Question 12.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a design of experiments approach would be appropriate.

I work for a FinTech company that analyzes Insider Trading data. Our primary method of delivery to our clients is to create pdf reports. The reports are created by saving report oriented html pages from our website into pdf format. These reports have links in them that refer back to the website. We would like these reports to generate better click throughs back to the site, and generate better client engagement with our product. We could design an A/B test of the report and tweak the design of the reports layout to see which version results in higher click throughs.

Question 12.2

To determine the value of 10 different yes/no features to the market value of a house (large yard, solar roof, etc.), a real estate agent plans to survey 50 potential buyers, showing a fictitious house with different combinations of features. To reduce the survey size, the agent wants to show just 16 fictitious houses. Use R's FrF2 function (in the FrF2 package) to find a fractional factorial design for this experiment: what set of features should each of the 16 fictitious houses have? Note: the output of FrF2 is "1" (include) or "-1" (don't include) for each feature.

Since this was generated using Jupyter Notebook, the corresponding R code was cut and paste into the markdown cell below. The FrF2 package has an option to name your factors, which is demonstrated below.

```
> require(FrF2)
Loading required package: FrF2
Loading required package: DoE.base
Loading required package: grid
Loading required package: conf.design

Attaching package: ‘DoE.base’

The following objects are masked from ‘package:stats’:

    aov, lm

The following object is masked from ‘package:graphics’:

    plot.design

The following object is masked from ‘package:base’:

    lengths

> set.seed(3836)
>
> home_factors = c("fireplace", "pool", "waterfront", "gated", "culdesac", "patio",
"lanai", "hvac", "garage", "basement")
> factors <- FrF2(16,factor.names = home_factors)
> factors
class=design, type= FrF2
> factors
   fireplace pool waterfront gated culdesac patio lanai hvac garage basement
1         -1    1          1     1       -1    -1     1   -1      1       -1
2          1    1          1     1        1     1     1    1      1        1
3         -1   -1         -1    -1        1     1     1    1     -1        1
4         -1   -1         -1     1        1     1     1   -1      1       -1
5         -1   -1          1     1        1    -1    -1   -1     -1        1
6          1    1         -1    -1        1    -1    -1   -1      1        1
7          1   -1          1     1       -1     1    -1    1     -1       -1
8          1    1          1    -1        1     1     1   -1     -1       -1
9          1   -1         -1     1       -1    -1     1    1      1        1
10        -1    1         -1    -1       -1     1    -1    1      1       -1
11         1   -1         -1    -1       -1    -1     1   -1     -1       -1
12         1   -1          1    -1       -1     1    -1   -1      1        1
13        -1   -1          1    -1        1    -1    -1    1      1       -1
14        -1    1          1    -1       -1    -1     1    1     -1        1
15         1    1         -1     1        1    -1    -1    1     -1       -1
16        -1    1         -1     1       -1     1    -1   -1     -1        1
class=design, type= FrF2
```

Question 13.1

For each of the following distributions, give an example of data that you would expect to follow this distribution (besides the examples already discussed in class). a. Binomial b. Geometric c. Poisson d. Exponential e. Weibull

a - binomial

A classic example of use for the binomial distribution is the flipping of a fair coin and counting the numbers of 'heads' and 'tails'. It can also be used for polling data, by asking independent voters who the voted for.

b - Geometric

Flipping of a fair coin can be used here, but you can count the number of flips until a 'head' or a 'tail' occurs. The flipping is now not independent, and the next flip is dependent on not getting the desired outcome on the prior flip. This can also be applied to families who keep trying for a boy or a girl, and don't stop having babies until they are successful in havig the sex they wanted.

c - Poisson

The poisson distribution would be good in analyzing number of website visits per hour. It could be used to determine the amount of customer engagement my employer is having with clients.

d - Exponential

The exponential distribution can be used to model continuous time markov chains, which are important for options pricing from the Black-Scholes Option Model.

e - Weibull

The Weibull distribution is most commonly used for failure analysis of a given, machine, product, etc. It can be used as a measure of quality control to detect if something is defective and/or is failing too early.

