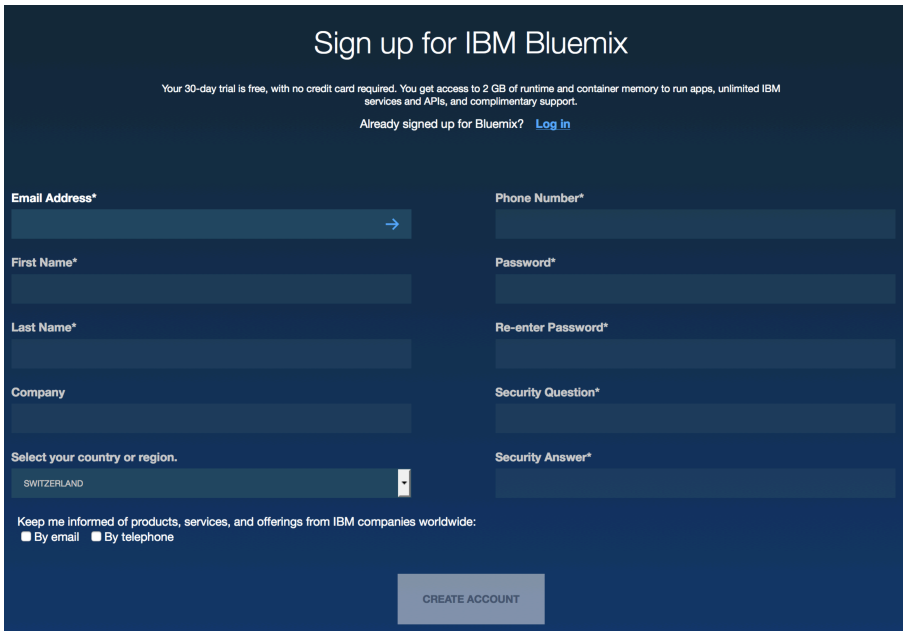
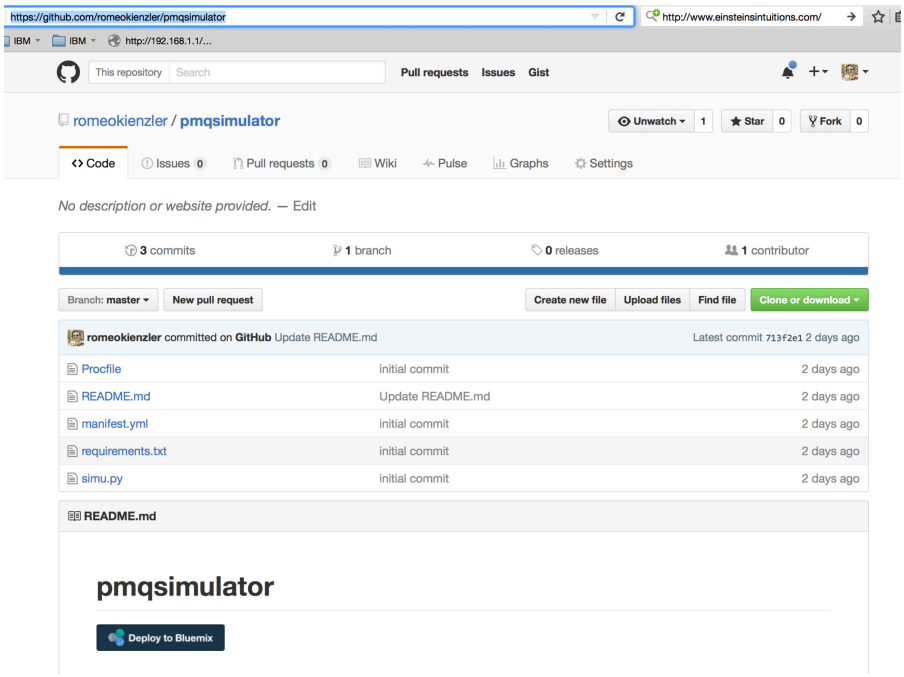
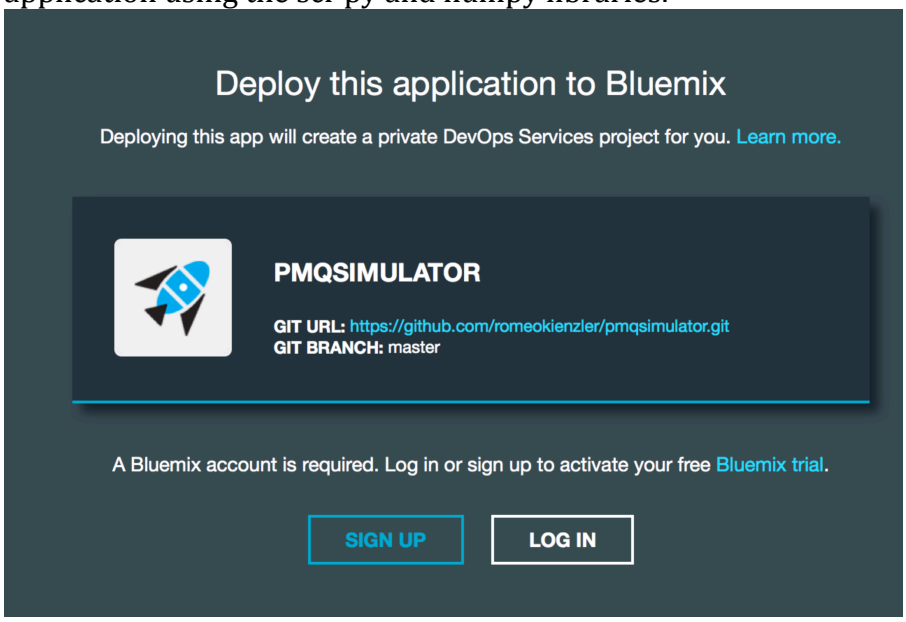
So first we have to create a account on the IBM Cloud – this is free of charge and after a 30 day trial period we'll end up in the free tier which allows us to deploy small application continuously without cost. Once the account is in place we can deploy this application using on-click deployment. This application creates a simulation of vibration data from a three-axis accelerometer attached to a hypothetical bearing in a machine. So it is a Industry 4.0 use case where we want to detect anomalies in the behavior of machine parts based on abnormal vibration patterns. One concern with artificially generated data is that is doesn't resemble real life experience. Therefore we will use the Lorenz Attractor Model – a data generator used in Chaos Theory because it is very hard to predict. On the right hand side we see the output of such a model where the system basically oscillates in two states. State transitions and amplitude are chaotic.



