Matrex Function Coding

Table of Contents

Matrex Function Coding....................1
- Why Jython?....................1
- The Matrex calculation model....................2
  - The IFunction and IFunctionData interfaces....................2
  - The Matrix abstract class and the classes that extend it....................5
  - Why columns before rows?....................7
  - An example of function....................8
  - Utility functions....................10
- How to add a function to Matrex....................19
- Unit testing....................19

When you build a new function template, you need to assign to it the executable code that calculates the function.

This code can be either written in Java or in a JVM scripting language. In Matrix version 1.0 the only available scripting language is **Jython**.

In this document I will explain how to write this code, both in Java and in Jython.

Why Jython?

I have chosen Jython as script language because:

- Python, which gives the syntax to Jython, is probably the best known scripting language.
- Jython seems pretty fast compared to the other JVM scripting languages (I considered mainly Groovy, JRuby and Beanshell).
- Jython is a stable product.
- Jython does not have the syntax of Java. I found unreasonable to give only the possibility to write scripts in Java or in something similar to Java. With Jython I open Matrex to people that uses Python and never wrote a line of Java.
- Jython works well with matrices. It is possible to write code in Jython that handles arrays in a 4-5 lines where other languages need 10-20.

This does not exclude the possibility of adding other JVM scripting languages in later releases of Matrex (if someone wants to do it, I can help him).

The Matrex calculation model

A function calculates its output matrices combining its input matrices and its parameters.

Matrex calls the function executable code to recalculate the function:

- When one or more of the input matrices **changed their content**.
- When a function is built with the **editor**.
- When the project containing the function is **loaded**.
- When a **timer** triggers the function recalculation.

Before calling the function, Matrex verifies that:

- The parameters are in the correct number, correct type and the mandatory parameters have a value.
- The input and output matrices are in the correct number (between the template's min and max values), and have correct type.
- The input matrices are not null.

So, you don't have to worry about this when you code your function. If in the template you wrote that the input matrix number 2 is of type DATE, Matrex will pass you a matrix of type DATE.

The IFunction and IFunctionData interfaces

A function consists of a class implementing the following interface:

```
package matrex.item;
/**
 * it is a function that calculates matrices from matrices
 */
public interface IFunction
{
            /**
             * Recalculates the output matrices
             */
   public void recalculate(IFunctionData data) throws FunctionCalculationException;
}
```

As you can see, the interface is very simple.

Every time Matrex needs to recalculate the function, it calls the *recalculate* method with all the data that is needed (an object instance of the IFunctionData interface).

The IFunctionData interface allows to access the input and output matrices and the parameters of the function:

```
package matrex.item;
/**
 * A function container, with the objects used by a function:
 * matrices, parameters and a template
 * @author SHZ Oct 4, 2004
 */
public interface IFunctionData
{
    /**
     * Returns the nth matrix input of the function
     */
    public Matrix[] getMatricesIn();


    /**
     * Returns the nth matrix result of the function
     */
    public Matrix[] getMatricesOut();


    /**
     * Returns the nth parameter
     */
    public Object[] getParameters();


   /**
     * Tells if the function can write debug information
     */
    public boolean isDebug();
}
```

Let's look at the methods:

- the *getParameters* method returns the parameters set in the function editor. The parameters are returned as an array of objects.

  The array contains all the parameters (mandatory and optional) in the order in which they are defined in the template. The optional parameters that was not filled by the user are null.

  Even if the parameters array is an array of objects, each object is actually of one of these specific types:

  1. For a TEXT parameter, the object is a String
  2. For a NUMERIC parameter, the object is a Double
  3. For a DATE parameter, the object is java.sql.Timestamp
  4. For a BOOLEAN parameter, the object is a Boolean

  So, since you know the order of the parameter definitions in the function template, you can cast the parameters to the correct type.

- the *getMatricesIn* method returns the input matrices of the function with their current content. The matrices are the same defined in the function editor, in the same order. Their number ranges in the input matrices min/max interval of the template. The recalculate method **is not supposed** to change their content, only to read it.

