

Online System for Photovoltaic Data Field Station

by

Michael A. Chu

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

December 14, 2001

[February 2002]

Copyright 2001 Michael A. Chu. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

BARKER

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUL 31 2002

LIBRARIES

Author _____
Department of Electrical Engineering and Computer Science
December 14, 2001

Certified by _____
Louis L. Bucciarelli
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Online System for Photovoltaic Data Field Station

by
Michael A. Chu

Submitted to the
Department of Electrical Engineering and Computer Science

December 14, 2001

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The online system for the PV data field station enables web-based presentation of real time and historic data for the purposes of PV system simulation. Programs have been written to read information from the (buffered) serial port of a server, parse and filter the data, and store it in a database – all done in real time. Also, search and selection ability have been implemented, so that a user anywhere in the world can access historic data; search for particular data sets; view incoming data in real time; simulate the performance of a PV system. The tools for the online system have all been written in Java, to facilitate cross-platform compatibility.

Thesis Supervisor: Louis L. Bucciarelli

Title: Professor of Engineering and Technology Studies

Table of Contents

Table of Contents	3
Table of Figures	4
1 Introduction	5
1.1 Photovoltaics	5
1.2 Thesis Statement	6
2 Design Overview	9
2.1 System Architecture	9
2.1.1 Hardware Architecture	10
2.1.2 Software Architecture	11
2.1.3 Webserver.....	12
2.1.4 Relational Database.....	12
2.2 Database / Data Model	13
2.3 Data Processing	15
2.3.1 Data Transmission.....	16
2.3.2 Data Upload.....	17
2.3.3 Data Filtering and Archival.....	20
2.3.4 Conversion details	22
2.3.5 Threading	22
2.4 Database Interactions	24
2.4.1 Database Connectivity.....	24
2.4.2 Remote Method Invocation (RMI).....	26
2.4.3 Searching.....	30
2.4.4 Optimization.....	31
2.5 User Interface	34
2.5.1 Java Applets	34
2.5.2 Compatibility / AWT only	34
2.5.3 RealTime.java	35
2.5.4 DataGrapher.java / Grapher.java.....	37
2.5.5 Sampling and Averaging.....	38
3 Conclusions	44
Appendix A - Sockets.....	45
Appendix B – Online Resources.....	46
Appendix D – Sample data file from 10/25/01	47
Appendix E – Sample Data.....	49
Appendix F – data model.....	52
Appendix G – Conversion details	53
Appendix H – HTML Converter	55
Appendix I – RMI Instructions.....	56
Appendix J – TTY.C Instructions	57

Table of Figures

- Figure 1: System overview..... 7
- Figure 2: System Architecture..... 10
- Figure 3: Data Model 14
- Figure 4: Sample GO block Insert statement 18
- Figure 5: Sample IV block Insert statement..... 18
- Figure 6: Sample GO block..... 21
- Figure 7: RMI's client-server architecture..... 26
- Figure 8: Marshalling and Unmarshalling 27
- Figure 9: Main.html real-time display applet..... 36
- Figure 10: Screenshot of ISC.html..... 38
- Figure 11: Graph of battery voltage - all points 39
- Figure 12: Graph of battery voltage - 1 point per pixel 40
- Figure 13: Graph of battery voltage - 8 points per pixel..... 41
- Figure 14: Graph of battery voltage - 8 points per pixel w/ averaging 42
- Figure 15: Graph of battery voltage - Min, average, max scheme..... 43

1 Introduction

1.1 Photovoltaics

Photovoltaic (PV) systems convert light energy into electrical energy. PV cells, also called “solar cells,” provide power for everyday devices – electricity for pumping water, power for communications equipment, and lighting for homes. PV systems are particularly well suited for remote sites, far removed from utility grids.

A PV cell consists of two or more thin layers of semi conducting material, usually silicon. Electrical charges are generated and conducted away by metal contacts as direct current (DC) when light hits the silicon. The electrical output from a single cell is small; so multiple cells are connected together and usually encapsulated behind glass to form a module (sometimes referred to as a "panel").

PV power has various benefits over diesel generators, primary batteries, and conventional utility power. PV modules have no moving parts and thus require minimal maintenance, although initial costs are high relative to more traditional power supplies. Although there are higher initial costs, PV power offers high reliability, reduced costs of operation, environmental advantages, and modularity to name just a few of the advantages.¹

The PV module is the principle building block of a PV system; any number of modules can be connected together to give the desired electrical output. This modular structure is

¹ For further resources about photovoltaics from the U. S. Department of Energy, the British Photovoltaic Association, Sandia National Laboratories, and others, see Appendix B.

a considerable advantage of the PV system; the system voltage can be different for different systems, depending on the number of modules connected in series. Further, strings of modules can be added to an existing system in parallel to increase current and power levels. There are various applications of PV technology, different types of PV cells, and an assortment of different PV system configurations.

A PV powered data field station has been installed atop MIT building 1. In addition to measuring ambient temperature, wind speed, and wind direction, the system measures the voltage of the battery storage subsystem and the current-voltage characteristics of the PV module itself. An "I/V" curve is taken every minute, whereas other variables are measured every two seconds. The primary purpose of the PV data field station is to generate PV data for the simulation of stand-alone systems – designed to supply any time varying load.

1.2 Thesis Statement

This PV powered station continuously transmits data to an RF receiver connected to a PC server inside a room facing the roof of an adjacent building. Approximately four megabytes of data stream into the server continuously over the course of a single day and are stored in a file. Each daily file contains data defining ambient temperature, module temperature, wind direction, wind speed, battery voltage, IV sets, etc.²

² Appendix D shows a portion of a sample data file. Appendix E contains plots of data from January 13, 2001 – plots of ambient temperature, module temperature, wind speed, battery voltage, and module short circuit current.

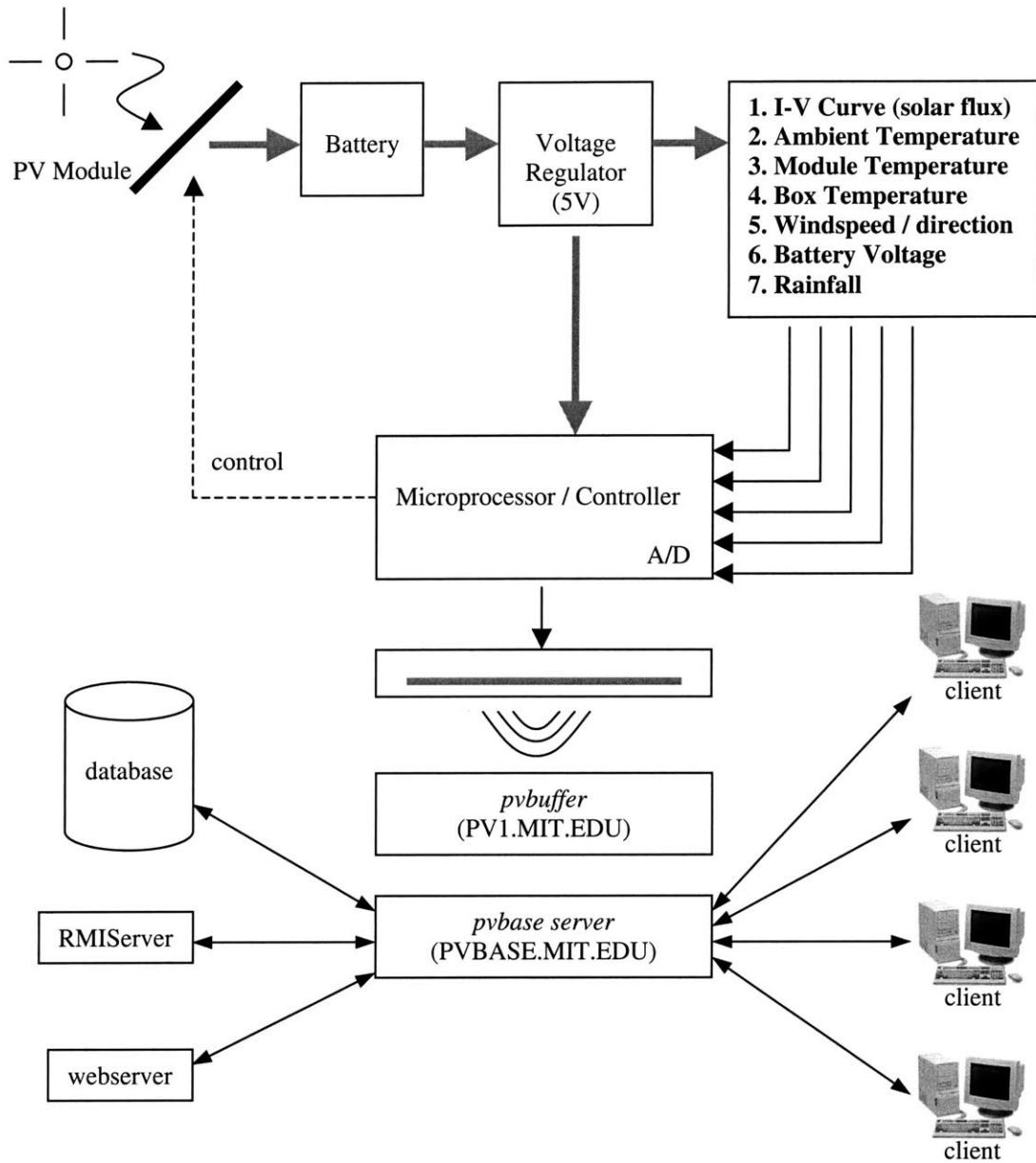


Figure 1: System overview

The overall objective of the project is to enable web-based presentation of real time and historic data for the purposes of PV system simulation. Programs have been written to read information from the (buffered) serial port of the server, parse the data, and store it in a database – all done in real time. Also, search and selection ability have been

implemented, so that a user anywhere in the world can access and download historic data; search for particular data sets; view incoming data in real time; simulate the performance of a PV system.

In pursuing this objective, several significant questions involving data storage, filtering, searching, optimization, real-time display, and system architecture were addressed initially:

- Should data be stored in a file system or a database?
- What structure should the storage system have?
- What data is relevant to store? How should data be filtered to eliminate noise?
- How can the data be searched and analyzed efficiently in spite of the large quantity of data stored? Can other optimizations be performed?
- How should incoming data from the weather station be displayed in real time?
- Should a cache and buffering be used?
- What system architecture – OS, webserver, and database – should be used?
- How should data be displayed?

There are five main areas of research:

1. **System Architecture** – determining hardware and software architecture, as well as decisions regarding the choice of a webserver, operating system, database, etc. The system architecture is discussed in detail in *Section 2.1*.
2. **Database / Data Model** – designing the structure of the database used in the project. The design considerations for the data model is discussed in detail in *Section 2.2*.

3. **Data Processing** – transmitting of data from the buffer machine, filtering noise for data integrity, uploading cleaned data into the database, converting of data, etc. A robust data archival scheme is discussed in-depth in *Section 2.3*.
4. **Database Interactions** – obtaining database connectivity, searching the database for desired data, remote method invocation, optimizing searches, etc. Interactions with the database are discussed in detail in *Section 2.4*.
5. **User Interface** – providing a simple interface for client / user applications to interact with real-time as well as archived data. The applets written for the project are described. Also, sampling and averaging considerations are discussed. The current interface between clients and data, whether real-time or archived, is explained in *Section 2.5*.

The completed online system can be visited at <http://pvbase.mit.edu/index.html>.

2 Design Overview

2.1 System Architecture

System architecture is the hardware and software platform that is used. The hardware platform is defined by specifying the processor, ram, hard drive, etc. The software platform is defined by specifying the operating system, type of database, webserver, etc. For this project, we decided to use PC systems running Windows 2000 and Windows

2000 Server³. The backend database and webserver is a SQL Server 2000⁴ and Internet Information Services 5.0 (IIS).⁵ The benefits of this system architecture include inter-compatibility, compatibility with existing systems⁶, cost⁷, and performance.

