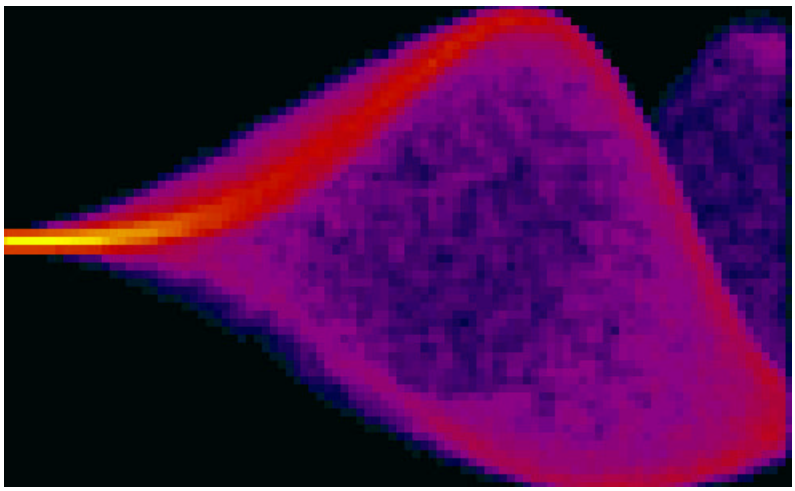
Global Minimizer for Java (GMJ) Version 2.0

Developer's and Power User's Guide



Modestas Grybauskas. 1999

1 INTRODUCTION 3

2 RUNNING GMJ 3

2.1 CONFIGURING APPLET 3
2.2 CONFIGURING STAND-ALONE APPLICATION 3

3 USER'S GUIDE 4

3.1 METHOD SELECTION 4
3.2 TASK SELECTION 5
3.3 OPERATION 6
3.4 ANALYSIS WINDOWS 6

4 EXTENDING GMJ 8

4.1 IMPLEMENTING MINIMIZATION TASKS 8
4.2 IMPLEMENTING NEW MINIMIZATION METHODS 9
4.3 CONFIGURING THE PROPERTYMANAGER 10
4.3.1 WRITING A CUSTOM PROPERTY PROVIDER 11
4.4 IMPLEMENTING NEW ANALYSIS OBJECTS 12
4.5 APPLET SECURITY ISSUES 13

5 CHANGES TO GMJ II 14

5.1 CUSTOM RESULT ANALYZERS 14
5.2 DOMAIN CONSTRAINT FUNCTION 14

6 PROGRAMMING TIPS 16

6.1 ABORTING CALCULATIONS 16
6.2 IMPLEMENTING INTERFACES 16

7 BUG REPORT 17

8 REFERENCES 17

1 Introduction

The GMJ system is class framework for implementing and testing custom minimization tasks and methods. The application is highly flexible and can be extended in many ways.

2 Running GMJ

The GMJ system can be run as a Java Applet or as a Java Application. The advantage of running it as a Java Applet is obvious - it can be used over the net. The advantage of the Java Application is that the newest Java features, which are supported by the Java Development Kit (JDK) can be used.

2.1 Configuring applet

The sample **gmj.html** file is provided, which shows, how to set configuration parameters of the GMJ applet. The <applet> tag looks like this:

```
<applet code="lt.ktu.gmj.ui.GMJ.class"
        codebase="LIB"
        archive="gmj.jar"
        width=800 height=500>
<param name=TASKS value="lt.ktu.gmj.tasks.Sin">
<param name=METHODS
value="lt.ktu.gmj.methods.Mig1|lt.ktu.gmj.methods.Bayes">
<param name=ANALYSIS value="lt.ktu.gmj.analysis.Convergence">
</applet>
```

code specifies the applet class and should not be changed.

codebase specifies the relative URL of the applet class archive file and user class files. The URL is relative to the location of the HTML file, where the <applet> tag resides or is an absolute path like "http://soften.ktu.lt/gmj/lib".

archive lists the class archive files.

width and *height* specifies the size of the applet, as it appears in a HTML page.

parameter *TASKS* lists tasks, which should be available, when the applet loads. Complete class names (package and class name) should be separated by '|' symbol.

parameter *METHODS* lists methods, which should be supported, when the applet loads. Complete class names should be separated by '|' symbol.

parameter *ANALYSIS* lists analysis classes, which should be available for running. Complete class names should be separated by '|' symbol.