- the *getMatricesOut* method returns the output matrices of the function with their current content. The matrices are the same defined in the function editor, in the same order. Their number ranges in the output matrices min/max interval of the template. The recalculate method needs to fill them with the result of the function calculation.

- the *isDebug* method tells if the function can write debug information in the log file. This is used to trace the function behaviour in case of problems and remove the trace if not needed to improve the performances.

Both input and output matrices are an array of Matrix objects. Let's give a look to the Matrix abstract class.

The Matrix abstract class and the classes that extend it

What follows is the abstract class Matrix written as if it was an interface (without methods bodies), to avoid too much text. To see the real Matrix class you can look at the Matrex CSV.

```
public abstract class Matrix implements IProjectItem, IMatrixNotifier
{
   // constructor
   public Matrix(NameAndPackage namePackage, Project project, boolean base);
   // returns the matrix width
   public int getSizeX();
   // returns the matrix height
   public int getSizeY() ;
   // returns the matrix type (TEXT, NUMERIC, DATE, BOOLEAN)
   public abstract MatrixType getType();
   // true if the matrix is not output of a function, therefore is editable
   public boolean isBase();
   // returns the matrix content as an array of objects (an array of anything is also an object)
   public abstract Object[] getValuesAsObjArr();
   // returns the matrix mutex (which avoids that two functions change it in the same time)
   public MatrixMutex getMutex();
   // change the matrix so that it becomes editable
   public void setAsBase(boolean base)
   // returns the project containing the matrix
   public Project getProject()
   // sets the matrix content as an array of objects (an array of anything is also an object)
   public abstract void setValuesAsObjArrAndNotify(Object[] values) throws ArgumentException;
   // the name-package of the matrix
   public NameAndPackage getNamePackage();
   // adds a listener (function, presentation, chart) of the matrix
   public void addListener(IMatrixListener listener);
```

```
    // removes a listener (function, presentation, chart) from the matrix
    public void removeListener(IMatrixListener listener);
    // returns the listeners (function, presentation, chart) of the matrix
    public List<IMatrixListener> getListeners();
    // new name-package for the matrix
    public void rename(NameAndPackage namePackage);
    // returns always MATRIX, to distinguish it from the other items (functions, presentations...)
    public ItemType getItemType();
    // returns the comment the user attached to the matrix
    public String getComment();
    // sets the comment the user attached to the matrix
    public void setComment(String comment);
}
```

The abstract class Matrix is extended by 4 different classes, one for each matrix type (NumericMatrix, TextMatrix, DateMatrix, BooleanMatrix) in the package matrex.item.matrix.

Each of the classes has two supplementary methods:

- *getValues* returns the content of the matrix as 2-dimensional array.
- *setValuesAndNotify* sets the content of the matrix as 2-dimensional array.

The reason because these methods are not in the Matrex object is that their parameters and return values depend by the matrix type:

- For NumericMatrix setValuesAndNotify sets and getValues returns a **double[][]** object.
- For TextMatrix setValuesAndNotify sets and getValues returns a **String[][]** object.
- For DateMatrix setValuesAndNotify sets and getValues returns a **Timestamp[][]** object.
- For BooleanMatrix setValuesAndNotify sets and getValues returns a **boolean[][]** object.

As you can see we use double[][] instead of Double[][] and boolean[][] instead of Boolean[][] to maintain a high level of **performance** using the numeric and boolean matrices, even if in the other way we could have the getValues and setValuesAndNotify methods directly in the Matrix abstract class (setting and returning Object[][] values).

Let's describe the usable methods of the NumericMatrix class (it is the most used and the other classes are not so different as functionalities):

- *getSizeX* returns the X size of the matrix, the number of columns.
- *getSizeY* returns the Y size of the matrix, the number of rows.
- *getValues* returns a 2-dimensional array of double values (double[][]).
- *setValuesAndNotify* sets a new 2-dimensional array of double values (double[][]) as content of the matrix.
- *getType* returns the type of the matrix. The type is a constant defined in the class matrex.item.type.MatrixType: TYPE_NUMERIC , TYPE_TEXT, TYPE_DATE, TYPE_BOOLEAN.
- *isBase* tells if the matrix was entered with an editor (a base matrix), in which case it is true, or is the result of a function.
- *getValuesAsObjArr* is equivalent to getValues, but uses a trick: since in Java an array is equivalent to an object, it returns an array of objects (Object[]), which can be casted to a double[][] array. I don't consider it a clean method, but can be very useful when the type of the matrix is not fixed (ANY in the template definition) and it has to be handled in a way that is independent by its type.
- *setValuesAsObjArrAndNotify* is equivalent to setValuesAndNotify, but uses the same trick seen for getValuesAsObjArr: the parameter is an array of objects (Object[]) which is converted to a double[][] array and set as the content of the matrix.