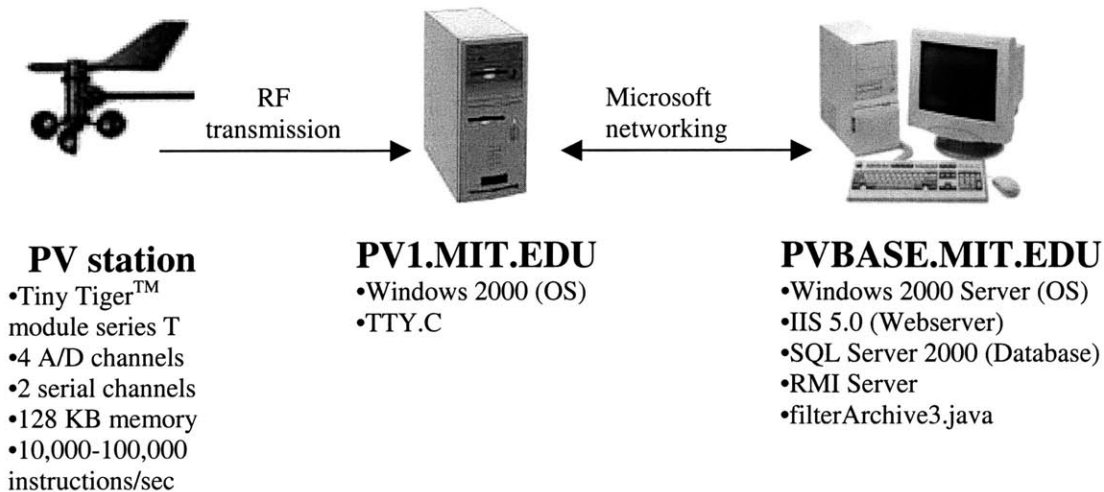


Figure 2: System Architecture

2.1.1 Hardware Architecture

PV1 . MIT . EDU (18.80.2.148) is a Windows 2000 machine, installed in a room in building 3, adjacent to building 1 and facing the PV station. It is an IBM Aptiva with a PI-150 processor, 32 MB ram, and a 2 GB hard drive⁸. These limited hardware specifications are sufficient for our purposes, as PV1 . MIT . EDU only needs to act as a

³ <http://www.microsoft.com/windows2000/guide/server/overview/default.asp>

⁴ <http://www.microsoft.com/sql/default.htm>

⁵ <http://www.microsoft.com/windows2000/guide/server/features/web.asp>

⁶ Currently, the Civil and Environmental Engineering Network (CEE-Net at <http://web.mit.edu/civenv/cee-net/index.html>) is largely comprised of Windows NT and 2000 machines.

⁷ Software provided by Microsoft, courtesy of David Mitchell

⁸ Graciously donated by Professor John Williams.

buffer machine. A simple C program, TTY.C⁹, runs continuously on PV1.MIT.EDU reading incoming information from its serial port (RF transmitter on the PV station sends data to a RF receiver connected to the serial port) and writing data to a file on the local hard drive.

PVBASE.MIT.EDU (18.38.1.96) is a Windows 2000 Server machine. It is a Dell Precision Workstation 330 with a P4-1.7 GHz processor, 512 MB ram, and a 36 GB hard drive. Installed on PVBASE.MIT.EDU are Windows 2000 Server, SQL Server 2000, and Internet Information Services 5.0. It continuously filters incoming data written to PV1.MIT.EDU and inserts into the database, acts as an RMI server (see section 2.4.2), acts as the webserver, and handles all incoming database requests from clients.

2.1.2 Software Architecture

Windows 2000 and Windows 2000 Server¹⁰ are used as the operating system. Since Windows 2000 is the operating system of choice for CEE-Net and most development work will be on Windows 2000 workstations, it made the most sense to keep Windows 2000 as the server operating system. We were thus restricted to use a webserver and database that are compatible with Windows 2000.

⁹ See Appendix J for instructions on restarting of tty.c.

¹⁰ Linux considered as an alternative: <http://www.redhat.com/software/linux/>

2.1.3 Webserver

For a webserver, IIS is used¹¹. A webserver such as IIS handles Hyper-Text Transfer Protocol (HTTP) requests. When one enters a URL into a web browser (e.g. Netscape or Internet Explorer), one performs an HTTP request for the webserver to retrieve and “serve” the desired web page. Most webserver allow interactions with a relational database and can generate web pages.

2.1.4 Relational Database

Microsoft SQL Server 2000¹² is used as a relational database, as it is designed for and runs on Windows 2000 Server. To perform many of the desired searches requires a relational database, a collection of data items organized as a set of formally-described tables from which data can be accessed or reassembled in many different ways without having to reorganize the database tables. The incoming information from the weather system is automatically inserted into a relational database. The information is stored on a file system as well for backup and archival purposes. Once the information for each

¹¹ Webserver options other than IIS include Apache¹¹, AOLServer¹¹, and others¹¹. Apache is compatible with both Windows 2000 and Linux, while AOLServer and most other webserver are mainly compatible with Linux

¹² Database options other than SQL Server 2000 include Oracle (<http://www.oracle.com/ip/dep/otn/database/oracle9i/>), MySQL (<http://www.mysql.com/information/index.html>), mSQL (<http://www.hughes.com.au/products/msql/>), PostgreSQL (<http://www.postgresql.org/index.html>), Sybase, and Informix. All of these major alternative relational databases products are developed to run under Linux – including commercial products such as Sybase, Oracle, and Informix; and open source software such as MySQL, mSQL, and PostgreSQL. Sybase and Informix databases are impractical to use because of their high cost. The open source databases are free; and Oracle is also viable because MIT has a license. To ensure compatibility with existing systems, cost effectiveness, performance, etc – the system architecture chosen was Windows 2000 Server, IIS, and SQL Server 2000.

record in a database is inserted, a standard Structured Query Language¹³ (SQL) statement is used to select the appropriate information from the relational database.

2.2 Database / Data Model

One benefit of using a relational database (Microsoft SQL Server 2000) is that the information is stored in an organized fashion. To organize this data efficiently, decisions regarding the structure of the database, what data should be stored, and how records should be indexed are crucial.

For the incoming data, two data models were considered. The first model would have a separate table for each day's information, each table indexed by time within the day. The second model would have one large table containing all of the data, each row indexed by day and time. Each of these designs has its advantages and disadvantages. The second data model is superior to the first because having a separate table for each day becomes inefficient and expensive.

The definition of the tables used to store data is called a "data model." A data model is a plan or map that defines the units of data and specifies how each unit is related to the others. A data model is contained in the database it describes and is available to any program that uses the database. The data model defines both the names of the data items and their respective data types. Here is a graphical depiction of the data model used.

¹³ <http://www.arsdigita.com/books/sql/>

Column Name	Data Type	Length	Allow Nulls
weather_detail_id	decimal	9	
[time]	datetime	8	✓
TA	float	8	
TM	float	8	
TB	float	8	
RN	float	8	
WD	float	8	
WS	float	8	
VB	float	8	
BF	bit	1	
IV	varchar	4000	✓
NP	float	8	✓
VO	float	8	✓
ISC	float	8	✓
VMAX	float	8	✓
IMAX	float	8	✓

Column Name	Data Type	Length	Allow Nulls
weather_hour_id	int	4	
[time]	datetime	8	
MPavg	float	8	
Iavg	float	8	

Figure 3: Data Model

Spending time and effort to design a “good” data model is worthwhile for various reasons. A good data model creates a database structure that allows for searches to be performed quickly and efficiently. Furthermore, a good data model is easily modifiable and maintainable.

A data model that is well designed is tailored to return data quickly for searches that are likely to be performed for a specific application. In the case of the PV station database, likely searches span different intervals of time. So the data model is designed with that in mind, as weather_detail is indexed by time. Indexing by time allows for those time interval searches to run faster because not all the rows in the table need to be examined.

The search simply needs to go to the index and where values in a certain interval are located.

A good data model also is easy to maintain and modify. Since it is likely that other students will continue work on this project and similar projects, easy maintenance and modifiability of the data model is important. Extra care is taken to name the tables and columns in an intelligent, consistent, and descriptive fashion. For example, the column for ambient temperature is “TA”, instead of something non-descriptive. Furthermore, “TA” and the other column names are consistent with the labels used on the PV station software installed on the roof. This may seem obvious, but it simplifies maintenance. Moreover, if the initial data model accurately represents the data that will be kept in the database, modification of the data model – adding new columns, deletion of columns, etc – is minimized. This is an important point, and minimizing modifications to the data model reduces later complexity.

Noise filtering is done prior to insertion into the database. This reduces insignificant entries in the table, which subsequently reduces the size of the database.

2.3 Data Processing

Data processing constitutes a major part of this project. In this real-time weather monitoring system, a data collection solution is developed. Our complete solution consists of both hardware and software components. Basically, hardware components on

the roof collect data; and then the data is transferred to servers for storage, filtering, analysis, and display with software.

Data is transmitted from the PV data field station to a buffer machine; data is read from the buffer machine into a separate database; and subsequently, relevant data is taken from the database and returned to a user on demand. Data transmission, data upload, searching, and filtering are discussed below.

2.3.1 Data Transmission

The PV powered field station continuously transmits data to an RF receiver connected to a PC machine, PV1.MIT.EDU at 4800 baud. The data must then be passed from PV1.MIT.EDU to the eventual webserver, PVBASE.MIT.EDU, where the data stream is read, filtered, and put into a database. Several different methods are available to accomplish this task.

To move the data from PV1.MIT.EDU to PVBASE.MIT.EDU, properties from both stream and datagram sockets¹⁴ are used to transmit data. We require reliable data transmission and packet transmission. Since there is no need for the data to arrive in a stream, we use a buffer to increase efficiency and lower network traffic. Reading are taken by the PV station every 2 seconds, so PV1.MIT.EDU acts as a buffer machine, holding data and sending packets of information every 2 seconds. In this manner,

¹⁴ See Appendix A for further details concerning sockets.

unnecessary network and database stress is avoided. Clearly, the buffer size needs to be carefully considered.

A program adapted from the Microsoft MSDN cdrom, TTY.C, facilitates serial communication. It resides on PV1.MIT.EDU and receives data from the PV station and writes that data into a local file. Microsoft Windows Networking facilitates method for data transmission. With Microsoft Windows Networking enabled, we simply map a drive of PV1.MIT.EDU onto PVBASE.MIT.EDU (where database is located) and have the database upload application on PVBASE.MIT.EDU read from that mapped network drive. This method is simpler than programming code to do this, as we can take advantage of the CEE-Net to accomplish the task of data transmission. Otherwise, one would need to write a program to achieve socket communication and possibly client/server communication code as well.

2.3.2 Data Upload

At PVBASE.MIT.EDU, incoming data is processed for real-time display and simultaneously inserted into the database.

There already exist data files that are on the file system of PV1.MIT.EDU. After the development phase of the system, these data files were uploaded into the database; this is done with a simple Java program. When the system is online and synchronized with the flat data files, live incoming data from the PV station is put into the database

continuously. The program that handles “live” incoming data regularly checks for new data, filters it, and inserts it into the database.

The periodicity of insertion into the database needs to be coordinated with the buffering interval. Simple SQL statements embedded in the Java upload programs (`filterArchive3.java` and `filterArchiveFlat.java`) are used to interact with the database.

```
INSERT into weather_detail
(time,TA, TM, TB, RN, WD, WS, VB, BF, IV, NP, VO, ISC, VMAX, IMAX)
VALUES ('2001-12-02
12:33:28',13.3744,12.4953,20.7288,0.0,
336.453,0.0,13.1828,0, null, null, null, null, null,
null);
```

Figure 4: Sample GO block Insert statement

```
INSERT into weather_detail
(time,TA, TM, TB, RN, WD, WS, VB, BF, IV, NP, VO, ISC, VMAX, IMAX)
VALUES ('2001-12-02
12:32:52',13.4248,12.5153,20.7222,0.0,
54.4919,0.0,13.1855,0, '0.374 2.315
0.374 4.758
0.354 9.902 ..... (100 pts) .....
0.0 19.29',100,19.3447,0.3839,16.075,0.335);
```

Figure 5: Sample IV block Insert statement

Batch insertion is a convenient way to insert multiple rows simultaneously into the database. Since there are approximately 40,000 rows inserted into the database daily, performing an insert every 2 seconds would cause unnecessary load on the database. We have grouped the inserts and perform one *batch insert* per x (x defined below) number of rows inserted. This significantly decreases load on the database by both decreasing the number of interactions with the database and increasing the amount of data inserted per interaction.

