SOUTHERN ILLINOIS UNIVERSITY
CARBONDALE

DEPARTMENT OF COMPUTER SCIENCE

Detection and Analysis of Hard Landing and Near Miss using Flight Data Monitoring System

Author
Avinash SORAB

Supervisor
Dr. Henry HEXMOOR

August 19, 2019

Detection and Analysis of Hard Landing and Near Miss using Flight Data Monitoring System

Avinash Sorab

August 19, 2019

Abstract

Hard landing and near miss are two important factors of flight safety in aviation management as miscalculation in any of them could lead to passengers discomfort or in some cases, a catastrophic disaster. This paper aims at detecting the hard landing and near miss and other important factors in one of the most used flights at SIU Aviation, Cessna 172 with G1000 and hence providing safety measures to avoid them in flight training for students. Flight Data Monitoring is a program which is organized in the school of aviation that intends to educate students and grade their performances with more sophisticated web based tool. This tool is developed to process, analyze, and visualize the flight data. The instructor uses this tool to fetch data from the flight and analyze performances of students. The tool uses some of the statistical methods and data analytics concepts for data visualization and thus helps improve the safety and quality of the FDM Program.

Acknowledgement

I cannot express enough thanks to Dr. Henry Hexmoor, Professor and Advisor for this project for his continued support and encouragement. I offer my sincere appreciation for the learning opportunities provided by the department of Computer Science.

My completion of this project could not have been accomplished without the support of the Aviation Department at SIU Carbondale. I'd also like to thank Kenneth Bro, Chief Flight Instructor at the Department of Aviation Management and Flight and Dr. Michael Robertson, Associate Professor of Aviation Management and Flight for their innovative ideas behind the project and guiding me throughout this since I was new to the topic of research. Your ideas and approaches were of great help for me to develop this software and my heartfelt gratitude goes to you.

Introduction

In any aircrafts, Hard landing is an unfavorable incident that occurs when the flight lands on the ground with a greater force than normal while landing. Hard landings are caused sometimes due to bad weather, mechanical problems or pilot error. There are several proposed methods that can be used to detect hard landing in flights. Hard landing can be measured by measuring the depression of the shock absorber of main landing gears. Hard landings can even be detected by measuring the pressure of hydraulic liquid of the shock absorber which is in direct relation with the vertical force that the plane experiences while landing. These devices are not integrated with all the flights so detection process can be tedious sometimes. There are other devices such as a piezoelectric meter that detects the change in the pressure of the fluid and converts it into electric signals.

Unfortunately most of the flights will not have those devices integrated to it. So detecting hard landing using the data collected during flying becomes important. Normal Acceleration is a parameter that can quantify the hard landing incident more accurately. Normal acceleration varies less as the flight descends and spikes up as soon as landing gears touches the ground. However this spike will not last for an entire second. It is because of this reason, sometimes it might miss to track this spike in our data, since the data is being recorded at a rate of 1Hz, i.e., data is captured at every 1 second interval. It is hence concluded that this method is not quite reliable. Even though normal acceleration is the only factor that shows a clear dependency on hard landing, there are other factors that contribute significantly towards hard landing if observed carefully.

Researches have shown that the factors like aircraft's pitch, descent rate and indicated air speeds also contribute to hard landing. In fact a steep decrement in the altitude followed by a pitch decrement spike at a sufficiently greater speed can be considered as hard landing. The aim of the paper is to quantify these parameter changes and hence be able to classify landings as hard or soft based on the landing data.

Near miss detection and analysis is yet another crucial part of the Flight Data Monitoring program. Near misses are identified by calculating the closeness of two flights on air. The geo co-ordinates of the flights are fetched when they are closer than the threshold specified. These co-ordinates are later used to plot the flight paths on a map. These plots help the flight instructor to see the whereabouts of the incident and help take appropriate action.

Background

Theory

AVMAF has a Flight Data Monitoring (FDM) program with the goal of improving the safety of its operations. The FDM program will assist the SRC with identifying potential hazards and accident precursors monitor trends in aircraft operations prior to experiencing an organizational accident, incident, or serious safety risk.

The FDM program is designed to continuously monitor and ensure that all flight operations are safe and in compliance with operating standards and regulations. An FDM program will assist the SRC in identifying and addressing operational deficiencies and trends that are not generally detectable with other procedures. Proper application of the FDM program will improve safety, evaluate and enhance training practices, revise operating procedures, and improve maintenance efficiency. The data acquired will never be used as the sole source of evidence for punitive action unless it is determined to involve the following circumstances:

- Criminal activity
- Substance or alcohol abuse
- Controlled substances
- Intentional falsification
- Intentional disregard for safety
- Intentional violation of the code of federal regulations

The objective of the FDM program is to facilitate the unrestricted flow of safety information. The FDM program will assist the SRC in doing the following activities.

- Collect operational flight data.
- Develop methods to analyze the collected flight data, such as triggered events and routine operational measurements.
- Compare established procedures and standards with the collected data to enhance safety in the following areas.
    - Flight training procedures.
    - Crew performance in all phases of flight.
    - Air traffic control procedures.
    - Aircraft maintenance.
- Perform trend analyses of FDM data to identify potential problem areas, evaluate corrective actions, and measure performance over time.

FDM Procedure

The FDM analyst will remove the SD cards of the G1000 equipped aircraft. This will occur during each semester as needed. The FDM Analyst will collect the data for analysis. It is the responsibility of the FDM analyst to keep all recorded flight data under the FDM program confidential. The FDM analyst will report hazards identified and trends as necessary to the Safety Officer / SMS Coordinator. The SRC will review FDM reports and make procedural or program changes to the Chief Flight Instructor and Accountable Executive if applicable.

The FDM program is modeled after the Flight Operational Quality Assurance (FOQA) program and follows the guidance found in AC 120-82. The FDM information can be included in audits and evaluations described in AC 120-82 to determine the causes of deficiencies and to suggest enhancements to operating practices. Title 14 of the Code of Federal Regulations (14 CFR) part 13 states the conditions under which information obtained from an approved voluntary FDM program will not be used in legal enforcement actions against an employee or student unless it is determined that there was an intentional disregard for safety, or intentional violation of the code of federal regulations.

Specification

The ultimate goal of this project is to build a cloud based web application and host it. The application should be able to provide these important functionalities.

Graphically represent various flight parameter values over time

Flight parameters can be simple parameters such as Air Speed, Pitch, Bank, Roll, Engine Temperatures, Fuel Quantity and so on. It can also be complex parameters that are derived using 2 or more simple parameters such as Hard Landing, Near Miss and so on. Analysing these parameters graphically is crucial for the end user/instructor in evaluating the performance of a pilot.

Generate Bar Graphs over years of data to analyse the trend.

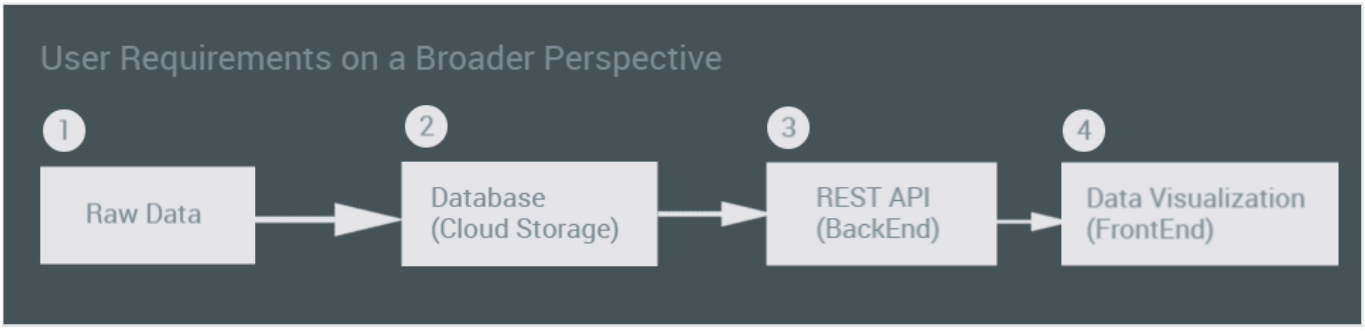
Trend analysis is a technique used in technical analysis that attempts to predict the future variable movements based on recently observed trend data. This technique can be used to see how the hard landing and near miss incidents have varied over the years and necessary actions can be taken using that.