Why columns before rows?

The content of the matrix is an array **double[columns (sizeX)][rows (sizeY)]**, which is **not intuitive**, at least for me. I would normally use an array double[rows][columns], so that I would first iterate by row, then by column.

The reason for this decision is the matrix editor: **it is much easier to write many rows than many columns**, because a cell is generally bigger horizontally than vertically.

To choose an extreme example, if you write a 1-dimensional array in the matrix editor, you will certainly write it vertically (only one column and many rows). When you have to iterate on the array in the code, since the column is only one and the rows many, it is faster to iterate on the column in the external loop and on the rows in the internal one:

```
for(int x = 0; x < sizeX; x++)
        for(int y = 0; y < sizeY; y++)
                doSomethingWith(arr[x][y]);
```

The same happens with 2-dimensional arrays with many rows and few columns, which as we saw before is the normality.

An example of function

To write the code for a function, you need to implement the IFunction interface that we saw before.

The typical schema for the implementation of the recalculate method is:

- Get the input matrices using the IFunctionData.getMatricesIn method.
- Get the output matrices using the IFunctionData.getMatricesOut method.
- Get the parameters (if any) using the IFunctionData.getParameters method.
- Get the content of the input matrices (as 2-dimensional vectors) using the getValues or getValuesAsObjArr methods.
- allocate and calculate the result 2-dimensional arrays.
- assign the result 2-dimensional arrays to the output matrices using the setValuesAndNotify or setValuesAsObjArrAndNotify methods.

Here we have the example of a very simple function that returns the content of the input matrix multiplied by two: 1 input matrix, 1 output matrix, no parameter.

In Java:

```
package matrex.fun.sys.standard;


import matrex.engine.errors.ArgumentException;
import matrex.item.FunctionCalculationException;
import matrex.item.IFunction;
import matrex.item.IFunctionData;
import matrex.item.Matrix;
import matrex.item.matrix.NumericMatrix;


/**
 * matrix multiplication by 2
 */
public class Times implements IFunction
{
    public void recalculate(IFunctionData data)
            throws FunctionCalculationException
    {
        Matrix[] arrMatrixIn = data.getMatricesIn();
        Matrix[] arrMatrixOut = data.getMatricesOut();
        Object[] arrParameter = data.getParameters();
        NumericMatrix matrixIn = (NumericMatrix) arrMatrixIn[0];
        NumericMatrix matrixOut = (NumericMatrix) arrMatrixOut[0];
        double[][] matIn = matrixIn.getValues();
        int sizeX = matrixIn.getSizeX();
        int sizeY = matrixIn.getSizeY();
        double[][] matOut = new double[sizeX][sizeY];
        for(int x = 0; x < sizeX; x++)
            for(int y = 0; y < sizeY; y++)
                matOut[x][y] = matIn[x][y] * 2;
        try
        {
            matrixOut.setValuesAndNotify(matOut);
        }
        catch (ArgumentException e)
        {
            throw new FunctionCalculationException("Filling the result matrix got", e);
        }
    }
}
```

In Jython:

```
from matrex.item import IFunction
from matrex.item import IFunctionData
from matrex.item import Matrix
from matrex.item.matrix import NumericMatrix

# returns a matrix that is number by number the double of the input matrix

class FunctionScript(IFunction):
    def __init__(self):
        pass
    def times2(self, arrIn):
        arrOut = [x * 2 for x in arrIn]
        return arrOut
    def recalculate(self, data):
        arrMatrixIn = data.getMatricesIn()
        arrMatrixOut = data.getMatricesOut()
        arrParameter = data.getParameters()
        matrixIn = arrMatrixIn[0]
        matIn = matrixIn.getValues()
        matrixOut = arrMatrixOut[0]
        matOut = [self.times2(x) for x in matIn]
        matrixOut.setValuesAndNotify(matOut)
```

Jython is very good with arrays and there is probably a way to simplify this code removing the times2 method.