After every valid block is read from the incoming data stream, we add an insert statement to the set of batch statements into the database. A simple line in java does this.

```
stmt.addBatch(insertStatement);
```

We keep track of the number of inserts in the batch insert waiting to execute (`numBatchInserts`); and execute the batch of inserts when there are more than a certain threshold of inserts x waiting to fire. By buffering the inserts in this manner, we reduce the number of time-consuming interactions with the database.

```
if (numBatchInserts > x) {  
    numBatchInserts = 0;  
    result = stmt.executeBatch();  
    stmt.clearBatch();  
}
```

By increasing the threshold x , we essentially enlarge the buffer. This causes the system to appear less real-time as the information in the database is updated less frequently. By decreasing the threshold, we essentially shrink the buffer. This causes the system to appear more real-time. When x is 25, a connection to the database is opened approximately every minute. When x is 100, a connection to the database is opened approximately every 5 minutes. The database is updated more frequently at the cost of more interactions with the database. For the system, x is set to 25. In this manner, the data is up to date within a minute and appears to be real time; also the load on the database is not significant.

Errors in data collection at the PV station can cause noise to be introduced. Transmitting data from the PV station to `PV1.MIT.EDU` and from `PV1.MIT.EDU` to `PVBASE.MIT.EDU` also adds noise. This necessitates data filtering and noise elimination.

2.3.3 Data Filtering and Archival

Noise in the data is introduced as a result of hardware imperfections, transmission errors, and other sources. Two files – `ivblock.java` and `goblock.java` – help in filtering noise from out of the data. Every minute, the PV station takes 27 GO block measurements, followed by 1 IV block measurement. “GO” and “ZZ” encapsulate each GO block. “IV” and “ZZ” encapsulate each IV block. Within each GO block, 9 measurements are taken – “GO, TA, TM, TB, RN, WD, WS, VB, and BF.” See

figure 6. Within each IV block, 3 measurements are taken – “IV, VO, IS.” Each “IV curve” consists of 100 current / voltage measurements. After each GO or IV block, a “ZZ” follows to signify the end of the block. Within each block, their respective labels separate the measurements.¹⁵

```

...
...
GO 457690004
TA 7006
TM 5517
TB 6851
RN 52
WD 0
WS 146
VB 9103
BF 0
ZZ
...
...
...
IV
 97 35
 95 35
...(100 total pts)...
 7 154
 7 153
VO 9920
IS 6179
ZZ
...
...

```

Figure 6: Sample GO block

The program `filterArchive3` runs continuously in the background of `PVBASE.MIT.EDU`. As data is transmitted to `PV1.MIT.EDU`, `filterArchive3.java` reads the data, filters out noise, and inserts it into the database. In filtering noise from the incoming data, various checks are performed to ensure the validity of any given GO or IV block. The following conditions make a block invalid:

- More or less than exactly 9 labels and 9 corresponding values in a GO block
- More or less than exactly 3 labels and 3 corresponding values in a IV block
- Variable names occur in the incorrect order within a given block (eg: TM preceding TA)
- A measurement is lower or higher than the correct physical values (eg: BF, a bit value which is either a 0 or 1, is greater than 1)
- Consecutive GO block times are more than an hour apart¹⁶

¹⁵ See Appendix D for an example of a data file.

¹⁶ If this value less than an hour, a string of successive bad GO blocks could compound errors. Meaning that if two valid GO blocks followed each other by more than an hour (every GO block in between is

- A value cannot be converted to a valid number (eg: a TA value is “77b03”, instead of the proper “7703”)
- There are less than 90 or more than 100 IV points

These validity checks ensure that all data inserted into the database are free of noise. We gain validity at the cost of losing data points, since the incoming data set is large. There are over 38,000 GO blocks and over 1,400 IV blocks read each day.

2.3.4 Conversion details

Each variable is converted into meaningful values before insertion into the database. For example, we store the temperatures in degrees Celsius and not in a sum of 16 word values – the form in which the values are transmitted – between 0 and $16 \cdot 2^{10}$. A prior conversion increases efficiency because the values do not need to be converted repeatedly when different users request overlapping data sets.

Specific conversion details are included in Appendix G.

2.3.5 Threading

Incoming data from the PV station is continuously archived with the aid of threads. At the beginning of each day, a new file is created by the `TTY.C` program running on `PV1.MIT.EDU`. For that entire day, data is written to that file continuously. At the beginning of each day, data is written to a new file. Even though the data is no longer

somehow corrupted), every GO block afterwards would be declared invalid, despite the fact that they are actually valid.

being written to the same file, the stream of incoming data from the PV station still needs to be archived into the database. The variables in `filterArchive3.java` need to be reinitialized and another instance of `filterArchive.java` needs to be started for the new file. We solve this problem simply by starting a new thread that will begin archiving data that is being written to the new file for the following day.

```
Thread t = new Thread(new filterArchive3());  
t.sleep(60000);  
t.start();
```

A new thread is started when the current day on the server (`PVBASE.MIT.EDU`) changes. For example, when the server clock changes from ‘2001-10-10 23:23:59’ to ‘2001-10-11 00:00:00’, a new thread is started to begin archival of the next day’s data. We use threading also so that there is not a pause in the archival of data. This allows for real-time behavior to continue even at midnight, despite the fact real-time behavior is not crucial at midnight because it is not a high traffic time.

Note that the thread is forced to wait – `sleep(300000)` – for 5 minutes before starting the new thread. This is to allow for any possible delay that `TTY.C` may have in creating a new file for the following day. Also, this eliminates the situation where we try to read a file that is non-existent. If the new file is still non-existent after 5 minutes, a problem exists on `PV1.MIT.EDU` and maintenance is required.

2.4 Database Interactions

2.4.1 Database Connectivity

We need to establish connectivity to the relational database in order to upload data and select data. Furthermore, the database connectivity needs to work with Java – our programming language of choice. We use both the Open Database Connectivity (ODBC) application programming interface (API) along with the Java Database Connectivity (JDBC) application programming interface specifications to connect our Java programs to our database. For our purposes the JDBC-ODBC bridge that comes with the Java SDK works well. The only disadvantage is that an ODBC data source needs to be added¹⁷ for every machine requiring database access. The API allows database access statements written in SQL to be encoded and then passed to a program managing the database.

There are two methods for client database access. The first method requires a client to add the ODBC data source and download a JDBC driver every time an applet is requested. The second method has a server perform database queries and simply transmit results to the client. It was decided that it is best to run the queries on a server for efficiency and security reasons.

Our data source, Microsoft SQL Server 2000, is compatible with the Microsoft supported ODBC driver. We add this ODBC data source on the Windows 2000 server machine that will access the database, PVBASE.MIT.EDU. ODBC allows programs to use SQL

¹⁷ To do this, go to the “Control Panel”, click on the “Administrative Tools” icon, click on the “Data Sources (ODBC)” icon, click on “add” under the “User DSN” tab, select the “SQL Server” option. Then configure the data source / database as desired.

statements accessing databases to do so without knowledge of proprietary database interfaces. Basically, ODBC will take the incoming SQL request and convert it to something that the individual database – SQL Server 2000 – understands.

The standard JDBC (Java Database Connectivity) API¹⁸ is used allow our Java programs to connect to the database. A class for database access was created, called `Database.java`. In the class, the driver is registered with the driver manager one time, and then applets and other programs that need a database connection call a static method to return a connection to the database. This makes changing database drivers easier. Also, connection pooling can be easily implemented. We can use a simple line in Java in `Database.java` to create a bridge between JDBC and ODBC.

```
DriverManager.getConnection("jdbc:odbc:weather",  
                             "test",  
                             "weathertest");
```

This line attempts to establish a connection to the database specified in the first argument. In the code sample, we are using JDBC to connect to an ODBC database “weather”, which resides on the localhost `PVBASE.MIT.EDU`. The second and third arguments are the username and password required to access the database. Essentially, our Java programs use a JDBC-ODBC bridge to connect to a SQL Server 2000 database.

¹⁸ Initially, I attempted to use the `aveConnect 2.3` for MS SQL Server driver to establish connectivity: <http://www.atinav.com/products/aveconnect/aveconnect.htm>

2.4.2 Remote Method Invocation (RMI)

For our online system, every client will be accessing the database. From the previous discussion, it would seem that each client then would be required to obtain database connectivity through the JDBC-ODBC bridge. Each end user for the online weather system should not be required to add the data source and obtain connectivity directly with the database. Moreover, requiring each end user to connect to directly to the existing relational database can compromise the security of the system. We use the Java Remote Method Invocation (RMI)¹⁹ framework to make interaction with the database transparent to the end user so that each client does not need to add the data source and connect directly to the database.

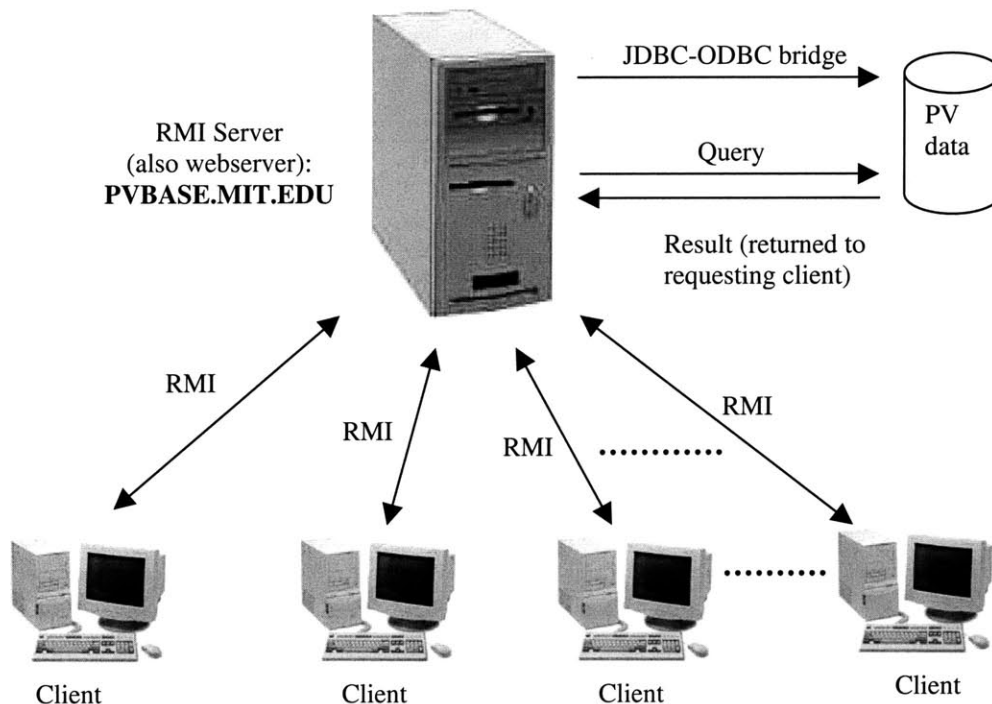


Figure 7: RMI's client-server architecture

¹⁹ <http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html>

RMI provides an easy way for Java programs to communicate with each other and allows for distributed applications. Basically, RMI is Java's implementation of RPC (remote procedure call) mechanism. RMI allows a client program (applet running on the machine of an end user) to call methods that belong to a remote object, living on an RMI server, PVBASE.MIT.EDU. The client program passes arguments (ie: search criteria) to the methods of the remote object and obtains return values (ie: search results), as if the client program is invoking methods of a local object.

The operation of a remote method call proceeds quite simply. The client program calls a dummy method (stub) residing locally. The stub gets the function arguments, serializes them (aka "marshalling") and then sends them over the network to the RMI server. On the server side, a corresponding method (skeleton) deserializes the arguments (aka "unmarshalling") and passes them to the real server method. The server method finally processes the function call. The RMI server serializes the results, sends the results back over the network, the client receives and deserializes the information.

