Implementation of the Temporal Field Data Type as a Java Library

Joel Whitney

School of Computing and Information Science,
University of Maine, Orono, ME 04469, USA
Email: joel.whitney@maine.edu

1. Introduction

Advances in technology, particularly those in regards to micro-computing and wireless communication devices, have resulted in ubiquitous sensor networks aimed to measure the world around us. Instances of these technologies can be seen in almost any application setting, such as agriculture, industrial manufacturing, traffic management, disaster response, air quality control, pollen monitoring, and numerous other applications [5, 6]. Sensor networks made up of thousands of sensors can be spread over geographic locations, deployed to collect live or near-live data [1]. The networks are then connected to an off-site database management system or data stream engine over the Internet where analysis can be performed by a more robust computer system.

Although individual sensors can only collect point-based data this information is still useful when monitoring environmental phenomena. However, when visualizing and analysing continuous phenomena, such as air quality or elevation models for example, thousands of sensors need to be integrated to generate representations that humans perceive more intuitively. As a result, the discrete data from many sensors has to be digitally transformed into a seemingly continuous representation [2, 4] that is often continuous over *both* geographic space as well as time. To handle the different sampling characteristics of many sensors, especially if they are live streaming, can be cumbersome and complex. For example, if a user is interested in a spatial snapshot at a certain point in time, all the samples of all sensors at this time point or near this time point have to be identified. The field model aims to alleviate this problem by behaving like a wrapper over the observation data in the form of temporal, spatial, or spatio-temporal fields [7]. A field is a mathematical concept, which describes a phenomenon's continuity very well; mathematically a field is a function f from a domain D to a range R. An information system level implementation of a field concept aims to provide the same seamless continuity and hide the fact of a discrete observation set. By giving the user an interface to treat the field as being continuous, that is he/she can request a value at *any* point within the domain over which a field is defined, and the field abstraction will retrieve and/or calculate the value on-the-fly based on existing observation data [7]. Thus, instead of pre-compiling continuous phenomena into fixed resolution grids or other representations, a field is a wrapper over the original data set and can produce any type of continuous representation on-the-fly and on demand.

The *objective* of this graduate project is a prototypical implementation of temporal fields. While fields are well-known in geographic information science, actual implementations of fields are still rare, despite obvious benefits. This project focuses on implementing temporal fields that utilize static data as well as streaming data. The field model protects the original data while allowing the user to request a value for any timestamp over which the temporal field is defined. As a consequence, it is easier to integrate several streams wrapped as temporal fields because the user can just request a value at a requested timestamp and does not need to worry about the original dataset. This is particularly useful when working with large numbers of streams. Utilizing temporal fields, for example 100 unique streams, allows the user to return a spatial snapshot for all of the fields at a user-required time on the fly.

For the remainder of this report, fields will be defined in more detail first. In Section 2, the rationale for the temporal field implementations and the temporal field object type system will be described. In Section 3 the usage of the field type implementation using an example of soil moisture sensor data streams will be shown. Section 4 describes extensions using the temporal field type implementation followed by the conclusion in Section 5.

1.1 Problem Statement

Traditionally, representations of continuous phenomenon over geographic areas are produced from the data collected by raw sensors at point-based locations. Individual nodes are deployed within the sensor network to collect data for a particular measurand. More than likely, samples are gathered asynchronously depending on the characteristics of how the sensor network is set up, both programmatically as well as given hardware limitations. However, when generating a continuous representation of a phenomenon these values need to be integrated correctly over space and time [3]. In traditional GIS, typical users transform sensed data into visual and continuous representations, e.g. rasters, TINs, or Voronoi diagrams, using various interpolation methods (i.e. IDW, st-IDW, kriging, etc.). Once this is performed, the original data is discarded. However, problems can arise when users are tasked with trying to integrate rasters that do not have matching resolutions, requiring the user to resample at least one of the rasters in question. This introduces error into the calculation due to approximation during resampling before integration.

1.2 Defining fields

A (geospatial) field is a mathematical defined as a function *f* from a domain *D* to a range *R*. All elements of the domain are mapped to an element of the domain. For example, a *spatial* field represents a phenomenon that is continuous over a spatial region. Given a bounded spatial domain S and a potentially infinite attribute domain V, a spatial field is a function $f: S \rightarrow V$ from all the elements of set S to the attribute domain V. The elements of S can be seen as locations; the attribute domain V can represent temperature values, for instance. A *temporal field* is often defined over an

individual sensor's measurements. A sensor's measurements establish a time series, that is, we can map timestamps to measurements. A temporal *field* is defined as a mapping from the elements of a *time domain* T to attribute values $f: T \rightarrow V$. A temporal field does not necessarily have a spatial dimension. It simply represents change of an attribute over time. The attribute can be a, the value of a stock symbol, or the reported value of a sensor or the reported location of a moving object.

1.3 Implementation of temporal fields

An *implementation* of a field is supposed to keep the original observation data and through the field's methods, generates any values the user needs on the fly, thus, behaving like a seamless, continuous entity. "Underneath the covers", the field implementation will use a statistical interpolation method to calculate the requested value based on the observation data. For a temporal field, a value can be requested at any time point within the interval over which the temporal field is defined. For a spatial field, output could result in a raster, TIN, Voronoi diagram, or interpolated values at user-specified intervals, all while not affecting the underlying data. Thus, in the case of integrating rasters at incompatible resolutions, the field model allows the user to create several fields over different observation data sets, and regenerate rasters at compatible resolutions directly from the observation data, resulting in minimal error.

In this implementation, a temporal field framework was developed. Using this framework, users can easily create fields over their data to query, store, and plot the data as if it were a continuous phenomenon. The power of the temporal field data types lies in its' ability to wrap many individual sensor data streams into temporal fields. This allows users to query specific times over each field's validity window, and/or slice the field into user-defined snapshots. The result of the latter method provides a dataset with values that can be integrated over time at user-defined density. By instantiating multiple fields over different sensor nodes, the user can compare all sensor streams at the same time instant. The implementation also enables wrapping data from different underlying storage technologies (DBMS, CSV/text file, or real-time stream) as temporal fields.

