Advanced Modeling Features of Micro Saint Sharp

Murat M. Gunal

Draft v.0.2

June - 2010

Contents

1 INTRODUCTION

Micro Saint Sharp's motto is "Everything you need in simulation". This is very correct indeed, since Micro Saint Sharp really provides everything you need to model a system.

To clarify the terminology, Micro Saint Sharp uses C# programming language. This is quite a contemporary language which is an Object-Oriented language. If you are familiar with C and Java, it would be very easy to get used to this new language. The "#" sign is pronounced as "sharp" and therefore I will use MS# to name Micro Saint Sharp in this book.

In this book, I mainly write about the ways the modeller can benefit from advance features of MS#. Real case studies are given to clarify the subjects covered.

Some advanced data-related features covered in this book are;

- Hashtables
- ArrayLists
- Excel
- Object Designer

In the *Time management* section, a method to convert virtual time units to meaningful units is given.

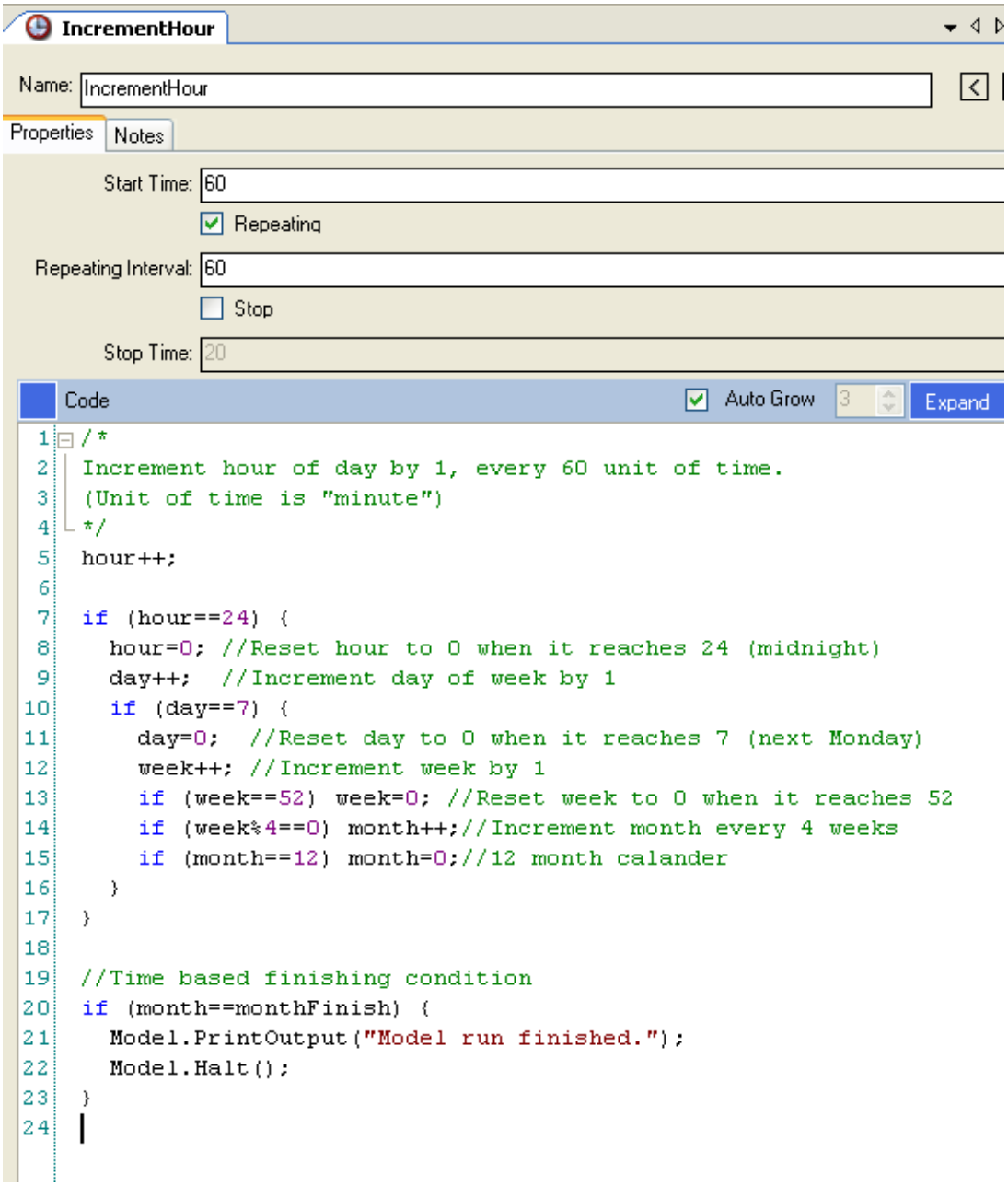
Data Collection – Snapshots (histograms)
Distributions (Hyperexponential)
Non-linked Tasks