Enable user to input more flight raw data in csv format

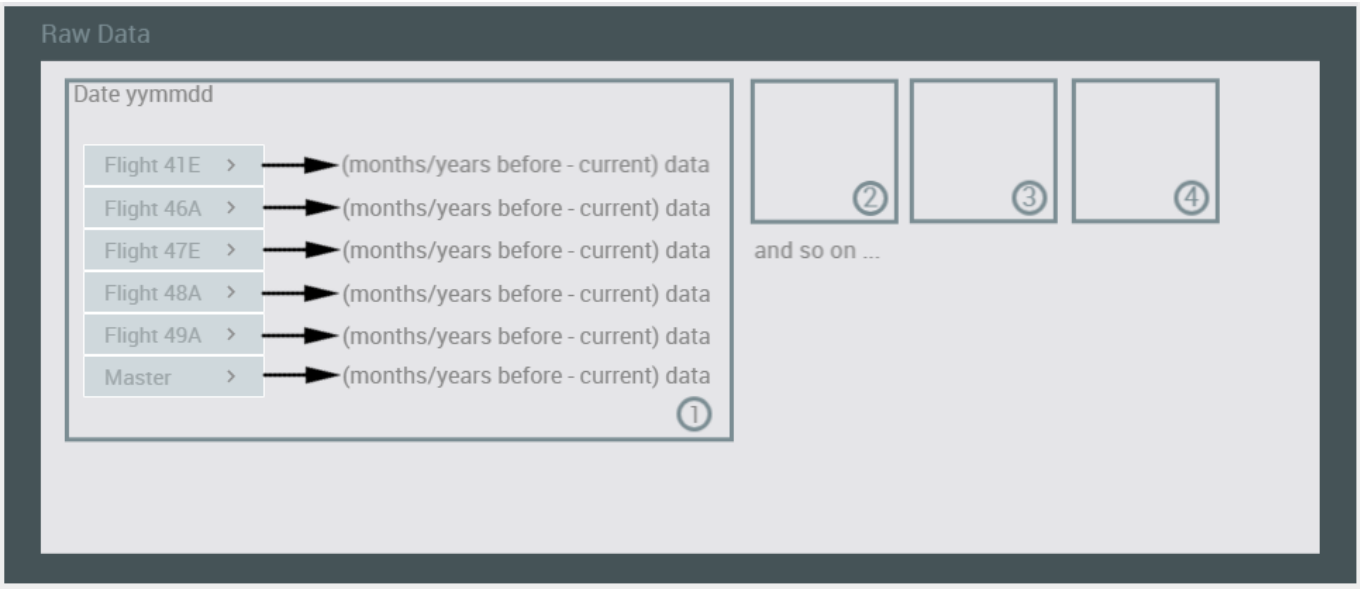
FDM Analyst collects the data from each flight that are stored in a microchip. These data are in raw form and needs pre processing. There are various stages for these pre processing before appropriate csv file is generated that is ready to be uploaded to the cloud database.