2. Design and Implementation

One of the goals when designing and implementing the temporal field data type was to create an object-based framework in such a way that can be easily extended. The library should be extensible to include more `'Data'/'Interpolation'` class objects, or even modified in the future to account for implementing a spatial field component. The temporal field data type was designed to allow users to construct fields over their data and subsequently query, store, and plot the data as if it were a continuous phenomenon. Calculations are performed on-the-fly by the application and returned to the user.

The design of the TF framework follows an object-based approach. Figure 1 shows the conceptual design of the TF framework. Starting in the top-center of the conceptual model at the

'TemporalField' class, the reader can follow the framework to understand of how the temporal field data type works. Since the main goal of this project was to implement the temporal field data type, the emphasis was building the underlying skeleton of the framework without worrying about the numerous ways that the data type could be used within an interface or separate application. Furthermore, the initial conceptual model was designed without consideration to the language that would be later used in the implementation; however, it assumes an object-oriented programming language.

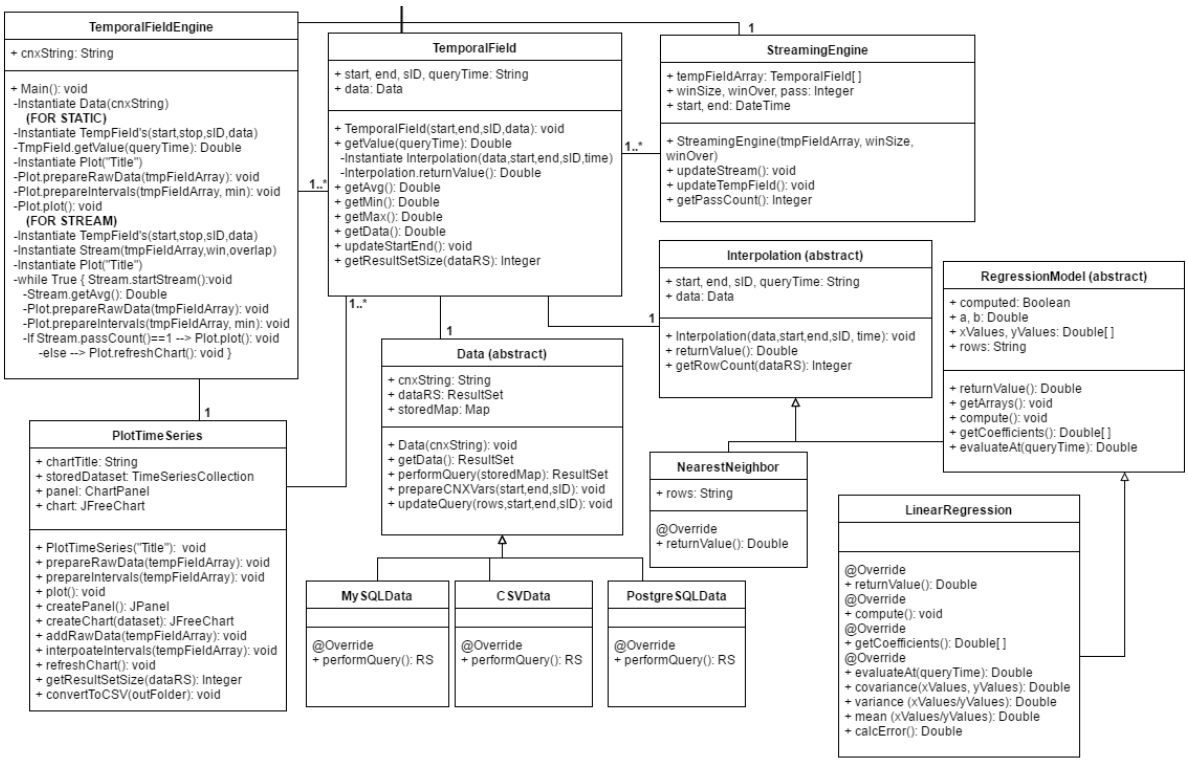

Figure 1. Conceptual design of the temporal field data type.

2.1 Code Walkthrough

In order to instantiate a 'TemporalField' object (upper middle box in Figure 1) the user must define the temporal field by passing in the appropriate arguments. The arguments needed for the 'TemporalField' include: **start** datetime (String), **end** datetime (String), **sensor ID** (String), and where the data is located through an instantiated 'Data' object (Data). Allowing the temporal field to take a data object opposed to the actual data (i.e. a ResultSet) means that the temporal field does not need to store or cache large amounts of data. The data object 'knows' where the data located is and how optimize access and caching. This is extremely important for implementations that will be calling temporal field objects for large numbers of sensor nodes over long periods of time.

The 'Data' object has methods to acquire the data depending on which subclass is instantiated. For the initial design of this framework, the data objects can be in the form of either 'MySQLData',

'PostgreSQLData', or 'CSVData'. These types are all subclasses of the super 'Data' class, which makes it easy to add new data sources that know exactly how to retrieve data from the underlying data storage technology.

The design of the temporal field data type should remain the same when working with static data as well as streaming data. When implementing the usage of the framework at the 'TemporalFieldEngine' level, the execution varies significantly; I will discuss them separately. The benefit of the current library design is that the data objects don't care whether the data is static or still being entered into the data storage system. The only thing that changes with the use of the library is how the temporal field objects are handled at runtime. This is a benefit, because designing an interface that is going to work exclusively with streaming data opposed to static data means the user only needs to model the execution of the application, from the example seen in the '(FOR STREAMS)' section of the temporal field engine's main() method (Appendix A – Figure 2).

