation() calls are airport.storm_check() and airport.departure_update() These functions work in the airports to determine if there are delays to the schedule caused by storms and also to determine if there are planes available and send out flights based on this information. These two functions also keep track of delays in each time step and help us to track our final output data.

```
def storm_check(self):
    '''checks storm info and update airport schedule with delays'''
    storm_info = atc.storm_info
    self.storm_delays = 0
    for time_step in range(len(self.airport_schedule)):
        schedule_step = self.airport_schedule[time_step]
        if schedule_step[0] == current_time:
            flight_time = find_flight_time(airport_dict, airport_names, self.code, schedule_step
            # storm_info is formatted as storm_time, storm_location, storm_length
            if (storm_info[0] <= (current_time + flight_time)) and ((current_time + flight_time)
                self.airport_schedule[time_step][0] += 1
                self.storm_delays += 1
            elif (storm_info[0] <= current_time <= (storm_info[0] + storm_info[2])) and (self.co
                self.airport_schedule[time_step][0] += 1
                self.storm_delays += 1
    return self.storm_delays
```

```
def departure_update(self):
    '''tells planes when to leave and for where'''
    global current_time
    planes_departing = [] # log what planes leave in this update
    self.delays = 0
    for time_step in range(len(self.airport_schedule)):
        schedule_step = self.airport_schedule[time_step]
        if schedule_step[0] == current_time: # if there is a flight scheduled now
            if len(self.planes_ready) > 0: # if we have planes available
                plane_flying = self.planes_ready[0] # send out the next available plane
                flight_time = find_flight_time(airport_dict, airport_names, self.code, schedule_
                plane_dict[plane_flying].assign_new_destination(schedule_step[2], flight_time, s
                self.planes_departing.append(plane_flying)
                if len(self.planes_ready) == 1: # if that was the last available plane, the list
                    self.planes_ready = []
                else: # otherwise just remove the plane from the list
                    self.planes_ready.remove(plane_flying)
            else: # if we don't have planes available
                self.airport_schedule[time_step][0] += 1 # push back that flight
                self.delays +=1 # add an hour of delay
    self.planes_departing_log.append(len(self.planes_departing)) # track how many planes left in
    return self.delays
```

Then we track all of our data for that simulation run in global variables depending on if we were running a storm or not.

```
#keeping track of metrics about how many storms and delays there were
if storms:
    storm_flights = planes_in_air_at_time
    storm_delays = delayed_per_hour
    for airport in airport_names:
        storm_planes_arriving.append(airport_dict[airport].planes_arriving_log)
        storm_planes_departing.append(airport_dict[airport].planes_departing_log)
else:
    no_storm_flights = planes_in_air_at_time
    no_storm_delays = delayed_per_hour
    for airport in airport_names:
        no_storm_planes_arriving.append(airport_dict[airport].planes_arriving_log)
        no_storm_planes_departing.append(airport_dict[airport].planes_departing_log)
```

In our main section, we set up all of out variables and also loop through our simulation twice, once without a storm and once with a storm. Throughout this we keep track of the number of delays and planes flying during each step so they can be used later when plotting out data.

```
'''allows us to run through the simulation twice with the same schedule to see how storms
affect the pattern'''
'''these global variable are changed throughout the simulation'''
current_time = -2
flight_time_array = []
```

```
plane_names = []
airport_names = read_csv(flight_time_array)
'''setting up the airports, planes, and schedule by initializing classes'''
airport_dict = create_airports()
plane_dict = create_airplanes()
schedule = create_schedule(airport_names, airport_size, length_of_schedule, airport_dict)
storm_planes = schedule.copy()
atc = ATC()
atc.storm_schedule()
'''looping through the same schedule with and without a stom'''
for storm in range(2):
    #resetting the airports and planes so each time they start from their initial values
    airport_dict = {}
    plane_dict = {}
    plane_names = []
    airport_dict = create_airports()
    plane_dict = create_airplanes()
    plane_tracking = []
    current_time = -2
    if storm == 0:
        atc.divide_schedule(schedule, airport_dict)
        run_simulation(False)
    else:
        atc.divide_schedule(storm_planes, airport_dict)
        run_simulation(True)
```

We plotted the arrivals, delays, and number of flights occurring at each time for the two different conditions. We also plotted a comparison of the flight times and delays between the data set with a storm n without to compare how changing some of the parameters affected delays.

3 Results

We ran our model using a variety of starting conditions in order to compare the results. Our base conditions included a flying probability of 95% and an initial plane distribution of 3 planes per airport, run with a 100-hour schedule in a 125-hour time frame. For our base case, the storm was generated in ORD at time 12 and lasted for 4 hours. Figure 1 shows the base case without a storm, Figure 2 shows with a storm, and Figure 3 shows the summary. For clarity, we are only including the summary graphs of the other test cases here.