TO BE WRITTEN

2 TIME MANAGEMENT

Time is a virtual entity in MS#. A unit of time is determined by the modeller. Therefore it may be useful to convert the time to meaningful time in the model. The time management in MS# is left to the modeller and here in this section a method is presented. This method utilizes Scenario Events in MS#.

Figure 2-1 A Scenario Event for time management.



It assumes that you already defined the variables hour, day, week, and month as integers and their default values are 0. You also define monthFinish and set its value to a positive integer which represents number of months you want to simulate the model for.

As you know Scenario Events are the bunch of codes that the model executes at given times, once or in regular intervals. In this example we wanted the “Increment Hour” Scenario Event to run at time 60, with regular intervals of 60, since an hour has 60 minutes and this continues through time. Note that we assume, as we discussed earlier, that the unit of time is minutes in this model.

What is amazing here is that, since you have full flexibility of C# you can actually set some rules in this function. For example in line 7 you can check whether the hour variable has reached 24. This is to check that the time is 24-hour circular and the hour variable only takes values between 0 and 23 (inclusive). When this happens you reset the value to 0, to represent the midnight. When the midnight is reached, that is a day has elapsed, it is time to increment the day variable (see line 9). In the same way, we need to make sure that the day variable is circulating 7 days.

It is also easy to check whether a month has elapsed. By using a simple mathematic operator “%” we can see if a week is divisible by 4. If the value of week is a multiple of 4 then it yields 0. We assume that every month is exactly 4 weeks, and calendar months are not used.

In the end of this example you see four lines of codes (lines 20-23). This part is to demonstrate how you can use time related variables to control the run length of your model. For example since this Scenario Event is executed regularly, it is also checked if a pre-defined time has come. If so, you can, for example, Halt the model. Note that Model.PrintOutput function prints the string value to MS#’s output window. This is a very useful function which can be used anywhere in the model code.

3 DATA STRUCTURES

3.1 Hashtables

Hashtables are one of the most efficient data structure in C# programming. They can also be used in MS#. In the variables you can define a variable of type Hashtable.

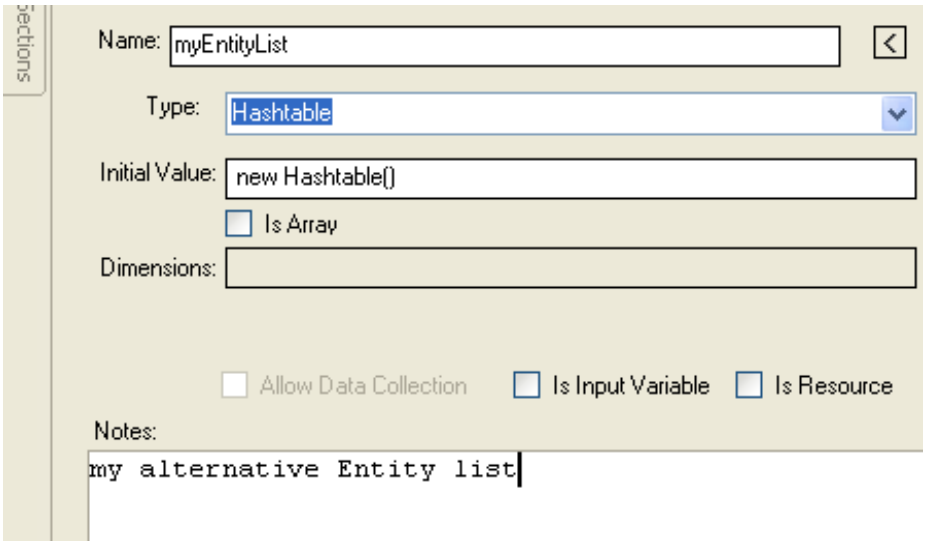
Hashtables are dynamic arrays which means the user can add any number of variables to it. There is no need to predefine its size. This feature of hashtable is quite usefull since its alternative, the arrays, you need to define its size.