2.2 Working with static data

When working with static data, once the 'Data' and 'TemporalField' objects are instantiated, the user can call one or more of the accessible options over the temporal field, including: **getValue**(), **getAvg**(), **getMin**(), and **getMax**(). More often than not, the user will want to call the **getValue**() method on the 'TemporalField' object, which takes a date and time as an argument. The method will return a **value** (Double) at the specified time. Within the 'TemporalFieldEngine', using the example seen in the '(FOR STATIC)' section of the main() method (Appendix A – Figure 2), the user can call the **getValue()** method on any of the temporal field objects that are instantiated. Depending on which interpolation method is instantiated within the temporal field object by the user, the value will either be calculated using the 'NearestNeighbor' or 'LinearRegression' interpolation classes. In order to change which interpolation method shall be used in the calculation, the user would need to change the 'Interpolation' subclass that will be instantiated internally of the temporal field object. The benefit of this method is that users can easily introduce new subclasses to be used for interpolation without having to change anything in the rest of the temporal field data type.

The user can also instantiate the 'PlotTimeSeries' object, which allows the user to not only plot the raw data as a continuous field, but also has the ability to interpolate values at specified intervals. Thus, the output is not returned as individual tuples but the results can be visualized. The 'PlotTimeSeries' object takes a **title** (String) as an argument in order to instantiate itself and then can call the **prepareRawData**() or **prepareIntervals**() methods to generate the data to be graphed. Both of the preceding methods take the temporal field objects as an Array, while the **prepareIntervals**() method takes an additional **minutes** (Integer) argument to determine the intervals from start to calculate values at. An array of temporal field objects to the **prepareRawData**() and

prepareIntervals() methods were required so the graphing library can use all of the temporal field objects when creating a single graph.

After the **prepareRawData**() and/or **prepareIntervals**() methods are called for the plot time series object, the data is ready to be plotted. Calling the **plot**() method will take the generated data and output a graph with the plotted traces for each dataset. The result of the previous method has generated traces for the raw data as well as the interpolated intervals (if called) at finer/coarser grained resolutions than just plotting the raw data. With the plot time series object, the user can also convert all of the interpolated datasets into individual CSV files. For each series of data generated by the temporal field objects (raw data and/or interpolated values for specific intervals) a unique csv file will be generated containing a tuple ID, x value, and y value. For the temporal field, the x and y values are the datetime and measurand value respectively.

2.3 Working with streaming data

The temporal field data type also works with streaming data in the same capacity as it does with static data. The way the user can accomplish this is to handle the data with sliding windows, such as seen in Figure 2. In principle, a data stream is an infinite update relation from a sensor that sends new updates over time live. Streams are processed in units of windows over the streams; for instance, always processing the most recent 4 tuples would establish of a window of size 4. Once the window is full the system can for instance calculate the sum of the values of the 4 measured values. For each subsequent window, two of the previous values are kept in the new window. This establishes a sliding window of size 2. The sliding window works particularly well for instances where the user is looking for trends in the data and gradual changes are desired from the output.

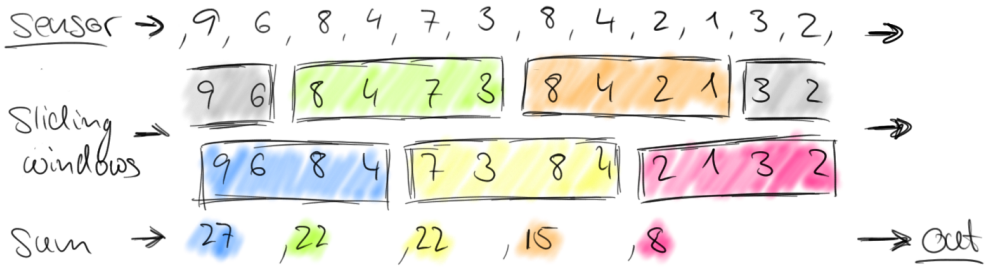

Figure 2. Sliding window premise (source https://flink.apache.org).

The temporal field framework works in the same manner as the sliding window example above when handling data streams with the temporal field objects. When the stream starts, the system waits until the specified window is full and then processes the data. Simultaneously, the temporal field objects' start times are updated to be that of the current time minus the overlap desired, so the previous data can be included in the next frame. By default, it is assumed that the user will be working with sliding windows, therefore this is how the framework was designed. However, it would be easy to modify this, or, if using the framework as a wrapper in a specific application, to pass it values that would nullify the overlap (i.e. overlap of zero).

Looking at the example code in the '(FOR STREAM)' section of the `'TemporalFieldEngine'` one can have the data and temporal field objects set up in the same way as if working with static data (Appendix A – Figure 2). The only difference is the start/stop dates and times don't matter for each temporal field object during their instantiation. Instead, the `'StreamingEngine'` object that will update the temporal field objects based on the window size and the overlap between windows. As far as the framework goes, there is no need to change the temporal field data type when working streaming data.

After setting up the temporal field objects and the variables for the sliding window in the `'TemporalFieldEngine'`, the user needs to initialize the streaming engine object. The streaming engine object will control the temporal field objects throughout the course of the next part of the program. The parameters needed to initialize the streaming engine include, an array of **temporal field objects** (Array), the **window size** (Integer), and **overlap** (Integer) between windows. The window size and overlap are both taken in minutes.

Next, the temporal field engine starts an infinite loop, which will run forever unless an exception is introduced into the program. At the start of each loop, the streaming engine updates the stream with the **updateStream**() method. This will update the start of each temporal field objects window to the current date minus the overlap between windows. It also updates the **end** datetime to that of the **start** datetime plus the window size. If it's the stream's first pass it ignores the subtracting the overlap until the next pass. After all of the temporal fields in the array are updated, the stream engine waits until the windows **end** datetime to process the data.

At this point, the user can call any of the temporal field objects methods to analyze the data before the next pass of the loop starts. The example in the `'TemporalFieldEngine'` returns the minimum, maximum, average, and result size (i.e. number of records) for each temporal field object. The raw data is then plotted for each temporal field object as well as interpolated values at one, two, and three-minute intervals. If the graph isn't open it will open a new graph, otherwise it will update itself. The streaming engine will continue to the next pass of the loop, repeating the above executions, until the application is killed or some sort of fatal exception is caught.

2.4 Implementing the temporal field data type as a Java library

The framework for the temporal field data type described above was implemented as a Java library to create a sample of how the field model can be used in practice. Java was used as the language of choice due to its object-oriented nature. Through inheritance, sub-classes are utilized to extend the use of existing super classes. This was an extremely important aspect of the field data type since many different methods of data access and calculations will need to accommodate different scenarios. Using a Java IDE, you can take the code accompanied with this project and run all of the scenarios described in the previous design sections.