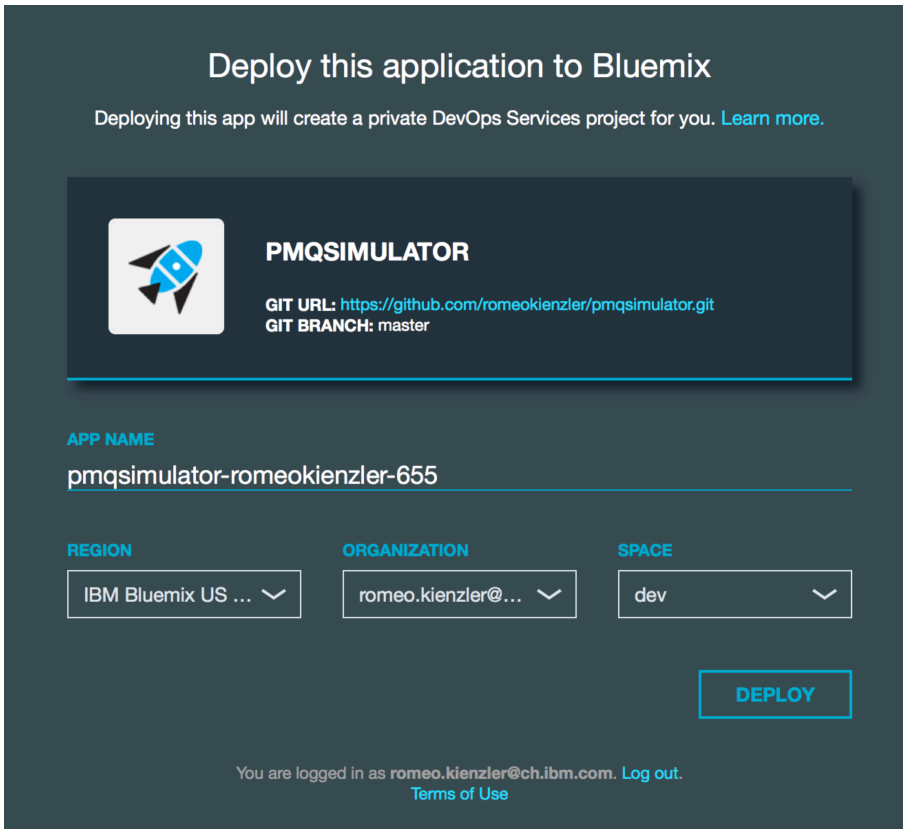
So let's get started with deployment of this test data generator. First we'll have to register for the free IBM Cloud account. Just fill-in those ten fields and click on "create account" – once done you'll receive a verification email where you have to click on the link provided.



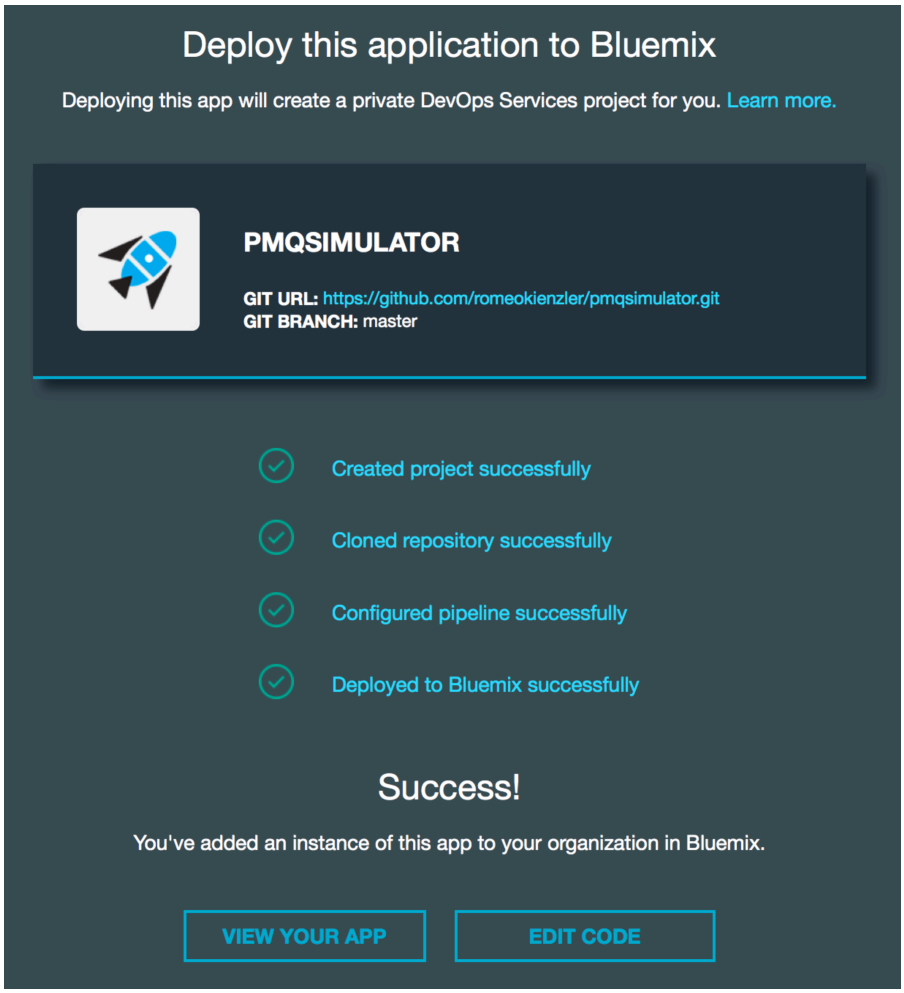
Open the following GitHub page (https://github.com/romeokienzler/pmqsimulator) and click on the Deploy to Bluemix button. As a side note you can also have a look at the source code of the test data generator if you are interested. Just click on simu.py – it is a Python application using the sci-py and numpy libraries.