A Hashtable is a dynamic array which can store key-value pairs. They can be seen as a dictionary, for every word there is a string to tell the meaning of the word. In C#, key and value is of type Object, which the user can later cast these variables to any type needed.

3.1.1 Creating a Hashtable

Hashtables can be created using MS# user interface. In the variables node of the tree view you can create a variable and then choose its type as Hashtable.

Figure 3-1 Defining a Hashtable



When a variable of type hashtable is defined, as you can see in Figure 3-1 its initial value is assigned to “new Hashtable()”. This actually means “to instantiate a new instance of Hashtable class”. The reader is advised to revise Object-Oriented programming concepts.

There is another way of creating a hashtable. Inside the code, for example in a Scenario event (ScenarioEvent1), you can explicitly write the following;

```
Hashtable myTable=new Hashtable();
```

The variable “myTable” is a local variable and its scope is only the Scenario Event. As you know, however, a variable which is defined in the variables is a global variable and its scope is whole model.

3.1.2 Adding an Item

In the example, to keep a list of Entities in MS# a hashtable with the name “myEntityList” is created. In this simple example after creating an Entity (using an Entity generator) the entity is added to myEntityList with the command shown below. Note that the key in our hashtable is the Entity’s tag number. The value, on the other hand, is the Entity itself.

```
myEntityList.Add(Entity.Tag,Entity);
```

By doing this, you keep a copy of every entity that is created in the model. Why we did so is up to the user. It is important to note that the unique identifier is Entity.Tag since we need unique key numbers in the hashtable.

3.1.3 Removing an Item

To remove an item in a hashtable you simply call “Remove” method. In the example model, Ending Effect of “Access an item and dispose” you can write the following;

```
myEntityList.Remove(Entity.Tag);
```

The parameter is the key of the key-value pair. The code above will remove every entity that is processed in the ending effect.

3.1.4 Accessing an Item

Hashtables are actually arrays and therefore to access an item in a hashtable you simply use [ ] operators. In the example model, Beginning Effect of “Access an item and dispose” task you can write the following;

```
Entity anItem = (Entity) myEntityList[Entity.Tag];

Model.PrintOutput("The Entity's Tag Number:"+anItem.Tag);
```

In the first line, the hashtable “myEntityList” is accessed using “[Entity.Tag]”. This index is the key in the hashtable. Since the key in our example is the tag number, this code will fetch the values, which are Entities and assign the entity to a temporary, local variable “anItem”. “(Entity)” is used just before myEntityList. This essentially tells the compiler that the returned type will be of “Entity” class type. This operation is called “type casting”. You can then use the returned value, or Entity, however you like, such as to print its attributes.

3.2 ArrayLists

3.2.1 Creating an ArrayList

ArrayLists can also created using MS# user interface.
…

3.2.2 Adding an Item

You can add an item to an arrayList object just like you do in Hashtables. However since an arrayList can only record one value to, in the Add function there is only one parameter which is “value”.

…..

3.2.3 Removing an Item

To remove an item in an arrayList,
….

3.2.4 Accessing an Item

To access an item in an array list you simple use …
…

Example: Accessing an Item in a MS# Queue

It is difficult to access an item in a queue in MS#. One way of accessing an individual item is to use built-in function “Model.Find”. This functions returns a list of Entities in a given task. For example;

```
ArrayList theList=Model.Find(“ID”,”3”);
```

will return a list of entities which the task ID is 3. Note that the type ArrayList is a C# type which is similar to HashTables, but in ArrayLists there is only a list of items not key-value pairs. An item in an ArrayList can be directly accessed using [ ] and the index number. Therefore to access the first Entity in Task 3, and the Entity's attribute, you can write the following ;

```
((Entity) theList[0]).anEntityAttribute
```

You can then use this statement anywhere you like e.g. to check for a condition.

