A Procedural and Object-Oriented Statistical Language

Ikunori Kobayashi[1], Takeshi Fujiwara[2], Junji Nakano[3] and Yoshikazu Yamamoto[1]

[1]Tokushima Bunri University, 1314-1 Shido, Kagawa 769-2193, JAPAN
[2]The Graduate University for Advanced Studies, 4-6-7 Minami-Azabu, Tokyo 106-8569, JAPAN
[3]The Institute of Statistical Mathematics, 4-6-7 Minami-Azabu, Tokyo 106-8569, JAPAN

Summary

A language of a statistical system is important, even though it has an effective graphical user interface. A language may be used to control the statistical system at will and to implement new statistical procedures which are not realized in the system at the beginning. This paper introduces the features and the syntax of the language of the statistical system Jasp (Java based statistical processor). We use a procedural function-based script language Pnuts as the basis of the language, and add object-oriented mechanism thinking much of ease, flexibility and extendibility.

Keywords: Jasp, Java, Object-Oriented Language, Procedural Language, Statistical system

1 Introduction

Many statistical systems have been developed from the dawn of the computer, and have continued to adopt the new computer technology of each age. Famous examples are, to name a few, SAS (SAS Institute Inc. 2001) and SPSS (SPSS Inc. 1999). Recently, the dissemination of cheap and powerful personal computers and the Internet offer new posibilities to data analysis environments. For example, we are able to browse many kinds of data easily on the Web, to stock huge data and to handle them even on our laptop computers. In order to use such modern technologies in statistical analyses effectively, several statistical systems are newly designed using recent technologies such as the Java language (Arnold, Goling & Holmes 2000) and the distributed computing. Our statistical system named Jasp (JAva based Statistical Processor) (Nakano, Fujiwara, Yamamoto & Kobayashi 2000) is one of them.

Statistical systems are required to be able to express various computation procedures easily and clearly, to draw graphics flexibly, and to customize functions for routine tasks. For realizing these purposes, most statistical systems use their programming languages as interfaces between systems and users. Jasp also has a programming language, i.e., the Jasp language. The Jasp language is based on the Pnuts language (Tomatsu 2000), which is a script language written in and for the Java language. We modified Pnuts for statistical users to be able to describe matrix handling and basic statistical computations simply, and to get graphical results easily. As the Jasp language is a procedural language based on functions, the syntax for it is relatively simple.

Procedural languages are flexible and intuitive to express formulae, functions and tentative programs. Examples are the S language (Chambers & Hastie 1992) and the XploRe language (Härdle, Klinke & Müller 1999). Procedural languages, however, tend to have many small functions and users sometimes have difficulty finding desired functions among a lot of others. Object-oriented languages are accepted to be good at bundling existing small functions or well-organized knowledge. Therefore, we designed a class mechanism for constructing class objects from Jasp functions without much modifications.

Through the Internet, many programs, functions and subroutines for statistical computations are freely available now. Some of them are written in the Java language, and Jasp can use them directly from inside the language. Some of them are written in traditional compiler languages such as Fortran, C and C++. Jasp can also use these programs relatively easily by using the JNI (Java Native Interface) mechanism (Liang 1999).

In the recent computing environment, GUIs (Graphical User Interfaces) are

popular, especially for beginners in statistics and computing. Although almost all statistical systems have GUI environments now, the GUI and the language are sometimes not well unified. It is clear that a statistical language should be kept simple even for considering the GUI support. We designed the Jasp language to be able to support the GUI operations without increasing the syntactical complexity of the language.

Next section describes the basic principles of the Jasp language design. In section 3, we show outline of the Jasp language with some examples.

2 Basic Design Principles

We first decided to use the Java language for developing our statistical system. Java is a general purpose and pure object-oriented programming language developed by Sun Microsystems since 1991. It is a portable, architecture-neutral language, whose program codes are compiled into bytecodes which are interpreted and executed by Java Virtual Machines (Java VMs). Any system that supports a Java VM can execute all pure Java programs. Java has a rich set of libraries, which contains classes for a wide variety of functions, including network communications, security, graphical user interfaces, remote method calls and database accesses. Recently using the Internet, we can download many well written Java programs which is able to be used freely.

As Java is a powerful general purpose language, it is too complicated for statisticians, who are not professional programmers. We use Pnuts script language as a basis for our Jasp language, because it has simpler syntax than Java and is still able to use Java classes directly. Pnuts is also a general purpose language and lacks special functions for statistical computing. Then we develop a pre-processor to add the ability to handle statistical algorithms naturally. Pre-processor approach is adopted mainly because of the ease of implementation. Jasp programs are internally translated to Pnuts programs by the pre-processor, then processed by the Pnuts interpreter.