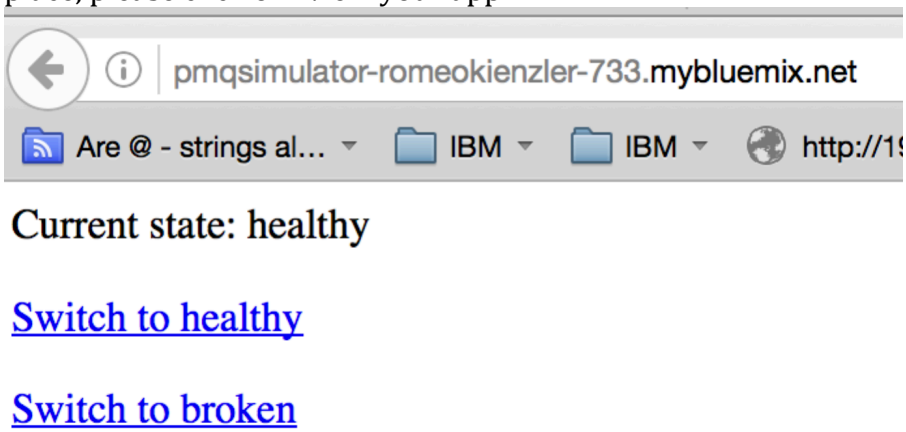
Now click on Log-In and enter the credentials you've obtained when you registered for the free cloud service.



Wait until the deploy button appears and click on it



Congratulations, once this is done you have your personal test data generator in place, please click on “view your app” …

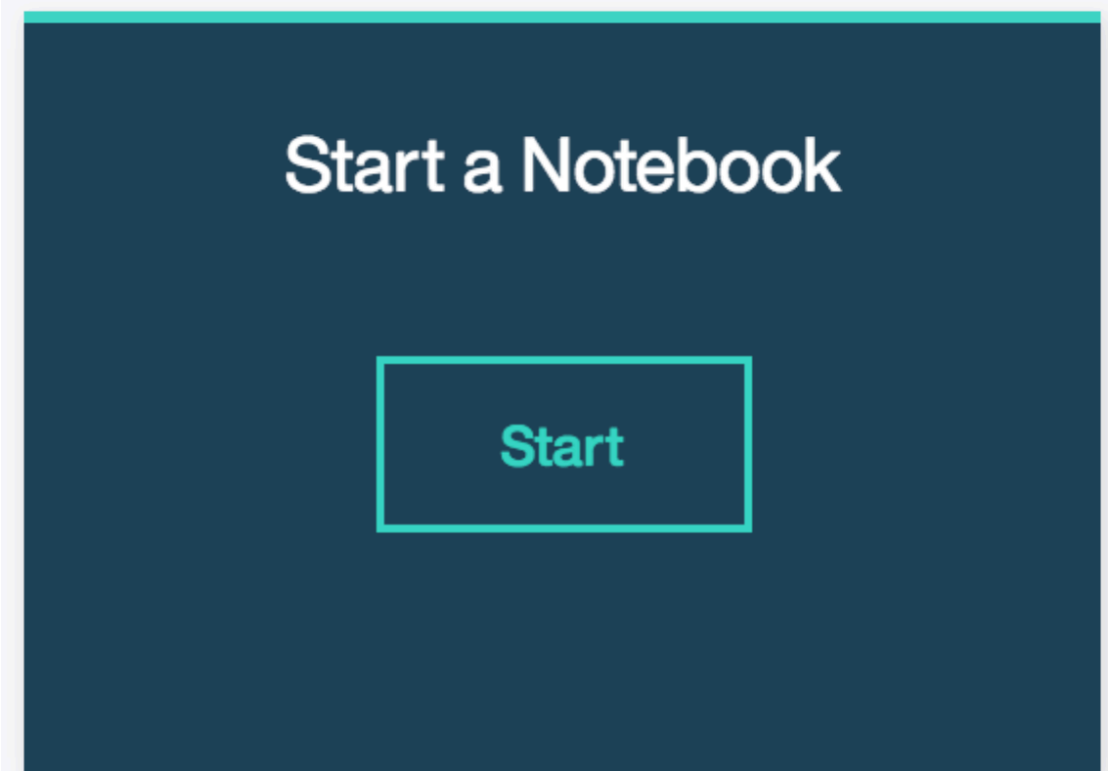


… and note down the URL since we will use it now.