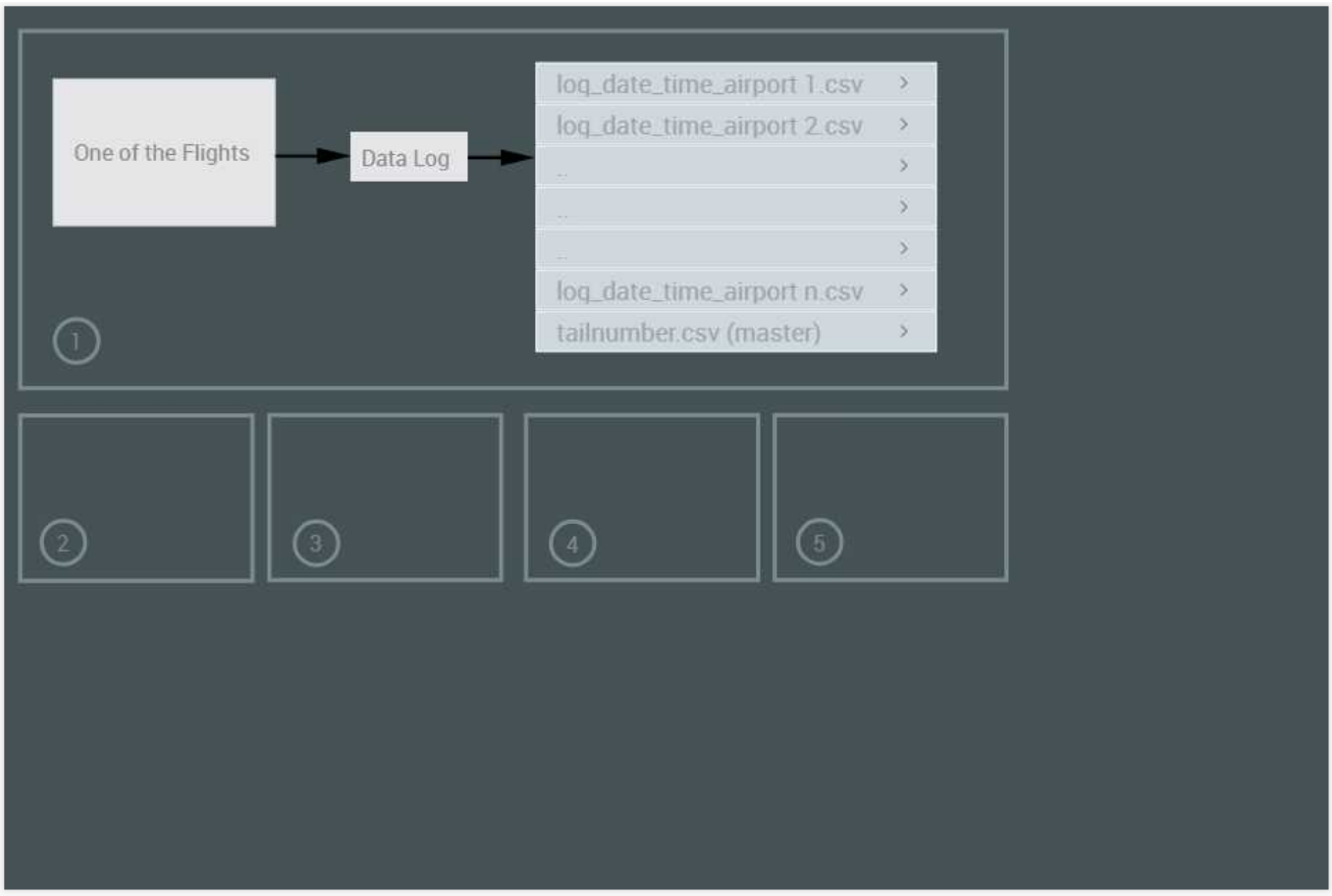
Design

User Data

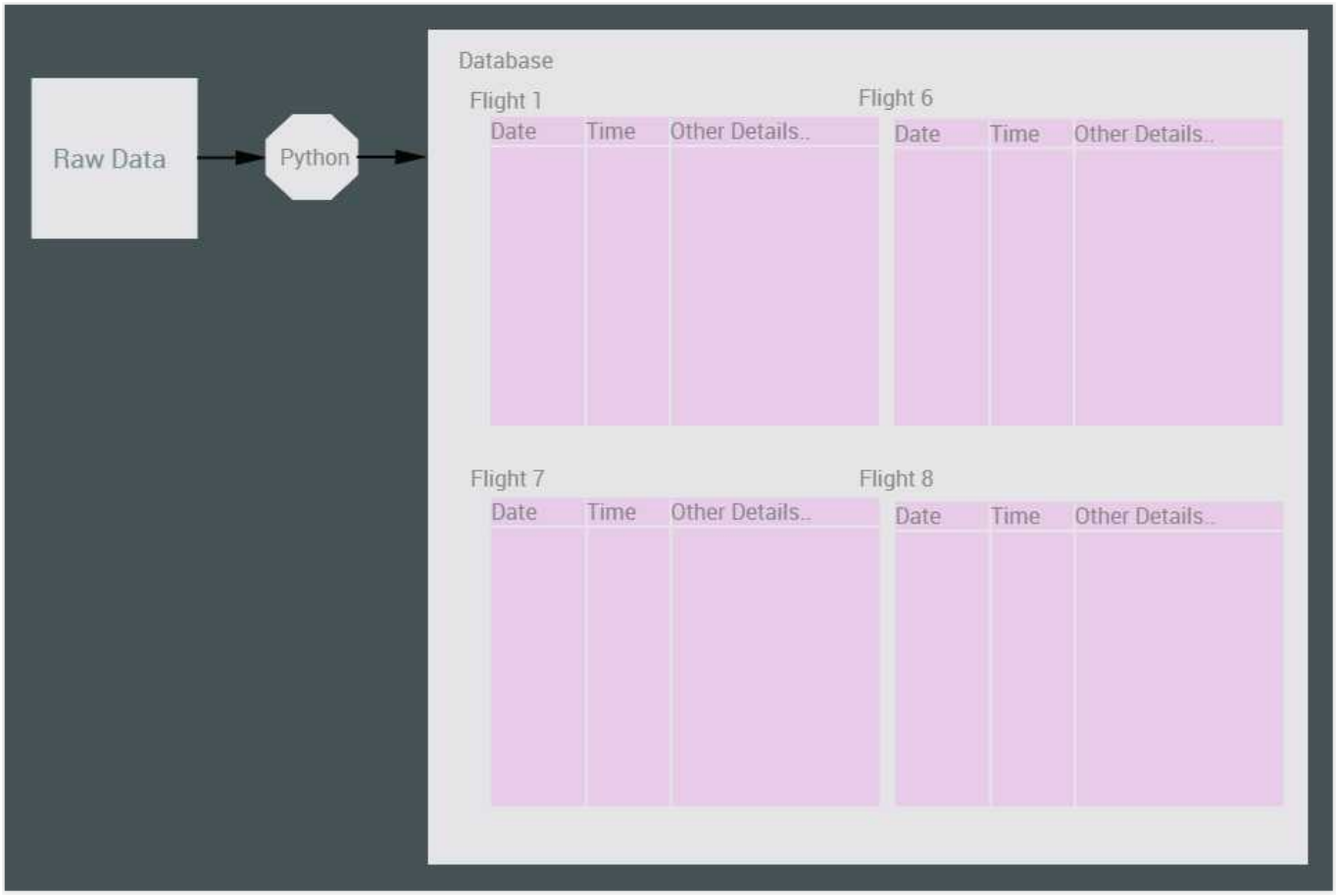


- The raw data will be processed and sent to a database after cleaning.
- We use REST API to query the data and display it in the frontend with good visualization





- Raw data is in .csv format (comma separated values)
- The csv files with name in the date format indicates that it has all data from the dates that are past to it. Ex: master_170330.csv will have the details of all flights that flew on dates till 30th March 2017.



Visualization

The visualization mostly consists of the following

- Displaying the historical data in the form of charts
- Performance Analysis for each students
- Deviation of Student's Performance Graph from mean value
- Flight status visualization and repair alerts if any

Technologies

Since this was a web-based application, individual technologies are chosen at different levels of architecture.

Database

BigQuery - Google's cloud storage service that enables us to store and query big data efficiently than the other relational databases.

BackEnd

NodeJS and ExpressJS - The powerful, fast tool that integrates well with the database that we've chosen and easily communicable with front end and the database.

FrontEnd

Angular 7 - Google's popular JavaScript framework for building interactive web application based on Model View Control Architecture.

HTML5 and CSS3 - Markup language and stylesheet technology used to design and develop our web application front end.

Visualization

amCharts 4 - Amcharts is a JavaScript/HTML5 charts and maps data-visualization library for websites and applications. This framework integrates well with Angular on the front end.

OpenLayers 5.3 - OpenLayers is an open-source JavaScript library for displaying map data in web browsers as slippy maps. It provides an API for building rich web-based geographic applications similar to Google Maps.

Implementation

Front End Architecture

A very powerful Model View Controller Architecture based framework, Angular is used to develop single page applications. Some of angular's features such as template binding, property and event binding, using services to transfer data between various components, observables and subscribers and many more have been very helpful in designing the front end layout for this project.

Folder Structure

- src
  - app
    - app module
    - app component
    - app routing module
  - assets
    - images
  - config
    - config json file
  - environments
    - environment file
  - shared
    - components
      - analysis
      - csvupload
      - dashboard
      - login
      - flights
      - bottom bar (footer)
      - side bar (navbar)
      - top bar (header)
    - models
      - classes

        - enums
    - services
        - ajax
        - authentication
        - flightData
        - authguard
    - templates
        - formTemplate
- vendor
    - css and js dependencies

Features

Template Binding

Since all the analysis more or less have the same form elements we could use a common form template instead of having one template for each component.

> Template Code

```
<ng-container *ngTemplateOutlet="formTemplate; context: {$implicit: analysis}">

</ng-container>

<ng-template #formTemplate>

<div>
    Your template goes here
    .
    .
    {{analysis.header}}
    .
    .
</div>
```

context: {$implicit: analysis}

is used to bind the data to to injected to the template from the component so that "analysis" contains the json data passed from the component that can be used in the template.

> Component Code

```
@Component({
  selector: 'ias-component',
  templateUrl: './../../url_where_the_template_exists.html',
  styleUrls: ['./ias.css']
})

// Data to be passed to the template
```

```
public analysis = {
    header    : 'Overspeed Analysis',
    chart     : 'Indicated Air Speed Max'
};
```

Using services as a data mediator between components

Observables is new standard of managing asynchronous data. Observables are objects that the consumers can subscribe to in order to receive the latest update of data from it.

They are different from promises as in promises only return the object once whereas observables can return the update multiple times and also observables can be unsubscribed whereas promises cannot.

We use observables in services to send data to various components.

We create an rxjs Subject that allows to be multicasted to many observers since there are so many components observing it.

BehaviorSubject is a special type of rxjs Subject that needs an initial value to start with.

```
private loadDataChartSource = new Subject<any>();
sourceCalled$ = this.loadDataChartSource.asObservable();
```

sourceCalled$ is an observable of type Subject.

```
getData(formData) {
    let url =
`${config.protocol}://${config.host}/calculations/${formData.parameter}/${formData
.dateFrom}/${formData.dateTo}/`;
    this.http.get(url, this.httpOptions)
      .subscribe((data) => {
        // calling next on the observer so that the subscribers receives the
latest values PUSH
        this.loadDataChartSource.next();
    });
```

After receiving data from the ajax call, in the callback, (subscribe function) next is called on observable that makes the observable push the latest data to the subscribers.

```
this.ajax.sourceCalled$.subscribe(
    () => {
        this.loadChartsData(this.ajax.dataToUpdate);
    }
)
```

The components receiving the data should subscribe to loadDataChartSource as above to get updated data, the callback inside the subscribe function gets called whenever the data is updated in the sourceCalled$ observable

Authenticating a user to login

```
private currentUserSubject: BehaviorSubject<any>;
public currentUser: Observable<any>;

this.currentUserSubject = new BehaviorSubject<any>
(JSON.parse(getItem('currentUser')));
this.currentUser = this.currentUserSubject.asObservable();
```

Store the current user if any in a temporary variable

```
login(username, password) {
    return this.http.post<any>(`${config.apiUrl}/users/authenticate`, { username,
password })
        .pipe(map(user => {
            // store user details and jwt token in local storage to keep user
logged in between page refreshes
            localStorage.setItem('currentUser', JSON.stringify(user));
            this.currentUserSubject.next(user);
            return user;
        }));
}
```

When you're logged in successfully, store the username and password in the local storage item

```
logout() {
    // remove user from local storage and set current user to null
    localStorage.removeItem('currentUser');
    this.currentUserSubject.next(null);
}
```

When logging out, remove the current user from the localStorage and update the subject

Using router guards to prevent trespassing login page

```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    const currentUser = this.authenticationService.currentUserValue;
    if (currentUser) {
        // authorised so return true
```

```
            return true;
        }

        // not logged in so redirect to login page with the return url
        this.router.navigate(['/login'], { queryParams: { returnUrl: state.url }});
        return false;
    }
```

The AuthGuard component implements CanActivate interface to be a guard deciding if a route can be activated. If all guards return true, navigation will continue. If any guard returns false, navigation will be cancelled. If any guard returns a UrlTree, current navigation will be cancelled and a new navigation will be kicked off to the UrlTree returned from the guard.