Pnuts is a procedural language which uses function definitions mainly. Together with additional statistical functions, Jasp functions can realize common characteristics and features of modern statistical systems, such as S and XploRe, i.e., they can express algorithms directly according to the flow of data processing without thinking unimportant programming concerns like types of variables. Jasp functions are easy to write tentative and small programs. Statistical works can be performed fully by many such small programs.

Jasp functions are formally independent one another, even there are some dependencies among them. Functions have no means to describe their relations. To bundle related functions, object-oriented programming is useful.

In the object-oriented programming, a system is constructed by objects which encapsulate data (attributes) and procedures (methods). As objects can describe real objects directly, they are easy to understand. Objects are also well encapsulated and have enough modularity, therefore, they are easily reused as parts of other programs. Although the object-oriented programming is a powerful technology, it has some demerits. When users want to create new objects or use objects, they have to know the structure of objects well. If there are many classes in the system, it is not easy to find an adequate object. By these facts, the object-oriented programming is not suitable for tentative and small programs.

In statistical systems, Xlisp-stat (Turney 1990) used the object-oriented programming technology mainly for graphical programming. It adopts the Lisp language as its development platform. The Lisp is a pure functional language and good at list processing. It is usually implemented by interpreter and can be operated interactively. The Lisp does not offer the object-oriented programming originally. However, because the language design is very flexible, object-oriented programming abilities such as CLOS (Keene 1989) are added. Xlisp-stat uses Lisp functions and defined objects for staistical analyses. Simple analyses are performed by built-in functions, and more complicated statistical techniques such as linear regression and time series analysis are executed using objects. This approach is thought to be useful for both users and developers, and was adopted in the current S language (Chambers 1998).

We also think that the object-oriented programming is powerful and useful for bundling related functions and describes the structure of the statistical techniques. However, we do not suggest to use this programming technique at the beginning of statistical analysis. Usually at the first stage of a statistical analysis, we have few information about data and do not know what to do next exactly. We try to execute many possible statistical procedures to the data and seek clues for clearing the data generation mechanism. Therefore, we propose using many functions at first. When we have meaningful results after many trial and error of executing functions and find some of them are useful for other data, we suggest to use a object-oriented framework for arranging them for the future use.

To realize this end, we design the Jasp class to use Jasp functions without much modification. In the class definition, we do not need to define attributes, and can define methods for using already written functions as they are.

Although Jasp functions and Jasp classes are easy to write and powerful enough, we sometimes want to use programs written in Java, Fortran, C and C++, for extending system by using fast execution speed of them. This is possible by Java and Pnuts abilities. First, Java classes are directly available from Jasp language by Pnuts ability. Second, foreign language programs in Fortran, C and C++ are relatively easily used by the JNI mechanism of the

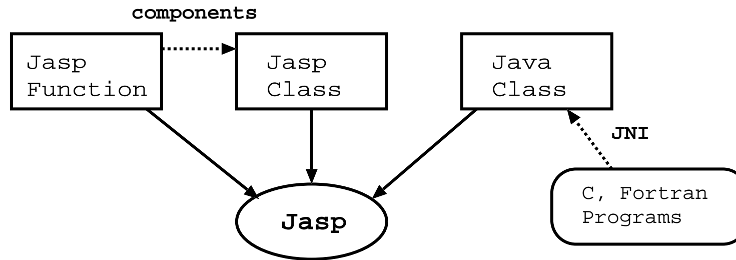

Figure 1: Jasp programming

Table 1: Purposes of Jasp programming

concept	purpose
Jasp Function	Description of a procedure of calculations
Jasp Class	Bundling related Jasp functions according to a meaningful statistical technique
Java Class	System extension

Java language. Java classes which use JNI are used directly from the Jasp language as same as other Java classes. Thus, Jasp programs are written in the way shown in Figure 1 and Table 1.