3. Testing and Applications

After the initial framework of the temporal field data type was determined, the design and implementation described in the previous section was tested and verified. This resulted in the need for test data to verify the functions and objects were behaving as they were intended. Since, the temporal field framework works just as well with static data as it does with streaming data, it was important to test both aspects simultaneously to make sure that their implementation doesn't affect the other. In the former of the two cases, previously collected data was used, whereas for the latter case, a simulated data stream was used to mimic the previously collected data.

3.1 Test data

The first application of the implementation was to use the field model on static data that was previously collected at a commercial blueberry farm in down-east Maine. A wireless sensor network was deployed in blueberry barrens to monitor soil moisture throughout the blueberry field. The temporal field was used at the end of the growing season to represent the data as a continuous phenomenon for identifying trends and consistency in the data. Figure 3 and 4 shows an example of how to instantiate a field over a sensor node and the resulting plot for raw data at the end of the season for all three nodes respectively. The field model allowed us to query the data at specific times and the on-the-fly interpolation works in the background to calculate the value at the queried time. The plotting function allows the user to plot the data in a fine-grained or coarse way by interpolating values at a user specified sample rate, regardless of the sampling rate of the raw data. Figure 5 shows how to interpolate intervals over the temporal field objects and Figure 6 shows the resulting output after plotting raw data of a much smaller window than the previous plot. The three temporal field objects were then used to prepare intervals at one, five, and fifteen-minute intervals.

```
// tempField1
String start1, end1, sensorID1, queryTime1;
start1 = "2015-08-03 09:19:00";
end1 = "2015-09-13 01:43:00";
sensorID1 = "1";
TemporalField tempField1 = new TemporalField(start1, end1, sensorID1, data);
```

Figure 3. Code snippet for instantiating a temporal field object over a sensor node.

3.2 General set-up

For testing purposes of the implementation, all changes that would be made to to create and access a temporal field are made within the **main**() method of the `'TemporalFieldEngine'`. The user also can use the TemporalField library in other programs directly. The `'TemporalFieldEngine'` is a test program using the temporal field library. Within the main method, the user can instantiate one or many `'TemporalField'` objects. Once instantiated, the user can call any of the temporal field methods on each temporal field object. If running the program as a standalone script, the user can run the application entirely from the temporal field engine, with exception of making minor changes to the some of the other classes.

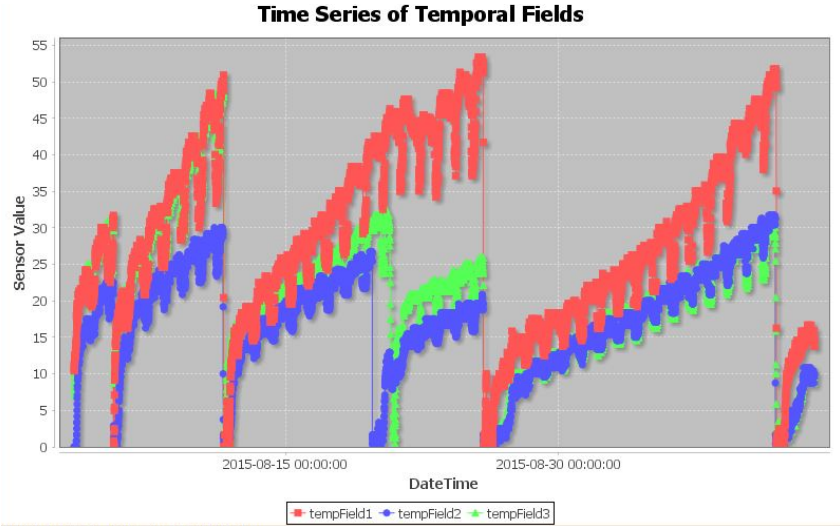

Figure 4. Raw data of the three sensor nodes plotted.

3.3 Temporal Fields based on static data

For testing purposes, this data was invaluable because it allowed for troubleshooting the temporal field library with real data. Prior to testing, the data was made available in CSV and PostgreSQL/MySQL database formats, to test the `'Data'` object subclasses were able to retrieve the data successfully from the same inputs to the temporal field objects. As mentioned earlier, the temporal field object does not care to know how to acquire the data. The temporal field object only needs to know I have a `'Data'` object and that object is responsible for knowing where the data is located and how to acquire it. This was an important aspect of the testing because each subclass of the data object requires a specific **performQuery**() method that overrides the superclass's **performQuery**() method. The testing needed to verify the temporal field data type could leverage the returned data regardless of which data object was being used to acquire the data.

```
/** 5a. Create PlotTimeSeries object and adjust the settings.
 */
// Create array of tempField objects and plot data
TemporalField[] temporalFieldsArray = {tempField1, tempField2, tempField3};
// Initiate PlotTimeSeries object
PlotTimeSeries plotData = new PlotTimeSeries("Time Series of Temporal Fields");
// Calculate the raw datapoints
plotData.prepareRawData(temporalFieldsArray);
/** 5b. User can plot raw data and/or interpolated values at various intervals
 */
// Calculate the interpolated datapoints at each interval
plotData.prepareIntervals(temporalFieldsArray, 1);
plotData.prepareIntervals(temporalFieldsArray, 5);
plotData.prepareIntervals(temporalFieldsArray, 15);
```

Figure 5. Code snippet for interpolating intervals over the temporal field objects.

The testing data also allowed validation of the interpolation and calculation methods, by extracting known subsets of the data for focused testing. By pulling out a subset of data, basic query tests can be run on some of the data to easily see if the `'NearestNeighbor'` and `'LinearRegression'` objects were returning the correct values when the **returnValue**() method was being called. One

simple test was to query known timestamps (i.e. the time of first sensor reading) to make sure identical values were returned. Initially, the linear regression model was designed to grab all of the tuples within a specific temporal range around the queried time. Depending on how close the data-points were together it was hard to get consistent R-squared values compared to the regression line.