Question 13.2

In this problem you, can simulate a simplified airport security system at a busy airport. Passengers arrive according to a Poisson distribution with λ1 = 5 per minute (i.e., mean interarrival rate □1 = 0.2 minutes) to the ID/boarding-pass check queue, where there are several servers who each have exponential service time with mean rate □2 = 0.75 minutes. [Hint: model them as one block that has more than one resource.] After that, the passengers are assigned to the shortest of the several personal-check queues, where they go through the personal scanner (time is uniformly distributed between 0.5 minutes and 1 minute). Use the Arena software (PC users) or Python with SimPy (PC or Mac users) to build a simulation of the system, and then vary the number of ID/boarding-pass checkers and personal-check queues to determine how many are needed to keep average wait times below 15 minutes. [If you're using SimPy, or if you have access to a non-student version of Arena, you can use λ1 = 50 to simulate a busier airport.]

```
Import the python modules needed to run the simulation.
```

In [11]:
```
import random
import simpy
import numpy as np
```

Set our initial variables

In [27]:
```
RANDOM_SEED = 3836
NUM_CHECKERS = 18 # number of boarding pass checkers
NUM_SCANNERS = 8 # number of security scanners
ARRIVAL_RATE = 5 # passengers per minute
MIN_SCAN = 0.5 # scanner min time for uniform distribution
MAX_SCAN = 1 # scanner max time for uniform distribution
CHECKTIME = 0.75 # boarding pass check rate in minutes
SIM_TIME = 1440 # simulation time (24 hrs)
REPLICATIONS = 100 # number of replications
```

Creat a SimulationStats class to hold our analysis results.

In [13]:
```
# global s is used to access SimulationStats inside the simulation
global s
class SimulationStats(object):
    def __init__(self):
        self.num_travelers = 1
        self.checkTimes = []
        self.scanTimes = []
        self.totalWaitTimes = []
```

Create our Security Class so we can model our simulation times.

In [14]:
```
class Security(object):
    """Airpor Security has a limited number of checkes and scanners (``NUM_CHECKERS`
    check people in through security.

    A traveler has to report to one of the checkers.
    They can then start the security process.
    The traveler has to go through boarding pass checking, and xray scanning.

    """
    def __init__(self, env, num_checkers, num_scanners, check_time, min_scan, max_sc
        self.env = env
        self.checker = simpy.Resource(env, num_checkers)
        self.scanner = simpy.Resource(env, num_scanners)
        self.checktime = random.expovariate(check_time)
        self.scantime = random.uniform(min_scan, max_scan)

    def check(self, traveler):
        """The security checking process. It takes a ``traveler`` and tries
        to process it."""
        yield self.env.timeout(self.checktime)
        #print("Security is %d%% through checking %s" %
        #      (random.randint(50, 99), traveler))

    def scan(self, traveler):
        """The security scanning process. It takes a ``traveler`` and tries
        to process it."""
        yield self.env.timeout(self.scantime)
        #print("Security is %d%% through scanning %s" %
        #      (random.randint(50, 99), traveler))
```

Create our traveler that enters the Security Point. We append the wait times of each traveler who goes through security to our SimulationStats class.

In [15]:
```
def traveler(env, name, sp):
    """The traveler process (each traveler has a ``name``) arrives at the security p
    (``sp``) and is processed.

    It then starts the security process, waits for it to finish and
    leaves to never come back ...

    """
    #print('%s arrives at the boarding pass security point at %.2f.' % (name, env.nc
    check_arrive = env.now
    with sp.checker.request() as request:
        yield request

        #print('%s enters the boarding pass check point at %.2f.' % (name, env.now))
        yield env.process(sp.check(name))

        #print('%s leaves the boarding pass check point at %.2f.' % (name, env.now))
    check_depart = env.now
    s.checkTimes.append(check_depart - check_arrive)
    scan_arrive = env.now
    with sp.scanner.request() as request:
        yield request

        #print('%s enters the boarding pass scan point at %.2f.' % (name, env.now))
        yield env.process(sp.scan(name))

        #print('%s leaves security at %.2f.' % (name, env.now))
    scan_depart = env.now
    s.scanTimes.append(scan_depart - scan_arrive)
    s.totalWaitTimes.append(scan_depart - check_arrive)
```

This initializes our simulation for a specific of parameters with the variables set earlier.