An easy-to-use operating environment is necessary for a statistical system, even though it has a sophisticated programming language. Now, most statistical systems have GUIs for visual operations. As a CUI (Character User Interface) in which a language is used for operating the system and a GUI have their own merits and demerits, they had better be united seamlessly and used almost alternatively. For example, when a Jasp Function is defined, an icon for representing it should appear on the GUI and the function should be executed by a mouse operation on the icon. For realizing such a user interface, Jasp has two windows for a CUI and a GUI. A GUI shows much information about functions on the screen, for example, an explanation of function, meanings and data types of inputs, etc. We have to include these additional information to definitions of functions and classes. We do not want to make the syntax complex for the GUI support, because users usually do not think the GUI support when they write programs for statistical calculation. Therefore, we decide to add these information as comments written in a specified special format. This style enables us to write programs without thinking the GUI first, to add necessary information on it when we decide to use it on the GUI.

3 Jasp Language

At a glance, the Jasp language is almost as same as usual procedural statistical languages. First two subsections explain such parts of the Jasp language. Third subsection shows object-oriented part, and fourth describes how to import Java classes. Last subsection explains the comments for the GUI support.

3.1 Data and Procedures

We are able to omit declarations of data type in Jasp language. A data type of a variable is determined by the type of data to which the variable is first assigned. Data types in Jasp are divided roughly into three categories: basic data types, Jasp classes and Java classes. Basic data types include integer, real, character string. Jasp classes are defined by original class declaration, and Java classes are written in Java and compiled into class files for Java VMs. They are treated almost equally in the Jasp language. Like other procedural languages, Jasp has the "`if`" statement, the "`for`" statement, etc. Each statement needs to be delimited by "`;`" or line feed.

As matrix operations are often used in data analyses, the Jasp language offers a data type to handle matrices as `JaspMatrix`, and equips expressions to manipulate them. For first example, `mat = JaspMatrix(2,3)` generates a 2×3 matrix named `mat`. We can extract elements of a matrix by the operator `[ ]` such as `mat[1,2]`. The expression `mat[!1,1:3]` extracts elements which include all rows except the first one, and columns between the first and the third. `JaspMatrix` has labels of rows and columns, we can specify elements by using them (e.g., `mat["no2", "x3"]`).

Operators for appropriate numeric matrices `x` and `y` are:

```
x + y     adds x and y,
x - y     subtracts y from x,
x * y     multiplies x and y,
x.inv     computes the inverse matrix of x,
x.trans   computes the transpose matrix of x,
x.det     computes the determinant of x,
x.eigen   computes eigen values and corresponded eigen vectors.
```

Here, we show an example to calculate the Durbin-Watson statistics (Ryan 1997) given by the expression

$$D = \sum_{i=2}^{n} (e_i - e_{i-1})^2 / \sum_{i=1}^{n} e_i^2$$

where e_i is an i-th residual. A Jasp program for it is:

```
dw = 0.0
for(i=2; i<=e.nr; i++){
  dw = dw + pow(e[i,1]-e[i-1,1], 2.0)
}
dw = dw / (e.trans * e)
```

where `e` is a JaspMatrix of residuals, and `e.nr` returns the number of the rows. The `pow` function calculates the second argument's power of the first argument.

3.2 Jasp Function

A Jasp function requires input arguments, performs calculation defined in it, and returns output values to the environment. It does not change the value of data or variables outside of the function. All Jasp functions can be used in the same way as built-in functions.

The general form of Jasp function is

```
function NAME ( ARGUMENTS ) BODY
```

where `NAME` is a name of the function, `ARGUMENTS` is a list of arguments, and `BODY` is statements for building the body of the function. The list of arguments can be empty or have several arguments delimited by ",". We do not need to specify types of arguments. Even if names of functions are same, Jasp distinguishes them by the number of arguments. If the `BODY` part includes two or more statements, they are enclosed with "{" and "}". Returning a result of the function to its caller, "`return`" statement is used.

Variables declared in a function are temporary. The scope of them is in the function, and they can not be modified by other functions. The scope of their arguments is also in the function. If the values of arguments are changed in the function, the changes are not reflected in the caller. We can define Jasp functions inside a Jasp function for using them internally. Jasp functions can be called recursively like C language.

3.3 Jasp Class

We can write all statistical calculations by Jasp functions. When we write a lot of functions for our particular analysis, we often notice that they are useful for other data and should be bundled and arranged in some way for reusing them in future easily. In addition, it often happens that some functions are

related to a particular kind of data set. These characteristics are expressed by the object-oriented approach so well that we add this structure to the Jasp language as one of the basic properties.

We consider a Jasp class as a set of related Jasp functions. Jasp functions can be used in the definition of a Jasp class with little modifications. We decide not to declare the type of the internal data (attributes) of objects.