Back End Architecture

Back End is completely done using Node and Express.

NodeJS is an cross-platform javascript run time environment that executes javascript code outside the browser.

ExpressJS is a framework written on top of NodeJS that handles most of the complex node js functionalities in a simpler fashion.

We use express to build our server and design api endpoints to communicate between our front end logic and database logic.

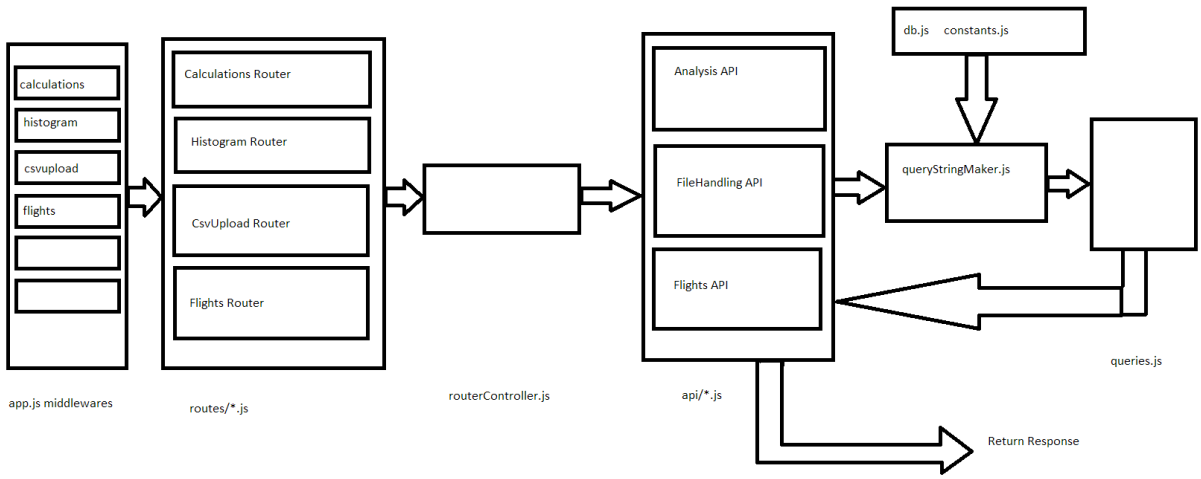
Express middleware are functions that execute during the lifecycle of a request to the Express server. Each middleware has access to the HTTP request and response for each route (or path) it's attached to. This chaining of middleware allows you to compartmentalize your code and create reusable middleware.

Folder Structure

- api
    - analysis api (for analysis components)
    - file handling api (for csv upload component)
    - app routing module (for flight compoents)
- bin
    - file to start server
- config
    - config credentials file json
- controllers
    - queryStringMaker
    - routerController
- models
    - constants
    - db
    - queries
- routes
    - calculations

    - csvupload
    - flights
    - histogram
- uploads

Pictorial Representation of Control Flow



Features of some Middlewares used

CORS

The cors middleware is used to allow the server to accept cross origin resource sharing.

```
app.use(cors());
```

body-parser

The body-parser middleware is used to allow form data to be parsed under the body of the request object sent, that way it's easier for the server to access what's been sent in the form.

We use this feature extensively since we are sending huge files through http protocol and also some user input paramters along with it.

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

cookie-parser

The cookie parser is used to parse cookie header and populate cookie object with names.

```
app.use(cookieParser());
```

multer

The multer middleware is used before making a post request by csvupload router component. We need this middleware to make sure that the huge stream of data sent by the front-end is received and converted to csv file format and stored at a place before moving on to processing.

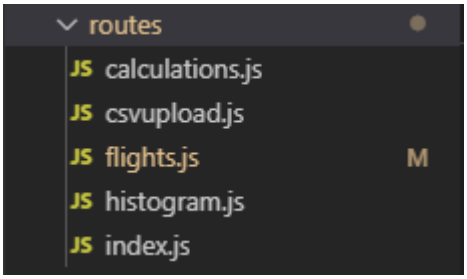
```
var storage = multer.diskStorage({
    destination: function (req, file, cb) {
        cb(null, './server/uploads/')
    },
    filename: function (req, file, cb) {
        cb(null, file.originalname);
    }
})

router.post('/', multer({dest: "./server/uploads/", storage:
storage}).array("uploads[]", 12), routerController.csvupload);
```

Designing API Endpoints

A single url `http://localhost:3000/flights` is used to handle all the create, read, update and delete functionality of the flights (RESTful API).

ExpressJS handles all of its routes in a separate files as below



This means we have

- flights
- csvupload
- calculations
- histogram

are the routes enabled.

Flight View, Add, Update, Delete API

> Front End

```
private httpOptions: any = {
    headers: new HttpHeaders({
        'Content-Type':  'application/json'
    }),
    body : {}
};

// GET Request for viewing flights
this.http.get(url, this.httpOptions)

// POST Request for adding a flight
this.http.post(url, flight, this.httpOptions)

// PUT Request for updating the flight
this.http.put(url, flight, this.httpOptions)

// DELETE Request for deleting the flight
this.httpOptions.body.id = id;
this.http.delete(url, this.httpOptions)
```

> Back End

```
// GET Request for viewing flights
router.post('/', routerController.flights.add);

// POST Request for adding a flight
router.get('/', routerController.flights.view);

// PUT Request for updating the flight
router.put('/', routerController.flights.update);

// DELETE Request for deleting the flight
router.delete('/', routerController.flights.remove);
```

Analysis API

The skeleton URL of any analysis would look like below

Note: ? denotes optional

`http://localhost:3000/calculations/parameter/dateFrom}/dateTo/?flightNumber/?visualize`

Examples:

- http://localhost:3000/calculations/overspeed/2019-01-01/2019-02-15/5/d
- http://localhost:3000/calculations/hardlanding/2018-01-01/2018-12-31/all/m
- http://localhost:3000/calculations/nearmiss/2019-01-01/2019-12-31/3-4-5-1-6-7-8-9

All analysis except near miss needs type of visualization and flight number along with from date and to date.

Near miss alone needs a list of flight number separated by hypens so that it can be decoded into an array at the backend. Visualization type is not needed for near miss.

> Front End

```
// form a hyphenated flight string if there are multiple flights

formData.flights.length < 2 ? url+=`${formData.flights[0]}` :
url+=`${formData.flights.join("-")}`;

// for near miss scenario

formData.visualize !== undefined ? url+=`/${formData.visualize}` : '';
this.http.get(url)
```

> Back End

```
// for all the parameters except nearmiss
router.get('/:attr/:from/:to/:flight/:visualize', routerController.calculate);

// for near miss
router.get('/:attr/:from/:to/:flights', routerController.calculate);
```

CSV Upload API

> Front End

```
let formData = new FormData();
for (var i = 0; i < this.filesToUpload.length; i++) {

    // File Name
    formData.append("uploads[]", this.filesToUpload[i],
this.filesToUpload[i].name);

    // Flight Number
    formData.append("flight", this.selectedFlight);

    // Will the content be written to master table or not flag
    formData.append("writeToMaster", this.writeToMaster.toString());
}
this.http.post(URL, formData)
```

> Back End

```
var multer = require("multer");
```

```
var storage = multer.diskStorage({
    destination: function (req, file, cb) {
        cb(null, './server/uploads/')
    },
    filename: function (req, file, cb) {
        cb(null, file.originalname);
    }
})

router.post('/', multer({dest: "./server/uploads/", storage:
storage}).array("uploads[]", 12), routerController.csvupload);
```

Using the multer module, file upload module is built to handle 12 files even though only one file is being uploaded at once.

Cloud Database - BigQuery

BigQuery is a RESTful web service that enables interactive analysis of massive datasets working in conjunction with Google Cloud Storage.

We have used bigquery to store as massive as 10 gigs worth csv data on the cloud in the database.