2.2 Configuring stand-alone application

The Java Runtime Environment (JRE) is required, when running GMJ as a stand-alone application. The JRE for Microsoft Windows NT and Solaris is distributed at no charge and can be obtained from Sun Microsystems at http://java.sun.com .

Before running GMJ, a *CLASSPATH* should be configured, so that the GMJ classes can be found by the JRE. For Microsoft Windows NT the command might look like this:

```
SET CLASSPATH=%CLASSPATH%;c:\gmj\gmj.jar
```

The application is started by loading the *lt.ktu.gmj.ui.GMJ* class:

```
java lt.ktu.gmj.ui.GMJ
```

The application accepts two optional parameters:

The first parameter lists tasks, which should be available, when the applet loads. Complete class names (package and class name) should be separated by '|' symbol.

The second parameter lists methods, which should be supported, when the applet loads. Complete class names should be separated by '|' symbol.

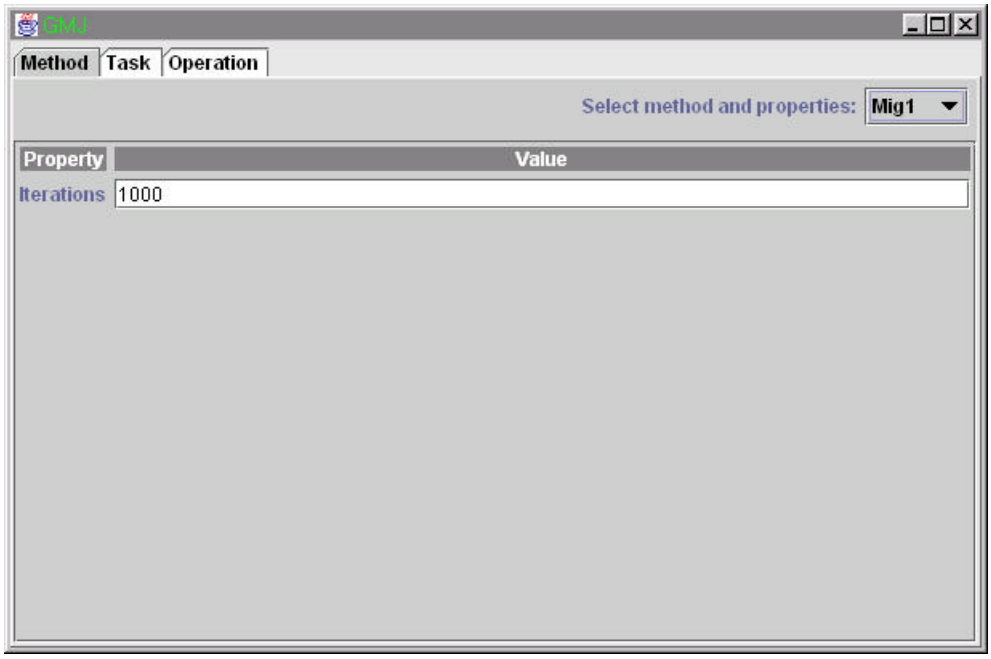
The third parameter lists analysis objects, which should be available, when the applet loads. Complete class names should be separated by '|' symbol.

3 User's guide

GMJ displays a tab control, which has three choices: method, task and operation. The appropriate pages can be selected by clicking on the page tabs.