Utility functions

The example function is very simple, but sometimes things become a little more complex.

For example one of the problem is if you have as input or output matrices matrices with unspecified type (ANY). In this case you should have a different piece of code for each type of matrix. If the ANY matrix is only one, that could be okay. If you need to combine more ANY matrices your code can become very big and complex.

For these cases Matrex contains some utility classes that can be very useful.

The *IMatrixMemory* interface uses the trick that we saw before, the fact that an array is equivalent to an object in Java, so that a 2-dimensional array is seen as an array of objects (Object[]). Not very elegant but effective. Using the classes that implement this interface you can work with the 2-dimensional arrays without to worry about their type.

```
package matrex.item.matrix.memory;
public interface IMatrixMemory
{
   /**
    * Returns the x,y item as a string
    */
   public String getItemAsString(int x, int y, Object matrix[]);
   /**
    * Sets the x,y item converting it from a string
    */
   public void setItemAsString(int x, int y, Object matrix[], String value);
   /**
    * Allocates a matrix
    */
   public Object[] allocate(int sizeX, int sizeY);
   /**
    * Returns the height of the matrix
    */
   public int getSizeY(Object[] matrix);
   /**
    * Makes a copy of the matrix
    */
   public Object[] copy(Object[] matrix);
   /**
    * Compares the elements of two matrices
```

```
     */
    public boolean equals(Object[] a, int ax, int ay, Object[] b, int bx, int by);
    /**
     * Assigns an item of matrix b to matrix a
     */
    public void assign(Object[] a, int ax, int ay, Object[] b, int bx, int by);
    /**
     * Converts a matrix to the same matrix containing objects
     * (double -> Double, String -> String, boolean ->  Boolean, Date -> Date)
     * No copy is done if not needed
     */
    public Object[][] convertToObjectMatrix(Object[] matrix);
    /**
     * Converts an object matrix to the same matrix containing pure types
     * (Double -> double, String -> String, Boolean ->  boolean, Date -> Date)
     * No copy is done if not needed
     */
    public Object[] convertFromObjectMatrix(Object[][] objMatrix);
      /**
        * Allocates a matrix containing objects (Double, String, Boolean, Date)
        * @param sizeX
        * @param sizeY
        * @return
      */
    public Object[][] allocateAsObjectMatrix(int sizeX, int sizeY);
      /**
        * Transposes the matrix (columns become rows, rows become columns)
        * @param matrix the matrix to transpose
        * @return the transposed matrix
        */
    public Object[] transpose(Object[] matrix);
}
```

An instance of a class implementing this interface can be obtained through the static method **matrex.item.matrix.memory.MatrixMemoryFlyweight.getAdapter**. For example:

```
Matrix anyMatrix = ...
...
ImatrixMemory memory =  MatrixMemoryFlyweight.getAdapter(anyMatrix.getType());
Object[] mat = memory.copy(anyMatrix.getValuesAsObjArr());
```

The *MatrixHelper* class is used to verify the input and output matrices. Matrex already verifies the number of matrices and their type, but it cannot verify the size. Using the functions of MatrixHelper you can:

```
package matrex.item.matrix;


import matrex.item.Matrix;
/**
 * some functions that are common among all types of matrices
 */
public class MatrixHelper
{
   /**
    * Returns true if the two matrices have same size
    */
   public static boolean sameSize(Matrix a, Matrix b);
   /**
    * Returns true if the two matrices have same width
    */
   public static boolean sameSizeX(Matrix a, Matrix b);


   /**
 * Returns true if the two matrices have same height
 */
public static boolean sameSizeY(Matrix a, Matrix b);
/**
 * returns true if all the matrices in the array have the same size
 */
public static boolean sameSize(Matrix[] arrMatrix);

/**
 * returns true if all the matrices in the array have the same height
 */
public static boolean sameSizeY(Matrix[] arrMatrix);

/**
 * returns true if all the matrices in the array have the same width
 */
public statc boolean sameSizeX(Matrix[] arrMatrix);

/**
 * Returns true if the two matrices have same type
 */
public static boolean sameType(Matrix a, Matrix b);

/**
 * Returns true if all the matrices in the array have the same type
 */
public static boolean sameType(sMatrix[] arrMatrix);

/**
 * Returns true if the two matrices have same type
 */
public static boolean sameType(Matrix a, Matrix b);
```

```
   /**
    * Returns true if the matrix is actually a vector (1 x dimension)
    */
   public static boolean isVector(Matrix matrix);
}
```