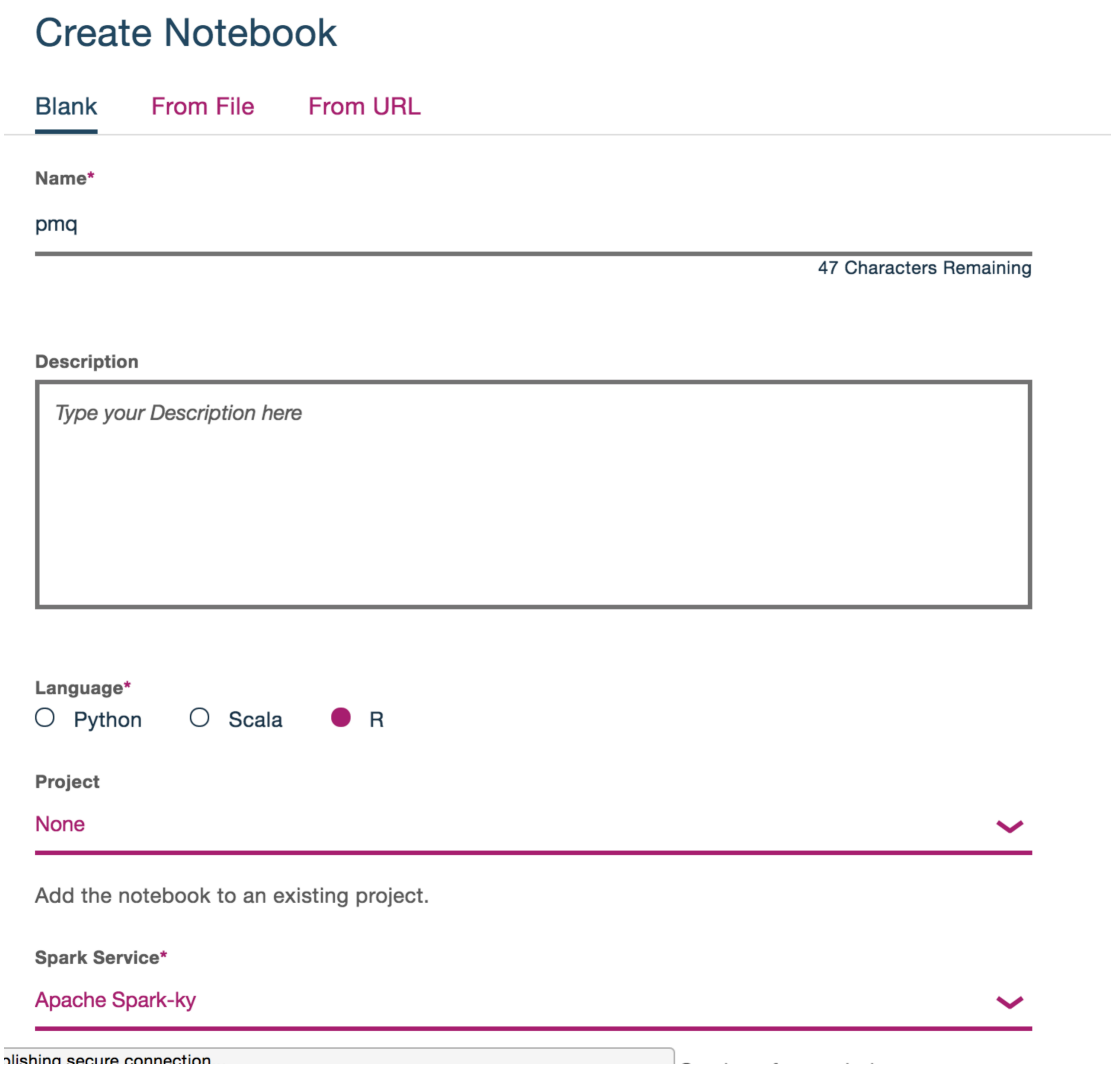
Append /data to the url and look at the CSV file you get back

Open this url and create an DataScience Experience (DSX) account using your existing Bluemix account http://datascience.ibm.com/registration/stepone

Open DSX on datascience.ibm.com and login

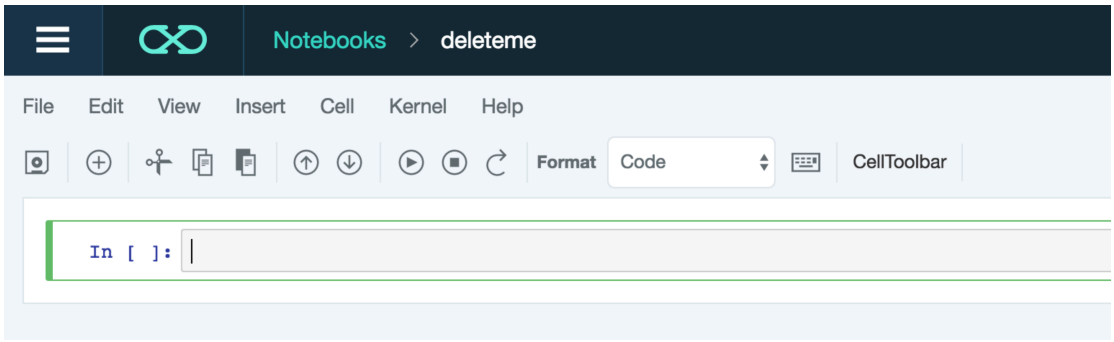


Click on “Start a new Notebook”