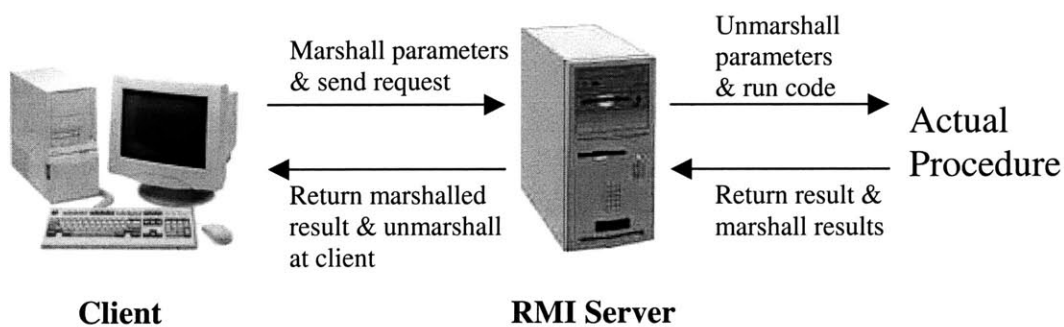


Figure 8: Marshalling and Unmarshalling

To implement RMI, both the server and clients must agree on a shared interface. This interface describes the methods that a client can invoke to be run on the server. In the weather system, we have *RMIInterface.java*:

```
import java.rmi.*;
import java.util.Vector;
import java.sql.SQLException;
import java.io.IOException;

public interface RMIInterface extends Remote {

    Vector getData(String var, Date startDate, Date endDate, int
        method) throws RemoteException, SQLException;
    Vector getTime(int method) throws RemoteException;
    String getRealTimeGO() throws RemoteException, IOException;
    String getRealTimeIV() throws RemoteException, IOException;
    ivcurve getOneIVCurve(Date d) throws RemoteException,
        SQLException;
    long getServerTime() throws RemoteException;
}
```

The client call to *Naming.lookup()* returns a reference, or a stub, for the *java.rmi.Remote* object associated with the specified URL string. The remote object is located on the RMI Server, where *myInterface* is an *RMIInterface* remote object as defined above. Now, methods defined on the server are run through the Remote object on the client as if they are local methods. The following code returns a reference to the remote object and returns the data between *startDate* and *endDate* for variable *var* into a vector *datav*.

```
myInterface = (RMIInterface) Naming.lookup("//pvbase.mit.edu/RMI");
datav = myInterface.getData(var, startDate, endDate, method);
```

The RMI Server makes its services available to clients by registering them with the RMI registry, using the *Naming.rebind()* call. Also, all methods defined in the interface – *getData* (shown below), *getServerTime*, etc – are all implemented in *RMIServer.java*. Here is a simplified version of *RMIServer.java*:

```
public class RMIServer extends UnicastRemoteObject
    implements RMIInterface {
    public RMIServer() throws RemoteException, SQLException {
        super();
    }

    public Vector getData(...) throws SQLException { ... }

    public static void main(String args[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            RMIInterface server = new RMIServer();
            Naming.rebind("//localhost/RMI", server);
        } catch(Exception e) {
            System.exit(-1);
        }
    }
}
```

As a result of using RMI, only *PVBASE.MIT.EDU* needs to have direct connectivity with the database. This increases security as RMI contains security measures that protect system integrity. Furthermore, RMI allows objects (in this case the database query results) to be easily passed between client and server. Also, RMI allows the RMI Server to handle multiple clients simultaneously, as RMI is multi-threaded. See Appendix I for instructions on how to start the RMI registry and RMI Server.

2.4.3 Searching

After data from the PV station has been inserted into the database properly, SQL statements are used to search and retrieve desired data. *RMIServer.java* acts as a database API that is used to abstract the details of database search and selection from the client. Using SQL, we can perform a variety of desired functions:

- select only desired data between two dates
- determine the average temperature for any given day
- ascertain which days had a cumulative rainfall above 3 inches
- and many more things...

Various SQL statements are used for insertion and selection of data from the database.

To select data from the database, a number of different SQL statements are used depending on what the end user has requested. For example, if the end user wants to view TA values from December 1, the following SQL statement is used:

```
SELECT time, TA
FROM weather_detail
WHERE TA IS NOT null
AND time > '2001-12-01 00:00:00'
AND time < '2001-12-02 00:00:00';
```

If the end user wants to view the IV curve taken at noon on December 1, the following SQL statement is used:

```
SELECT VB, IV, NP, VO, ISC, VMAX, IMAX
FROM weather_detail
WHERE time = '2001-12-01 12:00:00';
```

More complicated sets of data can also be requested. If the end user wants to view the max power values – the largest current times voltage product for each IV curve (taken every minute) – for December 1, the following SQL statement is used:

```
SELECT time, vmax*imax as MP
FROM weathertest_detail
WHERE vmax*imax IS NOT null
AND time > '2001-12-01 00:00:00'
AND time < '2001-12-02 00:00:00';
```

2.4.4 Optimization

There is a large quantity of data to be stored into the database. For this reason, the structure of the database and search optimization is of utmost importance. The most direct way to speed up selection of data is to use an *index*. An index is essentially a structure of pointers that point to rows of data in a table. An *index* optimizes the performance of database queries by ordering rows to speed access. To understand what a database *index* does, a simple analogy is the index of a book. To find something in the book, you simply flip to the index of the book and look up the page number the desired subject is located on. For a database, the idea of an *index* is the same – examining the *index* tells you where in the table the desired information is located.

A *clustered index* is created so that the data its respective table is ordered by the column(s) on which the index is defined. The difference between *clustered indexes* and non-clustered indexes is that a *clustered index* acts as a normal index **and** physically reorders the table so that the rows in the table are ordered based on the index fields. By

clustering data together in the same area physically on the hard drive, we take advantage of memory hierarchy to speed up searches. In computer systems, data is stored in “pages.” These “pages” of data are moved from slower memories (hard drive, etc) to faster memories (caches and RAMs). Moving “pages” of data from slower memories to faster memories is extremely time consuming. When a select statement occurs, the database program requests data from the memory manager, which in turn needs to move a page(s) of data from slow to fast memory. Because a “page” of data may contain many many rows of data, we can speed up searches if the entire page contains relevant information returned by the database search statement. When a select statement is performed, the database engine reads a page(s) of information. Because of the clustered index, those pages will consist of many rows of relevant data. Clearly only one *clustered index* can be created on a table because of the physical ordering that is performed, so choosing the correct column to create a *clustered index* on is important.

To speed up the searches, we create a *clustered index* on the `time` column²⁰. This *clustered index* will cause the rows of data to be physically stored in order of time. Since the majority of searches we perform are based on selection of a range of `time` values, the *clustered index* created on this column speeds up the search tremendously. With this index, the entire table does not have to be scanned as SQL Server can go straight to the location of desired rows and extract them. On the other hand, querying the table based on a desired value of temperature (eg: `TA`), SQL Server needs to examine each row of the

²⁰ See Appendix F for the formal definition of the data model. See Figure 3 for a graphical representation.

table to see if each row matches the desired criteria. Scanning the entire table needs is extremely costly and slow.

For example, when we query the `weather_detail` table for the all the data that has a time later than '2001-11-01' and before '2001-11-02', SQL Server can go straight to the relevant row and extract them because of the *clustered index* created on the `time` column. Not only does SQL Server go straight to the correct rows, the data is ordered physically as well. So the number of "pages" that the memory manager needs to move is minimized, which is crucial because moving data "pages" around is extremely slow.

On the other hand, querying `weather_detail` for all rows where TA is greater than 70 degrees, SQL Server needs to scan the entire table and check each row to see if the Ambient Temperature value is greater than 70. Since there are millions and millions of rows in the table, this query takes a great deal of time, as we do not create an index on this column since searches of this kind are never performed. If this type of search becomes often performed, a *non-clustered index* – but not a *clustered index* because only one can exist per table – on that column can be created to speed up the query.

Further, a *concatenated index* is an index created on more than one row.²¹ So if queries with criteria on more than one column are performed, a *concatenated index* can be created. Further optimizations can also be performed to speed up queries if needed.²²

²¹ Harrison, Guy, *Oracle Sql High-Performance Tuning*, Prentice Hall Computer Books, January 1997.

2.5 User Interface

Finally, the system allows display of user-selected data through applets and a website.

These applets have all been designed with targeted end user – researchers and students – in mind. Factors such as usability, clarity, simplicity, speed, etc have been considered in the design of the website and applets.

2.5.1 Java Applets

Java applets are used allow users to search, analyze, and display collected data. Applets are Java programs that can be included in an HTML page. The inclusion process is simple, as applets are included within an <APPLET> tag much in the same way an image is included with the tag. When a client runs an applet on a browser, the applet's code is sent to the client's machine. Subsequently, the client browser's Java Virtual Machine runs the applet code on the client machine.

2.5.2 Compatibility / AWT only

For greater browser compatibility, only AWT java is used. No Swing java is used in any of the applets written. AWT java contains classes that help with designing graphical user interfaces. Buttons, scrollbars, lists, checkboxes, and text fields are all components of the AWT package. Furthermore, it contains a graphics class that has methods painting graphics and images.

²² <http://msdn.microsoft.com/workshop/server/feature/tune.asp>

Also, AWT java allows for handling of outside events, such as clicking on a button and the movement of the mouse. For example, clicking on the “Graph” button causes a method to be run which retrieves data and draws it on the applet.

AWT java is compatible on more browsers than Swing java because it has been installed on, tested on, and supported by more browsers (the most recent release of the Java Development Kit – JDI – is 1.3.1. The AWT classes have been included as far back as JDK 1.0, so the classes are available on more browsers). Specifically, Athena machines on the MIT campus do not support Swing, as a plug-in needs to be installed before swing applets can be deployed on Athena browsers.

2.5.3 RealTime.java

RealTime.java is an applet that is embedded in the main page of the webpage (<http://pvbase.mit.edu/test/main.html>). When a user goes to the main page, real time information is displayed by this applet. Also, links to graph more detailed data are present. The real time information refreshes every 2 seconds, as that is also how frequently data is collected.

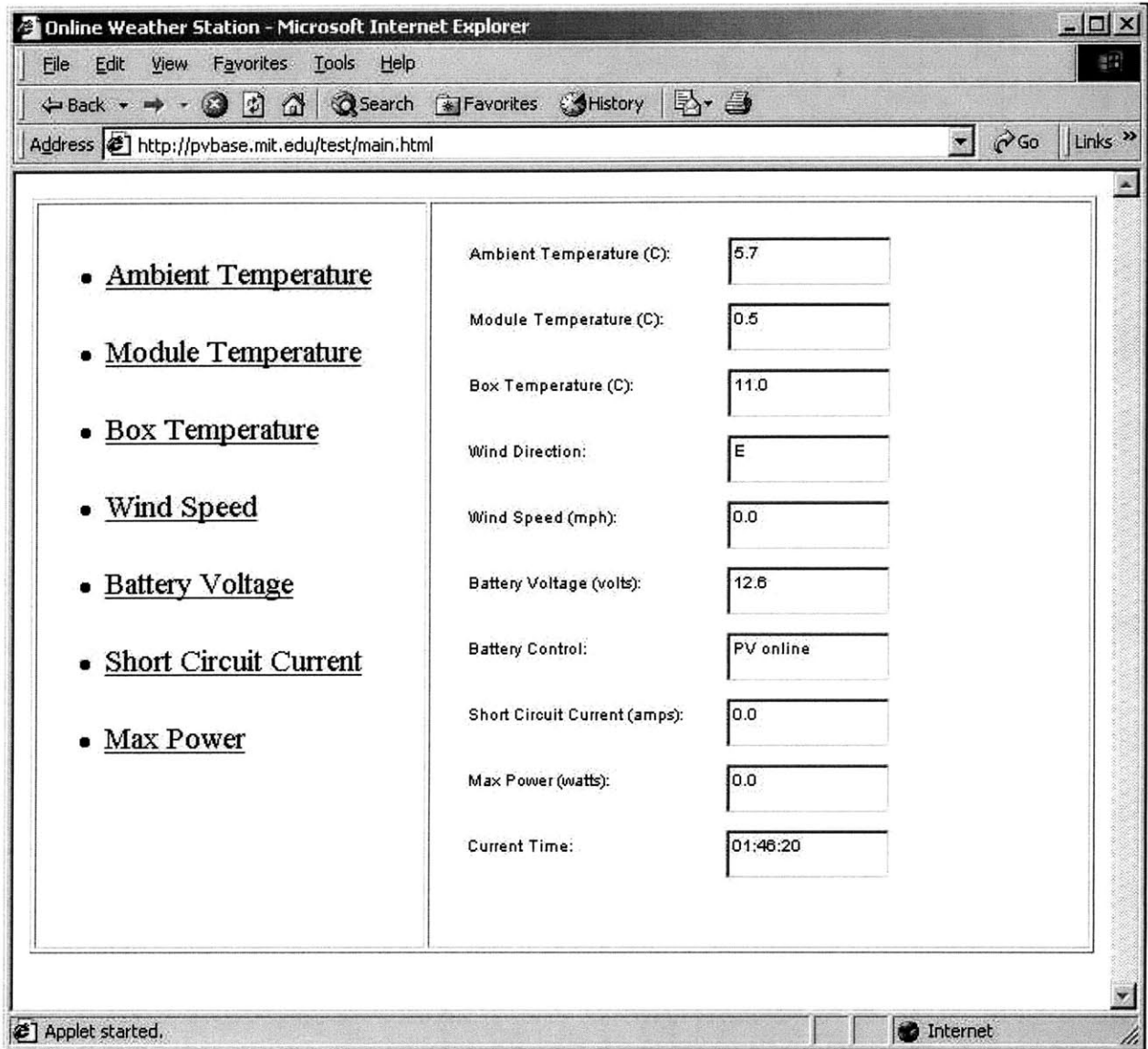


Figure 9: Main.html real-time display applet

RealTime.java functions similarly to Grapher.java in that it also will obtain an RMI object (see section 2.4.2). After obtaining this RMI object, RealTime calls a method of the RMI object asking it to return real-time information. Once the real-time information is returned, it is promptly displayed to the user.