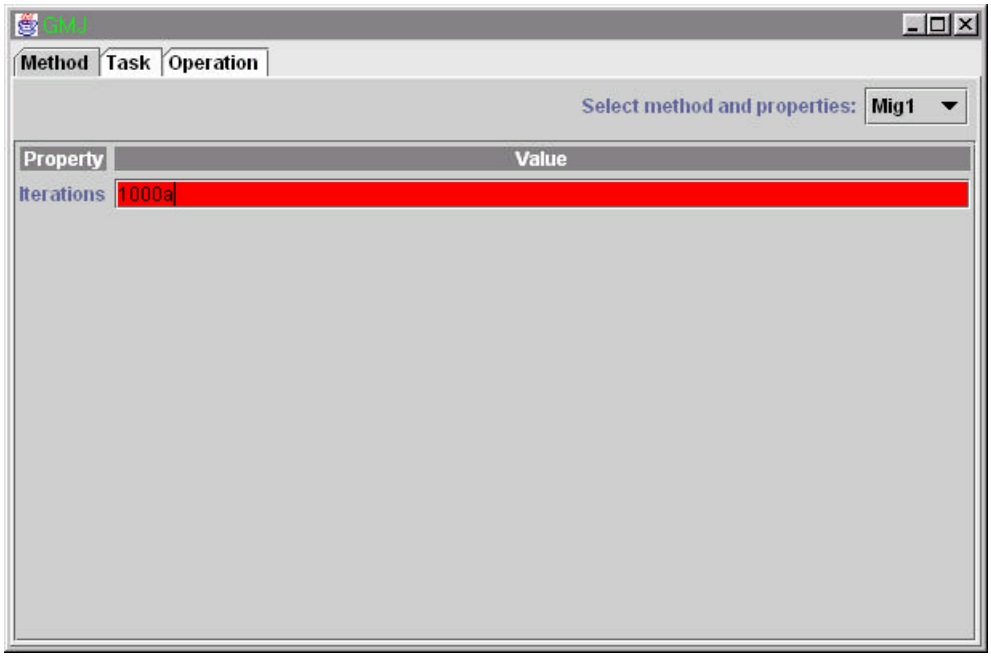
3.1 Method selection

The choice control at the top of the page shows the list of supported methods. One of them should be selected. After selecting a method, it's implementation class is loaded and the configuration window is rendered.



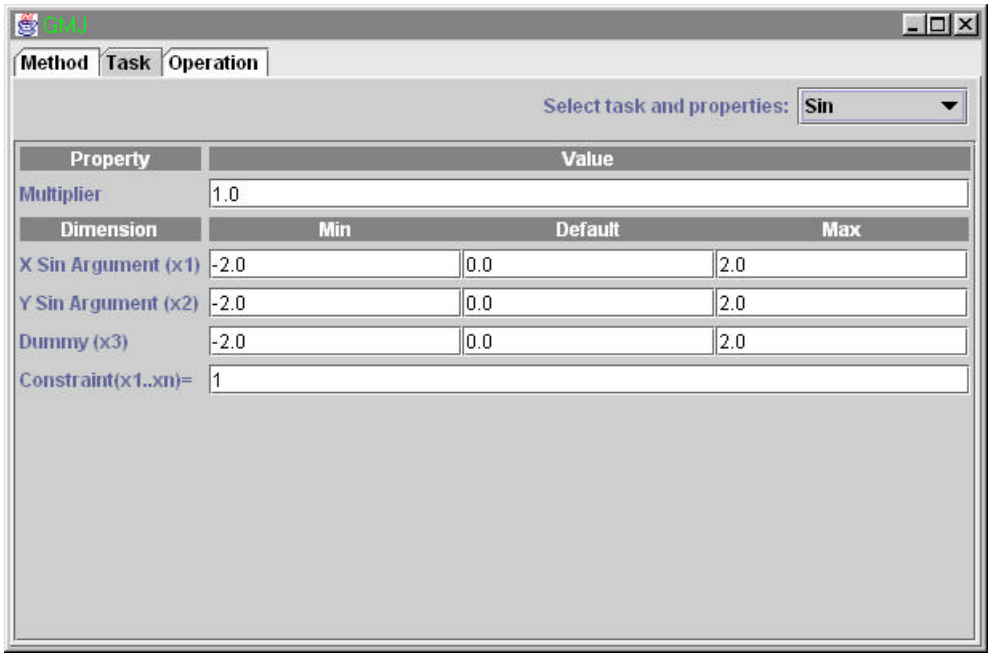
The configuration window depends on the actual method selected, but usually it will have an "Iterations" field, where the number of iterations, the method should do, can be changed.