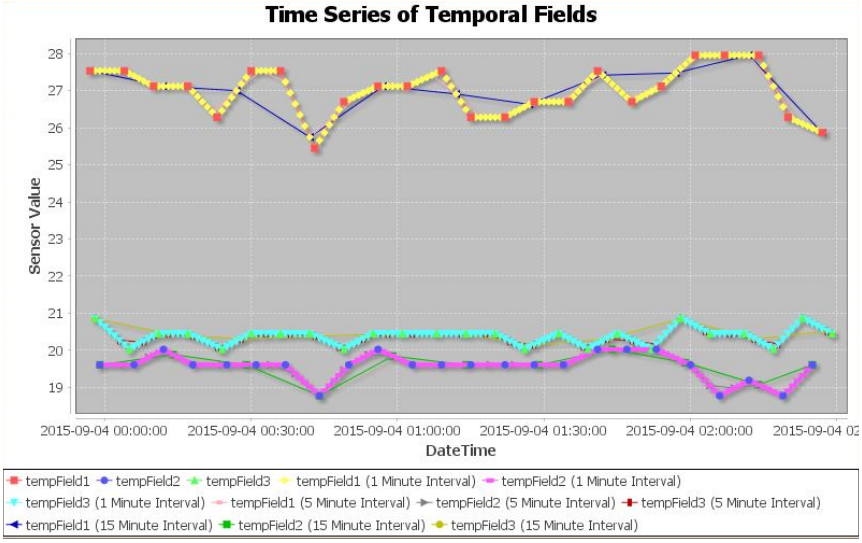

Figure 6. Three temporal fields plotted using the raw data and interpolated intervals at 1, 5, 15 minute snapshots.

This forced a redesign of the regression model to take a number of data points around the query-time opposed to a pure temporal range. This way, even if the data was highly variable (like it with the test data), the user could choose to select only two data points, returning an R-squared value of 1.0. With that said, it's worth noting that the linear regression class that was designed for the temporal field data type is a true regression model opposed to simply a linear interpolation. By adjusting the range around the query time (which is a variable in the form of number of tuples) the user can simulate a linear interpolation. When creating a linear interpolation, the user will always get a straight fit line between the two data points used in the calculation. Otherwise, the user has the ability to adjust how many data points he/she would like to use for each calculation, subsequently decreasing the R-squared value. In the event that users prefer to adjust these parameters, a **calcError**() method was included that can return the R-squared value for the user to calculate the variability between the fit line and the actual data.

3.4 Temporal Fields based on streaming data

The second testing application focused on using the temporal field model on streaming data. For the sake of simplicity, a simulated data stream was generated using a Python script that generates an output of values close to those of the previously used observation data with a similar throughput. The data was inserted into the database and the field model can use a streaming engine object to allow a user to work within a sliding window over their streaming data as described in the design and implementation section. In this test case, the temporal field allows the user to do the same functions

that they apply to static data, but handles the streaming data within a user-specified sliding window. This allows the user to do real-time analysis and plotting of the streamed data. Similar to the static data, the simulated streaming data was needed to verify the interoperability of the temporal field library for working with streaming data.

In the process of designing the library to work with streaming data, it was quickly identified that there needed to be a way to update the temporal field objects on-the-fly. This resulted in the development of the `'StreamingEngine'` class that was responsible for updating the temporal fields and keeping track of the stream status. An additional **updateStartEnd**() method was also added to the `'TemporalField'` class that wasn't originally in the design. This allows the streaming engine to iterate through each temporal field object in the array and update itself for each new window. Figure 7 shows the resulting graphs of the above process over streaming data with a ten-minute sized sliding window and an eight-minute sized overlap between windows. The red-line shows the progression of a single time instance progressing over four frames of the sliding window. Once the window is full, the user can call any of the temporal field object's methods over the field, such as: **getMin**(), get **Max**(), or **getAvg**().