2.5.4 DataGrapher.java / Grapher.java

DataGrapher.java is the main applet that is used to graph data that is pulled out of the database. It contains various labels, textfields, select boxes, and buttons. Most importantly, it contains a canvas called Grapher.java, on which all of the graphing takes place.

The DataGrapher applet interacts with the user to obtain information concerning the variable to be graphed, the interval to be graphed. Once these inputs are determined, DataGrapher passes the inputs to Grapher.java. Grapher is the canvas that subsequently graphs the data requested by the user.

Grapher obtains an RMI object (see section 2.4.2) and asks the RMI Server to return the proper data from the database. Once the data has been returned, various plotting and graphing routines are used to draw the graph on the canvas. Having a separate Grapher class is beneficial because of modularity and abstraction. If we want to have other instances of a graphing canvas that have different properties (such as size, axis scaling, graphing different variables, etc), we can simply create it with another instance of Grapher. For example, <http://pybase.mit.edu/test/ISC.html> contains two instances of Grapher, one with a dimension of 450 pixels and one with a dimension of 200 pixels. The 450 x 450 pixel instance graphs the short circuit current values, while the 200 x 200 pixel instance graphs the IV curve desired.

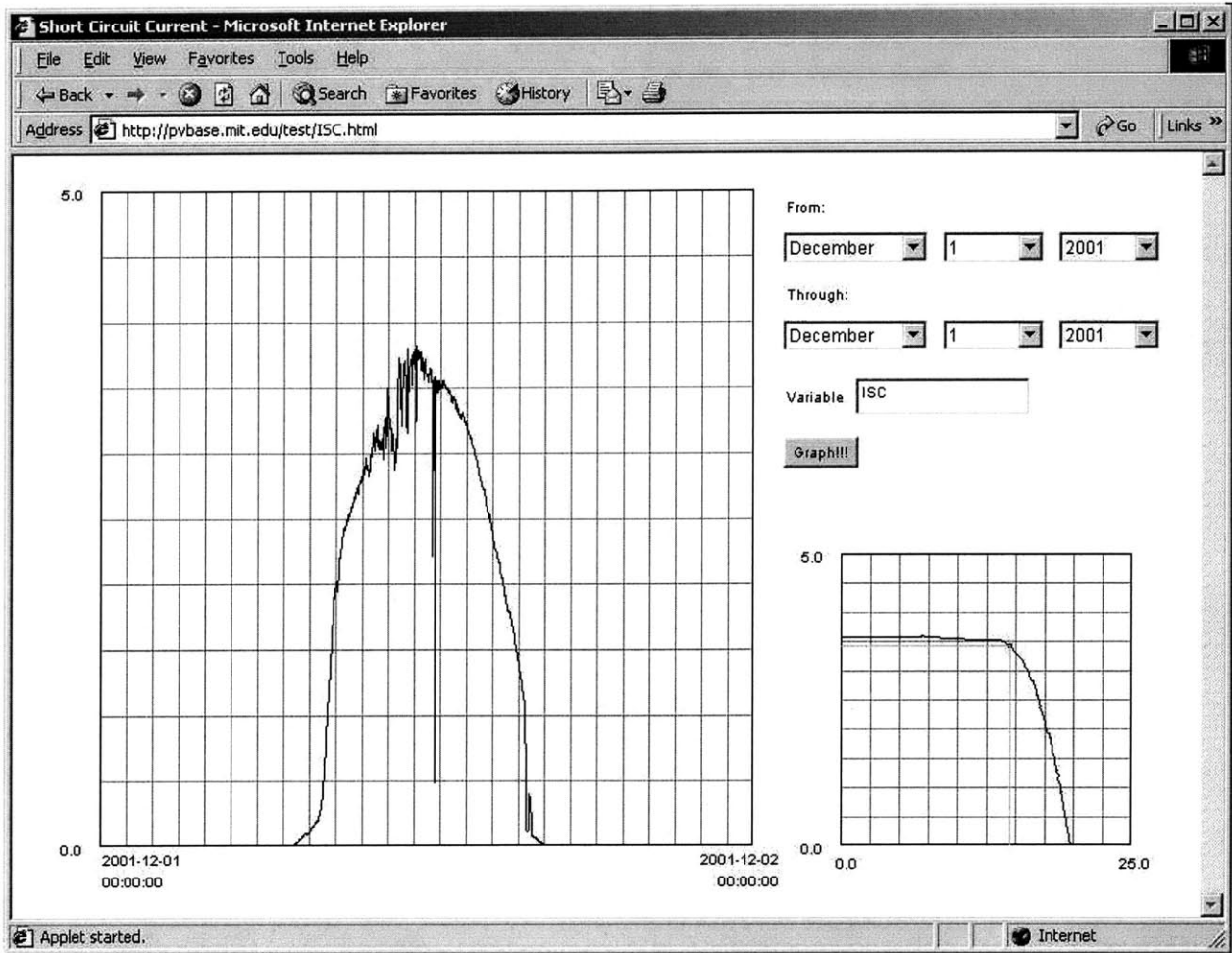


Figure 10: Screenshot of ISC.html

Another example is that the instance of `Grapher` at <http://pvbase.mit.edu/test/TA.html> has the y axis scaled from -20 to 60 , while another instance of `Grapher` at <http://pvbase.mit.edu/test/VB.html> has the y axis scaled from 10 to 16 .

2.5.5 Sampling and Averaging

Large quantities of data combined with a relatively small viewing size for graphs necessitate the need for averaging or sampling of some sort. For example, if a user wishes to view information concerning the battery voltage for December 1, 2001, there

are approximately 40,000 measurements taken on that day. The applet for graphing the data has a set viewable area of 450x450 pixels. Clearly, 40,000/450 gives you approximately 10 points per pixel. Variation in the 10 points will draw vertical lines. Here is the graph of battery voltage for 12/01/2001, with all of the points graphed.

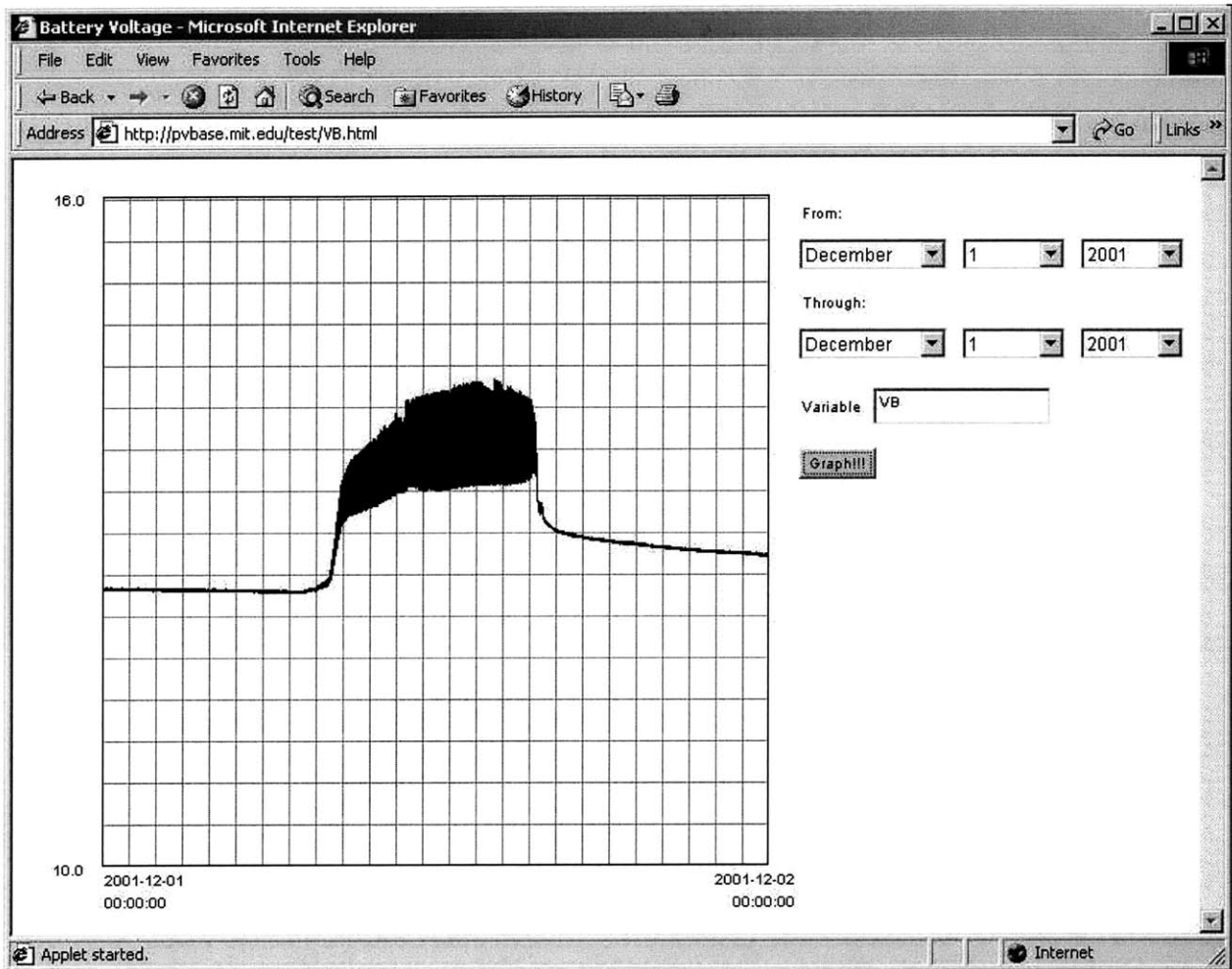


Figure 11: Graph of battery voltage - all points

The battery control is going on and off; this causes fluctuation in the battery voltage measurements. Putting time on the x axis and temperature on the y axis, it is clear that only 450 different data points are displayed to the user. Java can only plot on integer pixel values and the monitor only has a fixed number of displayable pixels. We cannot

display accurately more than that many points. The graph will not be able to show distinctions between two data points plotted on the same x coordinate.

Interpolation offers one solution. If the query returns n rows, we simply plot every $n/450$ th point. Thus, there are exactly 450 points plotted on the graph, each clearly distinguishable from the others. This not only makes points clearly distinguishable from each other, it significantly speeds the plotting of data, as exactly 450 points – instead of n points, where n can be orders of magnitude larger than 450 – are plotted.