IMatrixStringAdapter converts the content of your matrix in a 2-dimensional array of strings or sets the content of your matrix extracting it from a 2-dimensional array of string.

```
package matrex.item.matrix.string;


import matrex.engine.errors.ArgumentException;
import matrex.item.Matrix;
/**
 * @author SHZ Sep 24, 2004
 * to see the content of the matrix as if it was made by strings
 */
public interface IMatrixStringAdapter
{
   /**
    * Sets a String[][] array as content of a matrix
    */
   public void setValues(Matrix matrix, String values[][]) throws ArgumentException;
   /**
    * Returns the content of the matrix as a String[][] array
    */
   public String[][] getValues(Matrix matrix);
}
```

An instance of a class implementing this interface can be obtained through the static method **matrex.item.matrix.string.MatrixStringFlyweight.getAdapter**.

In case of functions that are applying mathematical functions to each element of one or more numeric matrices, there is a set of classes that can be very useful.

Instead to show directly the classes definitions, I will show some example that use them. I think it is better to understand what they do in this way.

They are all located in the package matrex.item.functions.numeric.

First of all, *UnaryScalarFunction* and *IUnaryCellFunction*, to calculate scalar unary functions (functions with one parameter) of matrices, like abs, exp, sin. The scalar function is applied to each item of the matrices.

This is the code used to realized the function exp (exponential of a matrix, the result matrix is composed by the exponential of all elements of the input matrix):

```
package matrex.fun.sys.standard;


import matrex.item.functions.numeric.IUnaryCellFunction;

import matrex.item.functions.numeric.UnaryScalarFunction;


public class Exp extends UnaryScalarFunction

{

   public Exp()

   {

      super(new IUnaryCellFunction()

      {

         public double apply(double a) { return Math.exp(a); }

      });

   }

}
```

Exp extends UnaryScalarFunction, which implements IFunction and does all the work. In the constructor Exp builds the parent class with a class extending IUnaryCellFunction that calculates the exponential of one single cell of the matrix.

UnaryScalarFunction calls the apply method for each cell of the matrix and uses the return value to build the output matrix.

BinaryScalarFunction and *IBinaryCellFunction* calculate binary functions (functions with 2 parameters) of matrices, like +, -, pow.

Let's see the function Plus (+):

```
package matrex.fun.sys.standard;


import matrex.item.functions.numeric.BinaryScalarFunction;

import matrex.item.functions.numeric.IBinaryCellFunction;


public class Plus extends BinaryScalarFunction

{

    public Plus()

    {

        super(new IBinaryCellFunction()

        {

                    public double apply(double a, double b) { return a + b; }

        });

    }

}
```

The concept is the same seen for UnaryScalarFunction and IUnaryCellFunction, but for binary functions.

AggregateFunction and *IVectorToScalar* are used to calculate aggregate functions (like sum and average). These aggregate functions are calculated vertically , which means that the content of each column of the input matrix is aggregated, producing an array from the matrix. If you want instead to calculate the sum of each row of the matrix, use the *Transpose* function on the matrix before the calculation.