If the entered value is invalid, or the number is out of bounds, the field will turn red and the value is not be stored.



3.2 Task selection

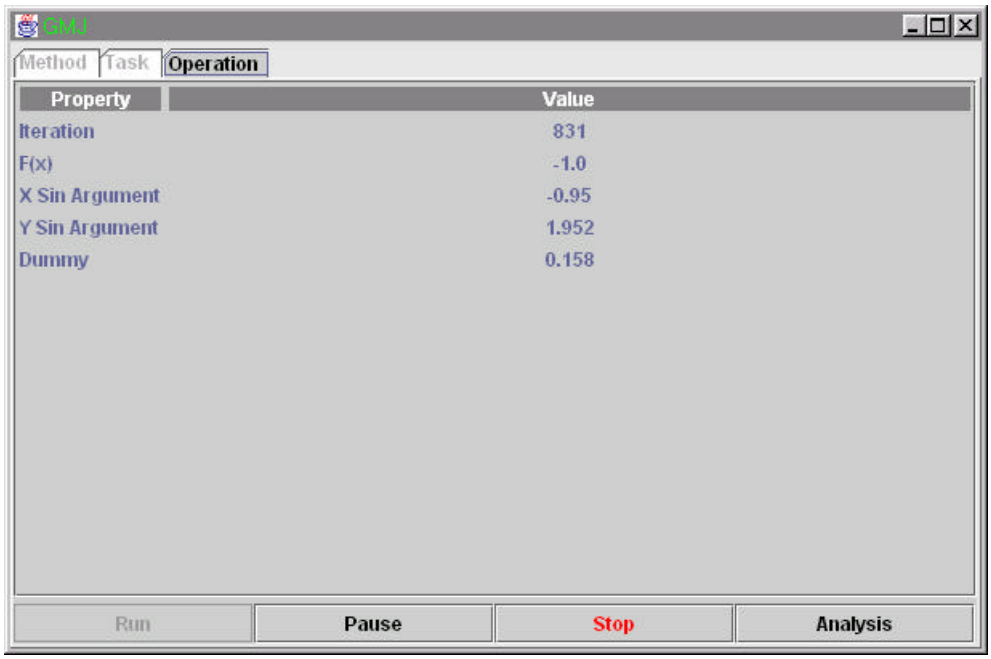
The choice control at the top of the page shows the list of available tasks. One of them should be selected and configured before calculations will take place. After selecting a task, it's implementation class is be loaded and the configuration window is rendered.



The configuration window depends on the actual task selected. It will first list the properties of the task and then the range of the minimization domain.

3.3 Operation

The operation page controls the minimization process. It has contains three control buttons: ***run, pause*** and ***stop***. ***run*** starts the minimization using the selected method and the selected task. ***pause*** temporarily stops calculations, which can then be continued by clicking ***run***. ***stop*** abandons calculations.

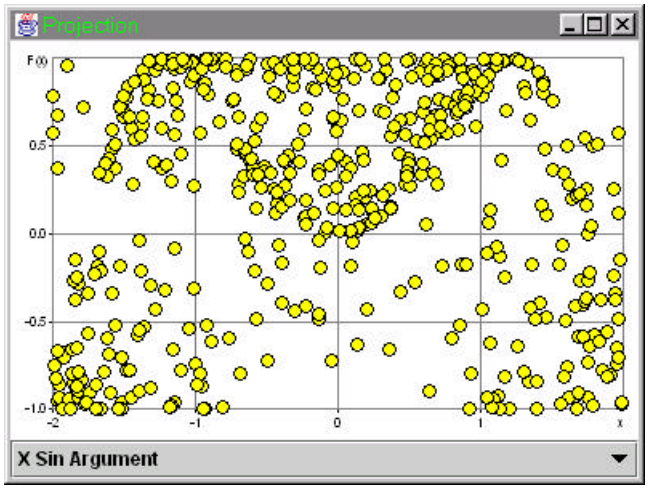


Results are showed in a formatted table:

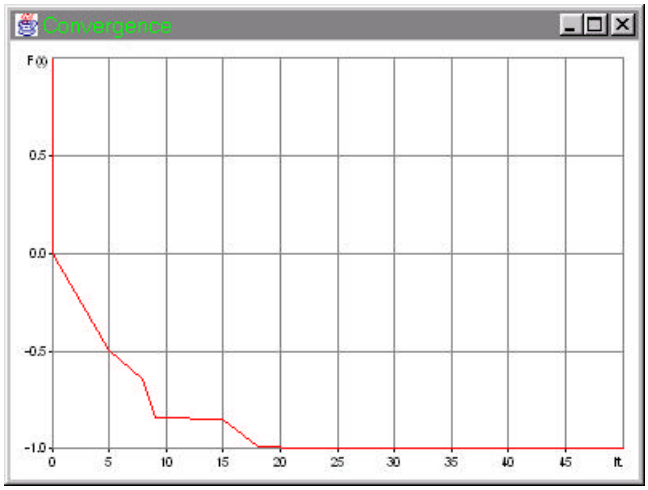
Iteration	Displays the iteration number, at which the current result was obtained
F(x)	Displays the value of the Task.f function at the result point
Calculation point values	Displays the coordinated of the point, at which the result was obtained
Analysis	Allows to invoke special analysis windows

3.4 Analysis windows

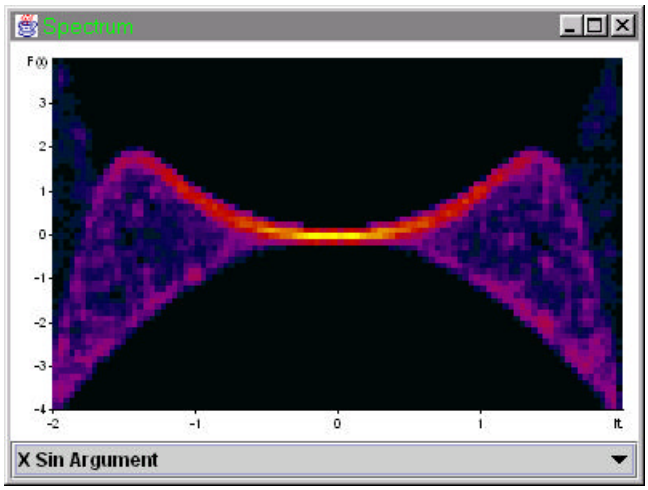
The projection windows show the relation between one of the dimensions of the domain of the minimization function and the value of the function.



The convergence window shows the progress of minimization on iteration basis



The spectrum window shows the distribution of the f(x) values with the respect of an iteration or any of the dimension variables.



4 Extending GMJ

The Java Development Kit (JDK) is required to compile additional Java files. Any Java development environment can be used, or the JDK for Microsoft Windows NT and Solaris from Sun Microsystems at http://java.sun.com can be obtained at no charge.

It is assumed, that the reader is familiar with the Java language and Object Oriented Programming in general.

4.1 Implementing minimization tasks

In order to solve custom minimization tasks, users have to develop and compile their own classes. The task class should implement a *Task* interface:

```
public interface Task extends Customizable  {
 Domain domain ();
 double f (Point pt);
};
```

The interface *Task* is derived from *Customizable*, which requires it to be configurable by the user. For further information on how to implement the configuration of the properties, refer to the section 4.3.

f calculates the value of the minimization function at the specified *Point*, which belongs to the *Domain* of the function.

domain returns the *Domain* of the minimization function, over which the function is to be evaluated. The task class should always return the same copy of the domain object, whenever *domain* is called. The *Domain* is a abstract class:

```
public abstract class Domain implements Customizable {
 public double min[];
 public double max[];
 public Point  defaultPoint;

 public abstract String[] dimensions ();

 ...
}
```

The abstract function *dimensions* should be overridden to return the descriptions of the dimensions of the domain of the minimization function. These descriptions are used in the analysis and configuration windows .