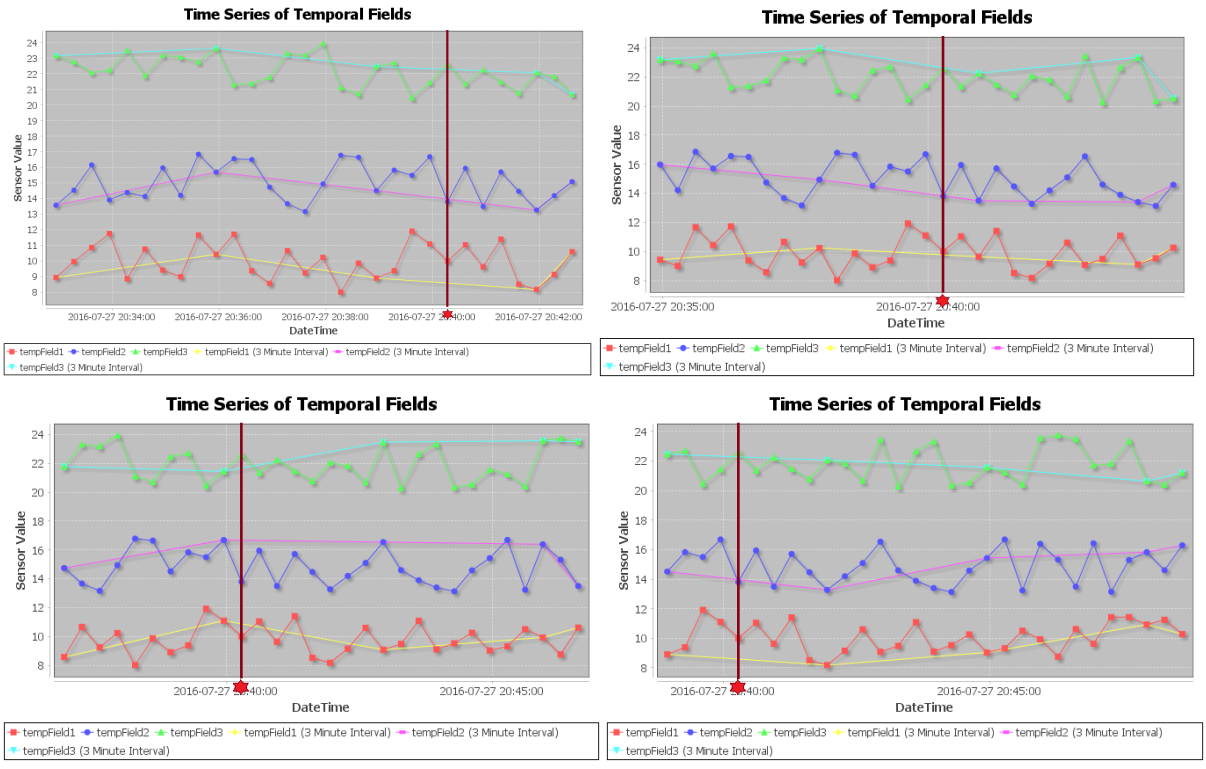

Figure 7. Four frames from a data stream with a ten-minute sliding window and an eight-minute overlap (top-left to bottom-right).

4. Temporal Field Python Extension for QGIS

A great deal of time was spent designing the temporal field data type in a manner that was clean and concise so that each class object has specific job to do. This allows the data type to be used in custom

applications very easily and also to be extensible. In one example to show this, a Python plugin for QGIS was developed to show the flexibility of the temporal field data type. The plugin allows a user to access the temporal field Java implementation and bypass the `'TemporalFieldEngine'` from a standard GIS software package. Figure 8 shows an example of the GUI the user interacts with to execute queries on temporal field objects much like they would in the temporal field engine. The Python plugin allows users who prefer to work with their data a GIS software environment to be able to query, analyse, and plot their data just as they would with the standalone program. The relationship between the Java application and the Python plugin are shown at the bottom of Appendix A - Figure 1.

```
java -jar "C:///…//plugins//TemporalFieldGateway//TemporalFieldGateway.jar" "
    dbtype;MySQL,host;localhost,port;8889,database;BlueberrySensors,table;moisturedata,username;root,password;root"

java -jar "C:///…//plugins//TemporalFieldGateway//TemporalFieldGateway.jar" "
    dbtype;PostgreSQL,host;localhost,port;5432,database;blueberrysensors,table;moisturedata,username;joelw,password;999jbw"
```

Figure 8. Two examples of how the 'TemporalFieldGateway' Java application can be started from the command line.

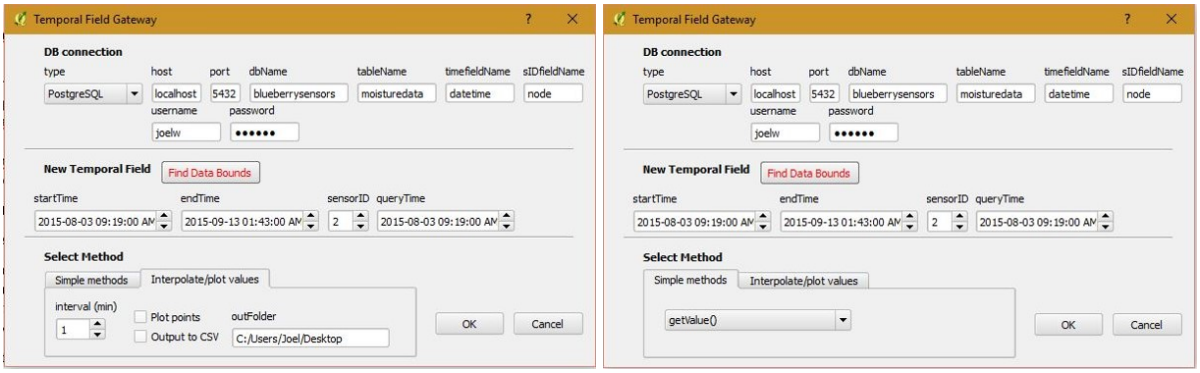

Figure 9. User interface for the QGIS 'TemporalFieldGateway' plugin.

While extending the Java application, additions were made to the code to accept PostgreSQL DB connections as a data source. A Python plugin also was developed that allows the user to access temporal field class objects within their GIS software environment. This meant developing a Python plugin that could access and run a Java application. The final product of the extension is a simple plugin in QGIS that gives the user access to the main functions of the Temporal Field class. The result for simple methods is the computed measurand's value at any time within the defined temporal field as a pop up. The user can also interpolate intervals, plot, and export the results as a CSV.

Normally, the Java application starts with the `'TemporalFieldEngine'` class, which houses the **main**() method. Within the main method the user will instantiate one or many temporal field objects. Once instantiated, the user can call any method on each temporal field object. If running the program from a Java IDE, the user can run the application entirely from the temporal field engine, with exception of making minor changes to the some of the other classes. In order to get the plugin to work

with the Java application a new Java object was built, e.g. `'TemporalFieldGateway'`, which bypasses the temporal field engine. The **main**() method in the temporal field gateway uses the Py4J library to open a gateway server. Once the Java application is running it listens at a specified port and allows other programs to access its' methods. Figure 8 shows two examples of command line prompt that can be used to start the Java application. In the Python plugin, once 'OK' is pressed, the plugin connects to the temporal field gateway using Py4J's JavaGateway. Once the connection is established, any method can be called from the Java object. Figure 9 shows an example of the GUI the user interacts with inside the GIS software to execute queries on temporal field objects much like they would in the temporal field engine.

5. Conclusions

Although individual sensors can only collect point-based data this information is still useful when monitoring environmental phenomena. However, when visualizing and analysing continuous phenomena, such as air quality or elevation models for example, often very large numbers with up to thousands of sensors need to be integrate to generate representations that humans perceive more intuitively. As a result, the discrete data from many sensors has to be digitally transformed into a seemingly continuous representation [2, 4] that is often continuous over *both* geographic space as well as time. To handle the different sampling characteristics of many sensors, especially if they are live streaming, can be cumbersome and complex. For example, if a user is interested in a spatial snapshot at a certain point in time, all the samples of all sensors at this time point or near this time point have to be identified. The field model aims to alleviate this problem by providing a wrapper over the observation data in the form of temporal, spatial, or spatio-temporal fields [7]. A field aims to hide the fact of a discrete observation set, and gives the user the interface to treat the field as being continuous, that is he/she can request a value at *any* point within the domain over which a field is defined, and the field abstraction will retrieve and/or calculate the value on-the-fly based on existing observation data [7]. Thus, instead of pre-compiling continuous phenomena into fixed resolution grids or other representations, a field is a wrapper over the original data set and can produce any type of continuous representation on-the-fly and on demand.