Provide a name and Choose “R” as Language”

Click on “Create Notebook”



Now you are ready to go

Install some required packages by pasting it into the field and hit the run button:

install.packages("h2o")

```
In [1]: install.packages("h2o")
```

```
Installing package into '/gpfs/global_fs01/sym_shared/YPProdSpark/user/s4c2-1e12ab68a45670-980ba6aaa6c3/R/libs'
(as 'lib' is unspecified)
also installing the dependency 'statmod'
```

install.packages("scatterplot3d")

```
In [4]: install.packages("scatterplot3d")
```

```
Installing package into '/gpfs/global_fs01/sym_shared/YPProdSpark/user/s4c2-1e12ab68a45670-980ba6aaa6c3/R/libs'
(as 'lib' is unspecified)
```

install.packages("ggplot2")

```
In [5]: install.packages("ggplot2")
```

```
Installing package into '/gpfs/global_fs01/sym_shared/YPProdSpark/user/s4c2-1e12ab68a45670-980ba6aaa6c3/R/libs'
(as 'lib' is unspecified)
```

Load those libraries:

library(h2o)
library(scatterplot3d)
library(ggplot2)
library(stats)

```
In [18]: library(h2o)
         library(scatterplot3d)
         library(ggplot2)
         library(stats)
```

Start the H20 DeepLearning Backend:
localH2O = h2o.init(nthreads = -1)

```
In [16]: localH2O = h2o.init(nthreads = -1)

 Connection successful!

R is connected to the H2O cluster:
    H2O cluster uptime:         26 minutes 53 seconds
    H2O cluster version:        3.10.0.6
    H2O cluster version age:    12 days
    H2O cluster name:           H2O_started_from_R_s4c2-1e12ab68a45670-980ba6aaa6c3_uqg059
    H2O cluster total nodes:    1
    H2O cluster total memory:   0.38 GB
    H2O cluster total cores:    48
    H2O cluster allowed cores:  48
    H2O cluster healthy:        FALSE
    H2O Connection ip:          localhost
    H2O Connection port:        54321
    H2O Connection proxy:       NA
    R Version:                  R version 3.3.0 (2016-05-03)

Warning message:
In .h2o.__checkConnectionHealth(): H2O cluster node 127.0.0.1:54321 is behaving slowly and should be inspected
 manuallyWarning message:
In .h2o.__checkConnectionHealth(): Check H2O cluster status here: http://localhost:54321/3/Cloud?skip_ticks=tru
e
```

Load the data and check for number of columns and rows. Paste the URL to the read.csv function and append slash /data to it, this is the endpoint delivering one second worth of data at a sampling rate of three thousand herz. Therefore the dimensionality is 3000 rows and four columns. One for time, and three for the vibration values of the bearing in each dimension:

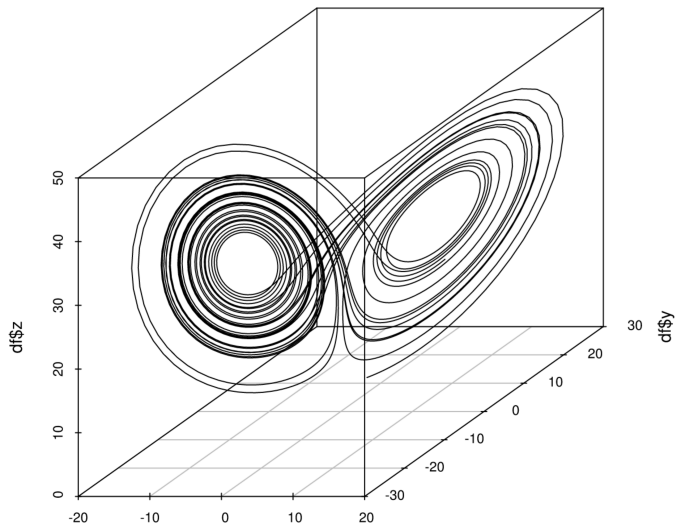
simuURL="https://pmqsimulator-romeokienzler-2310.mybluemix.net/data"
df = read.csv(simuURL,sep = ";",header = FALSE)
dim(df)

```
In [9]: simuURL="https://pmqsimulator-romeokienzler-2310.mybluemix.net/data"
        df = read.csv(simuURL,sep = ";",header = FALSE)
        dim(df)

Out[9]:     3000   4
```

Set the column names and 3D plot it, if we scatter plot the data we obtain the famous structure of the Lorenz attractor model:

colnames(df) = c("t","x","y","z")
scatterplot3d(dfx,dfy,df$z, type = "l")