Variables *min*, *max* and *defaultPoint* are created by *Domain*, but should be initialized by the derived class, as required.

The *Point* class implements a point in a n-dimensional space of *Domain*:

```
public class Point {
 public double x[];

 public Point (Domain domain)
  ...
```

```
}
```

It constructs from a *Dimension* object and creates the required array accordingly. The *Point* class also has several useful methods to calculate distance between points, norms and etc.

Summary

To implement a new task the following should be done:

1. Derive a class from a class *Domain*
2. Provide a constructor for it to initialize the default dimension range and default point.
3. Override the method *dimensions* and return descriptions of the dimensions of the domain.
4. Create a new class, which implements the interface *Task*. It is advised to extend an abstract class *AbstractTask* instead – this will make the code more compatible with the future GMJ releases.
5. Override the method *domain*, which returns the **single copy** of the custom domain class
6. Override the method *f*

Example

```
class SinDomain extends Domain {
 static final String []dimensions={
  "X Sin Argument"
};

 public String[] dimensions ()
 { return dimensions; }

 SinDomain ()
 {
   min[0]=-2;
   max[0]= 2;
   defaultPoint.x[0]=0;
 }
};

public class Sin implements Task {
 private SinDomain domain=new SinDomain();

 public Domain domain ()
 { return domain; }

 public double f (Point pt)
 {
   return Math.sin (pt.x[0]);
 }
};
```

4.2 Implementing new minimization methods

GMJ can be extended by providing additional calculation methods, which might be more efficient, faster or more flexible. The method class should implement the *Method* interface:

```
public interface Method extends Customizable {
 int iterations ();
 Result run (ResultLogger l, Task t);
};
```

The interface *Method* is derived from *Customizable*, which requires it to be configurable by the user. For further information on how to implement the configuration of the properties, refer to the section 3.3.

iterations returns the number of iterations, the method will do.

As many of the minimization methods will have a configurable "iterations" property, the *MethodBase* abstract class is implemented on top of *Method* interface, which overrides the *iterations* and *customize* methods. By inheriting from *MethodBase* only the method *run* has to be overridden.

run runs the method on the specified *Task* and writes results to the specified *ResultLogger*. The method should return the best result. *ResultLogger* gathers the generated results and stores them. The *ResultLogger* interface defines the following members:

```
public interface ResultLogger {
 void log (Result r);
 Result logAt (int it);
};
```

log should be used to log the result of every iteration of the method. Usually the class, which implements *ResultLogger* will update the user interface objects to reflect the progress of the method, when *log* is called.

Some methods require the history of the minimization to be available to move on. The *logAt* method returns the *Result*, which has been logged previously (in iteration *it*) using *log* method.

A *Result* class is a container of the iteration state:

```
public final class Result {
 public int iteration;
 public Point point;
 public double value;

 ...
}
```

It holds the iteration number, the point, where the value of the *Task.f* was calculated, and the value of *Task.f* at this point.

4.3 Configuring the PropertyManager

Most of the worker objects have to be configured by the user before usage. For example, the user may want to change the number of iterations, the method should do, or the range of the domain of the task function. A *PropertyManager* is used to list properties, which may be changed. A *PropertyManager* is a property list, which is described by the interface

```
public interface PropertyManager {
 void removeAll ();
 void add (Property property);
 void complete ();
};
```

Method *add* should be used to add new *Property* objects to the list. A property consists of a property name or label and value editor. A value editor is used to change the value of the property.

The most important component of the Property is a *PropertyProvider*. It's an object, which knows how to retrieve the value of the property and how to store it after it has been changed by the editor. Because there are no pointers in Java, it is not possible to pass primitive objects by reference. So custom value retrieval and storage methods should be implemented.