In [16]:
```
def setup(env, num_checkers, num_scanners, checktime, arrival_rate, minscan, maxscan
    """Create a security point, a number of initial travelers and have travelers arr
    approx. every ``t_inter`` minutes."""
    # Create the security point
    security = Security(env, num_checkers, num_scanners, checktime, minscan, maxscan

    while True:

        arrival_inter = random.expovariate(arrival_rate)
        yield env.timeout(arrival_inter)
        s.num_travelers += 1
        env.process(traveler(env, 'Traveler %d' % s.num_travelers, security))
```

The run_simulation function is used to loop through different values of checkers and scanners with a specific arrival_rate specified. In order to reduce randomness of the results, the simulation runs 100 times for each set of input values. Through trial and error I determined that the simulation is much more sensitive to the number of checkers assigned, so it is assigned to the inner loop to speed up the finding of a solution faster. The inner loop on checkers will exit if the average wait time is < 15 minutes. The information is then added to our wait_times array and returned for analysis at the end of the simulation. It will print the number of checkers, scanners, and the average wait time as the simulation progresses, just to keep an eye on it while running. There will be multiple scanner simulations for the same checkers parameter. We will choose from among the results returned for the best candidate inputs.

In [38]:
```
def run_simulation(scan_range, check_range, arrival_rate):
    random.seed(RANDOM_SEED)  # This helps reproducing the results
    wait_times = []

    for scanners in scan_range:
        for checkers in check_range:
            sim = {}
            total_travelers = []
            total_travelers_processed = []
            total_wait_time = []
            avg_wait_time = []
            sd_wait_time = []
            total_check_time = []
            avg_check_time = []
            sd_check_time = []
            total_scan_time = []
            avg_scan_time = []
            sd_scan_time = []
            for iteration in range(REPLICATIONS):
                global s
                s = SimulationStats()
                s.num_travelers = 0

                # Create an environment and start the setup process
                env = simpy.Environment()
                env.process(setup(env, checkers, scanners, CHECKTIME, arrival_rate,

                # Execute!
                env.run(until=SIM_TIME)
                #print "Checkers: %s, Scanners: %s, Iteration: %s" % (checkers, scan
                total_travelers.append(s.num_travelers)
                total_travelers_processed.append(len(s.totalWaitTimes))
                total_wait_time.append(np.sum(s.totalWaitTimes))
                avg_wait_time.append(np.mean(s.totalWaitTimes))
                sd_wait_time.append(np.std(s.totalWaitTimes))
                total_check_time.append(np.sum(s.checkTimes))
                avg_check_time.append(np.mean(s.checkTimes))
                sd_check_time.append(np.std(s.checkTimes))
                total_scan_time.append(np.sum(s.scanTimes))
                avg_scan_time.append(np.mean(s.scanTimes))
                sd_scan_time.append(np.std(s.scanTimes))
            iter_avg_wait_time = np.mean(avg_wait_time)
            if iter_avg_wait_time < 15:
                sim['Checkers'] = checkers
                sim['Scanners'] = scanners
                sim['Total Travelers'] = np.mean(total_travelers)
                sim['Total Travelers Processed'] = np.mean(total_travelers_processed
                sim['Total Wait Time'] = np.mean(total_wait_time)
                sim['Avg Wait Time'] = np.mean(avg_wait_time)
                sim['Std Dev Wait Time'] = np.mean(sd_wait_time)
                sim['Total Check Time'] = np.mean(total_check_time)
                sim['Avg Check Time'] = np.mean(avg_check_time)
                sim['Std Dev Check Time'] = np.mean(sd_check_time)
                sim['Total Scan Time'] = np.mean(total_scan_time)
                sim['Avg Scan Time'] = np.mean(avg_scan_time)
                sim['Std Dev Scan Time'] = np.mean(sd_scan_time)
                summary = [scanners, checkers, sim['Avg Wait Time']]
                print summary
                wait_times.append(sim)
                break
    return wait_times
```

Loop through the use of 5 to 25 checkers and scanners. I stepped through every 5 values, as an initial exploratory pass. Our arrival rate is 5 travelers per minute. The print values are ordered by scanner and checker count in ascending order.