The general form of a Jasp class definition is

```
jaspclass NAME(SUPER) BODY
```

where `NAME` is a name of the class, `SUPER` is the name of the super class. If there is no super class, `(SUPER)` can be omitted. `BODY` consists of constructors, methods, and private functions. Private functions are Jasp functions which are effective only in the class, and methods are used as interfaces of private functions to the outside of the class. Constructors are methods of the same name as its class name, and make instances of the class.

The next program shows a simple example of the Jasp class for fitting and diagnosing a linear regression model. For calculating diagnostic statistics of a regression model, we need information of the model. Thus, the `Diagnostics` class is derived from the `LinearRegression` class:

```
jaspclass Diagnostics(LinearRegression){
  method Diagnostics( file ){
    super.LinearRegression( file )
    y = this.depVar
    x = this.indepVar
    this.Diagnostics(y, x)
  }
  method Diagnostics(y, x){
    this.hat = hat(x) // projection matrix
    this.leverage = diag( this.hat )
    this.dw = dw( this.res ) // Durbin-Watson ratio
    ...
  }
  function hat(x){
    tmp = x.trans * x
    return x * tmp.inv * x.trans
  }
  function dw( e ){
    dw = 0.0
    for(i=2; i<=e.nr; i++){
      dw = dw + pow(e[i,1]-e[i-1,1], 2.0)
    }
```

```
        return dw / (e.trans * e)
      }
    }
```

where `hat`, `leverage` and `dw` are attributes to hold results of calculations, because variables prefixed by `this.` in methods are treated as attributes. Method calls prefixed by `this.` show the use of the corresponding method in the same class. As the identifier `super` in a method indicates the super class, `super.LinearRegression(file)` calls the constructor of the super class. The constructor generates a model from the specified file. In the above program, `diag` and `pow` are Jasp functions. The `diag` function returns the diagonal elements of a specified matrix. The content of the function `dw` is the same as the example in section 3.1. If an instance have a private function whose name is as same as a Jasp function, the private function has first priority. Note that `hat` and `dw` functions can be used separately as usual functions, and a Jasp class `Diagnostics` is defined for bundling these functions together.

We can generate an instance of this class by

```
    diag1 = Diagnostics(datafile.dat)
```

where `datafile.dat` is a text file in which data are written. If we send the instruction `diag1.leverage` to Jasp, it returns the leverage of the fitted model.

3.4 Using Java Classes

Pnuts, on which Jasp is based, can handle Java classes directly. Therefore, Jasp can use Java classes easily in Jasp programs. Although Jasp functions and classes are easy to write, they are translated by the pre-processor and interpreted by the Pnuts interpreter when they are executed. This mechanism causes slow excecution speed of them compared to compiled Java programs. Thus, heavy calculations or frequently used calculations should be implemented by Java programs to improve performance.

To import Java classes into Jasp system, we use the identifier `class` or the function `import`. For example,

```
    big = class java.math.BigDecimal
```

means that the variable `big` is assigned to the Java `BigDecimal` class. We can generate an instance of `BigDecimal` class named `pi` by the statement `pi = big(3.14159265358979323846264338328)`. The same tasks can be done by

```
import("java.math.BigDecimal")
pi = BigDecimal(3.14159265358979323846264338328)
```

For handling Java classes in Jasp program, the dot operator is used to access instance attributes and instance methods, and the double colon is used for accessing class attributes and class methods. For example, `pi.intValue()` and `Math::sqrt(4)` return `3` and `2.0`, respectively.

Some Java classes have been already built into the Jasp system; Jampack (Stewart 2000) is used for matrix computations, Ptplot (Lee & Hylands 2000) is for statistical graphs and Colt (Hoschek 2000) is for statistical distributions.

As an practical example of importing a Java class into Jasp, we show the use of an optimization package (Verrill 1998). We implement a function to calculate simple maximum likelihood estimators by using `Opti_F9` class of this package. The class is used in `mle` function as follows:

```
function mle(func, data, para){
  size = para.length
        .....
  opti = Opti_F9::define(model, size) // set a target model
  opti.initialize(para)               // set initial values
  ans = opti.optimize()
  return ans
}
```

where the variable `model` represents the likelihood function calculated by using `func`, `data` and `para`. The `size` holds a number of parameters, and the `para` is initial values of parameters. For having the maximum likelihood estimates of the normal distribution from 50 random numbers, the program

```
function nor(x, p){   // normal distribution
  pi = 3.14159265358979
  exp(-(x-p[0])*(x-p[0])/(2*p[1]*p[1]))
        /(sqrt(2*pi)*p[1])
}
data = normal(50) // generate random data for simulation
para = [0.1, 1.1] // initial values of parameters
ans = mle(nor, data, para)
```

is used, where `p[0]` and `p[1]` are a mean and a standard deviation of the normal distribution, respectively.