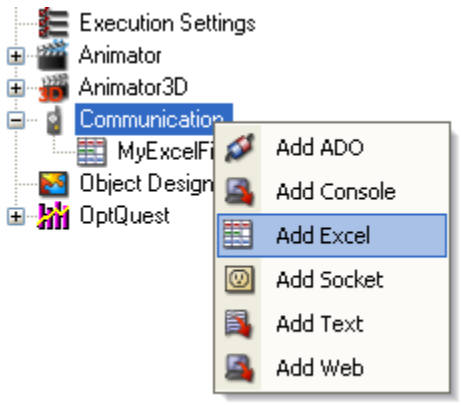
4 USING EXCEL

Microsoft Excel is one of the most frequently used spreadsheet applications in organisations. MS# can read and write to Excel files…

The model "ReadingWritingExcelFiles.saint" is an example of how to read data from Excel and how to write to. …
This model does nothing but reading data from the file, "ForMicroSaint-ReadWrite.xls", and writing some data to it. Task network has nothing. However you will see that two functions, readExcel() and writeExcel(), are called in the "Initialization Code". When the model start, these two functions are immediately called and executed.

From "Tree View" right click "Communications" and select "Add Excel" (see Figure 4-1). This will add item named "Excel_0". You can change this name by double clicking on it. In our example, we named the file "MyExcelFile". If you want to save changes when you write data in this file, you need to tick "Save on Model Complete".

Figure 4-1 Defining a Excel file

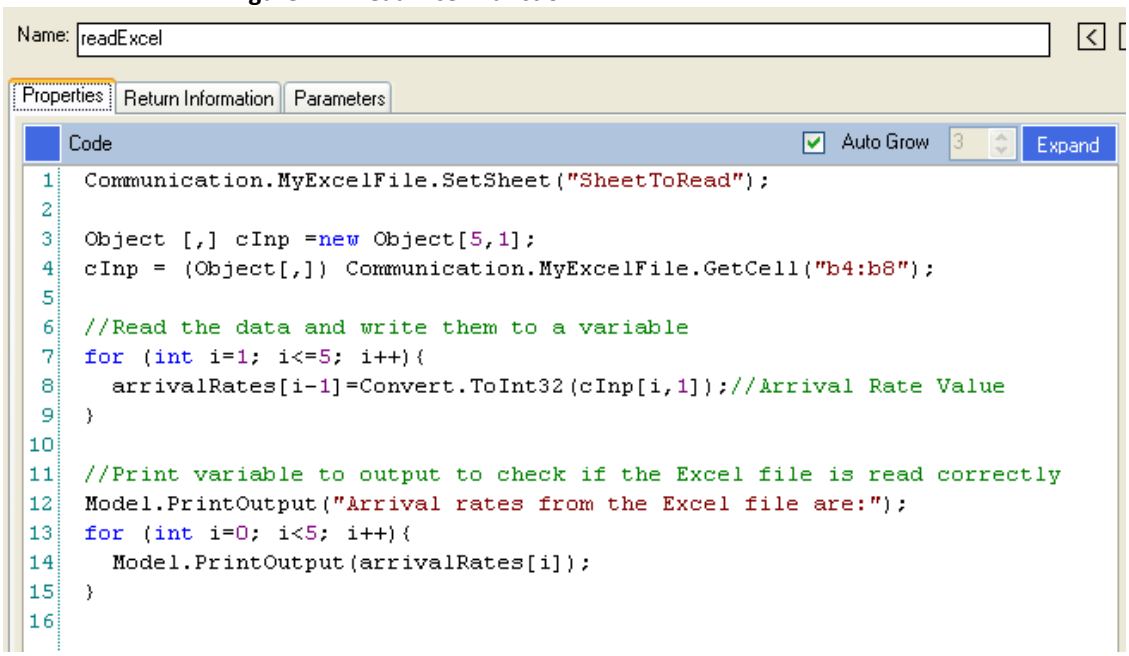


You need to choose an Excel file to read from and/or to write to. This must be done once you copy your model file to another computer, if the path to the Excel file is different.