BigQuery provides client libraries in various languages out of which Node is one of them. We use node.js client library to talk to bigquery.

```
await storageClient.bucket(bucketName).upload(fileNameAbs, {
    gzip: true
}).then(function(response){
    console.log(`${fileName} uploaded to ${bucketName}.`);
    that.push(res, fileName, flightID, writeToMaster);
    fs.unlinkSync(path);
});
```

NodeJS code to push the file to google cloud storage which can later be used to write to database.

```
const tableId = db.flights(flightID);
// Configure the load job. For full list of options, see:
//
https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load
const metadata = {
    sourceFormat: 'CSV',
    skipLeadingRows: 0,
    schema: {
        fields: [
        { name: 'Local_Date', type: 'STRING' },
        { name: 'Local_Time', type: 'STRING' },
            .
            .
            .
```

```
            .
            .
        { name: 'Magnetic_Variation_From_True_North', type: 'FLOAT64' },
        { name: 'Flight_Tail_Number', type: 'INT64' }
        ]
    },
    location: 'US',
};

// Load data from a Google Cloud Storage file into the table
let [job] = await bigqueryClient
    .dataset(datasetId)
    .table(tableId)
    .load(storageClient.bucket(bucketName).file(fileName), metadata);

// load() waits for the job to finish
console.log(`Job ${job.id} completed: Write to flight table`);
```

Get the table ID, dataset ID where you've to write the csv contents in the database. Also get the schema of the table to match with the schema of the csv file.

BigQuery Table Schema

Master Table and Individual Flight Table

Name	Type	Mode
Local_Date	STRING	
Local_Time	STRING	
Universal_Time_Offset	STRING	
Waypoint_In_GPS	STRING	
Latitude	FLOAT64	
Longitude	FLOAT64	
Barometric_Altimeter	FLOAT64	
Altimeter_Setting	FLOAT64	
GPS_Altitude	FLOAT64	
Outside_Air_Temperature	FLOAT64	
Indicated_Air_Speed	FLOAT64	
Ground_Speed	FLOAT64	
Vertical_Speed	FLOAT64	
Pitch	FLOAT64	

Name	Type	Mode
Roll	FLOAT64	
Lateral_Accelaration	FLOAT64	
Normal_Accelaration	FLOAT64	
Heading_Relative_To_Magnetic_North	FLOAT64	
Track	FLOAT64	
Voltage_1	FLOAT64	
Voltage_2	FLOAT64	
Amperage_1	FLOAT64	
Amperage_2	FLOAT64	
Fuel_Quantity_Left_Tank	FLOAT64	
Fuel_Quantity_Right_Tank	FLOAT64	
Fuel_Consumption	FLOAT64	
Oil_Temperature	FLOAT64	
Oil_Pressure	FLOAT64	
Engine_RPM	FLOAT64	
Cylinder_1_Heat_Temperature	FLOAT64	
Cylinder_2_Heat_Temperature	FLOAT64	
Cylinder_3_Heat_Temperature	FLOAT64	
Cylinder_4_Heat_Temperature	FLOAT64	
Cylinder_1_Exhaust_Gas_Temperature	FLOAT64	
Cylinder_2_Exhaust_Gas_Temperature	FLOAT64	
Cylinder_3_Exhaust_Gas_Temperature	FLOAT64	
Cylinder_4_Exhaust_Gas_Temperature	FLOAT64	
GPS_Altitude_Duplicate	FLOAT64	
Navigation_Needle_Indicator_Source	FLOAT64	
Horizontal_Situation_Indicator_Source	STRING	
GPS_Course	FLOAT64	
Frequency_Selection_For_Navigation_Radio_1	FLOAT64	
Frequency_Selection_For_Navigation_Radio_2	FLOAT64	
Communication_Frequency_For_Radio_1	FLOAT64	

Name	Type	Mode
Communication_Frequency_For_Radio_2	FLOAT64	
Horizontal_Displacement_For_Navigational_Needle	FLOAT64	NULLABLE
Vertical_Displacement_For_Navigational_Needle	FLOAT64	NULLABLE
WindSpeed	FLOAT64	NULLABLE
Direction_Wind_Coming_From	FLOAT64	NULLABLE
Active_Waypoint_Distance	FLOAT64	NULLABLE
Magnetic_Bearing_To_Active_Waypoint	FLOAT64	NULLABLE
Magnetic_Variation_From_True_North	FLOAT64	
Flight_Tail_Number	INT64	

Flights Table

Name	Type	Mode
ID	INTEGER	NULLABLE
Name	STRING	NULLABLE
Tail_Number	STRING	NULLABLE

Uploading csv file to cloud

- csvupload.js manages to receive the csv file sent over http protocol using an npm tool called multer.
- routerController.js appropriately redirects the control to fileHandlingAPI file sending relavant information.
- fileHandlingAPI.js does 4 important functionality
    - **upload**: uploads the file received to the cloud storage by gzipping it and deletes the file on the server later.
    - **push**: pushes the contents of the csv from the file on cloud to the database table. Formats are defined as metadata. Sends back the response to the front end.

Design and Implementation of Hard Landing

Unlike other parameters Hard Landing and Near Miss calculations are not pretty straight forward as they have more than 2 criteria that should be satisfied in order to be a hard landing/ near miss event.

A flight is said to have a hard landing if during the landing phase of the flight, the flight has the descent speed that's greater than 10 feet per second, the air speed of the flight is above the taxi limit and during the landing if the pitch changes by 6 degrees within a second.

> Query:

```
SELECT local_date, local_time, flight_tail_number, gps_altitude, air_speed, pitch
FROM ${db.dataset_id}.${flight}
WHERE local_date BETWEEN from_date AND to_date
AND gps_altitude BETWEEN ${constants.hardlanding.gpsAltitudeLow}
AND ${constants.hardlanding.gpsAltitudeHigh}
AND air_speed > ${constants.hardlanding.flightAirSpeed}
ORDER BY flight_tail_number, local_date, local_time
```

Basically you fetch all the landing data using this query and run some tests on it later to confirm if the landing is hard or not

```
for(var i=1; i<response.length; i++) {
    let pitchStartTime = new Date(response[i-1]["Local_Date"]+" "+response[i-1]
["Local_Time"]+"Z").valueOf()/1000;
    let pitchEndTime = new Date(response[i]["Local_Date"]+" "+response[i]
["Local_Time"]+"Z").valueOf()/1000;
    if(response[i-1]["Pitch"]-response[i]["Pitch"] > 6 && (pitchEndTime -
pitchStartTime == 1)) {
        for(let j = 0; j < constants.hardlanding.altPitchEventDiff; j++) {
            let altStartTime = new Date(response[i-j-1]["Local_Date"]+"
"+response[i-j-1]["Local_Time"]+"Z").valueOf()/1000;
            let altEndTime = new Date(response[i-j]["Local_Date"]+" "+response[i-
j]["Local_Time"]+"Z").valueOf()/1000;
            if(response[i-j]["GPS_Altitude"]-response[i-j-1]["GPS_Altitude"] < -10
&& (altEndTime - altStartTime == 1)) {
```

Here, we're scanning through the pitch records to see if there's any consecutive pitch with the difference of 6 degrees.

If we found that, we would scan atleast 15 records back in time to see if there's any altitude difference of 10 feet in the records.

If that matches as well, then we can be pretty sure that there is a hard landing event occured then and use the data to send it to the front end to plot a clean hard landing graph on charts.

The same process is carried out for all the different dates.

Finally we'll have an array of object that represent individual hard landing events. We'll send that array as the response.

Design and Implementation of Near Miss

Near Miss is also one of the important events to be analysed and detected.

Near Miss is said to have happened between 2 flights if they are on air and are as close as 500 ft apart at sufficiently high speeds.