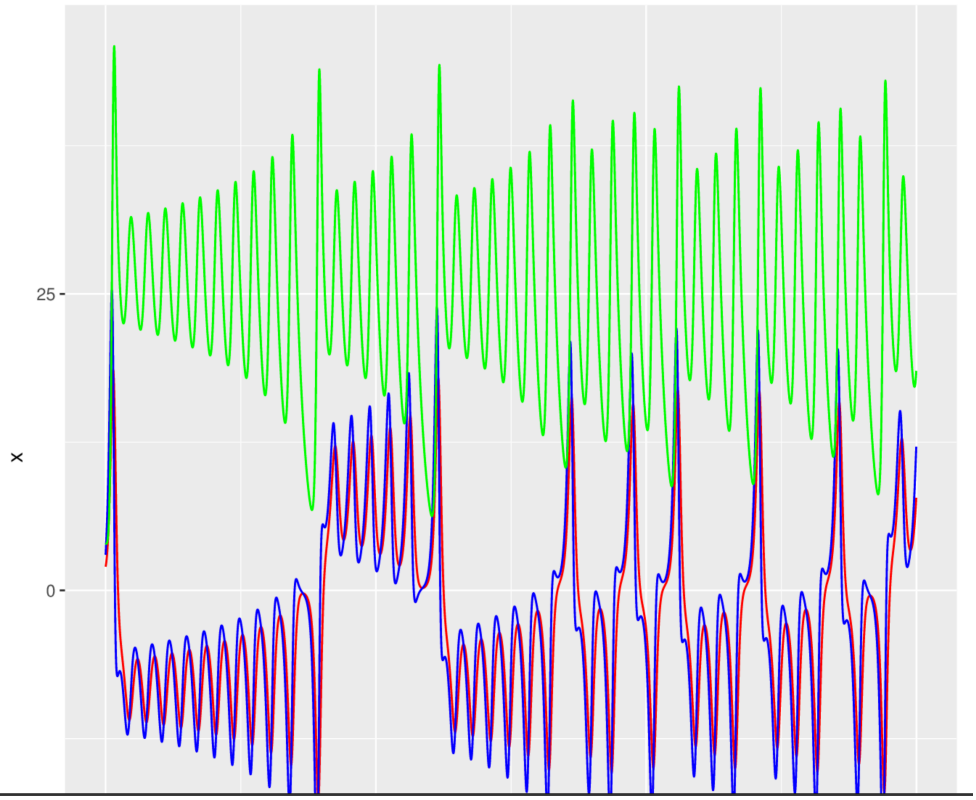
```
In [10]: colnames(df) = c("t","x","y","z")
         scatterplot3d(df$x,df$y,df$z, type = "l")
```



Plot each vibration sensor axis on a single line chart. If we plot all dimensions individually we see a typical vibration pattern on bearings – some increasing amplitudes until the system transitions into a second stable state where it remains for some time to oscillate back. As mentioned before, those parameters are chaotic and nearly impossible to predict. But how shall we implement an algorithm able to detect anomalies in such a system?

```
ggplot() +
  geom_line(data = df, aes(x = t, y = x),colour = "red") +
  geom_line(data = df, aes(x = t, y = y), colour = "blue") +
  geom_line(data = df, aes(x = t, y = z), colour = "green")
```

```
In [11]: ggplot() +
           geom_line(data = df, aes(x = t, y = x),colour = "red") +
           geom_line(data = df, aes(x = t, y = y), colour = "blue") +
           geom_line(data = df, aes(x = t, y = z), colour = "green")
```

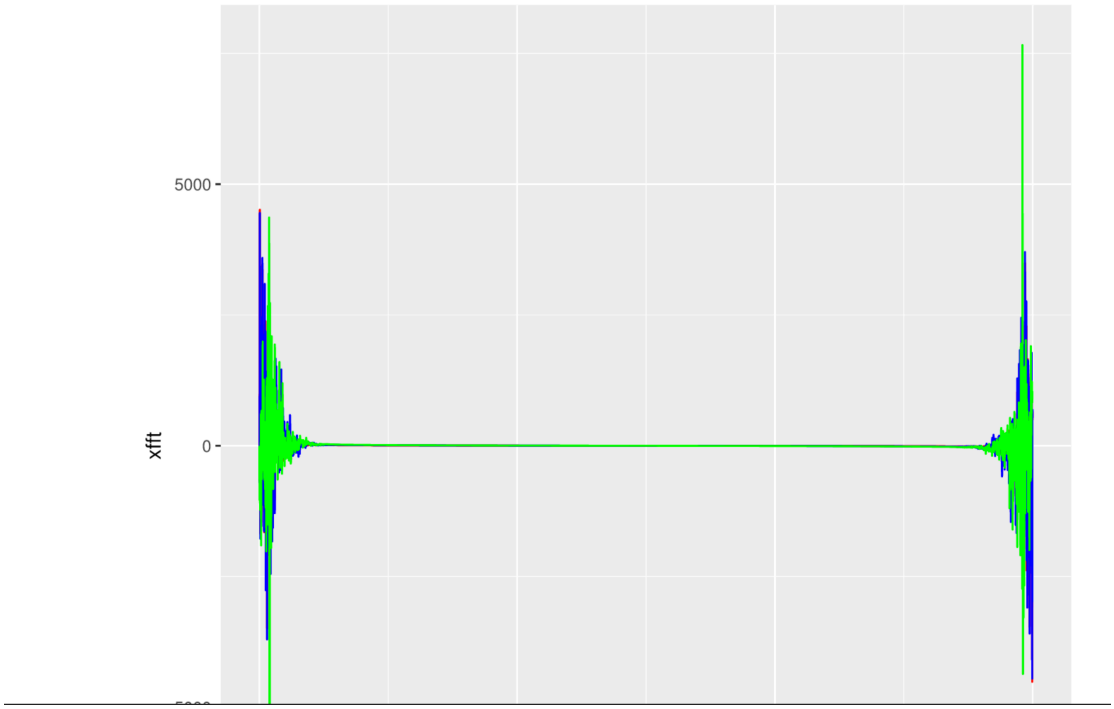


This first step is to transform this time series window to the frequency domain by using FFT. So here we see the frequency spectrum of bearing vibrations in a healthy condition. So let's switch our test data generator to produce vibrations in a condition where the bearing is about to break. Let's transform this time series from the time to the frequency domain using Fast Fourier Transformation (FFT):