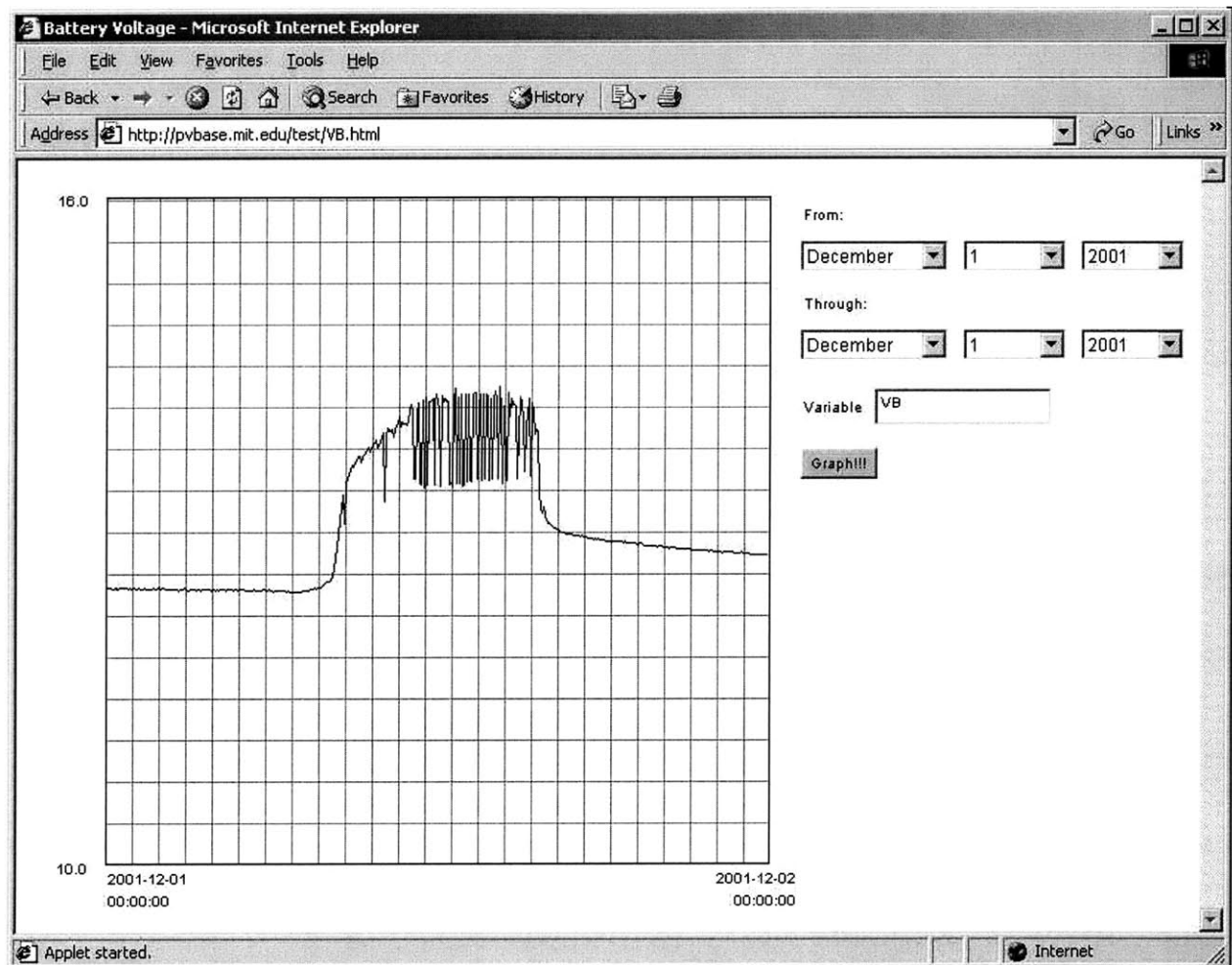


Figure 12: Graph of battery voltage - 1 point per pixel

Undoubtedly, the graph is cleaner and has plotted more quickly. But, some significant data has been lost because of the interpolation performed. The fluctuations that are oftentimes significant have been lost. For certain variables like battery voltage, this is oftentimes desirable. Another alternative is to graph more than 1 point per pixel to show the fluctuations in measurement. Below is the same plot, but with 8 points per pixel.

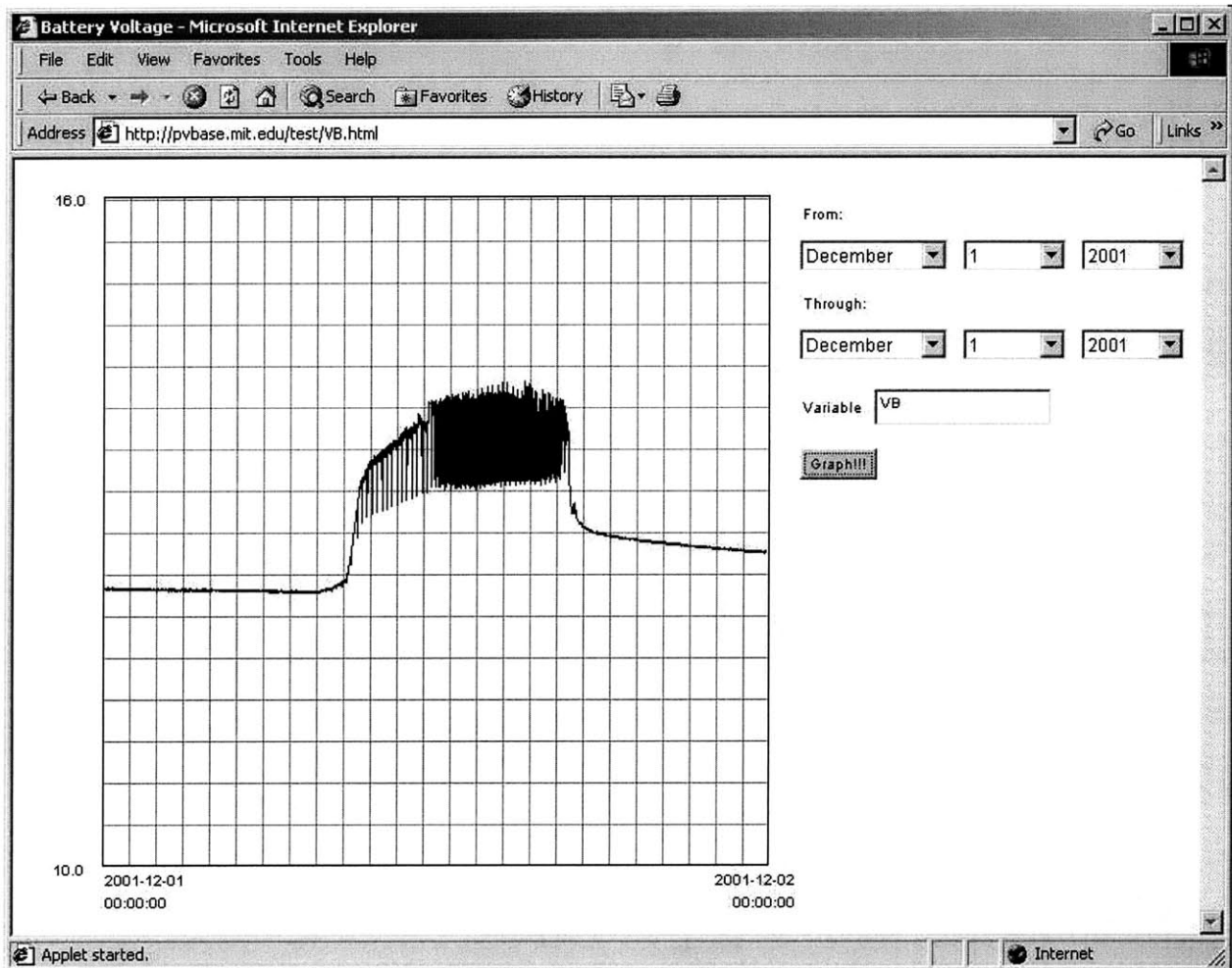


Figure 13: Graph of battery voltage - 8 points per pixel

Another viable alternative is to average the points over a given interval, and still plot 450 averaged data points. For example, we can split the query results into $n / 450$ intervals, where n is the number of points returned. Then, within each interval, we can average

over a specified number of evenly spaced data points. This average value would be the value plotted for the interval. Here is the same graph with 450 points plotted, each point the average of 8 data points – the same 8 points graphed above per pixel – evenly spaced within the interval.

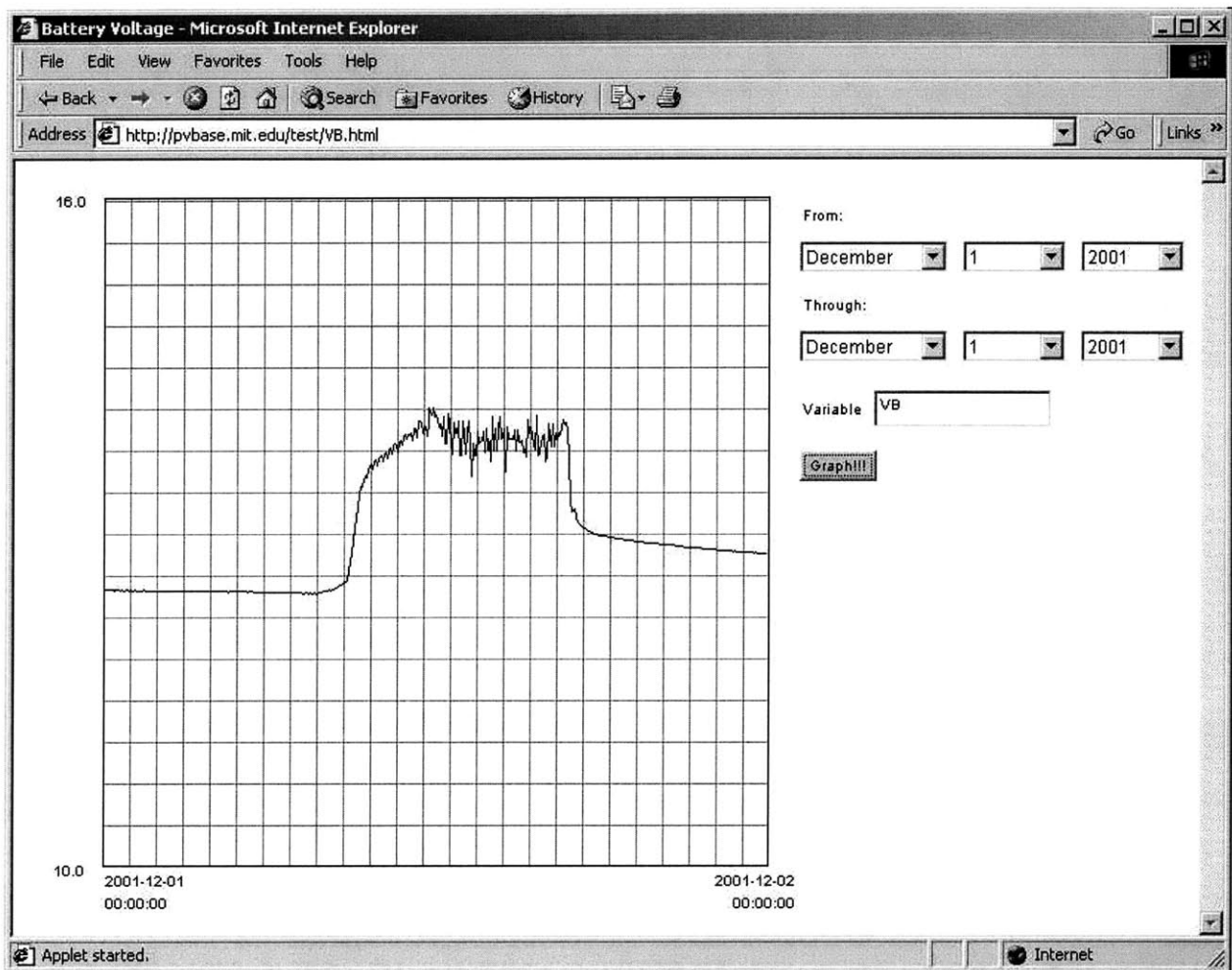


Figure 14: Graph of battery voltage - 8 points per pixel w/ averaging

However, oftentimes averaging and sampling schemes do not reveal fluctuations and excursions in the data set graphed for the user. These fluctuations may be reveal relevant information concerning the graphed data points. Another scheme for data display involves graphing 3 curves – one curve depicting the minimum values per interval, one

curve depicting the average values per interval, and one curve depicting the maximum values per interval.