The *objective* of this graduate project had been to achieve a prototypical implementation of temporal fields. While fields are well-known in geographic information science, actual implementations of fields are still rare, despite obvious benefits. This project focused on implementing temporal fields that utilize static data as well as streaming data. The field model protects the original data while allowing the user to request a value for any timestamp over which the temporal field is defined. As a consequence, it is easier to integrate several streams wrapped as temporal fields because the user can just request a value at a requested timestamp and does not need to worry about the original dataset. This is particularly useful when working with large numbers of streams. Utilizing temporal fields, for

example 100 unique streams, allows the user to return a spatial snapshot for all of the fields at a user-required time on the fly.

In this *implementation* of the temporal field model, a framework was developed where field data types abstract from a discrete observation set of samples to represent a continuous field. The temporal field data type, implemented as a Java library, provides a means to wrap sensor data streams as temporal field objects to be used in the analysis and interpretation of discrete data. Using this application, users can easily create fields over their data to query, store, and plot the data as if it were a continuous phenomenon. The power of the temporal field data types lies in its' ability to wrap many individual sensor data streams into temporal fields. This allows users to query specific times over each field's window, and/or slice the field into user-defined snapshots. The result of the latter method provides a dataset with values that are integrated over time at user-defined density. By instantiating multiple fields over different sensor nodes, you can compare all sensor streams at the same time instant. The implementation also enables access of data from different underlying storage technologies (DBMS, CSV/text file, or real-time stream).

The temporal field data type, implemented as a Java library, provides a means to wrap sensor data streams as temporal field objects to be used in the analysis and interpretation of discrete data in the form of an easy-to-use library. The data support numerous storage technologies, such as database management systems or simply as a text file. The data storage technologies supported include MySQL/PostgreSQL databases or a CSV file, however this could easily be extended by adding additional subclasses to the `'Data'` superclass as was done when creating the 'Temporal Field Python Extension' in section 4. When requesting values at any point over the temporal field's window, either explicitly through a **getValue**() call or internally through an **interpolateIntervals**() call, the user has the ability to choose what method of the interpolation they want to use. Currently, the interpolation can be in the form of a nearest neighbour or a linear regression model, but the library is easily extensible to include more methods for interpolation. For the linear regression model, which is the default method, the user can specify the range around the requested time in order to fine-tune the interpolation. The prototypical implementation created in this graduate project can be easily modified for a specific use, or used as is in an existing application.

References

[1] Murty, R.N. et al. 2008. CitySense: An Urban-Scale Wireless Sensor Network and Testbed. 2008 IEEE Conference on Technologies for Homeland Security. (May 2008), 583–588

[2] J.C. Whittier, Q. Liang, and S. Nittel, Evaluating Predicates over Dynamic Fields, 5th International Workshop "Geostreaming" in conjunction with SIGSPATIAL 2014, Dallas, TX, November 2014.

[3] G. Jin and S. Nittel: Towards Spatial Window Queries Over Continuous Phenomena in Sensor Networks, IEEE Transactions on Parallel and Distributed Systems (TPDS), Vol 19(4), pp. 559-571, April 2008.

[4] S. Nittel, J.C. Whittier and Q. Liang, Real-time Spatial Interpolation of Continuous Environmental Phenomena using Mobile Sensor Data Streams, SIGSPATIAL 2012, Redondo Beach, CA, November 2012.

[5] Campbell, A.T. et al. 2008. The Rise of People-Centric Sensing. *IEEE Internet Computing* (Jul. 2008), 30–39.

[6] Resch, B. et al. 2009. Real-Time Geo-awareness –Sensor Data Integration for Environmental Monitoring in the City. *2009 International Conference on Advanced Geographic Information Systems & Web Services* (Feb. 2009), 92–97.

[7] Q. Liang, S. Nittel and T. Hahmann, From Data Streams to Fields: Extending Stream Data Models with Field Data Types, 9th International Conference on Geographic Information Science (GIScience), Montreal, Canada, September 2016.

Appendix A.