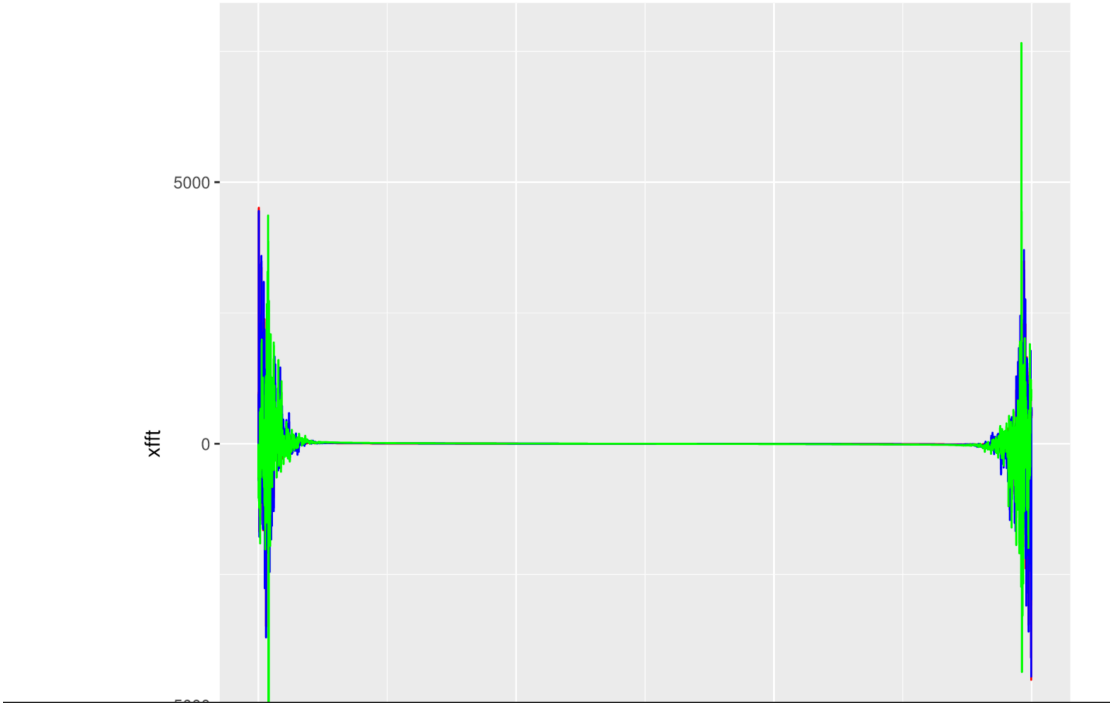
```
xfft = Im(fft(df$x))
yfft = Im(fft(df$y))
zfft = Im(fft(df$z))
ggplot() +
  geom_line(data = df, aes(x = t, y = xfft),colour = "red")  +
```

```
geom_line(data = df, aes(x = t, y = yfft), colour = "blue") +
geom_line(data = df, aes(x = t, y = zfft), colour = "green")
```

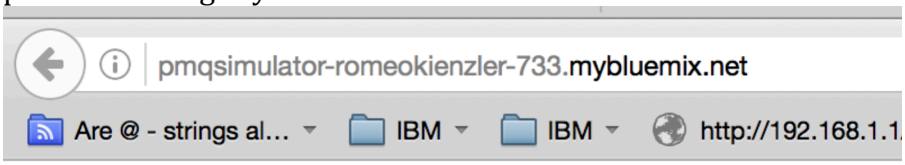
In [12]:
```
xfft = Im(fft(df$x))
yfft = Im(fft(df$y))
zfft = Im(fft(df$z))
ggplot() +
  geom_line(data = df, aes(x = t, y = xfft),colour = "red")  +
  geom_line(data = df, aes(x = t, y = yfft), colour = "blue") +
  geom_line(data = df, aes(x = t, y = zfft), colour = "green")
```



In [12]:
```
xfft = Im(fft(df$x))
yfft = Im(fft(df$y))
zfft = Im(fft(df$z))
ggplot() +
  geom_line(data = df, aes(x = t, y = xfft),colour = "red")  +
  geom_line(data = df, aes(x = t, y = yfft), colour = "blue") +
  geom_line(data = df, aes(x = t, y = zfft), colour = "green")
```



Let's go back to the test data generator page and click on broken. Now again, we will generate data according to the Lorenz attractor model but changed the parameters slightly.



Current state: broken

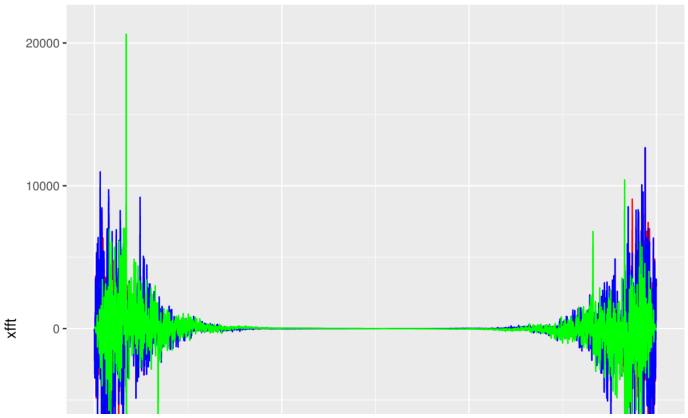
Switch to healthy

Switch to broken

If we now again plot the spectrum we see that it is different from the other. There are additional frequencies present indicating a bearing is about to break.