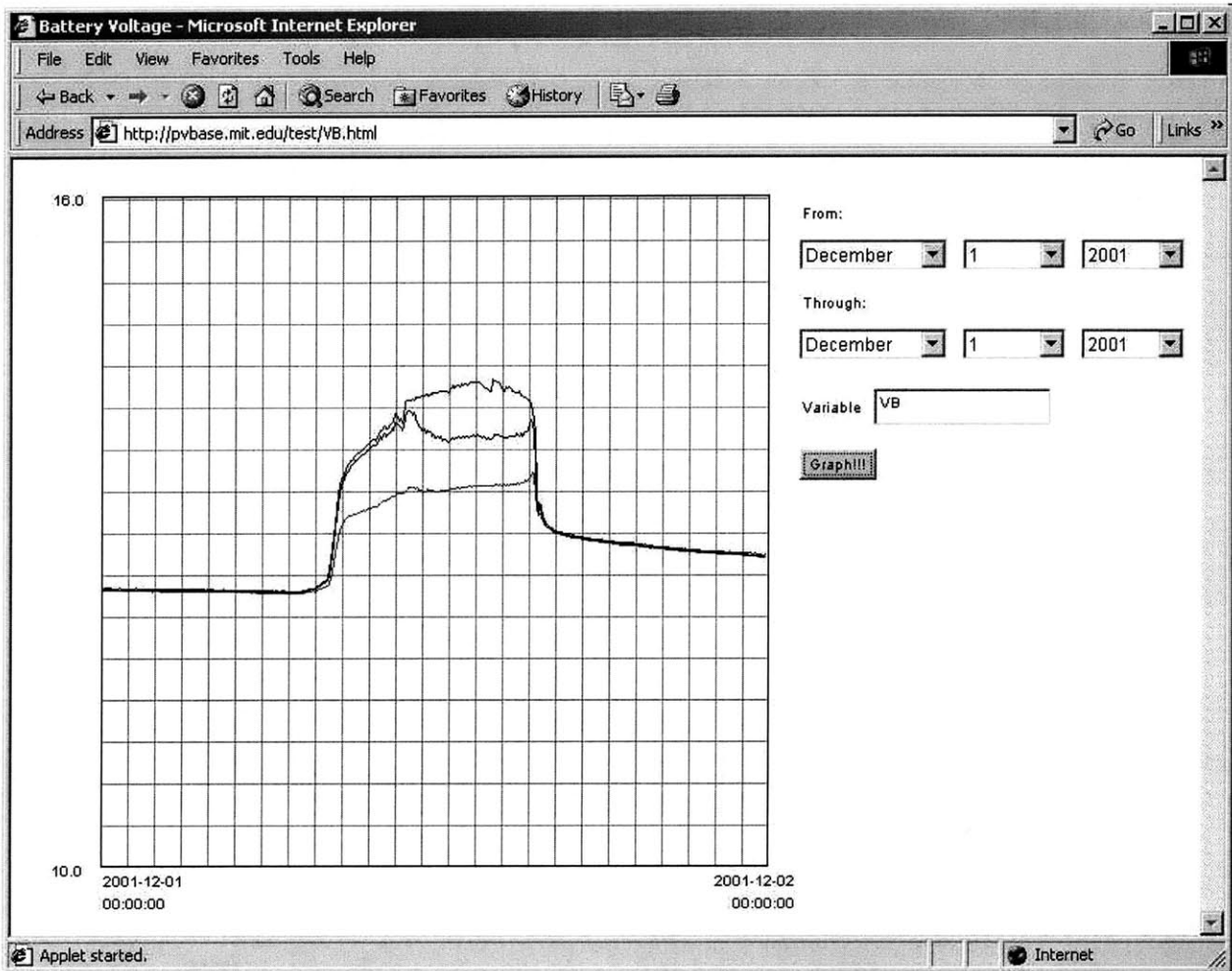


Figure 15: Graph of battery voltage - Min, average, max scheme

Essentially, three curves are drawn. The top curve represents the maximum value in the interval. The middle curve represents the average value of the points in the interval. The bottom curve represents the minimum value in the interval. This scheme – and schemes not involving averaging – allows major fluctuations to be clearly seen.

3 Conclusions

Many other difficulties – technical and nontechnical – also arose during the development of the system. When trying to obtain a new IP address for PV1.MIT.EDU, a conflicting roaming IP address was difficult to deal with. Much time was spent determining this problem and corresponding with MIT Information Systems. Also, the code red worm²³ infected PVBASE.MIT.EDU, forcing me to reinstall the operating system and install the necessary security patches. Furthermore, the limited number of pixels to graph on gave rise to the need to figure out averaging and sampling schemes for graphing. Also, since Java Virtual Machines (JVM) are not uniform across browsers, a we required that a plugin²⁴ had to be installed on client browsers. See Appendix H for information about how a HTML Converter was used to do this.

After completing this project, I am confident that a robust system has been put in place. All of our initial concerns have been addressed properly, and details have been handled appropriately. I hope that this thesis gives some insight into the large amount of work put into the project; and how it can serve as a guide to future students – students continuing this project and students building systems similar to this project.

²³ http://www.cert.org/incident_notes/IN-2001-09.html

²⁴ <http://java.sun.com/products/plugin/>

Appendix A - Sockets

“Stream sockets” are reliable, error-free two-way connected communication streams that use the Transmission Control Protocol (TCP). If you output two items into the socket in the order "1, 2", they will arrive in the order "1, 2" at the opposite end. Applications such as telnet, FTP, and the HTTP protocol use stream sockets. “Datagram sockets” use Internet Protocol (IP) routing and are connectionless and unreliable. With datagram sockets, you don’t need an open connection as you do with stream sockets. Simply build a packet, an IP header and other relevant destination information, and send it out. No connection is needed. Datagram sockets are generally used for packet-by-packet transfers of information. Applications such as NFS and tftp use datagram sockets.²⁵

²⁵ Bonner, Patrice, *Network Programming with Windows Sockets*, Prentice Hall, September 1995.
Quinn, Bob, *Windows Sockets Network Programming*, Addison Wesley Longman, Inc., November 1995.
<http://www.ecst.csuchico.edu/~beej/guide/net/>

Appendix B – Online Resources

PV Resources

- <http://www.sandia.gov/pv/> - Sandia National Laboratories Photovoltaics Program
- <http://www.fsec.ucf.edu/PVT/> - Florida Solar Energy Center
- <http://www.eren.doe.gov/pv/> - U. S. Department of Energy Photovoltaics Program
- <http://www.pv-uk.org.uk/> - British Photovoltaic Association (PV-UK)

Technical Resources

- <http://java.sun.com/j2se/1.3/docs/api/overview-summary.html> - Java™ 2 Platform, Standard Edition, v 1.3.1 API specification
- http://java.sun.com/products/plugin/1.3/docs/html_converter.html - Java™ Plug-in 1.3 HTML Converter
- <http://www.sql-server-performance.com/> - SQL Server Performance Tuning and Optimization
- <http://web.mit.edu/civenv/cee-net/index.html> - CEENET

Appendix D – Sample data file from 10/25/01

Data for 10_25_01.dat

GO 457688004
TA 7003
TM 5525
TB 6863
RN 52
WD 7
WS 144
VB 9098
BF 0
ZZ
GO 457690004
TA 7006
TM 5517
TB 6851
RN 52
WD 0
WS 146
VB 9103
BF 0
ZZ

.....

.....

GO 514434004
TA 8201
TM 7197
TB 8007
RN 94
WD 12594
WS 239
VB 10017
BF 1
ZZ
IV
97 35
95 35
96 35
96 55
95 79
95 97
93 106

..... (100 data points in IV curve)

10 153
9 153
9 153

8 154
8 153
7 154
7 153
VO 9920
IS 6179
ZZ
GO 514442004
TA 8196
TM 7208
TB 8001
RN 94
WD 11096
WS 2
VB 9857
BF 0
ZZ
GO 514444004
TA 8488
TM 7483
TB 7998
RN 94
WD 13484
WS 6
VB 11159
BF 1
ZZ
GO 514446004
TA 8187
TM 7224
TB 7998
RN 94
WD 11505
WS 10
VB 10392
BF 1
ZZ
GO 514448004
TA 8195
TM 7231
TB 7998
RN 94
WD 11514
WS 14
VB 10181
BF 1
ZZ

.....

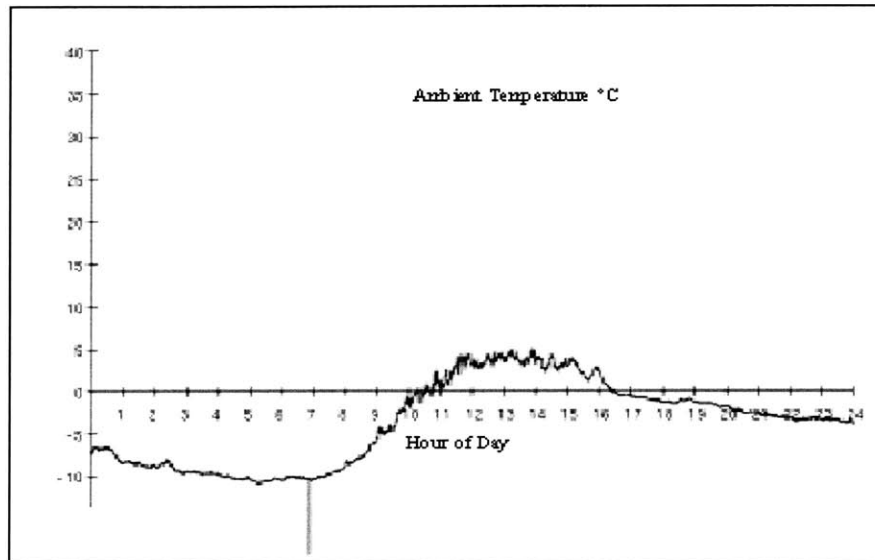
.....

.....

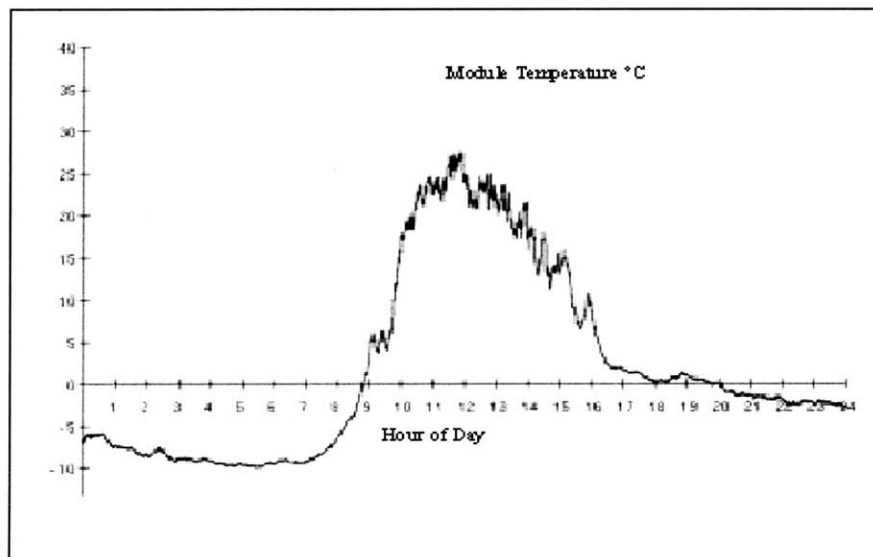
Appendix E – Sample Data

Sample Data from a Photovoltaic Powered, stand-alone, station atop Building 1, MIT

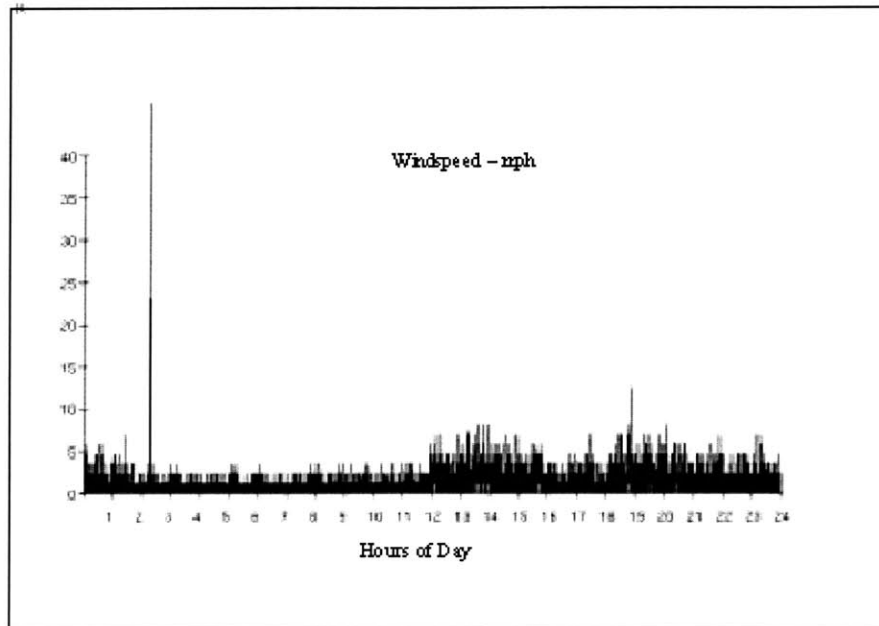
Ambient Temperature 13 January, 2001 More "cleaning" of data still needs to be done.