> Query:

```
SELECT first_flight.Local_Date AS Local_Date, first_flight.Local_Time AS
Local_Time, first_flight.Universal_Time_Offset AS UTC1, first_flight.Latitude AS
X1, first_flight.Longitude AS Y1, first_flight.GPS_Altitude AS ALT1,
first_flight.Flight_Tail_Number AS FTN1,
second_flight.Latitude AS X2, second_flight.Longitude AS Y2,
second_flight.GPS_Altitude AS ALT2, second_flight.Flight_Tail_Number AS FTN2
FROM aviation_dataset.flight_master first_flight
JOIN
aviation_dataset.flight_master second_flight
ON first_flight.Flight_Tail_Number <> second_flight.Flight_Tail_Number
AND first_flight.Local_Date = second_flight.Local_Date
AND DATETIME(CAST(CONCAT(first_flight.Local_Date, " ",first_flight.Local_Time) as
TIMESTAMP), first_flight.Universal_Time_Offset) =
DATETIME(CAST(CONCAT(second_flight.Local_Date, " ", second_flight.Local_Time)
as TIMESTAMP), second_flight.Universal_Time_Offset)
WHERE first_flight.Flight_Tail_Number < second_flight.Flight_Tail_Number
AND first_flight.Flight_Tail_Number in (3,4,5,1,6,7,8,9) AND
second_flight.Flight_Tail_Number in (3,4,5,1,6,7,8,9)
AND first_flight.Local_Date BETWEEN "2019-01-01" AND "2019-12-31"
AND first_flight.Indicated_Air_Speed > 50
AND second_flight.Indicated_Air_Speed > 50
AND ABS(first_flight.Latitude - second_flight.Latitude) < 0.0015
AND ABS(first_flight.Longitude - second_flight.Longitude) < 0.002
AND ABS(first_flight.GPS_Altitude - second_flight.GPS_Altitude) < 500
AND first_flight.GPS_Altitude > 600
AND second_flight.GPS_Altitude > 600
ORDER BY FTN1, FTN2, DATETIME(CAST(CONCAT(first_flight.Local_Date, " ",
first_flight.Local_Time) as TIMESTAMP), first_flight.Universal_Time_Offset)
```

Basically we need to perform join operation on the results we fetch, as in we need to join the different instances of same table but we need to make sure that the flight numbers are always different.

- Date and Time, considering the UTC shift has to be equal,
- Join results will create duplicate entries so only one has to be considered (ex: flight 1 and flight 6 in 1 entry, flight 6 and flight 1 with same records as another entry)
- It has to be between the local date specified
- It has to match all the other regulations such as latitude difference, longitude difference, altitude difference, true altitude of the flight and air speeds of both the aircraft.

The result is still raw and needs some processing

```
if(startTimeFirstFlight == nextTickFirstFlight && startTimeSecondFlight ==
nextTickSecondFlight
                && nextTick - startTime <
constants.nearmiss.twoDistinctIncidentsInterval) {
```

After sorting the raw data based on flight number date and time, it's now the time to gather all the data and separate them, group them into distinct nearmiss events. As in if the next record has a time difference of more

than 120 seconds (2 minutes) it will be grouped under a separate near miss event. Also an event to be considered as near miss, it needs to atleast have 5 or more timestamps, in other words it has to last for atleast 6 seconds within the constraints interval, otherwise it is discarded.

Finally we'll have an array of object that represent individual near miss events. We'll send that array as the response.

Result and Evaluation

Hard Landing Results

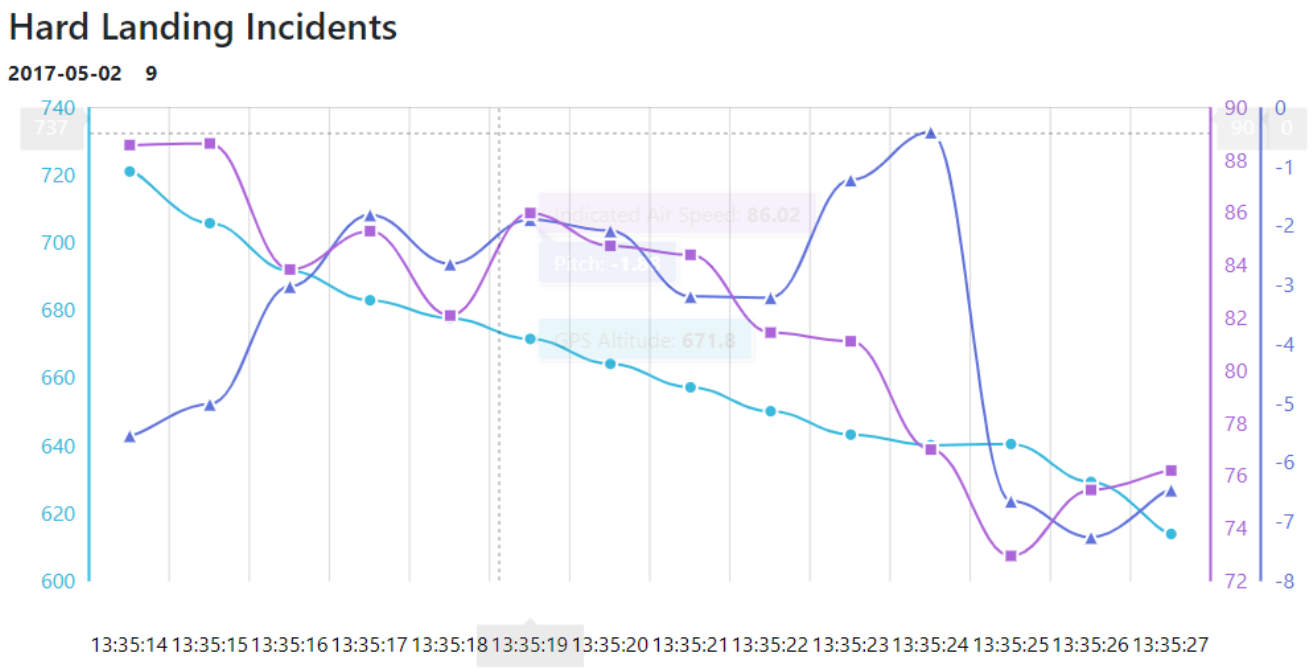
> Ex 1

Input Parameters

- Date From : 2017-01-01
- Date To : 2017-12-31
- Flight : all
- Analysis : hardlanding

Output Paramters

- Response Time : 43.54 seconds
- Hard Landings : 8



> Ex 2

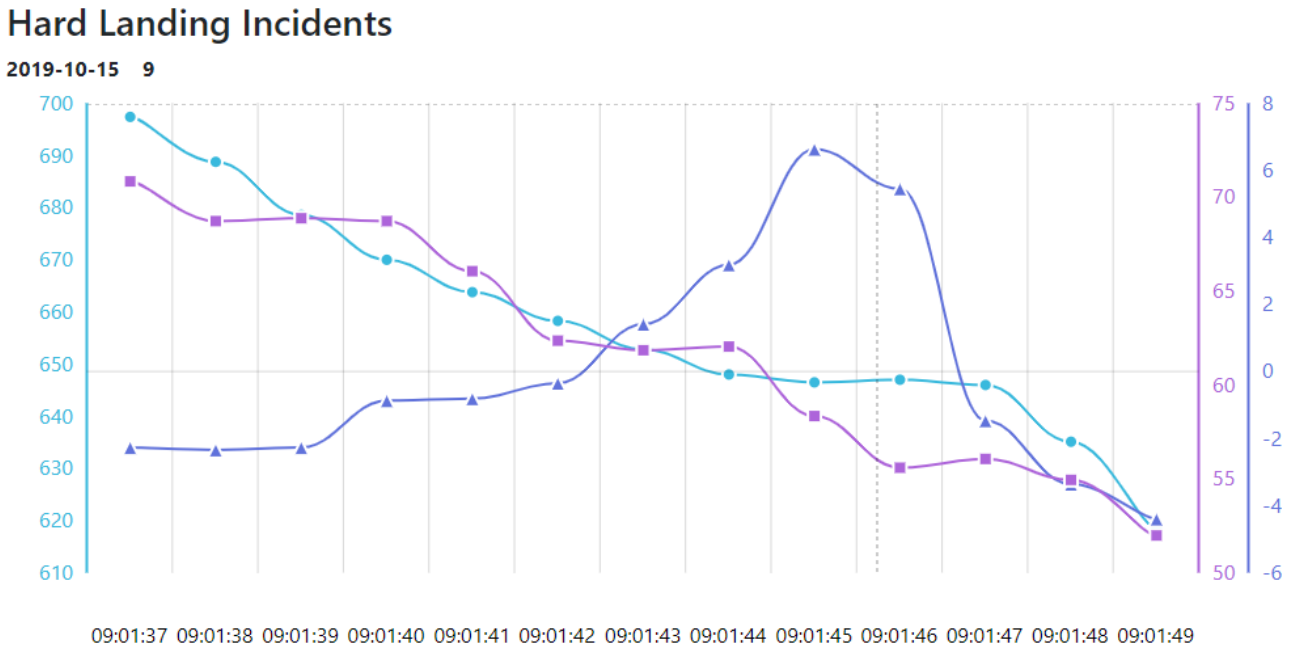
Input Parameters

- Date From : 2019-01-01
- Date To : 2019-12-31
- Flight : all

- Analysis : hardlanding

Output Paramters

- Response Time : 31.07 seconds
- Hard Landings : 6



Near Miss Results

> Ex 1

Input Parameters

- Date From : 2018-01-01
- Date To : 2019-12-31
- Flight : all
- Analysis : nearmiss