The GMJ package provides a set useful property providers:

DoubleArrayProvider accesses the double array elements.

FieldProvider uses Java 1.1 Reflection classes to provide access to any data field of a class. It virtually eliminates the need to write custom property providers.

Example

The sample code shows how to add a property named "Initial Speed", which is stored as a second element of the array *customData*. Initial speed can vary in a range from 10 to 50 km/h.

```
manager.add (
     new DoubleProperty (
      "Initial Speed (km/h)",
       new DoubleArrayProvider (customData, 1),
      10, 50
     )
);
```

The following sample code shows how to add a property named "Offset", which is stored in a public variable *offset* of the calling class.

```
manager.add (
     new DoubleProperty (
      "Offset",
       new FieldProvider (this, “offset”),
      0, 1
     )
);
```

When property is added to the property manager, an editor of the appropriate type is created. When the user changes the value in the editor, the value is automatically checked, validated and stored in the second element of the *customerData*.

4.3.1 Writing a custom Property Provider

When a property value is stored as a field of the class, custom access class should be provided to access that value.

This class should implement the *PropertyProvider* interface, which requires a number of functions to be overridden. Most of these functions are implemented in a simple helper *SimplePropertyProvider* class, which can be used as base.

Example

The sample code shows how to provide access to a field *multiplier* of a *Sin* class, which is defined as *double*.

```
class SinMultiplierProvider extends SimplePropertyProvider {

 public Sin sin;

 public SinMultiplierProvider (Sin _sin)
 { sin=_sin; }

 public Object get ()
  { return new Double(sin.multiplier); }

 public void set (Object value) throws InvalidPropertyException
  { sin.multiplier=((Double)value).doubleValue(); }

};
```

Since GMJ II runs on Java 1.2, Java reflection is always available, so you may safely use *FieldProvider* class instead of writing your custom providers.

4.4 Implementing new analysis objects

GMJ can be extended by adding new analysis objects. The analyzer may be derived from any desirable base class. It can be implemented as a window or it may log data to external files. The only requirement for it is to implement a ResultAnalyser interface:

```
public interface ResultAnalyser {
 void prepare (ResultRepository r);
 void dispose ();
};
```

prepare is called, when the analysis object is constructed. It asks the analysis object to initialize. A *ResultRepository* object is passed as a parameter. It contains all required information for the analyzer to initialize.

Dispose is called when the analyzer is being deleted. It should flush any open buffers or close associated windows.

The data observer-notifier pattern is used to deliver real time calculation results to the analysis objects. The data notifier has a list of data observers, which are notified when new data is available. To receive these notifications, the object should implement the *ResultObserver* interface and register on the appropriate *ResultRepository*. The time to do the registration is in the *prepare* method. Be sure to unregister, when you decide to leave. If you are coding a graphical analysis window, you should unregister, when the window is closed by the user.

```
public interface ResultObserver {
 void newResult (ResultRepository l);
 void newProgressResult (ResultRepository l);
 void finalResult (ResultRepository l);
```

```
 void logStarted (ResultRepository l);
 void logDisposed (ResultRepository l);
};
```

logStarted signals that a new calculation is going to be started.

newResult signals, that a new result has just been added to the *ResultLogger*.

newProgressResult signals, that the method has made progress. It has just found a new minimum value.

finalResult signals that minimization is over.

logDisposed signals that the *ResultLog* is deprecated and can no longer be used. The observers should delete all references to the *ResultLogger* objects.

