# GRIDS_1.0 core

# A Java™ package for processing numeric 2D square cell raster data

CCG, School of Geography, University of Leeds

Working Paper Version 0.4

December, 2005

Andy Turner

A.G.D.Turner@leeds.ac.uk

## 1. Introduction

This is the first of a planned set of publications about GRIDS_1.0(beta) - software written in Java™ 2 (Java) based on the Java 1.4.2 Stadard Development Kit - that is geared for the analysis of geographic raster data. The software is geared for processing very large heterogeneous two-dimensional (2D) square celled raster data sets (*grids*) that ordinarily would not fit in available fast access memory of computers regardless of the data structure used. This is Version 0.4 of this document and is the first to be put online.

GRIDS_1.0(beta) is open source and was first released in beta form under the GNU Lesser General Purpose Licence via the following URL in March 2005:

http://www.geog.leeds.ac.uk/people/a.turner/src/java/grids

This paper details the core package. The other packages that currently make up the software are:

- exchange
- process
- utilities

The exchange package contains classes with methods relevant for importing and exporting data. The process package contains classes of methods for manipulating, combining and generating new *grids*. The utilities package contains classes that are used in the other packages and that are likely to be of more general use. The core package is dependent on the utilities package. Some classes of the core package import classes from third party software. All third party software is distributed with the GRIDS_1.0(beta) software bundle and details are provided in the relevant sections below.

To date the software has been a closed development produced by a single individual supporting a very small user community. This has allowed for the code to be refactored

radically during development, but is not sustainable in the long term. Others are encouraged to get involved in a more open user/developer community to work top produce a more robust offering with many tests put in place so that functionality and capabilities can be preserved. There are several ways forward. Perhaps the best option is to integrate this software into a more extensive library adhering to more rigid syntax, testing and documentation requirements.

Section 2 outlines the basic data grid data framework. Section 3 describes memory handling. Section 4 provides some details on data structures employed. Section 5 is to describe factory and iterator classes for building and populating the data structures and going through the data values respectively. Section 6 details the use of statitics objects which are attached to data structures for providing fast and easy access to summary information. Section 7 is for detailing further work.

# 2. The Basic Grid Arrangement

The basic unit of a *grid* is a *cell*. *Cells* are arranged in *rows* and *columns*, aligning with orthogonal axes *x* and *y*. Each *cell* can be thought of as being square shaped, with a value at centre. Really *cells* are little more than a *value* at a specific point location with abstract boundaries. Currently each *cell* has a value stored as a `Java primitive`, either an `int` or a `double`.

A *grid* is made up of *chunks* of *cells* arranged in *rows* and *columns*. *Chunks* are 'sub raster blocks' (Mineter, 1998), i.e. discrete rectangular block of cells. *Chunks* like *cells* are arranged in *rows* and *columns* aligning with the orthogonal *x* and *y* axes. The organisation of *cells* into *chunks* offers flexibility. A *grid* can be comprised of many different types of *chunk*, so each *chunk* can be stored using a different data structure. Also, *chunking* enables a simple means of *caching* and *swapping* data.

*Chunks* are *serializable* (as are *grids*) because they are constructed from classes that implement the `java.io.Serializable` interface. This means that they can readily be swapped via `ObjectOutputStreams` and `ObjectInputStreams`. Operationally, this feature is geared for a specific type of memory management. Many computers have a fast access memory as well as a larger slower type of memory commonly referred to as *disc*. Computation is fastest if the data being integrated is loaded in the fast access memory. It is slower if this data first needs loading, and is slower still if storage in the fast access memory needs organising to enable the data to be loaded. If the fast access memory becomes full whilst a calculation is being performed, some memory management has to be done to save overwriting data or program that is required. Usually an attempt must be made to suspend the calculation while an attempt is made to *swap* or transfer data (that is not in use) to another store. If the data that was *swapped* is needed in a subsequent calculation, then attempts can be made to load it into the fast access memory again. Most operating systems perform some form of *swapping* operation. Within the memory limits of the Java Virtual Machine `GRIDS_1.0(beta)`explicitly handles the swapping of data to attempt to provide sufficient memory for calculations to be made.

There is a computational cost of transferring data from one store to another. The more transfers that are necessary, the higher the cost will be. Additionally there are transfer bottlenecks which mean that time can be saved by loading or swapping in advance so

that the data in the fast access memory is that required for the upcoming calculations. The part of a *grid* that is loaded in the fast access memory of a computer is called the *cache*.