3.5 Comments for GUI

We decide not to specify data types of variables and arguments in the Jasp language for ease of programming. This syntax is convenient for users when they write many small temporal functions, and it might be the main reason that many script languages do not have the type declaration syntax. The type declaration, however, has some merits. For example, it is useful for making clear the structure of the program for users who do not know the content of the program well. It is also useful for supporting the graphical user interface (GUI). Then, we decide to use comments for specifying data types of arguments.

If there are lines which are started by `@summary`, `@param` or `@return` in comment parts, they have special meanings for the GUI. For example, in the program

```
/**
* @summary "estimate coefficients of a regression model"
* @param   JaspMatrix y "dependent variable"
* @param   JaspMatrix x "independent variables"
* @return  "regression coefficients"
*/
function estimate_coefficients(y, x){
  beta = (x.trans * x).inv * x.trans * y
  return beta
}
```

the string enclosed by "`/**`" and "`*/`" is the comment part just like Java programs. The `@summary` line shows an explanation of the function, the `@param` lines show types and explanations of arguments, and the `@return` line shows the meaning of the return value. The format of `@param` part is

```
@param TYPE NAME "STRINGS"
```

where `TYPE` is a data type of the argument named `NAME`. These lines are omitted by the Jasp interpreter on the Jasp Server as usual comments, but is used by the Jasp GUI client to help GUI operations. When we try to execute methods or functions which need arguments from the GUI, a pop-up input window appears. The data types of inputs are checked by the `@param` information and it prevents users from making mistakes.

Functions and constructors without this information are thought to be private ones and are not directly available from the GUI window.

4 Conclusion

In a general purpose statistical system, the language design is important for users to control full system abilities. The language should be able to express statistical algorithms easily and clearly, and to make clear the structure and relation of whole programs naturally. For the first purpose, function-based procedural language is known to be useful, and for the second purpose, object-oriented language is admitted to be preferable.

In the design of the Jasp language, we propose to use simple procedural functions at the first stage of analysis, and to arrange them with little modifications for constructing classes for general usage later. This approach is useful for both naive and expert users, because procedural part is simple and easy for naive users, and object-oriented part is enough powerful for advanced aims such as grouping statistical procedures systematically. The Jasp language can also import Java classes directly, therefore, can be extended easily by using various Java features such as importing Fortran or C programs using the JNI mechanism.

References

Arnold, K., Goling, J. & Holmes, D. (2000), *The Java Programming Language, Third Edition*, Addison Wesley. (`http://java.sun.com/`)

Chambers, J. M. & Hastie, T. J. (ed.) (1992), *Statistical Models in S*, Pacific Grove: Wadsworth.

Chambers, J. M. (1998), *Programming with data: a guide to the S language*, Springer.

Härdle, W., Klinke, S. & Müller, M. (1999), *XploRe – Learning Guide*, Springer. (`http://www.xplore-stat.de/`)

Hoschek, W. (2000), Colt, `http://nicewww.cern.ch/hoschek/colt/`

Keene, S. (1989), *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Symbolics.

Lee, E. A. & Hylands, C. (2000), Ptplot, `http://ptolemy.eecs.berkeley.edu/java/ptplot`

Liang, S. (1999), *The Java Native Interface: Programmer's Guide and Specification*, Addison Wesley.

Nakano, J., Fujiwara, T., Yamamoto, Y. & Kobayashi, I. (2000), A statistical package based on Pnuts. In: *COMPSTAT2000 Proceedings in Computational Statistics*, 361–366. Heidelberg: Physica-Verlag.

Ryan, T. P. (1997), *Modern Regression Methods*, John Wiley & Sons.

SAS Institute Inc. (2001), Statistical Analysis System,
http://www.sas.com/

SPSS Inc. (1999), *SPSS Base 10.0 Applications Guide*, Prentice Hall.
(http://www.spss.com/)

Stewart, W. (2000), Jampack,
http://math.nist.gov/pub/Jampack/AboutJampack.html

Tomatsu, T. (2001), Pnuts, http://javacenter.sun.co.jp/pnuts/

Turney, L. (1990), *LISP-STAT*, John Wiley & Sons.

Verrill, S. (1998), Package-optimization,
http://www1.fpl.fs.fed.us/Package-optimization.html