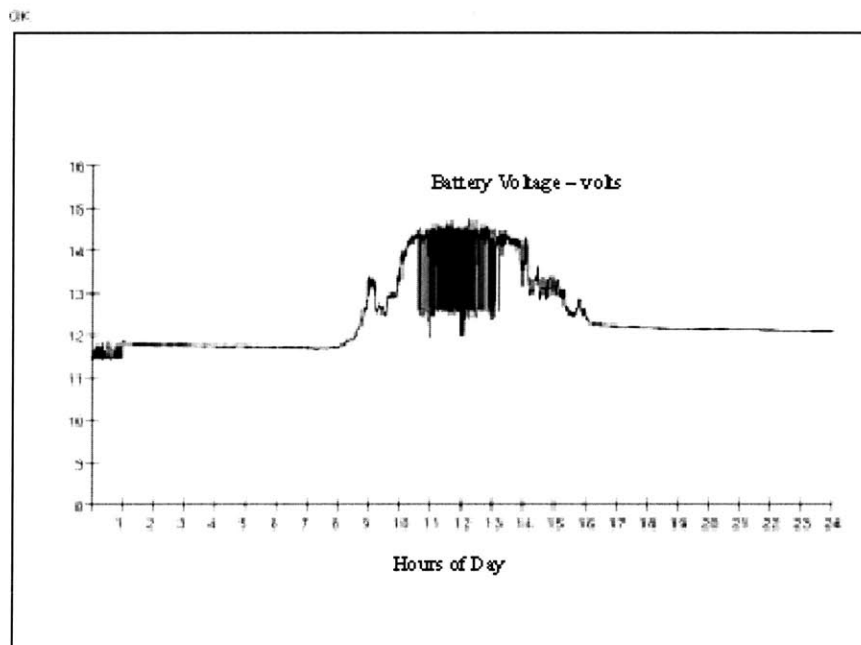
Module Temperature 13 January, 2001. The module temperature tracks the module power.



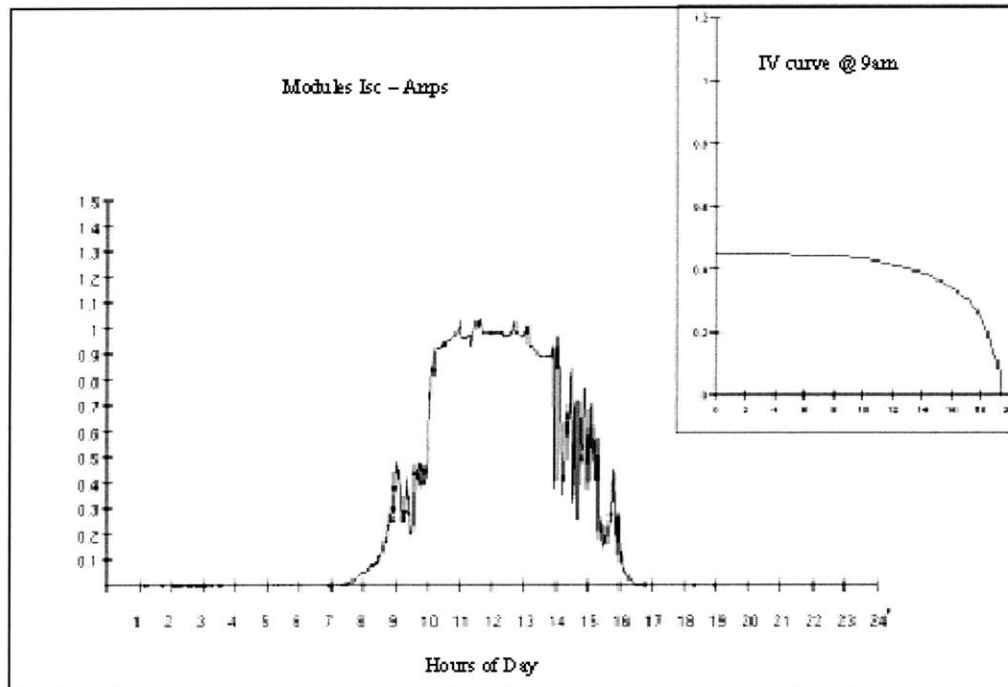
Wind speed 13 January, 2001. A clear sunny day with light winds in the afternoon.



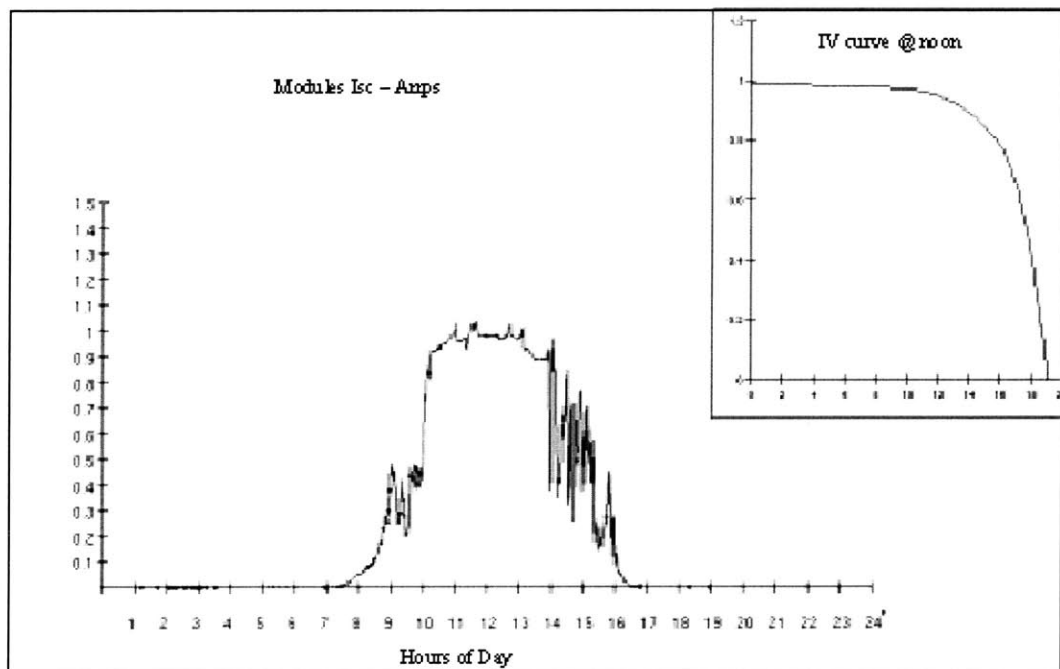
Battery voltage 13 January, 2001. Battery discharges due to load of approximately 100ma. When sun comes up, battery charges. Note battery control when voltage exceeds 14.5 volts. Note also higher voltage at end of day indicates net energy gain over the course of the day. Total battery storage is 30 ah.



Module short circuit current 13 January, 2001. Some clouds passing are evident but overall a good sunny day. The module is rated at 15 watts and shows a short circuit current at "full sun" of 1 amp. The I/V curve was taken at 9am.



The next plot shows the same Isc time history but the I/V curve was taken at noon. An I/V curve is taken every minute!



Appendix F – data model

```

/***** Object: Table [dbo].[weather_detail] *****/
/***** data model to store the 2 second data *****/
/***** 12/16/2000 mchu@mit.edu *****/

CREATE TABLE [dbo].[weather_detail] (
    [weather_detail_id] [decimal](18, 0) IDENTITY (1, 1) NOT NULL ,
    [time] [datetime] NOT NULL ,
    [TA] [float] NOT NULL ,
    [TM] [float] NOT NULL ,
    [TB] [float] NOT NULL ,
    [RN] [float] NOT NULL ,
    [WD] [float] NOT NULL ,
    [WS] [float] NOT NULL ,
    [VB] [float] NOT NULL ,
    [BF] [bit] NOT NULL ,
    [IV] [varchar] (4000) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [NP] [float] NULL ,
    [VO] [float] NULL ,
    [ISC] [float] NULL ,
    [VMAX] [float] NULL ,
    [IMAX] [float] NULL
) ON [PRIMARY]

/***** create clustered index on column "time" *****/

CREATE CLUSTERED INDEX [IX_weather_detail] ON
[dbo].[weather_detail]([time]) ON [PRIMARY]

/***** Object: Table [dbo].[weather_hour] *****/
/***** stores hourly averages - for simulation *****/
/***** 11/12/2001 mchu@mit.edu *****/

CREATE TABLE [dbo].[weather_hour] (
    [weather_hour_id] [int] IDENTITY (1, 1) NOT NULL ,
    [time] [datetime] NOT NULL ,
    [MPavg] [float] NOT NULL ,
    [Iavg] [float] NOT NULL
) ON [PRIMARY]

/***** create clustered index on column "time" *****/

CREATE CLUSTERED INDEX [IX_weather_hour] ON
[dbo].[weather_hour]([time]) ON [PRIMARY]
```

Appendix G – Conversion details

Ambient Temperature (TA) – °C

$$TA = 100 * \left(\frac{5.04}{5.50} * \frac{\text{wordvalue}}{2^{10} * 16} + 2.53 \right) - 273.15 = 0.005593 * \text{wordvalue} - 20.15$$

Module Temperature (TM) – °C

$$TM = 100 * \left(\frac{5.04}{4.61} * \frac{\text{wordvalue}}{2^{10} * 16} + 2.52 \right) - 273.15 = 0.006673 * \text{wordvalue} - 21.15$$

Box Temperature (TB) – °C

$$TB = 100 * \left(\frac{5.04}{4.62} * \frac{\text{wordvalue}}{2^{10} * 16} + 2.52 \right) - 273.15 = 0.006658 * \text{wordvalue} - 21.15$$

Rainfall (RN) – inches

$$RN = \delta\text{count} * \frac{1}{100}$$

Wind Direction (WD) – degrees

$$WD = \frac{\text{wordvalue} * 5.04}{2^{10} * 16} * 88.0 = 0.02707 * \text{wordvalue}$$

Wind Speed (WS) – miles per hour

$$WS = \frac{\delta\text{count} * 100}{\delta\text{time} * 44}$$

Battery Voltage – volts

$$VB = 3.74 * \left(\frac{\text{wordvalue} * 5.04}{1.707 * 2^{10} * 16} * +1.807 \right) = 0.000674 * \text{wordvalue} + 6.7582$$

Current (I_{module}) – amps

$$I_{\text{module}} = \frac{5.04}{256} * \text{bytevalue} = \text{bytevalue} * 0.01969$$

Voltage (V_{module}) – volts

$$V_{\text{module}} = 6.53 * \frac{5.04}{256} * \text{bytevalue} = \text{bytevalue} * 0.1286$$

Open Circuit Voltage – volts

$$VO = \frac{6.53 * 5.04}{2^{10} * 16} * \text{wordvalue} = \text{wordvalue} * 0.002009$$

Short Circuit Current- amps

$$IS = \frac{5.04}{2^{10} * 16} * \text{wordvalue} = \text{wordvalue} * 0.0003076$$

Appendix H – HTML Converter

Also, since Java Virtual Machines (JVM) are not uniform across browsers, we required that a plugin had to be installed on client browsers. This simply involved using a HTML Converter²⁶ provided by Sun on html pages which makes the client browser use the Sun standard JVM. This ensures that applet behavior on all client machines is uniform.

To run the latest version of the HTML Converter (1.3):

1. open command prompt
2. cd download\htmlconverter\converter\classes\
3. java -classpath . HTMLConverter
4. select input directory where html files are
5. select backup directory
6. convert

To install the plugin for the latest version of netscape (4.78) running on athena:

1. Click on download plugin - opens up browser window
2. Click on Archive link
3. Download the plugin into /mit/\$USER
4. go to prompt
5. cd
6. sh <FILENAME>
7. .netscape/java/ControlPanel
8. make sure plugin is enabled
9. logout and log back in

OR

start the old version of netscape on athena:

```
athena% add infoagents
athena% netscape-4.61
```

BUT

now the default athena runs the old JVM, so should work ok.

²⁶ http://java.sun.com/products/plugin/1.3/docs/html_converter.html

Appendix I – RMI Instructions

Start the RMI Registry

1. jdk window
2. cd RMI
3. javac *.java
4. rmic RMIServer
5. set classpath=
6. rmiregistry

Start the RMI Server

7. open another jdk window
8. cd RMI
9. java -Djava.security.policy=policy RMIServer

Appendix J – TTY.C Instructions

Instructions for restarting the TTY.C program on PV1.MIT.EDU.

1. Stop the TTY.C program
2. Rename the file for the current day (eg: c:/tty/11_15_01.dat to c:/tty/11_15_01a.dat)
3. Reboot the machine (or do whatever maintenance required)
4. run c:/tty/TTY.exe
5. Go to the "Settings..." menu and set options :
 - set to 4800 baud
 - set to com2 (i think it's com2. if it doesn't work, try com1)
 - check the 3 boxes "autowrap", "new line", "local echo"
 - change font size if desired
6. Under "Action", select "connect"
7. You should see incoming data displayed on the screen
8. Check to see that data file for today exists (eg: c:/tty/11_15_01.dat)