The main advantages of the chunk structure are that:

- potentially, each *chunk* can be stored optimally using any of a number of data structures; and,
- each *chunk* can be readily swapped between different memory stores of a computer.

To recap, there are two almost identical types of *grid* one type deal with *cell* values that are of an `int` type, the other deal with *cell* values that are of a `double` type. The classes for handling these both extend an `abstract class Grid2DSquareCellAbstract` which provides inner classes for `CellID` and `ChunkID` which can be used to uniquely identify a *cell* and a *chunk*, and which provides general referencing and geometry methods relevant to all extended classes. The abstract class also acts as an interface controlling what methods extended classes must implement.

# 3. Memory Handling

Memory handling has been implemented by essentially wrapping each method body in a `try{}catch(java.lang.OutOfMemoryError e){}` statement block. If memory handling is enabled and a `java.lang.OutOfMemoryError` is thrown during the execution of the code in the `try{}` block, then an attempt is made to clear some data from the cache and then the method is called again in a recursive manner. If the method is set not to do `OutOfMemoryError` handling the caught error is simply thrown. Clearing data from the cache involves three steps. Firstly, a small amount of pre-initialised memory is cleared. This freed memory is enough to allow for an `ObjectOutputStream` to be constructed to write data out to a file. Secondly, data is written to file via the constructed `ObjectOutputStream`. Finally, a small amount of memory is initialised for future use. Figure 3.1 is a Java code block of a method illustrating the memory handling.

Figure 3.1 Java code block illustrating memory handling enclosure

```
/**
  * This method does not exist in the grids_1.0 package. It is an illustration of
  * how OutOfMemoryErrors are handled. The function of the method is to execute a
  * chain of methods based on args.
  * @param args An Object of arguments to be used in this method.
  * @param handleOutOfMemoryError If true then OutOfMemoryErrors are caught in
  *   this method then swaping and cache clearing operations are called prior to
  *   retrying. If false then OutOfMemoryErrors are caught and thrown.
  */
public void method0( Object args, boolean handleOutOfMemoryError ) {
  try {
    // Calculate result1;
    Object result1 = method1( handleOutOfMemoryError );
    // Calculate result1;
    method2( method1( args, handleOutOfMemoryError ), handleOutOfMemoryError );
  } catch ( OutOfMemoryError oome0 ) {
    if ( handleOutOfMemoryError ) {
      clearMemoryReserve();
      clearChunk( swapChunk() );
      initMemoryReserve();
      return method0( args, handleOutOfMemoryError );
    } else {
        throw oome0;
    }
  }
}
```

Since an `OutOfMemoryError` is only encountered at runtime developing comprehensive testing suites for this code is challenging.

# 4. Different Types of *Chunk*

There are two sets or broad types of *chunk* that are almost identical, one for `int` type *cells* and the other for `double` type *cells*. For each set there are the following types of *chunk*:

- `64CellMap`
- `2DArray`
- `JAI`
- `Map`
- `RAF`

Each type of *chunk* employs a data structure to store data and is distinguished by the way in which it stores and retrieves data values from the structure. The various data structures offer specific advantages in specific circumstances. Essentially, individual *cell* values in each *chunk* can be retrieved and set to new values. Additionally, each *chunk* either contains or has access to methods which can provide information about the data content of the *chunk*.

The number of *cells* each type of *chunk* can have is limited. For some types of *chunk* the limit is based on the number of *cells*, or the number of *rows* and *columns*; for others, the limit is based on the diversity of *cell* values. Although *chunks* may address a large number of *cell* values, they are best to be of a size that allows an entire row to be loadable into the fast access memory of computers intended to use them. *Chunks* are intended to be relatively small compared to the overall *grid*, in that, it is generally best to have; more rows and columns of *chunks* in the *grid*, than *rows* and *columns* of *cells* in each *chunk*. It may also be likely advantageous to specifying some binary round number (e.g. 64, 128, 256, 512, 1024, etc...) for the number of *rows* and *columns* of *cells* in each *chunk*.

*Chunks* are lightweight in that they hold almost nothing except the *cell* values (or in one case, a reference to them). Individual *chunks* do not store any statistics (e.g. the mean) of the *cell* values it has, yet, *chunks* contain methods for calculating such statistics that override more general methods.