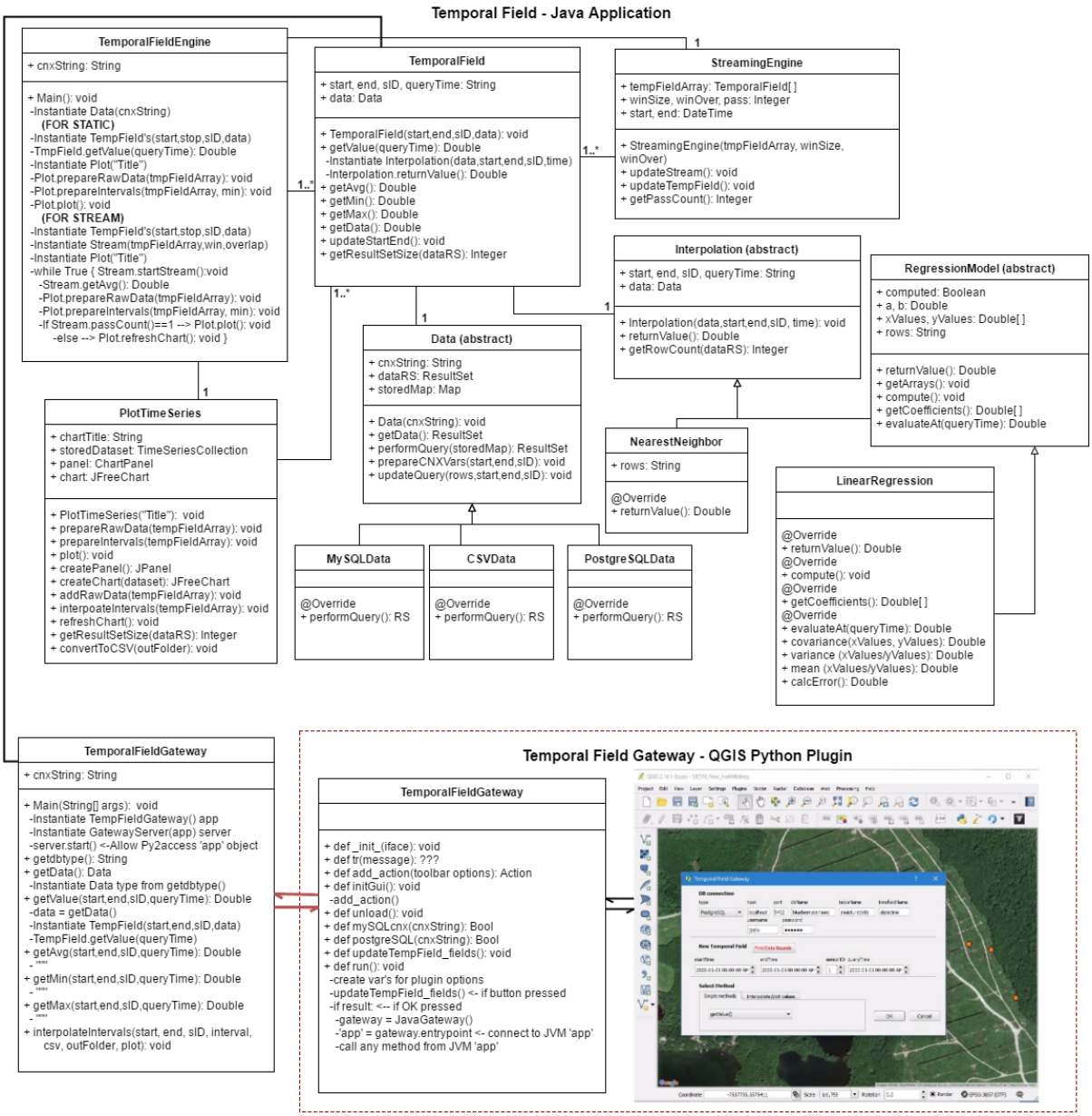

Figure 1. Full conceptual model, including Python plugin for QGIS to interact with the Temporal Field Java library.

```java
import java.io.FileNotFoundException;
import java.sql.SQLException;
import java.text.ParseException;

/**
 * Created by Joel on 1/5/2016.
 * The engine that runs the TemporalField application.
 */

public class TemporalFieldEngine {
    /** 1. Choose connection string
     */
    public static String cnxString = "dbtype;MySQL,host;localhost,port;8889,database;BlueberrySensors,table;
MoistureDataFAKE,username;root,password;root";
    // main method
    public static void main(String[] args) throws ParseException, SQLException, FileNotFoundException {
        /** 2. Choose which Data object to instantiate
         */
        MySQLData data = new MySQLData(cnxString);
        /** 3. Choose whether to use the static (false) or streaming (true) functionality
         */
        Boolean streamingData = true;
        // streaming v static data logic
        if (!streamingData) {
            /** 4. Create tempField object(s) and adjust the settings.
             */
            // tempField1
            String start1, end1, sensorID1, queryTime1;
            start1 = "2015-09-03 09:19:00";
            end1 = "2015-09-13 01:43:00";
            sensorID1 = "1";
            TemporalField tempField1 = new TemporalField(start1, end1, sensorID1, data);
            // query value
            queryTime1 = "2015-09-04 01:22:00";
            System.out.printf("\nThe tempField1 object returned: %s at '%s'", tempField1.getValue(queryTime1),
queryTime1);
            /** 5a. Create PlotTimeSeries object and adjust the settings.
             */
            TemporalField[] temporalFieldsArray = {tempField1};
            PlotTimeSeries plotData = new PlotTimeSeries("Time Series of Temporal Fields");
            plotData.prepareRawData(temporalFieldsArray);
            /** 5b. User can plot raw data and/or interpolated values at various intervals
             */
            plotData.prepareIntervals(temporalFieldsArray, 5);
            // export data as CSV then plot the data
            plotData.convertToCSV("C:/Users/Joel/Desktop");
            plotData.plot();
        }
        else {
            /** 4. Create window for tempField object(s) over streaming data.
             */
            String start1, end1, sensorID1, queryTime1;
            start1 = "now";
            end1 = "future";
            sensorID1 = "1";
            TemporalField tempField1 = new TemporalField(start1, end1, sensorID1, data);
            /** 5. Instantiate StreamingEngine object
             */
            // Stream window settings
            Integer windowSizeMIN = 10;
            Integer windowOverlapMIN = 8;
            // Create array of tempField objects and instantiate StreamEngine objects for each tempField
            TemporalField[] temporalFieldsArray = {tempField1};
            StreamingEngine stream = new StreamingEngine(temporalFieldsArray, windowSizeMIN, windowOverlapMIN);
            /** 6a. Create PlotTimeSeries object and adjust the settings.
             */
            PlotTimeSeries plotData = new PlotTimeSeries("Time Series of Temporal Fields");
            while (true) {
                // Update streams at start of loop
                stream.updateStream();
                System.out.printf("\nThe tempField1 object returned a min/max/avg value of '%.4f'/'%.4f'/'%.4f' on %s
tuples",
                        tempField1.getMin(), tempField1.getMax(), tempField1.getAvg(), tempField1.getResultSetSize());
                /** 6b. User can plot raw data and/or interpolated values at various intervals
                 */
                // Calculate the raw datapoints and some interpolated values
                plotData.prepareRawData(temporalFieldsArray);
                plotData.prepareIntervals(temporalFieldsArray, 3);
                // If first pass open plot, otherwise just refresh
                if (stream.getPassCount() == 1) {
                    plotData.plot();
                } else {plotData.refreshChart();}
            }
        }
    }
}
```

Figure 2. 'TemporalFieldEngine' class object code from Java application.