We decreased the probability of a flight being scheduled at a particular time to compare situations with more or less planes available to recover from a delay. This case had a 55% flight probability. The storm occurred at LAS at time 14 and was of length 5. This summary can be seen in Figure 4.

We also decreased the length of the simulation and schedule, to confirm the scalability of our model. This case had a 50-hour schedule run in a 75-hour simulation. The storm occurred at BOS at time 10 and was of length 4. This summary can be seen in Figure 5.

We increased the number of planes each airport started with to 10, partially to confirm scalability and partially to test if more planes available would mean a faster recovery from a delay. The storm occurred at ATL at time 6 and was of length 3. This summary can be seen in Figure 6.

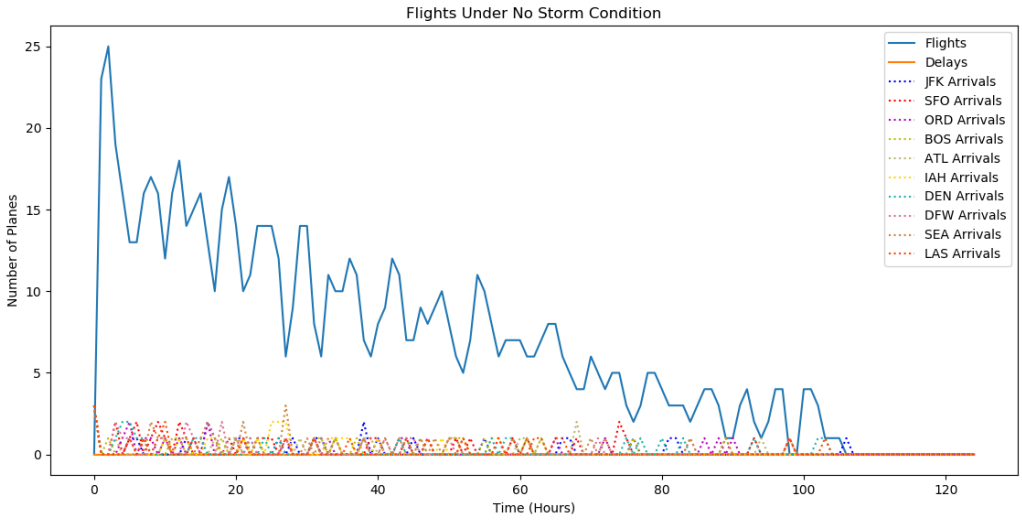

Figure 1: Graph of overall flights, overall delays, and airport-specific arrivals, with conditions of 95% flying probability, 125 hour simulation, 3 planes per airport initially, 100 hour schedule, with no storms.

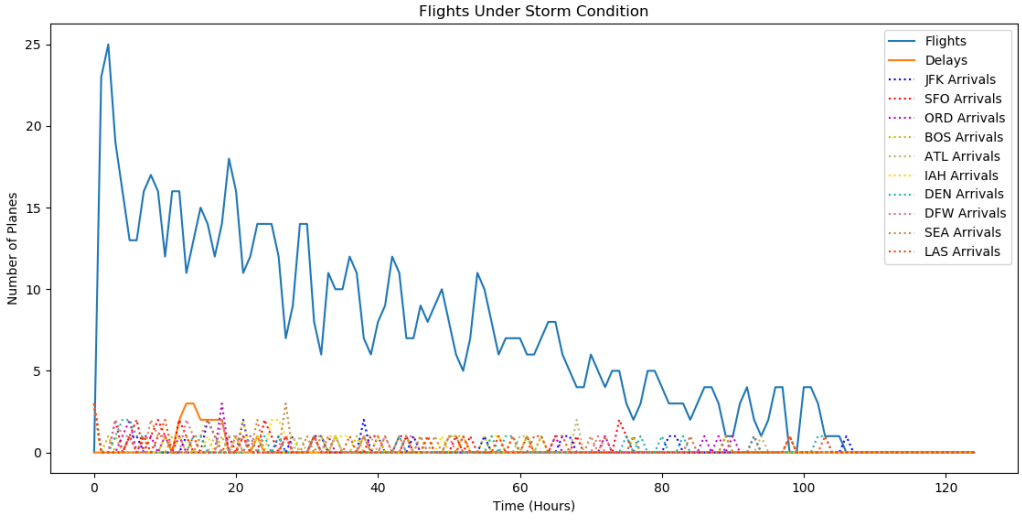

Figure 2: Graph of overall flights, overall delays, and airport-specific arrivals, with conditions of 95% flying probability, 125 hour simulation, 3 planes per airport initially, 100 hour schedule, with a storm at ORD at time 12 of length 4.