4.1 Reading from Excel

The readExcel function, shown below, reads data from the Excel file and writes its content to arrivalRates array and then prints what is read to output screen. When you run the model you will see that the content of b4:b8 cells will be displayed in the output screen.

Figure 4-2 "readExcel" function.



The statement in line 1 establishes a link with the Excel file's "SheetToRead" worksheet. To read data from this worksheet, you need to call GetCell function (line 4). This function returns an Object array. The type Object is a kind of generic type which can later be converted to any time appropriate. This conversion is done in a for loop (line 7-9) by using the function Convert.To…. Since we know that the data is integer, we converted them to Int32. arrivalRates is an array of integers which we defined in MS#. You can then use the values in arrivalRates array in the model, for example we printed them in the output screen in lines 12-15.

4.2 Writing to Excel

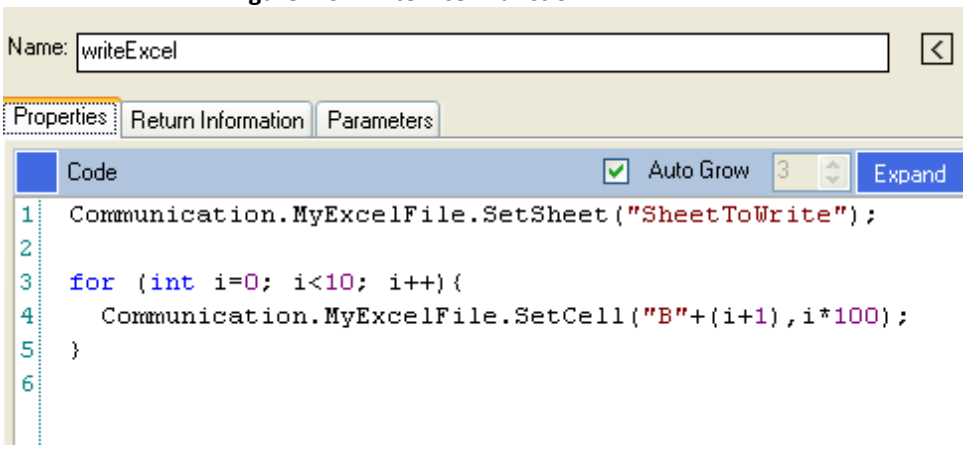
Writing data to an Excel file is also possible in MS#. First, you need to define a communication link with the Excel file that is to be written. You must follow the same steps as you did for reading from Excel. You can define a different link or use the same link to

read and write.

In our example model, the function writeExcel writes some data to a worksheet. This function is shown in Figure 4-3. In the first line, as we did in readExcel function, we establish a link with the Excel file's worksheet. In the Excel file the data is written to "SheetToWrite" worksheet.

To write a data to this worksheet you need to call "SetSheet" method. This has two parameters, first the address the data will be written to, and second, the data to be written. In our example, multiples of 100 is written to B column's rows 1 to 10.

Figure 4-3 "writeExcel" function.



5 OBJECTS AND THE DESIGNER

You can define your own classes in MS# using "Object Designer". This feature is particularly useful when there are multiple types of classes in the system you are modelling. For example, patients, doctors, nurses can be different classes of objects.
To be written.

6 DATA COLLECTION

Snapshots are easy ways of collection data in MS#. However you may need to write some code in order to format the output data.

6.1 Histograms

Histograms are useful to understand the distribution of a data set. By drawing a histogram you actually count the number of data points which falls into a specified interval.

To draw a histogram in MS#;

- Define a variable “waitingTimesBin” of integer array of size 53.
- Define a variable “endWarmup” of Boolean. Make its default value true.
- Define a snapshot “Histogram” which will collect data at the end of run. In this snapshot “waitingTimesBin” data will be collected. Since its an array you need to add each element of this array one by one. An easy way to do this is to use Excel and import the string from it. A sample worksheet is given.
    - In this Excel file, in “Template” worksheet, use the formula to concatenate the variable name and the index number.
    - Copy the contents of B column and paste it to a text editor (e.g.Windows Notebook). Then copy the text and paste it to “Expressions” column in Snapshots worksheet.
    - In MS#, read the “Snapshots” worksheet from “Utilities | Import Data from Spreadsheet”.
- Call the function waitingTimeHistogram with the value the histogram to be drawn.