Let's see the sum (Sum) class:

```
package matrex.fun.sys.mathstat.statutils;


import org.apache.commons.math.stat.StatUtils;
import matrex.item.functions.numeric.AggregateFunction;
import matrex.item.functions.numeric.IVectorToScalarFunction;
/**
 * Sums the columns of a matrix
 * @author SHZ May 19, 2005
 */
public class Sum extends AggregateFunction
{
        public Sum()
        {
                super(new IvectorToScalarFunction() {
                                public double apply(double[] in)  {
                                        return StatUtils.sum(in);
                                }
                        });
        }
}
```

As you can see it is very similar to Pow and Plus. But it depends by an anonymous class, which implements IVectorToScalar.

AggregateFunction calls the method *apply* for each column of the matrix. Apply must return the aggregated value for that column.

MatrixToMatrixApply and I*MatrixToMatrixFunction* can be used for the functions with one matrix as input and one matrix as output. Most of the functions used in Matrex are of this type, so these classes can be very useful. To use them you need to implement the interface IMatrixToMatrixFunction:

```
package matrex.item.functions.numeric;
public interface IMatrixToMatrixFunction
```

```
{
        public double[][] apply(double[][] mat);
}
```

where you calculate and return a double-array from an input double-array, and create a MatrixToMatrixApply instance with it.

ComparisonFunction and IComparison are used to obtain a boolean matrix from two numerical matrices. It is the basis for functions like **equal** and **different**. ComparisonFunction applies the IComparison implementation to the input matrices item by item:

```
package matrex.item.functions.numeric;
public interface IComparison
{
        boolean compare(double a, double b);
}
```

and builds a boolean matrix with the result.

You will also find *VectorApply* (with IVectorToVector and IVectorToScalar) which breaks a matrix in columns and calls the IVector... function for each column.

Similar help classes you will find also for generic, string and boolean matrices.

How to add a function to Matrex

You can add one or more Java functions (classes that implement the *IFunction* interface) compiling them and adding them to a jar file, then adding the jar file path to the classpath of Matrex (the -cp or -classpath parameter of the java executable).

For a Jython script, put the file in a directory under the ./script directory of Matrex.

When you write the template you need to select the java class or the script that calculates your function.

Unit testing

With the addition of the ISQL function template in version 1,2, Matrex functions unit testing has become a need.

Matrex functions unit testing has always been problematic, because of the complexity of the objects involved in the calculations, especially Matrix and FunctionHandler.

To cope with this problems I added some special classes, all in the matrex.item.test package, which allow to write Junit test classes very easily.

- **TestFunctionData** is the simplest way to implement IFunctionData and can be created completely with its constructor:

```
TestFunctionData(Object[] arrParameter, Matrix[] arrInMatrix, Matrix[] arrOutMatrix)
```

  Once it has been built, it can be passed to the recalculate method of the function that has to be tested.

- **MockObjects** contains several static methods to build Matrix objects (of numeric, text, boolean and date type) from simple and two levels arrays.
  The methods have names of the getMock...Matrix
  It also has a getISODates methods that converts an array of ISO date strings to an array of dates.

- **JunitMatrices** contains Junit assertEquals methods for simple and two levels arrays. They are needed because they are missing from the Junit library and only avalable in separate libraries.
  With these methods you can test the content of a matrix in one single call.

This an example of test case code (for the matrex.fun.sys.date.Add function, which adds days, months ... to dates), that can be contained in a Junit testRecalculate method:

```
// input matrix with the dates
Matrix dateMatrix = MockObjects.getMockDateVector("test.date",
                    MockObjects.getISODates( new String[] { "2008-01-01", "2009-01-01" }));
// input matrix with the days to add to the dates
Matrix numberMatrix = MockObjects.getMockNumericVector("test.number", new double[] { 10, 20 });
// result matrix (will be populated by the Add function)
DateMatrix out = MockObjects.getMockDateMatrix("test.out", null);
// function call
new Add().recalculate(
          new  TestFunctionData(
                    new Object[] { "DAY" }
                    , new Matrix[] { dateMatrix, numberMatrix }
                    , new Matrix[] { out }
          ));
// result matrix content verification
JUnitMatrices.assertEquals(out.getValues(),
                    MockObjects.getISODates( new String[] { "2008-01-11", "2009-01-21" }));
```

It is generally very easy and takes a few minutes to write the test case for a new function.

All the unit testing classes for the Matrex functions have this structure and are contained in some lines of code.