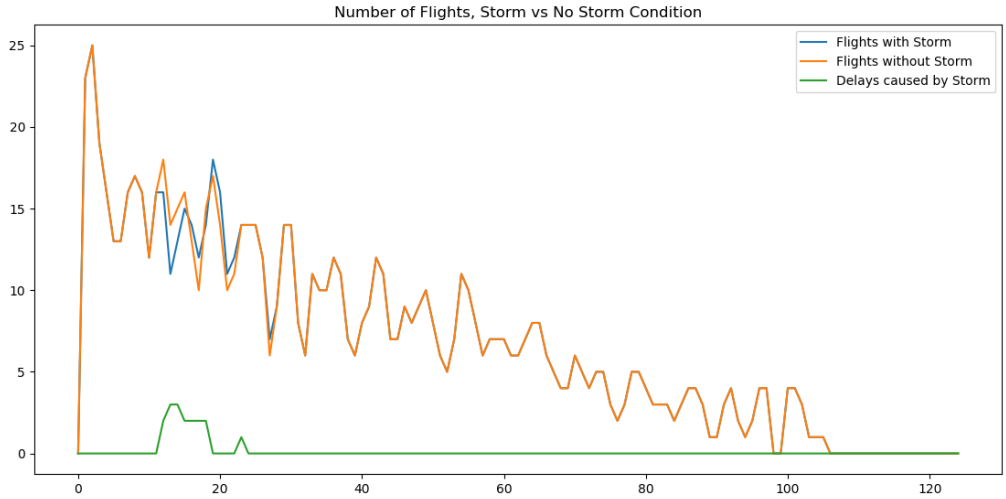

Figure 3: Graph of flights for the above conditions with and without a storm, and delays due to the storm.

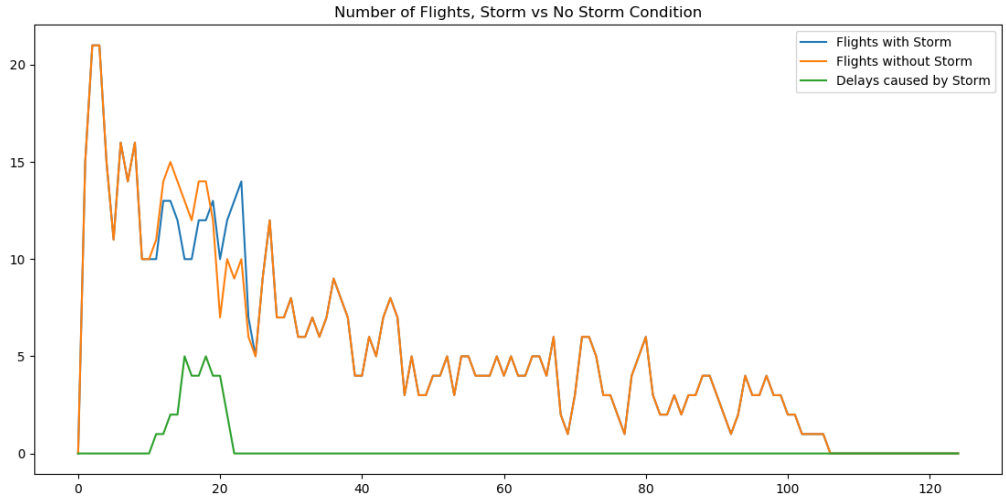

Figure 4: Graph of flights with base conditions modified to have a 55% flying probability, with and without a storm at LAS at time 14 of length 5, and delays due to storm.