Function waitingTimeHistogram() returns void and its parameter is waitingTime of type FloatingPoint.

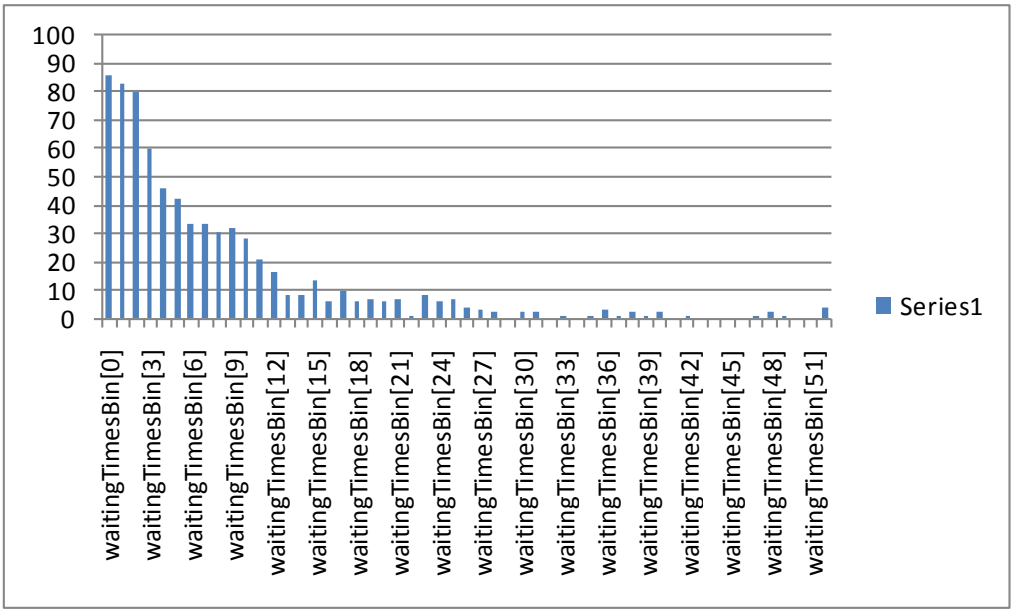
Code Box 6-1 waitingTimeHistogram function

```
double binSize=3.0;

if (endWarmup) {
  for (int i=0; i<52; i++){
    if ((waitingTime>=(i*binSize))&&(waitingTime<((i+1)*binSize))){
        waitingTimesBin[i]++;
      }
  }
  if (waitingTime>=(52*binSize)) {
    waitingTimesBin[52]++;
  }
}
```

At the end of the run, you can export results of “Histogram” and open the html file using Excel. Then, in Excel, you can draw a simple bar chart, such as below. Note that this is the result of one simulation run. If you run the model multiple times, the result of each run will be displayed as one row in the Excel file.

Figure 6-1 A Histogram (intentionally not formatted).



7 DISTRIBUTIONS

MS# provides built in functions to sample from common statistical distributions. However you may need to write code to sample from some special distributions.

7.1 Hyper-exponential Distribution

This distribution is a version of exponential distribution. It is useful to model long tail distributions and is actually n-phase distribution.

Function Hyperexponential() returns a floatingPoint and its parameters are mu1, mu2, and alpha of type FloatingPoint. In this example the distribution is a two-phase hyper-exponential

Code Box 7-1 Hyperexponential function.

```
double U1=Model.Random();//Random number for phase selection
double returnValue=0.0;

if (U1<=alpha)
  returnValue=Distributions.Exponential(mu1);
else
  returnValue=Distributions.Exponential(mu2);

return returnValue;
```

7.2 Empirical Distribution

Some data is difficult to fit to stationary distributions. In these cases, modellers may prefer to use empirical distributions. This is an easy option because frequencies in the data are used to sample.

To Be Written

7.3 Sampling from Non-Stationary Distributions: Thinning Algorithm

In real life most time dependent distributions are non-stationary, or in other words the parameters of distributions change in time. For example, let's think about number of patient arrivals in an Emergency Department. More patients arrive in day time than in night time.

To Be Written