In [ ]:
```
check_range = range(5, 26, 5)
scan_range = check_range
wait_times_5 = run_simulation(scan_range, check_range, ARRIVAL_RATE)
```

In [39]:
```
wait_times_5
```

```
[5, 20, 5.1436895558580513]
[10, 15, 14.16820236127419]
[15, 15, 9.7331163105859613]
[20, 20, 6.5799041910286213]
[25, 15, 12.781209695725149]
```

Out[39]:
```
[{'Avg Check Time': 3.5518044491711538,
  'Avg Scan Time': 1.592859152163375,
  'Avg Wait Time': 5.1436895558580513,
  'Checkers': 20,
  'Scanners': 5,
  'Std Dev Check Time': 1.4133228552669386,
  'Std Dev Scan Time': 0.72443789582112583,
  'Std Dev Wait Time': 2.1258696464924549,
  'Total Check Time': 23384.358888316463,
  'Total Scan Time': 11395.659874034063,
  'Total Travelers': 7184.1800000000003,
  'Total Travelers Processed': 7145.4099999999999,
  'Total Wait Time': 34760.703870569407},
 {'Avg Check Time': 13.415643195240552,
  'Avg Scan Time': 0.76070144083996194,
  'Avg Wait Time': 14.16820236127419,
  'Checkers': 15,
  'Scanners': 10,
  'Std Dev Check Time': 7.021044778575722,
  'Std Dev Scan Time': 0.0095401463607037613,
  'Std Dev Wait Time': 7.0243647461448608,
  'Total Check Time': 57526.991914886239,
  'Total Scan Time': 5393.7790716324598,
  'Total Travelers': 7204.5900000000001,
  'Total Travelers Processed': 7074.3299999999999,
  'Total Wait Time': 62852.762803979364},
 {'Avg Check Time': 8.9892399115407979,
  'Avg Scan Time': 0.74999266508348983,
  'Avg Wait Time': 9.7331163105859613,
  'Checkers': 15,
  'Scanners': 15,
  'Std Dev Check Time': 4.5739950495816837,
  'Std Dev Scan Time': 5.8292616322089959e-05,
  'Std Dev Wait Time': 4.5706266838213727,
  'Total Check Time': 41872.401333632668,
  'Total Scan Time': 5342.2206621638152,
  'Total Travelers': 7198.9099999999999,
  'Total Travelers Processed': 7110.3299999999999,
  'Total Wait Time': 47159.094912983921},
 {'Avg Check Time': 5.8183985800160656,
  'Avg Scan Time': 0.76369393690353948,
  'Avg Wait Time': 6.5799041910286213,
  'Checkers': 20,
  'Scanners': 20,
  'Std Dev Check Time': 2.6349868894511279,
  'Std Dev Scan Time': 3.210998030157525e-14,
  'Std Dev Wait Time': 2.6338485358309756,
  'Total Check Time': 35308.538498472575,
  'Total Scan Time': 5473.2432437369089,
  'Total Travelers': 7213.7299999999996,
  'Total Travelers Processed': 7159.6000000000004,
  'Total Wait Time': 40750.188676895923},
 {'Avg Check Time': 12.057546798085092,
  'Avg Scan Time': 0.73519121044874169,
  'Avg Wait Time': 12.781209695725149,
  'Checkers': 15
```

After inspecting the results of the initial exploratory pass, the number of checkers needed for an average wait time < 15 minutes is likely between 11 and 15. The number os scanners is likely between 5 and 15. We will run our simulation over the range of these values to find when our under our target wait time of 15 minutes.