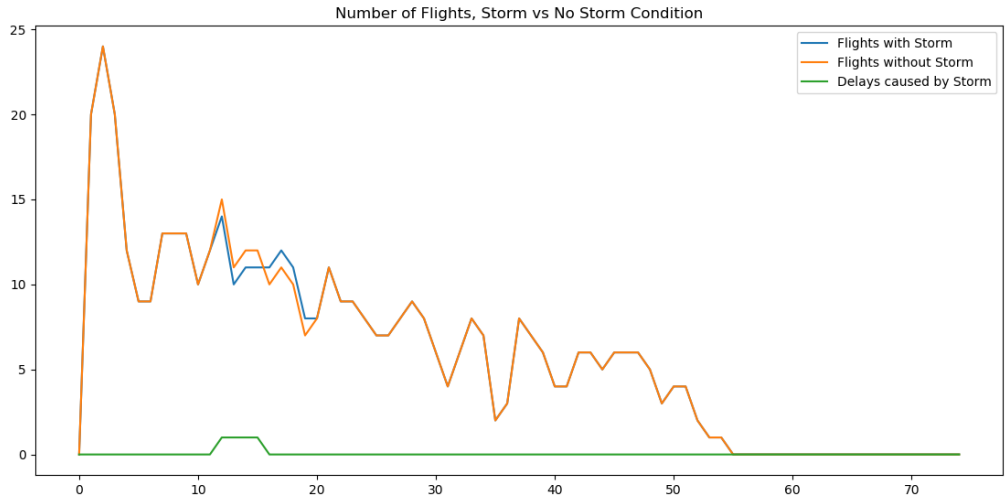

Figure 5: Graph of flights with base conditions modified to have a 50-hour schedule and 75-hour simulation, with and without a storm at BOS at time 10 of length 4, and delays due to storm.

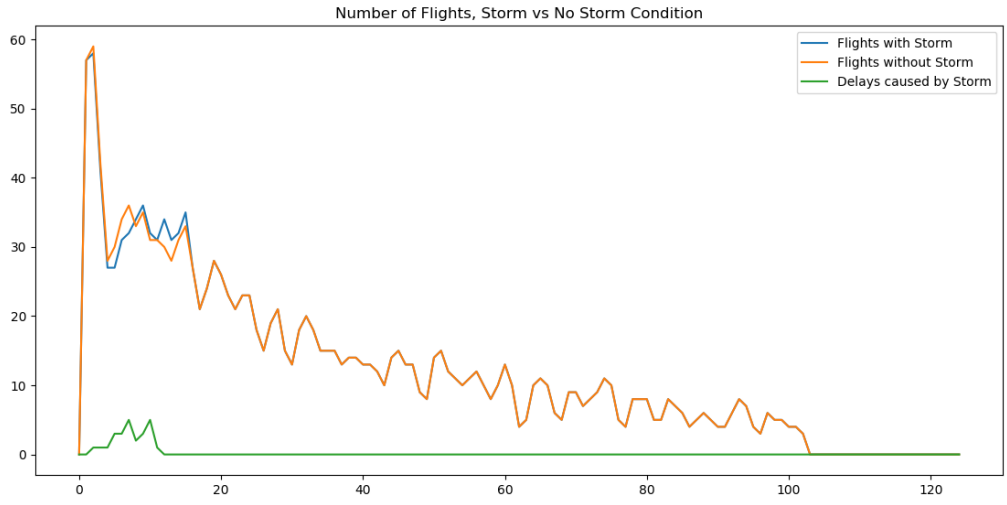

Figure 6: Graph of flights with base conditions modified to have 10 planes per airport initially, with and without a storm at ATL at time 6 of length 3, and delays due to storm.

4 Interpretation

Our data shows how a storm changes the flight pattern relative to flights on the same schedule lacking delays. One interesting thing we found in our data was the pattern that the storm flights had, which was a wave following a sort of $-1 * \sin(x)$ path relative to the non-storm flights. This reflects how there is a large delay in flights prior to and during the storm so there are less planes in the air, however after the sky is clear to fly again there is a spike in delayed flights as more planes are sent out to make up for all of the loss in flights. The area on our storm vs. non-storm flights graph where the two lines differ shows the amount of time that the flights are impacted by the storm before they are able to recover.

Our base case didn't take very long to recover from its storm. This makes sense because the storm was of length 4, and we can see its impact in the delay spike just before time 20. During this spike, the number of flights in the situation with a storm decreased, as expected, and the flights increased soon after to compensate. It is unclear what the small spike in delays just after time 20 is.

With a lower probability of flights, it took longer for the with and without storm conditions to converge again. The flights with a storm vary more, as seen in the large spike of recovery flights directly after the delay spike ends. This indicates having less frequent flights leads to a more erratic recovery.

In a shorter time frame, the model displayed similar behavior to the base case. The number of flights in the case with a storm decreased during the delay but promptly recovered afterward and the two situations converged again without much incident. From this we can infer that our model works for a variety of time frames.

With more planes per airport, we saw a large spike in initial flights, as each airport had more planes available to send out. The storm delays, though, behaved in much the same way as the base case. After an initial dip, the flights in the storm case recovered. However, in this case we can see two spikes in delays, which may be a sign of heightened delays at the beginning and end of a storm. This also may be related to the schedule of the airport whose departures were delayed.

Our model might not be exactly accurate to life since it relies on a few assumptions about the interchangeability of planes and how long some planes might be free before flights. The lack of some of the factors which play into flight times like runway availability aren't taken into account which definitely limits our model. However, on the whole it gave us some insight into how a flight schedule reacts to delays, particularly that having less frequent flights results in less predictable ways and that having a different airport size or a different length of time to study does not have a large impact on how the model delays behave overall.