There are advantages to be gained in using different structures to store *chunks* and retrieve information about their *cell* values. Different *grids* of data can be stored optimally in different ways depending on what is required. Optimisation involves:

- retrieving *cell* values, information about the *grid* and regions of it as fast as possible;
- changing/setting *cell* values as fast as possible; and,
- working within memory limitations, (attempting to use as little memory, but as high a proportion of fast access memory as possible).

For any data set (*grid*), given a memory limit and sequence of operations to perform; there may exist an optimal configuration with respect to the number of *rows* and *columns* in each *chunk*, and the types of each *chunk*.[1] Chancing on this configuration is highly unlikely, but it is equally unlikely that a complete optimisation is necessary.

*Grids* can be modified by changing *chunks* from one type to another. Where one *chunk* type will be more efficient, both in terms of memory storage and individual *cell* value retrieval, having this *chunk* type is highly desirable, but, the construction of it comes at a cost.

In `Grids_1.0(beta)`, *grid* constructor methods do not automatically switch to more optimal *chunk* types during *grid* construction, although this could be a good time to do it. What happens is that a *chunk factory*, that produces a particular type of *chunk*, is passed to the *grid* constructor and it is these types of chunk that are constructed.

The number of *rows* and *columns* each *chunk* will be comprised of can be specified, as can what type a *chunk factory* to use (there are defaults). It is unlikely for any *chunk* type to simultaneously be optimal for memory and computational speed for a given set of operations. Usually one data structure will offer the fastest access to specific types of information about its values, and another will offer the most memory efficient storage. Switching from one structure to the next is somewhat expensive, and keeping track of changes in more than one structure is also expensive.

The remainder of this section describes each type of *chunk* in turn.

## *4.1* ***64CellMap Chunks***

This somewhat sophisticated data structure is limited to *chunks* with at most 64 *cells*. The advantages of this data structure are potentially huge and require further testing as a storage model for other kinds of 2D and three dimensional (3D) spatial data. For this, *grids* handling `Java` *primitives* of a `boolean` type are wanted.

The *chunk* `64CellMap` class imports part of the GNU Trove library (GNU, 2004) to store data in a fast, lightweight implementation of the `java.util` Collections API

[1] Potentially more than one configuration will perform the processing in an equal amount of time.

(Java, 2005)[2]. Each different *value* in a `64CellMap` *chunk* has a *key* entry in a `HashMap`. Each *key* is mapped to a `long` value which in this instance codes the locations of the 64 cells that have this particular value. Each *key* and each *value* are unique.

There are $2^{64}$ `long` values and these give all the possible combinations of a `boolean` mapping to the 64 cells in the *chunk*. The bit mapping of the long value is what codes the locations to which the *key* applies.

So, for *chunks* that contain a single *cell* value there is a single mapping in the `HashMap` and for *chunks* with 64 different *cell* values there are 64 mappings in the `HashMap`. Iterating over (going through) the *keys* in the `HashMap` is necessary to get and set *cell* values, so generally this works faster, the smaller the number of mappings.

The speed of data retrieval is uncertain, since it may be the first or the last *key* presented by an iterator that is the one being sought.

A mapping of *keys* (cell values) and *values* (cell identifiers) is a general way of storing *grid* data. It is very efficient in terms of memory use where a default value can be set, and if there are only a small number of non-default mappings in the *chunk* (compared to the number of *cells* in the chunk). Such maps also offer the means to generating some statistics about a *chunk* very efficiently. In particular, the diversity (number of different values) can be calculated readily and the mode (the mean of the most commonly occuring values) is also tends to be able to be calculated quickly. For other types of statistics the efficiency is more constrained by the number of mappings than it is by the generic speed of retrieving an individual *cell* value.

## *4.2 2DArray Chunks*

2DArray *chunks* are effectively arrays of arrays organised in *rows* and *columns*. They are primitive arrays of either `double[][]` or `int[][]` types. This *chunk type* is generally the most efficient if the diversity of values in a *chunk* is large. The 2D arrays are indexed via `primitive int` values, so setting and retrieving values is straightforward. The arrays can be initialised with `java.util.Arrays.fill( array, value )` where array is the array to be filled and value is the value to fill it with.

## *4.3 JAI Chunks*

JAI *chunks* uses classes from the Java Advance Imaging API (JAI, 2001). The data is stored in a `javax.media.jai.TiledImage` and can be readily visualised or written to an image file. To use this kind of chunk the Java Advance Imaging software is required.

---

[2] Since the release of `Java 1.5` the utility of the GNU Trove library is questionable due to the new features of the language allowing for autoboxing/unboxing.