Example

This is a sample analyzer, which simply prints the progress to the console window.

```
public class DemoAnalyzer implements ResultAnalyzer, ResultObserver {
 protected ResultRepository results=null;

 public void prepare (ResultRepository n)
 {
   results=n;
   results.addObserver (this);
 }

 public synchronized void logStarted (ResultRepository n)
 {
   System.out.println (“START”);
 }

 void newResult   (ResultRepository l) {}
 void finalResult (ResultRepository l) {}

 void newProgressResult (ResultRepository l)
 {
   System.out.println (“Progress was made at iteration “+
                       l.lastResult().iteration);
 }

 public void logDisposed (ResultRepository r)
 {
   System.out.println (“STOP”);
   dispose ();
 }
};
```

4.5 Applet security issues

Some potential uses of GMJ may involve reading/writing a file on a client machine or connecting to a host, different than the one, from which the applet was loaded. When GMJ is loaded as a stand-alone application, there are no security restrictions, but when it’s loaded as an applet it is forced to run in a so called “sandbox”. To escape the sandbox, the developer has to prove, that he didn’t have any evil intensions and that the applet is “safe”. The approach used in Java environment is a digital signature.

Digital signature is a digital proof, that the code was not changed by a third person, and that it was packed and prepared to run by a developer, who's responsible for the code. To create a digital signature a developer has to get a personal key with a certificate, that that key is valid. Companies, such as Versign provide these services.

Due to standard incompatibilities, signing applets for Netscape, Microsoft Internet Explorer and Sun Java Runtime Environment differs. Signing for Netscape involves code changes, thus requires writing non-standard Java code, and thus is not recommended at all.

This security topic is out of scope of current document. More information can be obtained at a Java Signing FAQ at http://www.fastlane.net/~tlandry/javafaq.txt

5 Changes to GMJ II

5.1 Custom result analyzers

There are cases, when a task specific output should be shown. The output class can be implemented as an analysis object, as described in 4.4. The only inconvenience with this approach is that the analysis object will be available, when any task will be selected.

To make the analysis object show up only when the desired task is selected, modify the implementing Task class to implement a *TaskWithAnalyzers* interface.

```
public interface TaskWithAnalyzers extends Task {
 Class[] analyzers () throws ClassNotFoundException;
};
```

The *analyzers* function should return a class list of the custom analyzers to the task.

Example

The sample code shows a sample task implementation, which states, that there is a custom analyzer, called "Diagram".

```
class DemoTask extends AbstractTask implements TaskWithAnalyzers {
...
public Class[] analyzers () throws ClassNotFoundException
{
  Class diagramClass=Class.forName("lt.ktu.gmj.demo.Diagram");
  return new Class[]{ diagramClass };
}
...
};
```

5.2 Domain constraint function

Sometimes, minimization domain, bounded only by minimum and maximum values is not sufficient. A special constraint function can and should be used to enable specification of any nonrectangular domain.

Class *Domain* now has a function

```
public double constraintAt (Point pt)
```

which calculates a constraint value at the specified point. It returns 1 by default.

When writing new methods, one should always check the constraint function before calling the function *f* of the Task. If the method doesn't support constraints, simply multiply constraint by *f* to get the resulting value, to do minimization on.

When implementing a task, one can derive the custom domain class from a *DomainWithConstraint* class, to allow the user specify the constraint function at run time. *DomainWithConstraint* adds an entry field in the task configuration panel, where any algebraic expression can be entered and compiles it at run time. This way, the user can specify any constraint function, he wants.

6 Programming tips

6.1 Aborting calculations

In very intensive calculations it is advisable to check for abortion user request. When the user presses STOP button, or when the applet is about to exit, an *interrupted* flag is set for the calculation thread. The task and method writers are advised to check for:

```
if (Thread.interrupted())
  throw new ResultLogger.AbortedError ();
```

condition and abort any calculations as soon as possible by throwing the *AbortedError* exception.

6.2 Implementing interfaces

When creating custom method or task, your class should implement the required interface. It is advised to extend a abstract class (if there is a matching one) instead – this will make the code more compatible with the future GMJ releases.

For example extend *AbstractMethod* instead of implementing *Method*.

7 Bug Report

Bugs should be reported by e-mail to modestas_g@equinoxlt.com. Please put "GMJ Bug" in the subject field.

8 References

The author of this document and designer of GMJ	modestas_g@equinoxlt.com
The built-in minimization method author	jonas@optimum.mii.lt
Java Home Page	http://java.sun.com