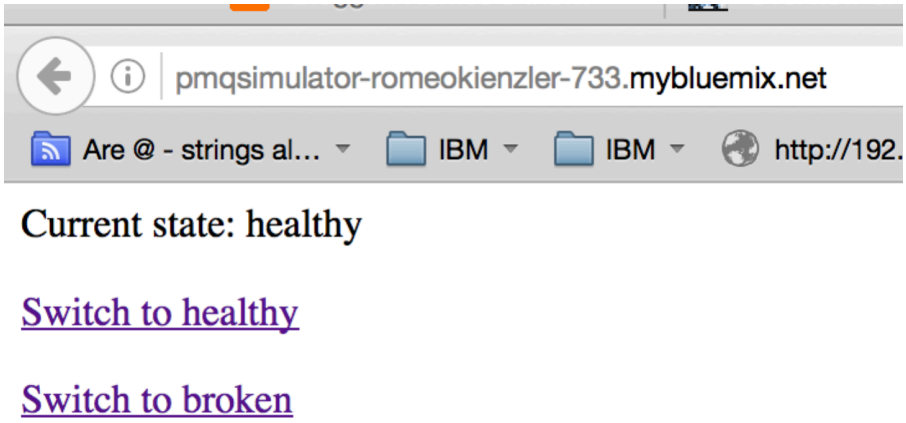
```
In [13]: df = read.csv(simuURL,sep = ";",header = FALSE)
         dim(df)
         colnames(df) = c("t","x","y","z")

         xfft = Im(fft(df$x))
         yfft = Im(fft(df$y))
         zfft = Im(fft(df$z))
         ggplot() +
           geom_line(data = df, aes(x = t, y = xfft),colour = "red")  +
           geom_line(data = df, aes(x = t, y = yfft), colour = "blue") +
           geom_line(data = df, aes(x = t, y = zfft), colour = "green")
```

```
Out[13]: 3000  4
```



But how can we decide which frequencies are indicating a problem? Just by looking at the two plots it is hard. Fortunately there is a solution to this, let's use a neural net.

First, let's switch back to a healthy condition again.



Since we want to analyze data in real-time we are using a loop continuously reading and analyzing data. We apply fft and load this transformed data to H2O

```
while (TRUE) {
  print("reading data...")
  df = read.csv(simuURL,sep = ";",header = FALSE)
  print("done")
  colnames(df) = c("t","x","y","z")
  xfft = Im(fft(df$x))
  yfft = Im(fft(df$y))
  zfft = Im(fft(df$z))
  tsData = as.h2o(t(data.frame(xfft,yfft,zfft)))
```

Now we create a so called neural network auto encoder with three hidden layers. This neural network will read the data and try to reconstruct the very same data. But there is a catch. Note that the middle layer has only two dimensions. Therefore when reconstructing the data it has to flow through this bottleneck basically dramatically reducing the dimensions from three thousand to two and throwing away all noise and irrelevant information.

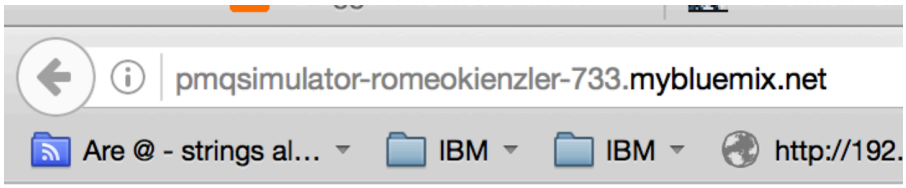
```
NN_model = h2o.deeplearning(
  x = 1:3,
  training_frame = tsData,
  hidden = c(3000, 2, 3000),
  epochs = 300,
  activation = 'Tanh',
  autoencoder = TRUE
)
```

Now we simply obtain an anomaly rate, which is the same as the reconstruction error given a neural net trained with a data set and applying new data to it. If the data is something the net already has seen the reconstruction error is low, otherwise it is high. Let's run the code

```
  anomalyrates = as.data.frame(h2o.anomaly(NN_model,tsData))
  print(sum(anomalyrates))
  flush.console()
  anomalyrates = as.data.frame(h2o.anomaly(NN_model,tsDataPr
  print(sum(anomalyrates))
  flush.console()
  tsDataPrev=tsData;
}
```

When seeing healthy data the error is small. Now let's switch back to the broken condition:

```
Warning message:
In .h2o.startModelJob(algo, params, h2oRestApiVersion): Dropping constant columns: [V1].
  |======================================================================| 100%
[1] 4.717214e-05
[1] 4.717214e-05
[1] "reading data..."
[1] "done"
  |======================================================================| 100%
```



Current state: healthy

Switch to healthy

Switch to broken

Now you can see that the reconstruction error on the already seen data is still very low but on the new, unseen data it is very height. And voila, this is our real-time IoT sensor anomaly detector.

```
Warning message:
In .h2o.startModelJob(algo, params, h2oRestApiVersion): Dropping constant columns: [V1].
  |======================================================================| 100%
[1] 0.001589089
[1] 27.06687
[1] "reading data..."
[1] "done"
  |======================================================================| 100%
```