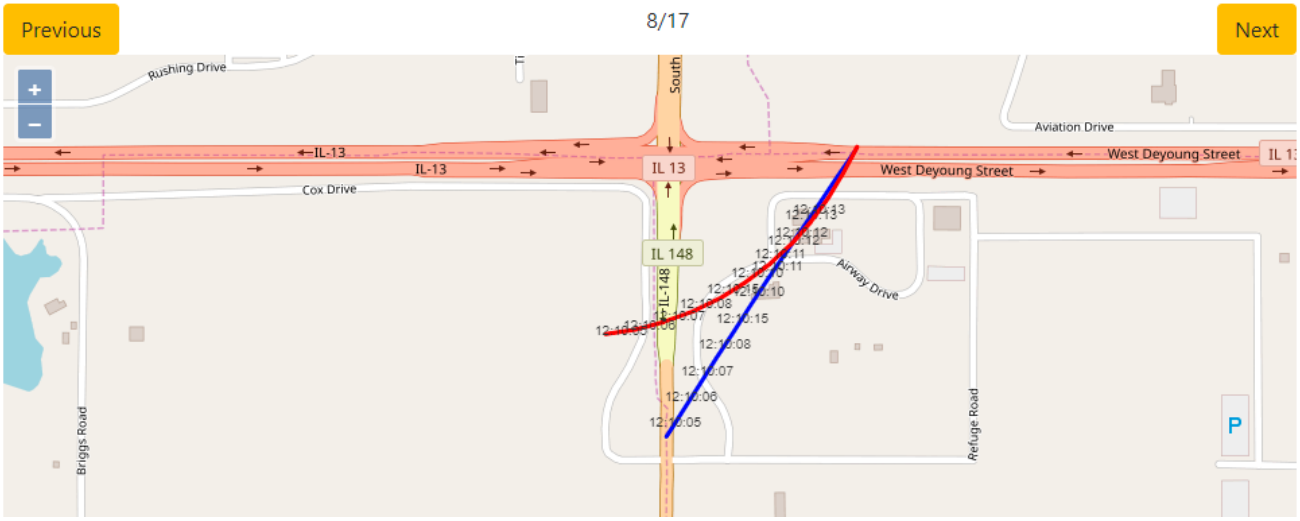
Output Paramters

- Response Time : 5.2 seconds
- Near Misses : 17

Near Miss Map

Flight 1 : 541E Flight 2 : 549A

Date : 2018-03-30

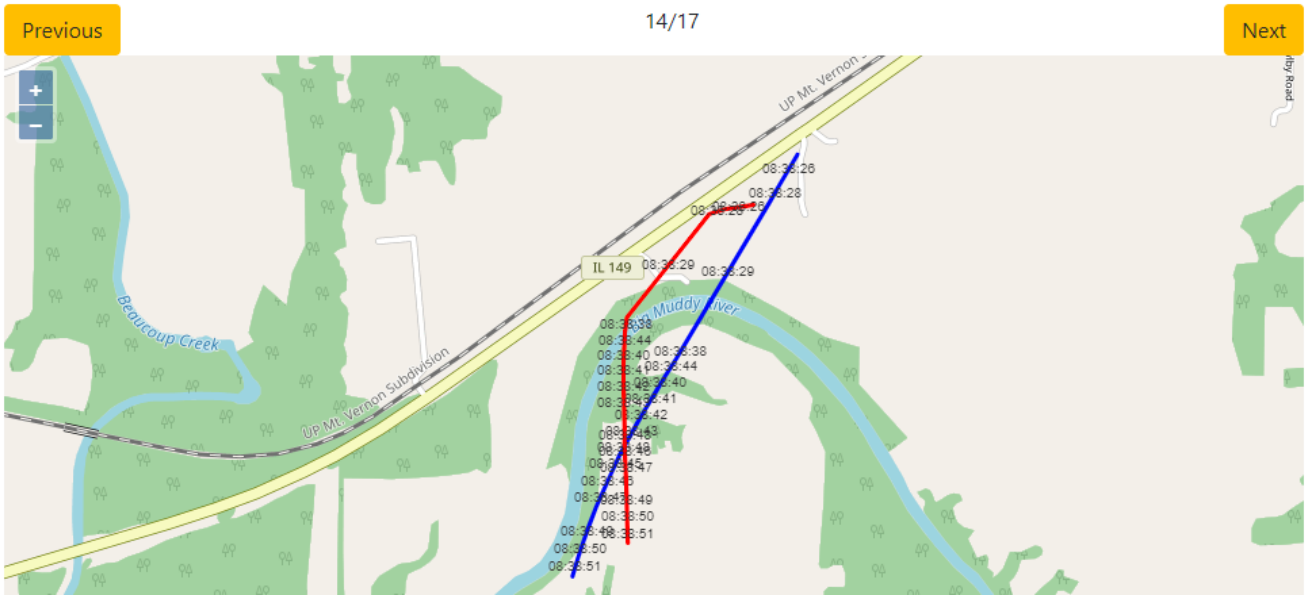


As close as 324 feet

Near Miss Map

Flight 1 : 546A Flight 2 : 548A

Date : 2019-09-18

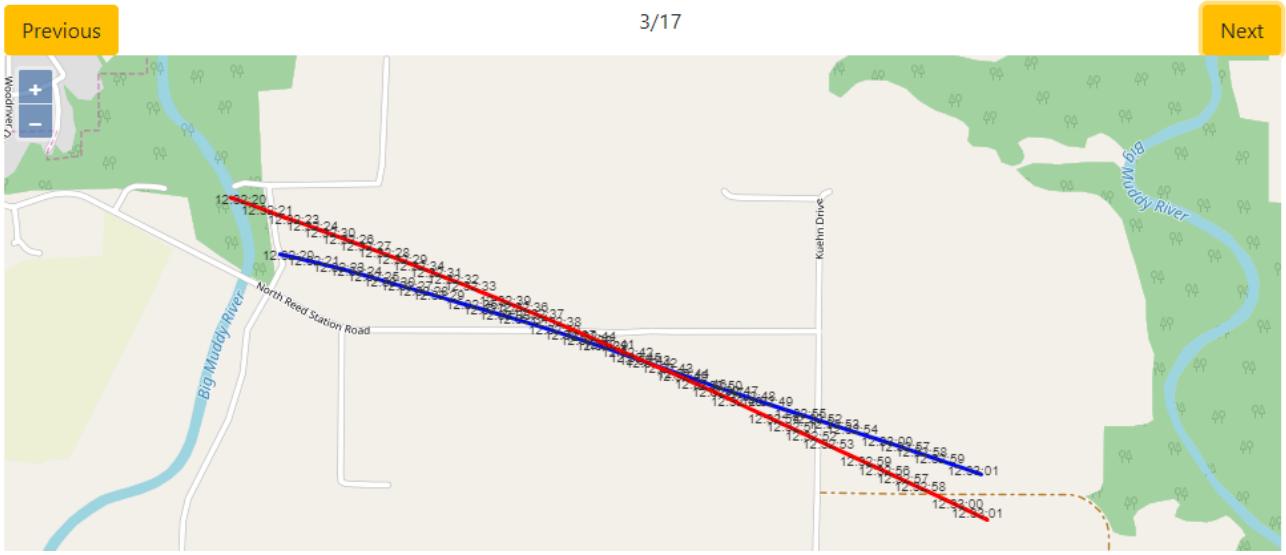


As close as 271 feet

Near Miss Map

Flight 1 : 541E Flight 2 : 548A

Date : 2019-02-27



As close as 432 feet

Indicated Air Speed Max Results

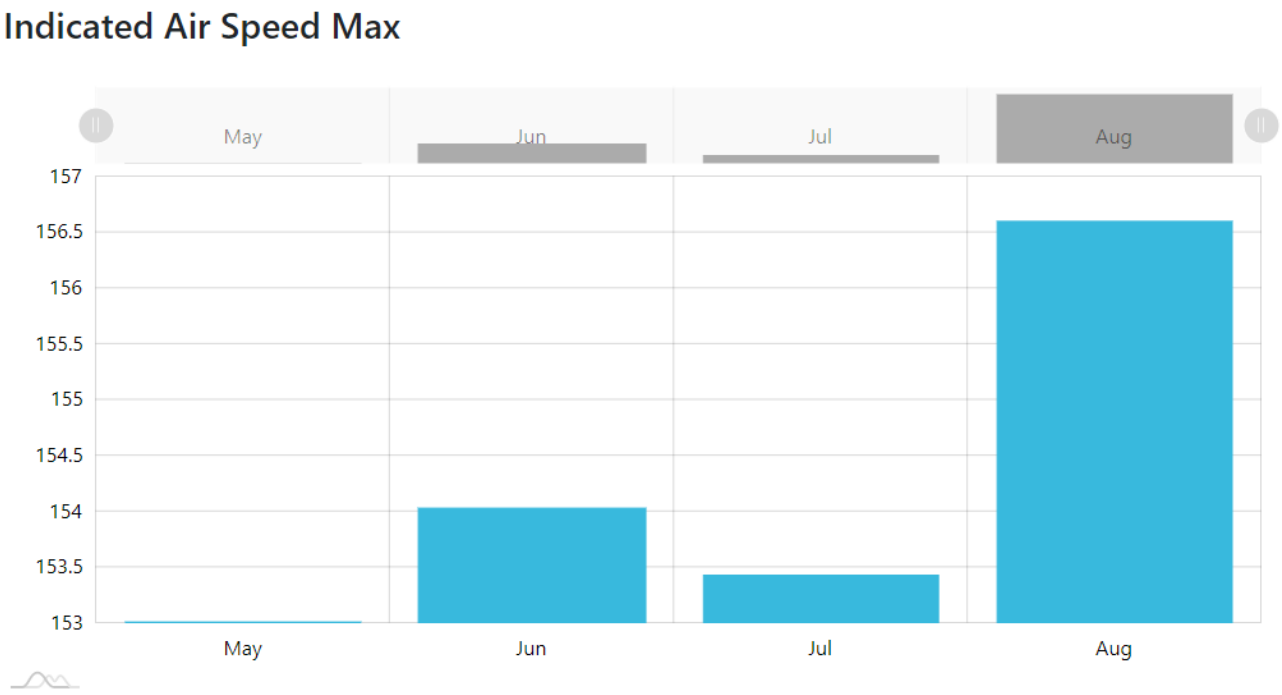
> Ex 1

Input Parameters

- Date From : 2019-05-01
- Date To : 2019-08-22
- Flight : all
- Analysis : overspeed

Output Paramters

- Response Time : 3.1 seconds



High Engine Cylinder Heat Temperature Results

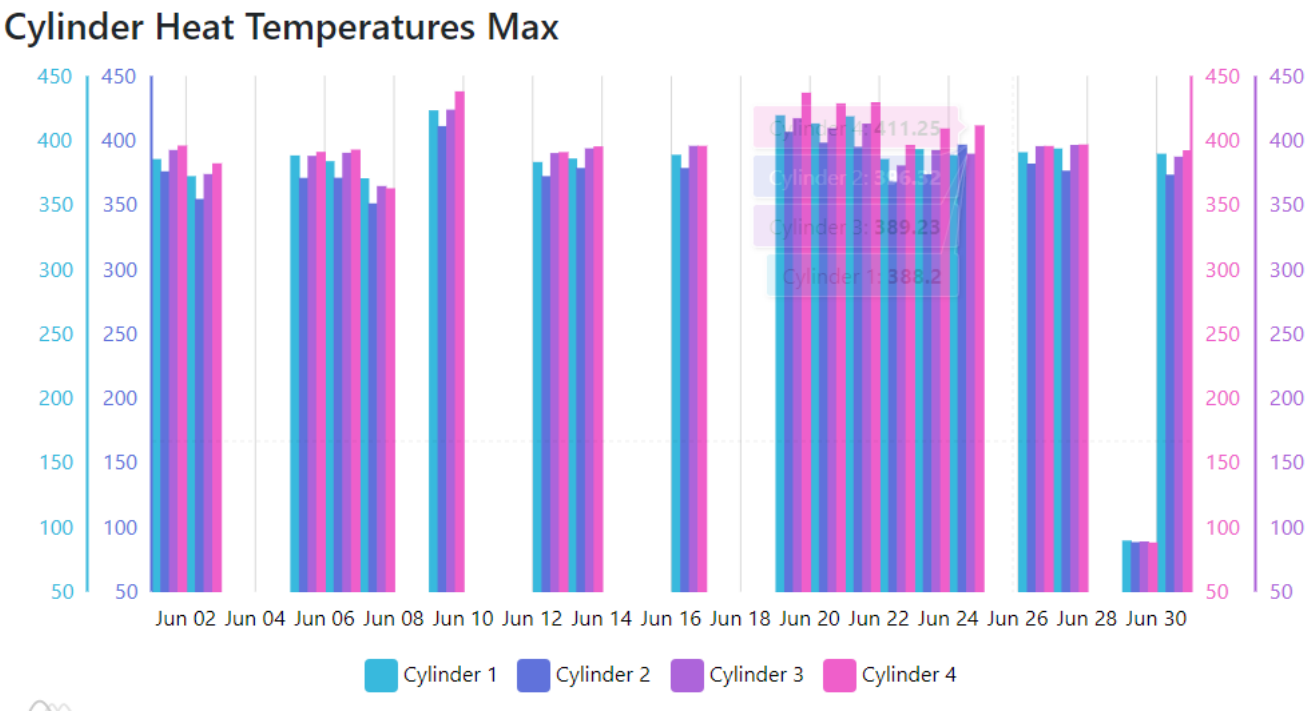
> Ex 1

Input Parameters

- Date From : 2019-06-01
- Date To : 2019-06-30
- Flight : 541E
- Analysis : high cht

Output Paramters

- Response Time : 2.2 seconds



Future Work

- **Find out a hard landing pattern.**
- **Better algorithm for hardlanding and nearmiss**
- **Increase the percentages of successful hardlanding events.**
- **3D View of Near Miss using KML Maps.**
- **Undo CSV upload operation to database.**
- **Login Registration with Admin Approval**

Conclusion

The flight data monitoring system has definitely made parameter analyses easier compared to prior method using Microsoft Excel to inspect records and plot charts.

The charts and maps give better insights on the incidents and are interactive and verbose.

The aim of this project is to grade students better and take appropriate measures so that the training can be more effective and in future we would see fewer hard landings or near miss incidents.

Glossary

Table of Abbreviations

Abbreviations	Full Forms
AJAX	Asynchronous Java and XML
AVMAF	Aviation Management and Flights
CFR	Code of Federal Regulations
CHT	Cylinder Heat Temperature
CORS	Cross-Origin Resource Sharing
FDM	Flight Data Monitoring
FOQA	Flight Operational Quality Assurance
GCS	Google Cloud Storage
HTTP	Hyper Text Transfer Protocol
IAS	Indicated Air Speed
JSON	JavaScript Object Notation
SRC	Survey and Research Center

Appendices

FAA AC 120-82

https://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_120-82.pdf

14 CFR Part 13

https://www.law.cornell.edu/cfr/text/14/part-13/subpart-C

References