### *4.4 Map Chunks*

Map chunks are very similar to 64cellmap chunks except that they are not limited to 64 cells. Rather than using a primitive-primitive Trove HashMap, these use a primitive-Object Trove HashMap. The Object (map value) is either a ChunkCellID as defined in the abstract chunk class, or a collection of them.

### *4.5 RAF Chunks*

RAF chunks store data on disc accessed via a `java.io.RandomAccessFile`. These chunks are little used since the caching/swapping and OutOfMemoryError handling functionality has been implemented.

## 5. Factory and Iterator classes

Every type of *grid* and *chunk* has an associated factory class for constructing instances of it and an iterator class for iterating over the *cell* values in them.

## 6. Statistics classes

Every grid is adorned with one of two types of statistics object. With one type, a number of statistics about the grid are initialised as the grid is constructed and these fields are kept up to date as the underlying data cell values are modified. The computation involved is wasted if the statistics are not going to be used, but if they are, then time may be saved by having them readily available. The following statistics fields may be kept up to date:

- nonNoDataValueCountBigInteger – number of cells with non noDataValues.
- sumBigDecimal - the sum of all non noDataValues as a BigDecimal.
- minBigDecimal - the minimum of all non noDataValues as a BigDecimal.
- minCountBigInteger - the number of min values as a BigInteger.
- maxBigDecimal - the maximum of all non noDataValues as a BigDecimal.

## 7. Further Work

At this stage development might best be split:

- `GRIDS_1.X` to continue with bug fixes and enhancements based on the `GRIDS_1.0 core`; and,
- `GRIDS_2.0` to be a new branch devoted to the development and application of an enhanced `core`.

A specification for `GRIDS_2.0` is perhaps the first step in this line of development. It may be that the first step is support for `boolean`, `BigDecimal`, and other `Object` type *cell* values.

It may also be that more fundamental and abstract improvements are wanted, such as; support for three-dimensional grids, or support for raster models based on triangular equidistant points of measurements.

Perhaps prior to any of this, open source development strategies should be considered and some model for development implemented. `GRIDS_1.0(beta)` has been developed by an individual, but this is not sustainable in the long term. To maximise the utility and use of the code it is probably best developed by a team. Maybe this is best achieved by integrating the code into other ongoing software development efforts, such as GeoTools (GeoTools, 2005). For the code to be developed by multiple users, a common repository for the source and the implementation of a full suite of unit tests are wanted. Much other work is necessary to make `GRIDS_1.0` robust, flexible and user friendly, and to implement the standards specified by the Open Geospatial Consortium (OGC, 2005).

Work is needed to evaluate the merits of different types of chunk. The 64CellMap chunk is described as being very sophisticated, but does it offer clear advantages for any data sets? It may be that this data structure is best suited to binary (boolean or two-values) grids. Indeed for this the mapping can be thought of as a possible substitute for a vector storage model. One number effectively storing the pattern of a points, lines or regions.

## Acknowledgements

The European Commission supported this work under the following contracts:
IST-1999-10536 (The SPIN!-project)
EVK2-CT-2000-00085 (MedAction)
EVK2-CT-2001-00109 (DESERTLINKS)
EVK1-CT2002-00112 (tempQsim)

The ESRC supported this work under the following contract:
RES-149-25-0034 (MoSeS)

Thanks to James MacGill and Ian Turton for encouraging me to code Java. Thanks also to all the academics and student that have encouraged me to develop this software. Without your help, encouragement and support, this work would not be published.

## References

JAI (2005) *Java Advanced Imaging API.* Accessed on 2nd February 2005 via http://java.sun.com/products/java-media/jai/

Java (2005) *Java™ 2 Platform Standard Edition 5.0 API Specification.* Accessed on 2nd February 2005 via http://java.sun.com/j2se/1.5.0/docs/api/

Mineter M. J. (1998) *Partitioning Raster Data.* Chapter 10 of Healey R. G., Dowers S., Gittings B.M., Mineter M.J. (eds.) Parallel Processing Algorithms for GIS. Taylor & Francis.

OGC (2005) *The Open Geospatial Consortium.* Accessed on 2nd February 2005 via http://www.opengeospatial.org/

GeoTools (2005) *GeoTools open source Java GIS toolkit.* Accessed on 2nd February 2005 via http://www.geotools.org/

GNU (2004) *GNU Trove: High performance collections for Java*. Accessed on 5th November 2004 via http://trove4j.sourceforge.net/