In [41]:
```
check_range = range(11, 16, 1)
scan_range = range(5, 16, 1)
wait_times_5_take2 = run_simulation(scan_range, check_range, ARRIVAL_RATE)
```

```
[6, 14, 14.610158172324367]
[9, 14, 5.521425188397691]
[13, 13, 14.05347827950632]
[14, 14, 11.544415010890274]
```

```
In [42]: wait_times_5_take2
```

```
Out[42]: [{'Avg Check Time': 13.819600862354015,
           'Avg Scan Time': 0.79696887578572184,
           'Avg Wait Time': 14.610158172324367,
           'Checkers': 14,
           'Scanners': 6,
           'Std Dev Check Time': 7.4014070264864253,
           'Std Dev Scan Time': 0.12046767337300816,
           'Std Dev Wait Time': 7.5071782618393907,
           'Total Check Time': 64247.383936965009,
           'Total Scan Time': 5632.0613503419236,
           'Total Travelers': 7197.7299999999996,
           'Total Travelers Processed': 7061.4700000000003,
           'Total Wait Time': 69818.315771448702},
          {'Avg Check Time': 4.7695997644601142,
           'Avg Scan Time': 0.75329319334024236,
           'Avg Wait Time': 5.521425188397691,
           'Checkers': 14,
           'Scanners': 9,
           'Std Dev Check Time': 2.2670572390424635,
           'Std Dev Scan Time': 0.016353537313367285,
           'Std Dev Wait Time': 2.2796932426950325,
           'Total Check Time': 26727.415772573655,
           'Total Scan Time': 5393.5137794005577,
           'Total Travelers': 7201.9099999999999,
           'Total Travelers Processed': 7155.2399999999998,
           'Total Wait Time': 32102.110161789995},
          {'Avg Check Time': 13.303214615387445,
           'Avg Scan Time': 0.75491677795614098,
           'Avg Wait Time': 14.05347827950632,
           'Checkers': 13,
           'Scanners': 13,
           'Std Dev Check Time': 7.074784958040726,
           'Std Dev Scan Time': 0.00081891683763204884,
           'Std Dev Wait Time': 7.0729297686489669,
           'Total Check Time': 72933.765188654143,
           'Total Scan Time': 5340.0737542319475,
           'Total Travelers': 7205.54,
           'Total Travelers Processed': 7078.7700000000004,
           'Total Wait Time': 78217.92766776013},
          {'Avg Check Time': 10.80015501978615,
           'Avg Scan Time': 0.74688971533803428,
           'Avg Wait Time': 11.544415010890274,
           'Checkers': 14,
           'Scanners': 14,
           'Std Dev Check Time': 5.7644698900952278,
           'Std Dev Scan Time': 0.00017970758603792926,
           'Std Dev Wait Time': 5.763008134065811,
           'Total Check Time': 51252.408300248448,
           'Total Scan Time': 5305.6458961743165,
           'Total Travelers': 7199.04,
           'Total Travelers Processed': 7093.5799999999999,
           'Total Wait Time': 56523.934323060486}]
```

Treating the weights of checkers and scanners equally, having 14 checkers, and 6 scanners has the lowest cost for getting the average wait time to < 15 minutes.

We will increase our arrival rate now to 50 travelers per minute. Since our arrival rate is 10x what it was in the previous simulation, we'll increase the numbers used to loop through so we will get results back, other wise our range values may be too low to break the average wait time of < 15 minutes constraint. We will loop through the the range of 150 to 200 checkers, and 50 to 100 scanners, stepping at every 5 values.

In [43]:
```
ARRIVAL_RATE_50 = 50
check_range = range(150, 201, 5)
scan_range = range(50, 101, 5)
wait_times_50 = run_simulation(scan_range, check_range, ARRIVAL_RATE_50)
wait_times_50
```

```
[50, 150, 12.010594220750857]
[55, 160, 8.2317944981782016]
[60, 160, 8.0983427562373169]
[65, 155, 14.776514945001646]
[70, 150, 6.2371430266300658]
[75, 150, 14.074811188061547]
[80, 165, 9.5078457341495568]
[85, 170, 8.0188136218788788]
[90, 155, 12.988114026366924]
[95, 150, 14.378708582050237]
[100, 165, 9.3167862834854596]
```

Out[43]:
```
[{'Avg Check Time': 11.197120392734337,
  'Avg Scan Time': 0.81879746638162887,
  'Avg Wait Time': 12.010594220750857,
  'Checkers': 150,
  'Scanners': 50,
  'Std Dev Check Time': 5.8395780872565615,
  'Std Dev Scan Time': 0.071620588257944096,
  'Std Dev Wait Time': 5.9067979193194571,
```

The main takeaway from running this simulation is there are multiple scenarios where you can arrive to an average wait time of < 15 minutes. The best solution to this problem would be arrived at by constructing a cost function of how much adding a checker vs adding a scanner would be. Based upon the values of the cost of adding an additional checker or scanner to the mix would determine which solution should be picked. The ideal solution would minimize the number of checkers and scanners needed. You would also want to do an analysis of the variation of the average wait time as well. If one model's average wait time is only a minute or 2 better than another model, but the model with the higher wait time has a much smaller variance, that may be the better model to choose.